A Declarative Approach to Modeling and Solving the View Selection Problem

Imene Mami

Université Montpellier II
— Sciences et Techniques du Languedoc —

Thèse

pour obtenir le grade de
Docteur de l'Université Montpellier II

Discipline : Informatique
Spécialité Doctorale : *Informatique*
Ecole Doctorale : *Information, Structure, Systèmes*

présentée et soutenue publiquement par

Imene MAMI
le 15/11/2012

A Declarative Approach to Modeling and Solving the View Selection Problem

Jury

Bernd Amann, Professeur, Université de Pierre et Marie Curie (Paris 6), ... Rapporteur
Ladjel Bellatreche, Professeur, Ecole Nationale Supérieure de Mécanique et d'Aérotechnique (ENSMA), ... Rapporteur

Pascal Giorgi, Maitre de conférence, Université Montpellier II, ... Examinateur
Mohand-Said Hacid, Professeur, Université Claude Bernard Lyon I, ... Examinateur

Zohra Bellahsene, Professeur, Université Montpellier II, ... Directrice de thèse
Rémi Coletta, Maitre de conférence, Université Montpellier II, .. Co-encadrant de thèse

Abstract

View selection is important in many data-intensive systems e.g., commercial database and data warehousing systems to improve query performance. View selection can be defined as the process of selecting a set of views to be materialized in order to optimize query evaluation. To support this process, different related issues have to be considered. Whenever a data source is changed, the materialized views built on it have to be maintained in order to compute up-to-date query results. Besides the view maintenance issue, each materialized view also requires additional storage space which must be taken into account when deciding which and how many views to materialize. The problem of choosing which views to materialize that speed up incoming queries constrained by an additional storage overhead and/or maintenance costs, is known as the view selection problem. This is one of the most challenging problems in data warehousing and it is known to be a NP-complete problem. In a distributed environment, the view selection problem becomes more challenging. Indeed, it includes another issue which is to decide on which computer nodes the selected views should be materialized. The view selection problem in a distributed context is now additionally constrained by storage space capacities per computer node, maximum global maintenance costs and the communications cost between the computer nodes of the network.

In this work, we deal with the view selection problem in a centralized context as well as in a distributed setting. Our goal is to provide a novel and efficient approach in these contexts. For this purpose, we designed a solution using constraint programming which is known to be efficient for the resolution of NP-complete problems and a powerful method for modeling and solving combinatorial optimization problems. The originality of our approach is that it provides a clear separation between formulation and resolution of the problem. Indeed, 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 for optimization purpose. The main contributions of this thesis are as follows. First, we define a framework that enables to have a better understanding of the problems we address in this thesis. We also analyze the state of the art in materialized view selection to review the existing methods by identifying respective potentials and limits. We then design a solution using constraint programming to address the view selection problem in a centralized context.

Our performance experimentation results show that our approach has the ability to provide the best balance between the computing time to be required for finding the materialized views and the gain to be realized in query processing by materializing these views. Our approach will also guarantee to pick the optimal set of materialized views where no time limit is imposed. Finally, we extend our approach to provide a solution to the view selection problem when the latter is studied under multiple resource constraints in a distributed context. Based on our extensive performance evaluation, we show that our approach outperforms the genetic algorithm that has been designed for a distributed setting.

Keywords materialized views, query processing and optimization, view selection, view maintenance, constraint programming.

TITRE en français : **Une approche déclarative pour la modélisation et la résolution du problème de la sélection de vues à matérialiser**

Resumé

La matérialisation de vues est une technique très utilisée dans les systèmes de gestion de bases de données ainsi que dans les entrepôts de données pour améliorer les performances des requêtes. Elle permet de réduire de manière considérable le temps de réponse des requêtes en pré-calculant des requêtes coûteuses et en stockant leurs résultats. De ce fait, l'exécution de certaines requêtes nécessite seulement un accès aux vues matérialisées au lieu des données sources. En contrepartie, la matérialisation entraîne un surcoût de maintenance des vues. En effet, les vues matérialisées doivent être mises à jour lorsque les données sources changent afin de conserver la cohérence et l'intégrité des données. De plus, chaque vue matérialisée nécessite également un espace de stockage supplémentaire qui doit être pris en compte au moment de la sélection. Le problème de choisir quelles sont les vues à matérialiser de manière à réduire les coûts de traitement des requêtes étant donné certaines contraintes tel que l'espace de stockage et le coût de maintenance, est connu dans la littérature sous le nom du problème de la sélection de vues. Trouver la solution optimale satisfaisant toutes les contraintes est un problème NP-complet. Dans un contexte distribué constitué d'un ensemble de noeuds ayant des contraintes de ressources différentes (CPU, IO, capacité de l'espace de stockage, bande passante réseau, etc.), le problème de la sélection des vues est celui de choisir un ensemble de vues à matérialiser ainsi que les noeuds du réseau sur lesquels celles-ci doivent être matérialisées de manière à optimiser les coût de maintenance et de traitement des requêtes.

Notre étude traite le problème de la sélection de vues dans un environnement centralisé ainsi que dans un contexte distribué. Notre objectif est de fournir une approche efficace dans ces contextes. Ainsi, nous proposons une solution basée sur la programmation par contraintes, connue pour être efficace dans la résolution des problèmes NP-complets et une méthode puissante pour la modélisation et la résolution des problèmes d'optimisation combinatoire. L'originalité de notre approche est qu'elle permet une séparation claire entre la formulation et la résolution du problème. A cet effet, le problème de la sélection de vues est modélisé comme un problème de satisfaction de contraintes de manière simple et déclarative. Puis, sa résolution est effectuée automatiquement par le solveur de contraintes. De plus, notre approche est flexible et extensible, en ce sens que nous pouvons facilement modéliser et gérer de nouvelles contraintes et mettre au point des heuristiques pour un objectif d'optimisation. Les principales contributions de cette thèse sont les suivantes. Tout

d'abord, nous définissons un cadre qui permet d'avoir une meilleure compréhension des problèmes que nous abordons dans cette thèse. Nous analysons également l'état de l'art des méthodes de sélection des vues à matérialiser en en identifiant leurs points forts ainsi que leurs limites. Ensuite, nous proposons une solution utilisant la programmation par contraintes pour résoudre le problème de la sélection de vues dans un contexte centralisé. Nos résultats expérimentaux montrent notre approche fournit de bonnes performances. Elle permet en effet d'avoir le meilleur compromis entre le temps de calcul nécessaire pour la sélection des vues à matérialiser et le gain de temps de traitement des requêtes à réaliser en matérialisant ces vues. Enfin, nous étendons notre approche pour résoudre le problème de la sélection de vues à matérialiser lorsque celui-ci est étudié sous contraintes de ressources multiples dans un contexte distribué. A l'aide d'une évaluation de performances extensive, nous montrons que notre approche fournit des résultats de qualité et fiables.

MOT-CLES vues matérialisées, optimisation de requêtes, sélection de vues, maintenance de vues, programmation par contraintes.

INTITULE ET ADRESSE DE L'U.F.R. OU DU LABORATOIRE
Le Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier
LIRMM, 161 rue Ada, 34095 Montpellier Cedex 5 - France

To the memory of my father

Acknowledgements

This thesis report would not have been possible without the guidance and the help of my supervisor Dr. Zohra Bellahsene. I thank her for being my mentor in research. Her guidance, encouragement and support were available any time I needed. I am also very thankful to my co-supervisor, Dr. Remi Coletta, who showed a lot of commitment and attended almost every meeting related to this work.

I would like to express my gratitude to the honorable reviewers, Dr. Bernd Amann and Dr. Ladjel Bellatreche for accepting the task to evaluate my thesis. I thank them for their valuable comments and remarks. I am grateful to Dr. Pascal Giorgi and Dr. Mohand-Said Hacid for taking out time from their hectic schedules, to act as examiners for my work.

My stay in France for pursuing the doctorate degree has been financially supported by the Government of France. The research work presented in this thesis has been performed in the Laboratoire d'informatique de Robotique et de Microélectronique de Montpellier (LIRMM), University of Montpellier 2, France. As a member of Zenith team (INRIA-LIRMM), I got a lot of help and encouragement from all the members of the group. I forward my extreme appreciations to all colleagues.

I would like to mention my family members for their encouragement, especially my mother who has always filled my life with generous love and unconditional support. I am deeply thankful to my fiance for showing continuous cooperation, backing and patience during the course of Phd. His support to pursue this goal had been enormous.

Montpellier, Novembre 2012

Imene MAMI

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations

Data-intensive systems i.e., commercial database and data warehousing systems allow businesses to get data and turn that data into a useful information. However, the time taken to compute query results exponentially grows as the amount of data increases leading to more waiting time on the user side. This delay is unacceptable in most business environments, as it severely limits productivity. A common and powerful query optimization technique is to materialize some or all queries of the workload rather than compute them from data source each time. There are several scenarios in which we investigate the issue of which views (or queries) to materialize in order to speed the query processing when it is too expensive to materialize all the views.

- **Performance of query processing**. The goal is to select a set of views to be materialized over a database, such that subsequent queries can make use of these views in query processing. In most cases, it is cheaper to read the content of a materialized view than to compute from scratch the associated query. Consequently, choosing an appropriate set of views to materialize in a database is crucial since the presence of the right materialized views can significantly improve the query performance. Many commercial database systems i.e., SQL database systems support creation and use of materialized views to answer queries in order to facilitate efficient query processing.

- **Warehouse design**. One of the most important tasks when designing a data warehouse is a judicious selection of materialized views. A data warehouse stores information that is collected from multiple, heterogeneous information sources, with the purpose of efficiently implementing decision support queries

(OLAP-style queries). The information in the data warehouse is typically organized in materialized views, which are designed based on the user's requirements e.g., pre-computed portions of the frequently asked queries. In this way, the query processor of the warehouse answers queries without interacting with the data sources that may contain several millions of tuples. Scanning these data may be time consuming and wasteful. Hence, the benefit of using only materialized views to answer queries is significant for improving performance in a data warehousing environment.

- **Data placement in a distributed setting**. Choosing which views to materialize can be considered in a distributed setting to optimize complex scenarios consisting of multiple computer nodes with different resource constraints, where each computer node issues different types of query characteristics. The key idea to improve query performance in such a context is the intelligent placement of data at different computer nodes of the network. For instance, query results may be stored as materialized views and placed closest to where they will most likely be accessed. When processing queries, the materialized views can be used to speed up local queries and reduce the amount of communication between the computer nodes of the network. Distributed query processing is a key factor in business environments in order to remain competitive.

We have been motivated by these scenarios to study how to select the right materialized view that can significantly improve performance and speed up the processing of queries by several orders of magnitude.

1.2 View Selection Problem

View selection can be defined as the process of selecting a set of views to be materialized in order to optimize query evaluation. To support this 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 (i.e., base relation) 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.

Low query evaluation cost can be obtained by materializing all the queries of workload. However, it is important to note that it is not always a possible solution because of the storage space limitation (i.e., the query result can be too large to fit in the available storage space) and the cost of maintaining the views in order to keep them consistent with the data at sources. Hence, there is a need for selecting a set of views to be materialized by taking into account the view maintenance and storage space constraints. The problem of choosing which views to materialize that speed up incoming queries constrained by an additional storage overhead and/or maintenance costs to keep the views synchronized with the base data (I.e., base relations), is known as the view selection problem. This is one of the most challenging problems in data warehousing [74] and it is known to be a NP-complete problem [35]. In a distributed environment consisting of many heterogeneous nodes with different resource constraints, the view selection problem becomes more challenging: Besides the issue of deciding which views have to be selected, the problem includes the question where these views should be materialized.

The problem of view selection can be defined as follows. Given a database (or a data warehouse) schema and a query workload defined over it, the problem is to select an appropriate set of materialized views that minimizes the cost of evaluating the queries of the workload under a limited amount of resources, e.g., storage space and/or view maintenance cost. This problem will be defined more formally in chapter 4. In a distributed scenario, multiple computer nodes are connected to each other. Each computer node may share data and issue numerous queries against other computer nodes. The view selection problem in a distributed context is to compute which view has to be materialized on what computer node, so that the full query workload is answered with the lowest cost subject to multiple resource constraints. Resources may be storage space capacity per computer node, maximum view maintenance cost and network bandwidth (i.e., communication costs). The view selection problem in a distributed context will be defined more formally in chapter 5.

1.3 Objectives of the Dissertation

The central goal of the dissertation is that, we design a novel and efficient approach for the view selection problem in relational databases and data warehouses as well as in a distributed setting.

Explicitly, the objectives are to:

- Define a framework which gives the main notions and the basic contents related

to the view selection context that are required to be known when attempting to address the view selection problem.

- Analyze the state of the art in materialized view selection to review the existing view selection methods by identifying respective potentials and limits.

- Design a solution to the view selection problem in a centralized context, which can provide the best balance between the computing time to be required for finding the materialized views and the gain to be realized in query processing by materializing these views. This solution will also guarantee to pick the optimal set of materialized views where no time limit is imposed.

- Provide a solution which can provide high performance, when the view selection problem is studied under multiple resource constraints in a distributed context.

1.4 Contributions

In this section we explicitly outline our contributions to fulfill the above mentioned objectives.

First, we introduce the mains notions and concepts related to the view selection context. We provide definitions and a glossary of key terms in the domain of view selection. This study defines a basic framework that maybe helpful to the beginner to understand the view selection problem. Then, as our work is based on constraint programming techniques, we describe the main features of these techniques and the basics of modeling and solving with constraint solvers such as CHOCO [2].

Our second contribution consists in analyzing the existing works in materialized view selection. We identify the main view selection dimensions along which existing view selection methods can be classified. More specifically, we classify them based on what kind of algorithms they use to address the view selection problem, pointing which resource constraints they consider during the view selection process and frameworks they use to represent the view selection. We introduce three main classes of view selection algorithms, namely: deterministic algorithms, randomized algorithms and hybrid algorithms. Based on this classification, we survey and review the related works.

Our third contribution is the design of a novel approach to address the view selection problem in a centralized context i.e., relational databases and data warehouses. The approach is based on constraint programming techniques. Our motivation to

use constraint programming is that it is known to be a powerful approach for modeling and solving combinatorial problems. It is also an effective paradigm for the resolution of NP-complete problems. The idea of constraint programming is to solve problems by stating constraints which must be satisfied by the solution. Hence, the effort in our approach has been to model the view selection problem as a Constraint Satisfaction Problem (CSP). Its resolution was supported automatically by the constraint solver. We have designed the constraint satisfaction model to the view selection problem and performed several experiments, demonstrating the benefit of our approach.

Our fourth contribution aims at extending the constraint satisfaction model, which we have designed to address the view selection problem in a centralized context, in order to capture the distributed features. As mentioned before, the view selection problem becomes more challenging in a distributed environment. Indeed, the resource constraints that we have considered in a centralized context i.e., storage space constraint will be per machine (computer node) in a distributed scenario. The view selection will additionally constrained by maximum global maintenance costs. Furthermore, resource constraints such as network bandwidth and the location of materialized views will have to be taken into consideration. To the best of our knowledge, no past work has addressed the view selection problem under all these resource constraints. Our constraint programming based approach fills this gap. Indeed, all these resource constraints have easily been modeled with the rich constraint programming language. Experiment results have shown that our approach provides high performance resulting from evaluating the quality of the solutions found by our approach in terms of cost saving.

In the context of this PhD work, the following articles were published.

- International Journal papers
  - Imene Mami and Zohra Bellahsene. A Survey of View Selection Methods. ACM SIGMOD Record, pages 20-29, volume 41, 2012.
- International Conference papers
  - Imene MAMI, Rémi Coletta and Zohra Bellahsene. Modeling View Selection as a Constraint Satisfaction Problem. International Conference on Databases and Expert Systems Applications (DEXA), pages 396-410, 2011.

- Imene Mami, Zohra Bellahsene and Rémi Coletta. View Selection Under Multiple Resource Constraints in a Distributed Context. International Conference on Databases and Expert Systems Applications (DEXA), pages 281-296, 2012.

- National Conference papers

  - Imene Mami, Zohra Bellahsene and Rémi Coletta. A Constraint Satisfaction based Approach to View Selection in a Distributed Context. Dans 28ème Journées des Bases de Données Avancées (BDA), Clermont-Ferrand, France, October, 2012.

1.5 Structure of the Dissertation

This dissertation is organized into 6 chapters.

Current chapter is introducing the application domains in which the view selection has to be investigated and the general problem of view selection in a centralized context as well as in a distributed setting.

Chapter 2 provides a brief introduction to the key elements of the view selection field. We introduce the main definitions and concepts related to this field. This chapter provides the basic content which is required to be known to the researchers who are going to work on materialized view selection.

In chapter 3, we provide a literature review of the state of the art in materialized view selection. It defines a framework for highlighting the view selection problem by identifying the main dimensions that are the basis in the classification of view selection methods. Based on this classification, we review most of the view selection methods by identifying respective potentials and limits.

Chapter 4 presents our approach which is based on constraint programming to address the view selection problem in a centralized context. After the problem definition, we introduce the concept of the AND-OR view graph that is needed to represent the view selection and which constitutes the input to our approach. Then, we describe how to model the view selection problem as a Constraint Satisfaction Problem (CSP). We conclude with our experimental evaluation and results.

Chapter 5 extends the constraint satisfaction model that we have designed in a centralized context in order to capture the distributed case. After giving the problem definition in a distributed environment, we propose an extension of the concept of the AND-OR view graph to reflect the relation between views and communication network within the distributed scenario. Then, we present the model that we have

used to formulate the view selection problem as a CSP in a distributed environment. Finally, we present our experimental validation.

Chapter 6 concludes this dissertation and highlights future directions of research.

Chapter 2

Preliminaries

In this chapter, we introduce the main notions and concepts related to the issue of view selection. We introduce definitions about view selection and provide the details of cost modeling. One of the goals of the dissertation is to solve the view selection problem, by designing a solution involving constraint programming. Therefore, we describe in the second part of this chapter, the main characteristic features of constraint programming techniques.

2.1 View Selection

View selection is the task which consists of selecting a set of views to be materialized in order to improve query performance. We use the term view selection interchangeably with materialized view selection. To understand the basic concepts related to the view selection field, we first introduce the notion of a view. Let us consider the query q_1 defined over a simplified version of the TPC-H benchmark [5]. In our work, we are dealing with SQL queries which include select, project, join and aggregation operations. Query q_1 finds the minimal supply cost for each country and each product having the brand name 'Renault'. The associated query is as follows:

Select **P.partkey, N.nationkey, Min(PS.supplycost)**
From **Part P, Supplier S, Nation N, PartSupp PS**
Where **P.brand = 'Renault'**
and **P.partkey = PS.partkey**
and **PS.suppkey = S. suppkey**
and **S.nationkey = N.nationkey**
Group by P.partkey, N.nationkey;

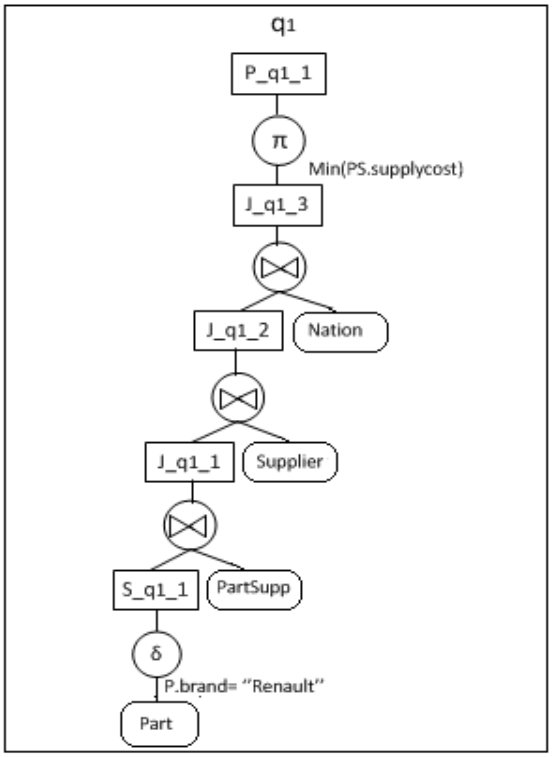


Figure 2.1: The query tree for query q_1.

A sample query tree for q_1 is shown in figure 2.1. Circles nodes represent algebraic expressions (Select-Project-Join) with possible aggregate function. Boxes nodes represent the result of evaluation of the relational algebra expression. The root node represents the query result and the leaf nodes represent the base relations.

2.1.1 Definitions

Definition 1 (View): A view is a derived relation, defined by a query in terms of base relations and/or other views. A view thus defines a function from a set of base relations to a derived relation. This function is typically recomputed each time the view is referenced (if the view is virtual). In figure 2.1, the views are represented by boxes nodes.

Definition 2 (Materialized View): A view is said to be materialized if its extent is computed and persistently stored otherwise it is said to be virtual. Our goal is to select a set of views to materialize. We refer to a set of selected views to materialize as a set of materialized views. In most cases it is cheaper to read the content of a materialized view than to compute the view from scratch.

Definition 3 (Candidate Views): The search space in the vocabulary of view selection represents the space of possible candidate views to be materialized. Our aim is to find among the candidate views, those optimizing the query performance. In our example (see figure 2.1), boxes nodes correspond to the views that are candidates to materialization.

Definition 4 (Workload): A workload or a query workload is a given set of queries defined over a database (or a data warehouse) schema. The set of materialized views is dependent on the query workload. In a distributed scenario, the queries are executed on different computer nodes. Each computer node has an associated query workload.

Definition 5 (View Benefit): A view benefit (or query benefit) is a useful notion is the view selection setting. This is defined as the reduction in the workload evaluation cost, which can be achieved by materializing this view.

Definition 6 (View Maintenance): Whenever a base relation is changed, the materialized views built on it have to be updated in order to compute up-to-date query results. The process of updating a materialized view in response to changes on the base relations is known as view maintenance.

Definition 7 (Incremental View Maintenance): Rather than refreshing the view by re-computing it from scratch, a process that may be time consuming and wasteful, a view can be maintained in an incremental fashion: only the portions of the view which are affected by the changes in the relevant sources (base relations) are updated. The process of computing only the changes in the view to update its materialization is known as incremental view maintenance.

Definition 8 (Distributed View Selection): In a distributed context, we consider more complex scenarios where multiple computer nodes are connected to each other and each computer node may share data and issue numerous queries against other computer nodes. Our objective thus is to compute which view have to be materialized on what computer nodes. We call this the distributed view selection.

Definition 9 (Solution Quality): In the context of view selection, the solution quality is evaluated by measuring the benefit of using materialized views to improve

query performance. More precisely, the solution quality represents the quality of the set of materialized views delivered by the view selection methods in terms of cost saving.

Definition 10 (View Selection Method): A view selection method implements one or many heuristic algorithms to efficiently search the space and find the appropriate set of materialized views within a reasonable time. Designing these heuristic algorithms also aims to find the right set of computer nodes on which these views should be materialized when the view selection is studied in a distributed environment.

2.1.2 Cost Model

The cost model is an important issue for the view selection process [16]. It assigns an estimated cost (e.g., query cost or maintenance cost) to any view in the search space. In our work, we use a cost model similar to [64, 46, 21]. Hence, the query and view maintenance costs are estimated with respect to CPU and IO costs. In a distributed system, a cost model should reflect the communication costs. In the following, we introduce these costs.

Query Cost: We also use the terms *query processing cost* and *query evaluation cost*. It refers to the amount of time necessary to compute the answer to a given query.

Maintenance Cost: The maintenance of views in response to changes at the sources (base relations) incurs what is known as *maintenance cost* or *maintenance time*. In other words, it is the time that can be allotted to keep up to date the materialized views.

Communication Cost: It is the time needed to transfer data e.g., transmitting views on the communication network. We also use the term *transfer cost* to refer to this cost.

The main factor for estimating the different costs is the size of the involved relations. This estimation is based on statistical information about the base relations and formulas to predict the cardinalities of the results of the relational operations. For instance in our example (see figure 2.1), to estimate the query cost corresponding to the view $J\_q_1\_1$, we require knowledge about the size of the view $S\_q_1\_1$

and the base relation *PartSupp*. We define the size of a given view v as follows.

$$size(v) = card(v) * length(v) \tag{2.1}$$

where $length(v)$ is the length (in number of bytes) of a tuple of v, computed from the lengths of its attributes. The estimation of $card(v)$ which is the number of tuples in v requires the use of the formulas given in the following section.

Evaluating the Cardinalities of the Views

Database statistics are useful in evaluating the cardinalities of the views. Two simplifying assumptions are commonly made about the database: (i) the distribution of attribute values in a relation is supposed to be uniform, and (ii) all attributes are independent, meaning that the value of an attribute does not affect the value of any other attribute. In what follows we give the formulas for estimating the cardinalities of the results of the basic relational algebra operations: selection, projection and join.

Selection. The cardinality of selection result is

$$card(\sigma_{relation}) = SF_S(F) * card(relation) \tag{2.2}$$

where $SF_S(F)$ is the selectivity factor which is dependent on the selection predicate and can be computed as follows [64].

$$\begin{aligned}
&SF_S(A = value) = \tfrac{1}{card(dv(A))} \\
&SF_S(A > value) = \tfrac{max(A) - value}{max(A) - min(A)} \\
&SF_S(A < value) = \tfrac{value - min(A)}{max(A) - min(A)} \\
&SF_S(p(A_i) \wedge (A_j)) = SF_S(p(A_i)) * SF_S(p(A_j)) \\
&SF_S(p(A_i) \vee p(A_j)) = SF_S(p(A_i)) + SF_S(p(A_j)) - (SFS(p(Ai)) * SFS(p(Aj))) \\
&SF_S(A \in \{values\}) = SF_S(A = value) * card(\{values\})
\end{aligned} \tag{2.3}$$

where A in an attribute of the $relation$, $dv(A)$ is the number of distinct values of the attribute A, $max(A)$ and $min(A)$ denote respectively the minimum and maximum possible values for A. $p(A_i)$ and $p(A_j)$ indicate the predicates over attributes A_i and

A_j, respectively.

Projection. The cardinality of projection result is simply the number of tuples when the projection is performed. We consider projection without duplicate elimination.

$$card(\pi_{relation}) = card(relation) \tag{2.4}$$

Join. To estimate the result cardinality of a join, we maintain the join selectivity factor SF_J as part of statistical information.

$$\begin{aligned} &card(relation_1 \bowtie relation_2)) = SF_J * card(relation_1) * card(relation_2) \\ &where \quad SF_J = \frac{card(relation_1 \bowtie relation_2)}{card(relation_1) * card(relation_2)} \end{aligned} \tag{2.5}$$

Cost Functions

The main objective in view selection problem is the minimization of the total query cost, defined by the formula:

$$QueryCost = \sum_{q_i \in Q} f_{q_i} * Qc(q_i, M) \tag{2.6}$$

Each query q_i has an associated non-negative weight f_{q_i} which represents the query frequency. $Qc(q_i, M)$ is the processing cost corresponding to q_i in the presence of a set of materialized views M.

The query cost is computed as the sum of all estimated costs incurred by the required relational operations. Recall that in this dissertation, 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:

- Formulae 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 estimate is the time necessary for fetching each tuple of the rela-

tion.

- All operation 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 number of 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*).

Because materialized views have to be kept up to date, the view maintenance cost has to be considered. This cost is weighted by the update frequency indicating the frequency of updating materialized views. The view maintenance cost is computed as follows:

$$ViewMaintenanceCost = \sum_{v_i \in M} f_u(V_i) * Mc(v_i, M) \tag{2.7}$$

where $f_u(v_i)$ is the update frequency of the view v_i and $Mc(v_i, M)$ is the maintenance cost of v_i given a set of materialized views M.

The view maintenance cost is computed similarly to the query cost, but the cost of executing the relational operation is computed with respect to updates. Different maintenance policies (deferred or immediate) and maintenance strategies (incremental or rematerialization) can be applied [26, 27, 55, 82]. In our work, 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. In what follows, we briefly review techniques for computing the differential of a join operation.

Computing the Differential of a Join Operation. Consider the view v which correspond to the result of $relation_1 \bowtie relation_2$. We assume for each relation $relation_i$ that there are two relations $\delta^+_{relation_i}$ and $\delta^-_{relation_i}$, denoting respectively the set of tuples inserted into and deleted from $relation_i$. Let $relation_1^{old}$ and $relation_2^{old}$ refer respectively to the contents of $relation_1$ and $relation_2$, before the update. The set of tuples that get added to the view v are denoted by δ_v^+ and can be computed as follows [51].

$$\delta_v^+ = (\delta^+_{relation_1} \bowtie relation_2^{old}) \cup (relation_1^{old} \bowtie \delta^+_{relation_2}) \cup (\delta^+_{relation_1} \bowtie \delta^+_{relation_2}) \quad (2.8)$$

View v is then updated as follows.

$$v = v \cup \delta_v^+ \quad (2.9)$$

Similarly, the set of tuples that get deleted from the view v are denoted by δv^- and can be computed as:

$$\delta_v^- = (\delta^-_{relation_1} \bowtie relation_2^{old}) \cup (relation_1^{old} \bowtie \delta^-_{relation_2}) \cup (\delta^-_{relation_1} \bowtie \delta^-_{relation_2}) \quad (2.10)$$

View v is then updated as follows.

$$v = v - \delta_v^- \quad (2.11)$$

Updates can be modeled as deletes followed by inserts. If both inserts and deletes are present in a relation, the view v is updated as follows.

$$\begin{aligned} v = v &\cup (\delta^+_{relation_1} \bowtie relation_2^{old}) \cup (relation_1^{old} \bowtie \delta^+_{relation_2}) \cup (\delta^+_{relation_1} \bowtie \delta relation_2{}^+) \\ &- (\delta^-_{relation_1} \bowtie relation_2^{old}) \cup (relation_1^{old} \bowtie \delta^-_{relation_2}) \cup (\delta^-_{relation_1} \bowtie \delta^-_{relation_2}) \end{aligned} \quad (2.12)$$

In the case when only one relation is updated i.e., $relation_1$, the view v with

respect to the changes is computed as.

$$
\begin{aligned}
v &= v \cup (\delta^{+}_{relation_1} \bowtie relation_2^{old}) \qquad //insert \\
v &= v - (\delta^{-}_{relation_1} \bowtie relation_2^{old}) \qquad //delete \\
v &= v \cup (\delta^{+}_{relation_1} \bowtie relation_2^{old}) - (\delta^{-}_{relation_1} \bowtie relation_2^{old}) \qquad //update
\end{aligned}
\tag{2.13}
$$

For more details about how to compute the differential of other relational operations, we refer the reader to [26].

The cost model is extended for distributed setting by taking into account the communication cost which is the cost for transferring data from their origin to the site (i.e., computer node) that initiated the query. Given a query q_i which is issued at the site s_j and denoting by v_k, a view required to answer q_i, the communication cost is zero if v_k is materialized at s_j. Otherwise, let s_l be the node containing v_k, then the communication cost for transferring v_k from s_l to s_j is:

$$
CommunicationCost_{(v_k, s_l \rightarrow s_j)} = \frac{size(v_k)}{Bw(s_j, s_l)} \tag{2.14}
$$

where $Bw(s_j, s_l)$ is the bandwidth between s_j and s_l (i.e., network transmission cost per unit of data transferred) and $size(v_k)$ is the size of the view v_k in number of bytes.

2.2 Constraint Programming and CHOCO solver

Constraint programming is currently applied with success to many domains [12, 13, 58], such as scheduling, planning, vehicle routing, configuration, networks and bioinformatics. More recently, constraint programming has been considered as beneficial in data mining setting [56]. Our motivation to use constraint programming in solving the view selection problem is that it is known to be a powerful approach for modeling and solving combinatorial optimization problems. Notice that the view selection problem is considered as a combinatorial optimization problem since the search space for the optimal solution (i.e., the optimal set of materialized views) entails a great number of comparisons between all possible combinations (subsets) of the set of candidate views. Constraint programming is also known to be efficient for the resolution of NP-complete problems since it can provide the optimal solution. The idea of constraint programming is to solve problems by stating constraints

which must be satisfied by the solution. Indeed, constraint programming allows to solve combinatorial problems modeled by a Constraint Satisfaction Problem (CSP) [72].

Constraint Satisfaction Problem (CSP). Formally, a CSP model 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. $DOM = \{dom_{var_1}, dom_{var_2}, ..., dom_{var_n}\}$ is the set of possible values that can be assigned to each variable var_i.
- Constraints. $CST = \{cst_1, cst_2, ..., cst_n\}$ is the set of constraints that describes the relationship between subsets of variables. Formally, a constraint Cst_{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., $Cst_{ijk} \subset dom_{var_i} \times dom_{var_j} \times dom_{var_k}$. 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.

In what follows, we provide a simple example showing how a constraint problem can be modeled as a CSP.

Example1. Let us solve a problem where unknows are the values of the variables x, y and z, knowing that each variable can take its value between 1 and 3, the value of x has to be greater than y and the value of y has to be greater than z.

To model the problem of example 1 as a CSP, one need is to define the CSP variables, their domains and the constraints defined over them:

Variables. x, y and z
Domains. $dom_x = dom_y = dom_z = \{1, 2, 3\}$
Constraints. $x > y$ and $y > z$

In our work, we use CHOCO [1, 2] for modeling and solving Constraint Satisfaction Problems (CSPs). CHOCO is a java library for CSPs and constraint programming which is built on a event-based propagation mechanism with backtrackable structures. It is an open-source software, distributed under a BSD license and hosted by sourceforge.net. Note that the constraint solvers such as CHOCO are structured around annual competitions [43]. In what follows, we present the

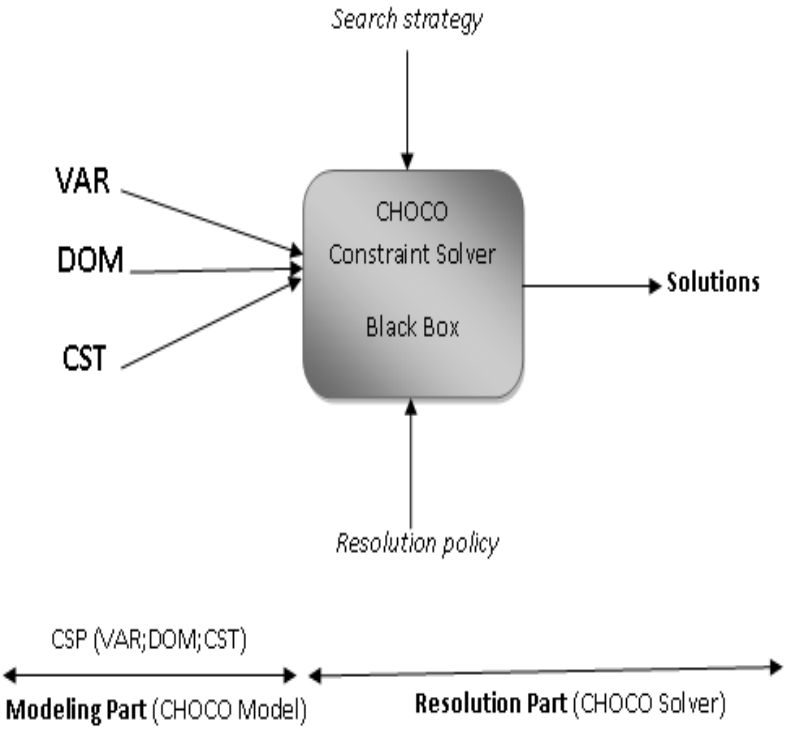


Figure 2.2: Modeling and solving with CHOCO.

basics of modeling and solving with CHOCO and the basic principles of constraint programming: propagation and search.

2.2.1 CHOCO Design

CHOCO is a java library that provides a clear separation between the formulation (CHOCO model) and the resolution (CHOCO solver) of the problems. The main interest of this separation is to propose to the user to model a problem without being interested in the way the problem is solved. Indeed, the user of CHOCO focus only on specifying the problem itself and the solver is then responsible for solving it (see figure 2.2). The different parts are clearly identified. The first part which is the modeling part is devoted to expressing the problem. It consists in modeling the problem as a CSP. For this purpose, the variables of the problem, their domains and the constraints to be satisfied, have to be defined. While, the second part which is the solving part, is devoted to solve the modeled problem. The CHOCO solver is mainly focus on the resolution part: reading the model, defining the resolution policy and the serach strategy. Once the model and the solver have been defined, the resolution of the modeled problem starts and produces as output one solution, all solutions or an optimal solution.

CHOCO Model

This section gives information on how to create a constraint programming model and introduce variables and constraints by using the API provided by CHOCO. It allows describing a problem in an easy and declarative way. It simply records the variables and the constraints defining the problem.

A constraint programming model within CHOCO is created as follows.

$$Model\ model = new\ CPModel(); \tag{2.15}$$

Variables A variable is defined by a name, type (integer, real, or set variable), the values of its domain and possibly with a given domain type i.e., bounded or enumerated. Bounded variables are related to large domains which are only represented by their lower and upper bounds. On the contrary, the domain of an enumerated variable is explicitly represented and every value is considered. The variables can be added to the model as follows.

$$\begin{array}{l} model.addVariable(var_1, "cp : bound"\ or\ "cp : enum"); \\ model.addVariables(var_2, var_3); \end{array} \tag{2.16}$$

Specific role of variables can be defined with options: non-decision variables or objective variable. The non-decision variables are also called implied variables because it is expected that, they will be instantiated by propagation as soon as all the decision variables will be all instantiated. Consider for example, a problem with two integer variables var_1 and var_2 linked by some implication $var_1 = 1 \Rightarrow var_1 = 2$, then the variable var_1 can be set as the decision variable, while the variable var_2 can be let implied. By default, each variable added to a model is a decision variable. To exclude the variable var_2 from the search strategy, we use the option "cp:no decision".

$$model.addVariable(var_2, "cp : nodecision"); \tag{2.17}$$

For optimization problems, one need is to define an objective variable within the model. An optimal solution is then a solution that minimizes or maximizes the objective variable.

$$model.addVariable(var_2, "cp : objective"); \tag{2.18}$$

Constraints. A constraint deals with one or more variables of the model and specifies conditions to be held on these variables. CHOCO allows the user to easily state its own new constraints by using the following method.

$$\begin{aligned} &model.addConstraint(cst_1);\\ &model.addConstraints(cst_2, cst_3); \end{aligned} \tag{2.19}$$

CHOCO provides a large number of simple and global constraints. Simple constraints may be binary or ternary constraints that involve respectively two and three variables. For instance, $eq(var_1, var_2)$ is a binary constraint which states that the two arguments are equal: $var_1 = var_2$. While the global constraints accept any number of variables and offer dedicated filtering algorithms which are able to make deductions. For instance, constraint $alldifferent(var_1; var_2; var_3; var_4)$ with $dom_{var_1} = dom_{var_2} = [1, 4]$ and $dom_{var_3} = dom_{var_4} = [3, 4]$ allows to deduce that var_1 and var_2 cannot be instantiated to 3 or 4; such rule cannot be inferred by simple binary constraints.

The list of simple and global constraints available in CHOCO can be found within the Javadoc API. Details and examples can be found in CHOCO documentation [3].

CHOCO Solver

The solver, along with the model is one of the two key elements of any CHOCO program. As mentioned before, the CHOCO solver is mainly focus on resolution part: reading the Model, defining the resolution policy and the search strategies. The creation of a solver can be easily done with the following methods available from the solver API.

$$Solver\ solver = new\ CPSolver(); \tag{2.20}$$

Reading the model. The reading of a model is compulsory and must be done after the entire definition of the model. The reading step is divided in two parts: variables reading and constraints reading. The solver gives the following API to read any CHOCO model.

$$solver.read(model); \tag{2.21}$$

The resolution of the model is performed automatically by the solver.

Resolution policy. The solver is able to find an optimal solution for any problem. However, computing the optimal solution for large problems with huge solution space may be very expensive. To this aims, the solver provides ways to limit the search regarding different criteria. Once a limit is reached, the search stops. These limits have to be specified before the resolution. Implementing a search limit such as time

Solver API	description
solve() or solve(false)	The solver runs until reaching a first feasible solution in which all constraints are satisfied (returns Boolean.TRUE) or the proof of infeasibility (returns Boolean.FALSE) or a search limit has been reached before (returns null).
solveALL() or solve(true)	The solver runs until computing all feasible solutions, or until proving infeasibility (returns Boolean.FALSE) or until reaching a search limit (returns Boolean.TRUE if at least one first solution was computed, and null otherwise).
maximize(Var obj, boolean restart)	The solver runs until reaching a feasible solution that is proved to maximize objective obj or until proving infeasibility (returns Boolean.FALSE) or until reaching a search limit (returns Boolean.TRUE if at least one first solution was computed and null otherwise). It proceeds by successive improvements of the best solution found so far: each time a feasible solution is found at a leaf of the tree search, then the search follows for a new solution with a greater objective, until it proves that no such improving solution exists. Parameter restart is a boolean indicating whether the search continues from the solution leaf (if set to false) or if it is relaunch from the root node (if set to true).
minimize(Var obj, boolean restart)	similar to maximize but for computing a feasible solution that is proved to minimize objective obj.

Table 2.1: The different API to solve a problem.

limit need only to specify the following methods.

- *Time Limit.* Performing the search until reaching a search time limit. A time limit is set using the solver API: *setTimeLimit*(*int timeLimit*).

For optimization problems, the resolution policy is to leave the constraint solver running until reaching a feasible solution that is proved to minimize or maximize the objective variable. Some of the API offered by the solver to launch the problem resolution, are presented above in table 2.1.

Search Strategy. A key ingredient of any constraint satisfaction approach is a clever search strategy. The search space is organized as an enumeration tree, where each node corresponds to a subspace of the search and each child node is a subdivision of the space of its father node. The tree is progressively constructed by applying a series of branching strategies that determine how to subdivide space at each node and in which order to explore the created child nodes.

In the CHOCO solver, branching has been applied to decision variables. The most common branching strategies in CHOCO 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 (value selector).

Defining a search strategy is very important since a well-suited search strategy can reduce the number of expanded nodes, the number of backtracks and hence the time that the solver incurs to compute solutions. The branching strategies i.e., variable and value selection strategies available in CHOCO can be found in [3].

Constraint Propagation and Search

Once the model and solver has been defined, the resolution can start. It is based on constraint propagation techniques. Indeed, algorithms for solving CSPs usually employ a search procedure that is based on constraint propagation [39]. Such algorithms are guaranteed to find a solution, if one exists, or to prove that the problem is unsatisfiable. In figure 2.3, we present the organigram of the propagation loop. When the search fixes the value of a variable (modification of a domain of a variable), an event is posted, storing information about the action done (event type, variable, values, etc.). We call this the *variableevent*. Then, constraint filtering algorithms have been called, *constraintevent*, in order to reach a fix point or to detect contradictions. This means that the variable event will be given to the related constraints

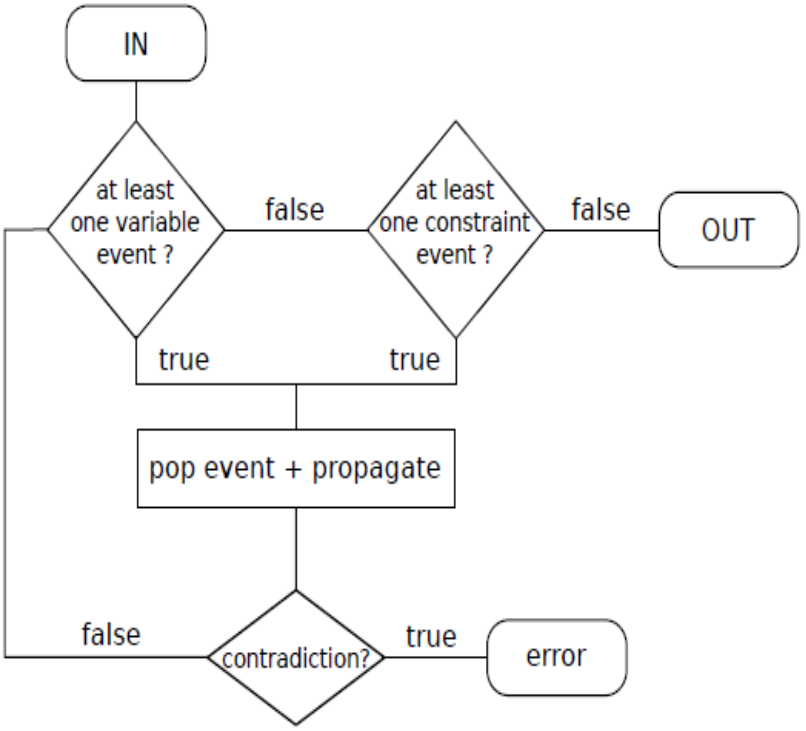


Figure 2.3: Organigram of the propagation loop [3].

of the modified variable, to check consistency and propagate this new information to the other variables. If the propagation of an event leads to a contradiction, the propagation engine stops the process.

Explicitly, the constraint programming system always starts by propagating the immediate effects of the constraint set which results in the reduction of the variable domains, through the withdrawal of inconsistent values. With reference to the problem of example 1, we show through figure 2.4 how constraint propagation and search can be applied to solve this problem.

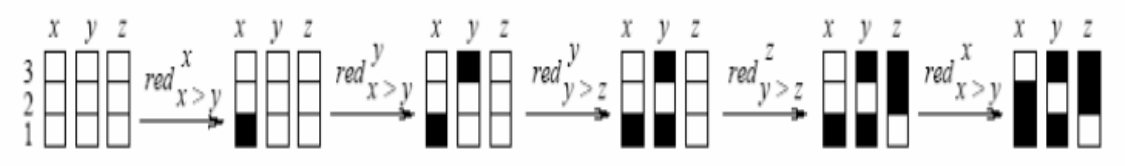


Figure 2.4: The variable domain reductions of three variables x, y and z [22].

Figure 2.4 shows the domain reduction of three variables x, y and z and two constraints $x > y$ and $y > z$. At the beginning, the initial variable domains, $dom_x = dom_y = dom_z = \{1, 2, 3\}$, are represented by three columns of white squares. Considering the constraint $x > y$, it appears that x cannot take the value 1 because

otherwise there would be no value for y such that $x > y$; $red^{x}_{x>y}$ filters this value from dom_x. The black square denotes the deleted value. Similarly, $red^{y}_{x>y}$ eliminates the inconsistent value 3 from the domain of y. Then, considering the constraint $y > z$, $red^{y}_{y>z}$ and $red^{z}_{y>z}$ withdraw respectively the sets {1} and {2,3} from dom_y and dom_z. Finally, $red^{x}_{x>y}$ reduces dom_x to the singleton {3}. The final solution is $\{x = 3, y = 2, z = 1\}$. If, after this stage, some variable domains are not reduced to singletons, the solver takes one of these variables and tries to assign it each of the possible values in turn. This enumeration stage triggers more reductions, which possibly leads to solutions.

In chapter 4 and chapter 5, we provide simple examples to illustrate how the constraint propagation and search can be applied to the view selection in a centralized context as well as in a distributed scenario.

2.3 Conclusion

In this chapter we have introduced the basic and necessary notions and concepts which are required to be known to address and to understand the view selection problem. We have first provided definitions related to the view selection context and then presented the cost model formulation that is an important issue for the view selection process. We have also introduced the basic principle of constraint programming that we have proposed to address the research problem targeted in this thesis. In next chapter we present the state of the art in view selection field in a centralized context as well as in a distributed setting.

Chapter 3

State-of-the-Art

In this chapter we discuss the state of the art in view selection research. Previous works on materialized view selection were developed in the context of query optimization, warehouse design, data placement in a distributed setting, etc. Many diverse solutions to the view selection problem have been proposed and analyzed through surveys [6, 30, 42]. The survey [30] concentrates on methods of finding a rewriting of a query using a set of materialized views. The study presented in [42] focuses on the state of the art in materialization for web databases. A critical analysis of methodologies for selecting materialized views in data warehousing is provided in [6]. However, none of the above mentioned surveys provides a classification of view selection approaches in order to identify their advantages and disadvantages. Our target in this chapter is to fill this gap. This chapter aims at studying the view selection in relational databases and data warehouses as well as in a distributed setting. It defines the view selection process that determines the main dimensions which are the basis in the classification of view selection methods. Based on this classification, this study reviews existing view selection methods by identifying respective potentials and limits. It also provides an overview of dynamic view selection methods. The content of this chapter is mainly based on our material published in [47].

3.1 The View Selection Process

The view selection process determines the set of views to be materialized. In a distributed environment, in addition to providing the set of materialized view, the set of sites (computer nodes) on which these views should be materialized has to be computed during the process. There are some other parameters which can extend the definition of the view selection process. Figure 3.1 outline these parameters which can be classified into three major groups, namely: (i) input parameters; (ii)

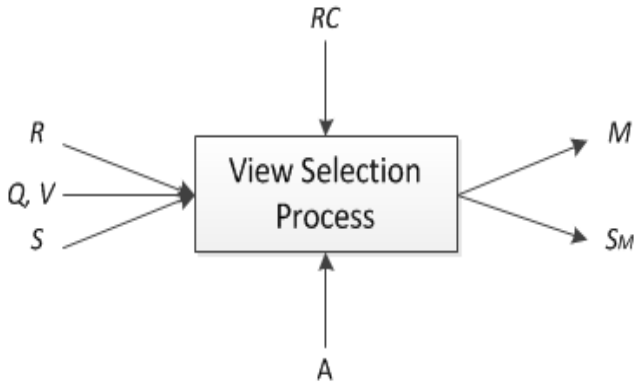


Figure 3.1: The view selection process.

process parameters; and (iii) output parameters.

Input Parameters (R, Q, V, S). These parameters concern the kind of input on which view selection methods operate. Approaches to the view selection problem take as input a database (or a data warehouse) schema R, a query workload Q defined over R and a set of sites S (computer nodes) of the network if the problem is studied in a distributed scenario. In order to solve the view selection problem, one need is to identify the candidate views V which are promising for materialization. Starting with the input queries Q, techniques based on multiquery DAG, syntactical analysis of the workload or query rewriting have been used to obtain the candidate views V (see next section for details).

Process Parameters (RC, A). The view selection process generates a set of materialized views by applying heuristic algorithms A given a limited amount of resource (resource constraints RC). Resources may be CPU, IO, storage space capacity and the view maintenance cost limit. In a distributed context, the resource constraints i.e., CPU, IO and the storage space will be per site (computer node). Also, resource constraints such as network bandwidth and the location of materialized views will have to be taken into consideration.

Output Parameters (M, S_M). The view selection process produces as output the set of views to be materialized M. It also computes the set of sites S_M on which the views M should be materialized, in the case where the view selection problem is addressed in a distributed environment. Once the views are selected and placed at the appropriate sites, the input queries will be answered using these views.

3.2 View Selection Dimensions

There are many dimensions that can be taken into account when attempting at classifying view selection methods in order to identify their advantages and disadvantages. As from figure 3.1, we may classify them according to (i) Frameworks used to obtain the candidate views, (ii) Resource constraints considered during the view selection process and (iii) Heuristic algorithms applied to address the view selection problem.

3.2.1 Frameworks

As mentioned in the previous section, techniques based on multiquery DAG, syntactical analysis of the workload or query rewriting have been used as a framework to obtain the candidate views which are promising for materialization. Based on the set of candidate views, the view selection methods compute the set of views to be materialized and the set of sites on which these views should be placed if the view selection problem is studied in a distributed environment.

Multiquery DAG

Most of the proposed view selection methods operate on query execution plans. The plans can be derived from multiple query optimization techniques or by merging multiple query plans. The main interest of such techniques relies in detecting common sub-expressions between the different queries of workload and capturing the dependencies among them. This feature can be exploited for sharing computations, updates and storage space. The dependence relation on queries (or views) has been represented by using a Directed Acyclic Graph (DAG). However, these methods require optimizer calls which can be expensive in complex scenarios.

The most commonly used DAGs in literature are:

- **AND/OR View Graph:** The union of all possible execution plans of each query forms an AND-OR view graph [59]. The AND-OR view graph described by Roy [61] is derived from the AND-OR DAG representation which is composed of two types of nodes: Operation nodes and Equivalence nodes. Each operation node represents an algebraic expression (Select-Project-Join) with possible aggregate function. An equivalence node represents a set of logical expressions that are equivalent (i.e., that yield the same result). The operation nodes have either one or two children that are equivalence nodes and one parent equivalence node. The equivalence nodes have edges to one or more

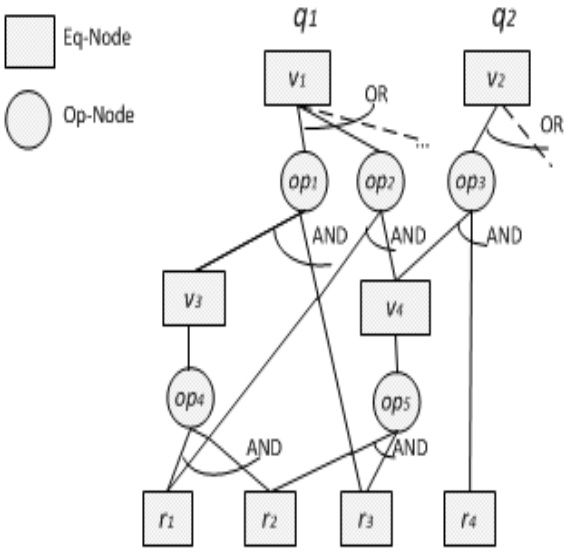


Figure 3.2: The AND-OR view graph of the two queries q_1 and q_2.

operation nodes. The root nodes are equivalence nodes corresponding to the query results. While, the leaf nodes are equivalence nodes corresponding to the base relations. An equivalence node can be calculated by computing one of its operation node children. While, an operation node can be calculated only by computing all of its equivalence node children.

A sample AND-OR view graph is shown in figure 3.2. Circles represent operation nodes (Op-Nodes) and boxes represent equivalence nodes (Eq-Nodes). For simplicity, we represent only two execution plans for the view v_1 which is the query result of q_1 and one execution plan for the view v_2 that is the query result of q_2 (where r_1, r_2 and r_3 represent the base relations).

q_1:$((r_1\ op_4\ r_2)\ op_1\ r_3) \cup (r_1\ op_2(r_2\ op_5\ r_3))$ // **two execution plans**

q_2:$((r_2\ op_5\ r_3)\ op_3\ r_4)$ // **one execution plan**

The remaining execution plans are just indicated in figure 3.2 by dashed lines. The dependence among the views is indicated by AND and OR arcs. The AND arcs mean that all of the child views are needed to compute the parent view. While the OR arcs specify that the parent view can be computed from

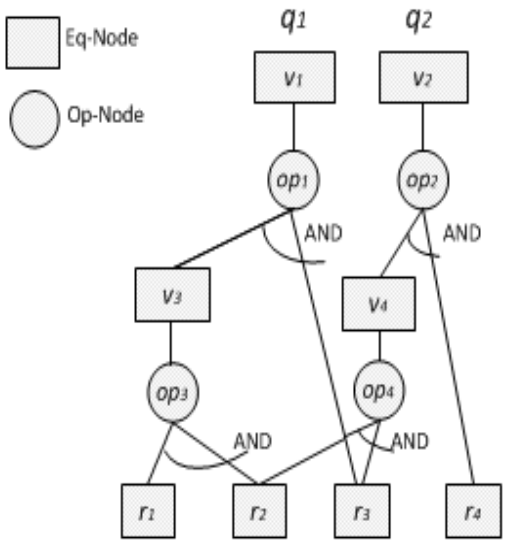


Figure 3.3: The AND view graph of the two queries q_1 and q_2.

any one of its children. For example, in figure 3.2, view v_1 corresponding to a single query q_1, can be computed from v_3 and r_3 or r_1 and v_4.

In the AND view graph (see figure 3.3), there is only AND arcs and hence there is only one way to answer or update a view (or a query). As can be seen in figure 3.3, the views v_1 and v_2 corresponding respectively to the result of the query q_1 and q_2 can be computed or updated on only one way:

q_1:$((r_1\ op_3\ r_2)\ op_1\ r_3)$

q_2:$((r_2\ \ op_4\ r_3)\ op_2\ r_4)$

If there is only one way to answer or update a given view (or a query), the graph becomes an AND view graph.

In the data cube which is a specific model of a data warehouse, the AND-OR view graph is an OR view graph, as for each view there are zero or more ways to construct it from other views, but each way involves only one other view [28]. In other words, an OR view graph is an AND-OR view graph in which every node is an equivalence node that can be computed from any one of its children. A sample OR view graph is shown in figure 3.4. For example, in this figure, view v_1 can be computed from any of the views v_2, v_3 or v_4. View v_2 can again be computed from any of the base relations r_1 or r_2. The same applies to computing the views v_3 and v_4.

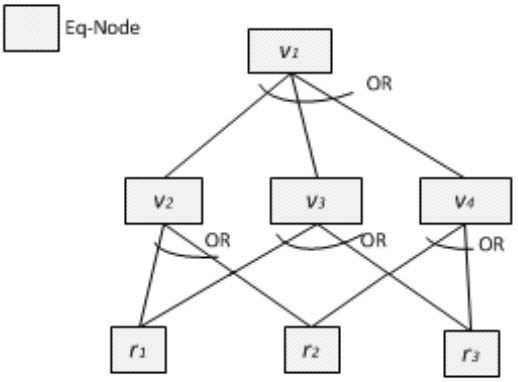


Figure 3.4: The OR view graph for four views.

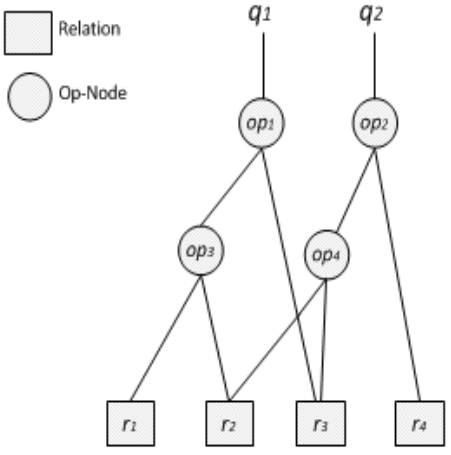


Figure 3.5: The MVPP of the two queries q_1 and q_2.

- **Multi-View Processing Plan (MVPP):** The MVPP defined by Yang et al [76] is a DAG in which the root nodes are the queries, the leaf nodes are the base relations and all other intermediate nodes are selection, projection, join or aggregation views that contribute to the construction of a given query. The MVPP is obtained after merging into a single plan either individual optimal query plans (similar to the AND view graph) or all possible plans for each query (similar to the AND-OR view graph). The difference between the MVPP representation and the AND-OR view graph or the AND view graph representation is that all intermediate nodes in the MVPP represent operation nodes. A sample MVPP is shown in figure 3.5.

- **Data Cube Lattice:** Harinarayan and al [31] propose the data cube lattice for modeling data in multiple dimensions. It is built from the queries involved in the data warehouse application, e.g., OLAP-style queries. The data cube lattice is a DAG whose nodes represent views (or queries) which are characterized by the attributes of the Group by clause. The edges denote the derivability relation between views. That is, if there is a path from view v_1 to a view v_2 (see figure 3.6), then grouping attributes on v_2 can be calculated from grouping attributes on v_1. The node labeled none corresponds to an empty set of group-by attributes (tuples are not grouped).

  The data cube lattice is used for representing queries with only aggregate functions involved for OLAP applications. It can be seen as an OR view graph where each view in the graph can be derived from a subset of other views in one or more ways, but each derivation involves only one other view. The benefit of this representation is that a query can be used to answer or update another query.

  An extension of the data cube lattice in order to adapt it to a distributed case was proposed in [7, 77]. Indeed, the cube has been modified by adding edges that mark the derivation relationship between views on different computer nodes. Therefore, in addition to the edges representing aggregation dependencies, further edges are introduced to denote the communication channels within the distributed scenario as illustrated in figure 3.7. In the example, we consider a scenario with two sites i.e., two data warehouses. To keep the example as clear as possible, only a tiny part of the full lattice is given in its full complexity (as shown in the dashed rectangle).

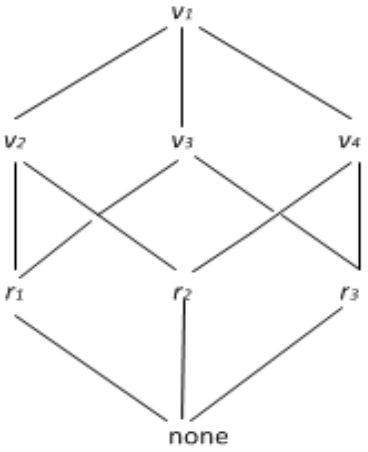


Figure 3.6: The four views constructible by grouping on some of r_1, r_2, and r_3.

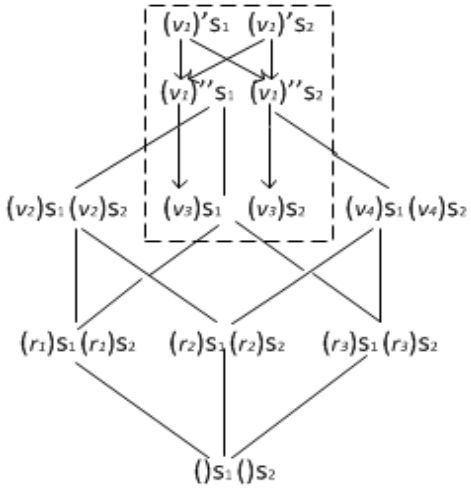


Figure 3.7: The distributed data cube lattice.

Query Rewriting

Here, the input to the view selection problem is not a multiquery DAG but the query definitions. The view selection problem is modeled as a state search problem. Each point in the search space, called a state, represents a set of views and a set of associated equivalent rewritings showing how to compute each query based on this set of views. Consider the queries $q_1 = \pi_k(\sigma_{k=3}(r_1) \bowtie_{b=c} r_2 \bowtie_{c=d} r_3)$ and $q_2 = \pi_a(r_2 \bowtie_{c=d} r_3 \bowtie_{d=e} \sigma_{a<6}(r_4))$. The state graph for q_1 and q_2 is defined as follows.

- The set of nodes is the set of of base relations $r_1, ..., r_i$. If a node r_i is labeled by an attribute label $v : a_1, .., a_k$, this means that the attributes $a_1, .., a_k$ are projected out in the query or view definition.
- For every selection operation involving attributes of the relation r_i, there is an edge from r_i to itself (a loop) labeled as v : *selection predicate*. This loop is called *selection edge*.
- For every join operation involving attributes of the relations r_i and r_j, there is an edge between r_i and r_j, labeled as v : *join predicate*. Such an edge is called *join edge*.

We define the initial state of the search for the two queries q_1 and q_2 as: $State_0 = (\{v_1, v_2\}, G_0, R_0)$ where v_1 and v_2 are views identical to the queries q_1 and q_2 and R_0 the rewriting set that consists of the trivial rewritings $\{q_1 = v_1, q_2 = v_2\}$. The related state graph G_0 is depicted in figure 3.8. Then a set of transformation rules have been applied in order to detect and exploit common sub-expressions between the queries of the workload and guarantee that all the queries can be answered using exclusively the selected views. In what follows, we introduce the two elementary transformation rules which are *SelectionCut* and *JoinCut*.

- *Selection Cut.* Let (V, G, R) be a state and v : *selection predicate* be a selection edge. A *selectioncut* on v : *selection predicate* yields a new state (V', G', R') such that V' is the new set of views by replacing v with a new view v', G' is the new state graph obtained by erasing the edge v : *selection predicate* and R' is obtained from R by replacing all occurrences of v with the expression $\sigma_{selection\ predicate}(v')$.
- *Join Cut.* Let (V, G, R) be a state and v : *join predicate* be a join edge. A *joincut* on v : *join predicate* yields a new state (V', G', R') such that V' is the new set of views by replacing v with two new symbols v'_1 and v'_2, G' is the new

state graph obtained by erasing the edge v : *join predicate* and R' is obtained from R by replacing v by $v'_1 \bowtie_{join\ predicate} v'_2$.

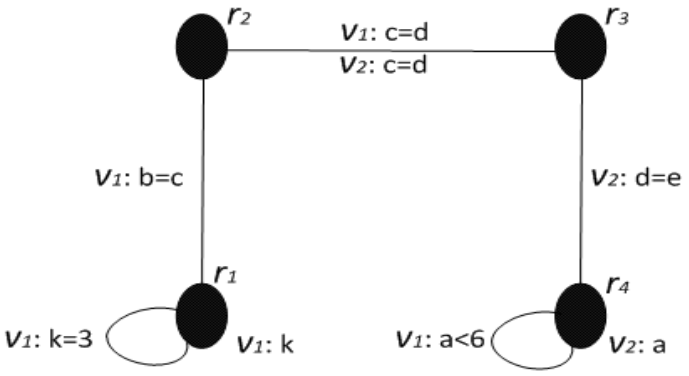


Figure 3.8: Sample query graph.

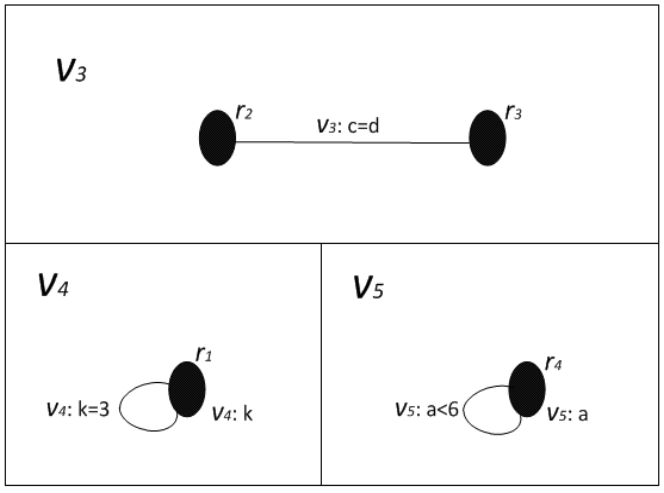


Figure 3.9: The query graph after two *joincut*

.

The state graph that is shown in figure 3.9 is obtained from G_0 (figure **??**) by applying two *joincut* which consist in removing two *join edge* $v_1 : b = c$ and $v_2 : d = e$. The resulting state is: $State_1 = (\{v_3, v_4, v_5\}, \{q_1 = \pi_k(v_3 \bowtie_{b=c} v_4), q_2 = \pi_a(v_3 \bowtie_{d=e} v_5)\})$

This set of transformation rules allow to rewrite completely all the input queries over the selected views and detect common sub-expressions between the queries.

For instance in the above state $State_1$, we can note that the view v_3 is a common sub-expression between the queries q_1 and q_2. Nevertheless, the completeness of the transformation rules makes the complexity of state search problem exponential, based on the number of states created by each transformation and time complexity of the transformation.

Syntactical Analysis of the Workload

Some view selection methods are based on syntactical analysis of the workload to identify candidate views. These approaches analyze the workload and pick a subset of interesting base relations from all possible base relation subsets for the workload. This subset is chosen according to the condition that if materializing one or more views on it has the potential to reduce the cost of the workload significantly. The remains base relation subsets are removed from the final result set according to two cost metrics [8]:

- $BRS - Weight(brs)$ which computes the detailed costs of processing all the queries in the query workload that use a given base relation subset brs.

$$BRS - Weight(brs) = \sum_{q \in Q_{brs}} f_q * Qc(q_i) * \frac{\sum_{rs \in brs} size(rs)}{\sum_{rq \in brq} size(rq)} \tag{3.1}$$

  Where Q_{brs} is the set of queries that refer to the base relations brs, brq is the set of base relations required to answer the query q, $Qc(q_i)$ is the query cost corresponding to q without using the materialized views, f_q is the frequency of q in the workload, $size(rs)$ is the size of the base relations in brs and $size(rq)$ is the size of all the base relations referenced in brq.

  $BRS-Weight$ is defined as the sum of query costs weighted by the sizes of the base relations. However, this definition is not suitable for distributed scenarios where the size of the base relations is not the only influencing factor. For this purpose, the study in [15] computes the cost $BRS - Weight$ as follows:

$$BRS - Weight(brs) = \sum_{q \in Q_{brs}} f_q * Qc(q_i) * \frac{\sum_{rs \in brs} cost_{baseRelation}(q, rs)}{\sum_{rq \in brq} cost_{baseRelation}(q, rq)} \tag{3.2}$$

  Where $cost_{baseRelation}(q, rs)$ and $cost_{baseRelation}(q, rq)$ return respectively the costs of a base relation rs and rq in the query q, i.e., the costs of base relation scans, selection and sending data, which depend on the resources of the sites storing the base relations rs and rq. This definition captures the size

(cardinality) of a base relation, its query frequency and the performance of its allocation site.

- $BRS{-}Cost(brs)$ that estimates the costs only, with $BRS{-}Cost(rs) \geq BRS{-}Weight(rs)$.

$$BRS - Cost(brs) = \sum_{q \in Q_{brs}} f_q * Qc(q_i) \tag{3.3}$$

As mentioned above, $Qc(q_i)$ is the query cost corresponding to q without using the materialized views and f_q is the frequency of q in the workload

Workload analyzing allows finding an interesting base relation subsets from among all possible base relations for the workload, and restrict the space of candidate views considered to only those base relation subsets. A base relation subset is interesting if materializing one or more views on it has the potential to reduce the cost of the workload significantly. However, the search space for computing the candidate views to be materialized may be very large since the number of possible combinations of base relations may be exponential based on the number of different base relations referenced by all queries in the query workload. The syntactical analysis of the workload can be substituted for example by using multi-query optimization techniques as described above which can significantly save a lot of work and cost as well. By using such techniques, the ideal search space can be found just by constructing the DAG representation of the entire workload which can recognize possibilities of shared computation among several queries of the workload.

3.2.2 Resource Constraints

Resource constraints considered during the view selection can be taken into account when classifying view selection methods. There are three main models presented in literature, namely: unbounded, space constrained and maintenance cost constrained.

Unbounded

In the unbounded setting, there is no limit on available resources (storage, computation etc.). Thus, the view selection problem consists in choosing a set of views to materialize that minimizes the query processing cost and the view maintenance cost. Formally thus, the problem is:

$$argmin \left(\sum_{q_i \in Q} f_{q_i} * Qc(q_i, M) + \sum_{v_i \in M} f_u(v_i) * Mc(v_i, M) \right) \tag{3.4}$$

Recall (see chapter 2) that f_{q_i} is the query frequency, $Qc(q_i, M)$ is the processing cost corresponding to q_i in the presence of a set of materialized views M, $f_u(v_i)$ is the update frequency of the view v_i and $Mc(v_i, M)$ is the maintenance cost of v_i given a set of materialized views M.

However, this approach may lead to two kinds of problems. First, sometimes the selected views may be too large to fit in the available space. Second, the cost of the view maintenance may offset the performance advantages provided by the view materialization.

Space Constrained

Due to the storage space limitation, materializing all views is not always possible. In this setting, a useful notion is that of a view benefit (or query benefit). This is defined as the reduction in the workload evaluation cost, which can be achieved by materializing this view. Also relevant in this context is the *per-unit benefit*, obtained by dividing the view benefit by its space occupancy. It has been shown [28] that the per-space unit benefit of a view can only decrease as more views are selected (monotonic property). The space constrained model minimizes the query processing cost plus the view maintenance cost under a space constraint.

$$\begin{aligned} &argmin \left(\sum_{q_i \in Q} f_{q_i} * Qc(q_i, M) + \sum_{v_i \in M} f_u(v_i) * Mc(v_i, M) \right) \\ &under \ \sum_{v_i \in M} size(v_i) \leq S_{max} \end{aligned} \tag{3.5}$$

where S_{max} is the storage space capacity.

Nevertheless, the view maintenance cost is unbounded in this model. Indeed, in many real applications, maintenance-cost is more likely to be the real constraint to keep the materialized views consistent with the data source (base relations), rather than storage space constraints. Besides, the storage space can be considered as cheap and therefore not regarded as a critical resource anymore.

Maintenance Cost Constrained

This model constrains the time that can be allotted to keep up to date the materialized views in response to updates on base relations. In the maintenance cost constrained model, the maintenance cost of a view may decrease with selection of other views for materialization. Therefore, the query benefit per unit of maintenance cost of a view can increase [29]. This non monotonic nature of maintenance cost

makes the view selection problem more difficult. The maintenance cost constrained model minimizes the query processing cost under a maintenance cost constraint.

$$\begin{aligned} &argmin \left(\sum_{q_i \in Q} f_{Q_i} * Qc(q_i, M)\right) \\ &under \ \sum_{v_i \in M} f_u(v_i) * Mc(v_i, M) \leq U_{max} \end{aligned} \tag{3.6}$$

where U_{max} is the view maintenance cost limit.

The models that we have presented: unbounded, space constrained and maintenance cost constrained, can be extended to the distributed setting by taking into account the distributed specific features i.e., the communication cost between the sites (computer nodes) and the location of the materialized views.

3.2.3 Heuristic Algorithms

In this section, we present the different kind of the most well-known heuristic algorithms proposed in literature to solve the view selection problem i.e., deterministic algorithms, randomized algorithms or hybrid algorithms.

Deterministic Algorithms

Algorithms in this class usually construct a solution in a deterministic manner by exhaustive search or by applying some kind of heuristics such as greedy algorithm to avoid having to traverse the solution space in an exhaustive search manner. 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. It is very difficult to find an optimal solution to the problems which belong to the class of NP-complete problems because of the fact that the solution space grows exponentially as the problem size increases. For instance in the context of the view selection problem, the number of possible views (view combinations) to materialize grows exponentially with the number of queries in the workload, the numbers of columns, join predicates, grouping clauses and tables referenced in each query and with the number of computer nodes if the problem is studied in a distributed scenario.

Randomized Algorithms

The most commonly used randomized algorithms in the context of view selection are simulated annealing algorithms [40] and genetic algorithms [24].

- *Simulated Annealing Algorithms.* Such algorithms are motivated by an analogy to annealing in solids. They are based on the iterative improvement technique which is applied to a single point that represent a solution in the search space and continuously tries to search its neighbors to find a better point (a better solution). In order to eliminate the dependency on the starting point of the search, simulated annealing algorithms use a probability for acceptance to decide whether or not to move to a neighboring point. Indeed, it is possible to move to a neighboring point (a neighboring solution) by random walk that may be further away from the optimum than the previous one in expectation that its neighbors will represent a better solution. The probability for acceptance is calculated according to a cooling schedule. The algorithms terminate as soon as no applicable moves exist or lose all the energy in the system.

- *Genetic Algorithms.* These algorithms generate solutions using techniques inspired by the natural evolution process such as selection, mutation, and crossover. The strategy search for these algorithms is very similar to biological evolution. Genetic algorithms use a randomized search strategy; they start with a random initial population containing individuals which represent possible solutions 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. In contrast with the simulated annealing algorithms, genetic algorithms use a multi-directional search by maintaining a pool of candidate points (candidate solutions) in the search space. Information is exchanged among the candidate points to direct the search where good candidates survive while bad candidates die. This multi-directional evolutionary approach allows the genetic algorithm to efficiently search the space and find a point near the global optimum.

Randomized algorithms are based on statistical concepts where the search space can be explored randomly until reaching a point near the global optimum. The can be applied for very large search spaces. Furthermore, they 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.

Hybrid Algorithms

Hybrid algorithms combine the strategies of pure deterministic algorithms and pure 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.

The combination of the power of randomized algorithms and deterministic algorithms may provide better solution quality than either randomized algorithms or deterministic algorithms used alone. However, hybrid algorithms are more time consuming since a considerable amount of time must be spent during the search. Therefore, such algorithms may be impractical due to their excessive computation time.

3.3 Review of View Selection Methods

In this section, we classify the view selection methods. More specifically, they have been classified based on what kind of algorithms they use to address the view selection problem pointing which resource constraints they consider during the view selection process and frameworks they use to obtain the candidate views (see figure 3.10). Based on this classification, we review most of the view selection methods that have been proposed in the literature.

3.3.1 Deterministic Algorithms Based Methods

Much research work on view selection use deterministic strategies to address the view selection problem. [60] seems to be the first paper that provides a solution for materializing view indexes which can be seen as a special case of the materialized views. Indeed, view indexes are similar to views except that instead of storing the tuples in the views directly, each tuple in the view index consists of pointers to the tuples in the base relations that derive the view tuple. The solution is based on A* algorithm [53] to compute the optimal set of view indexes.

An exhaustive approach is also presented in [57] for finding the best set of views to materialize in the context of SQL views. The authors have also examined the cost of maintaining a materialized view by materializing additional views. In addition to study how to select the set of views to be materialized, the work in [41] address the index selection problem. By running experiments, the authors were able to indicate that building indexes on key attributes in the primary view lead to solid maintenance

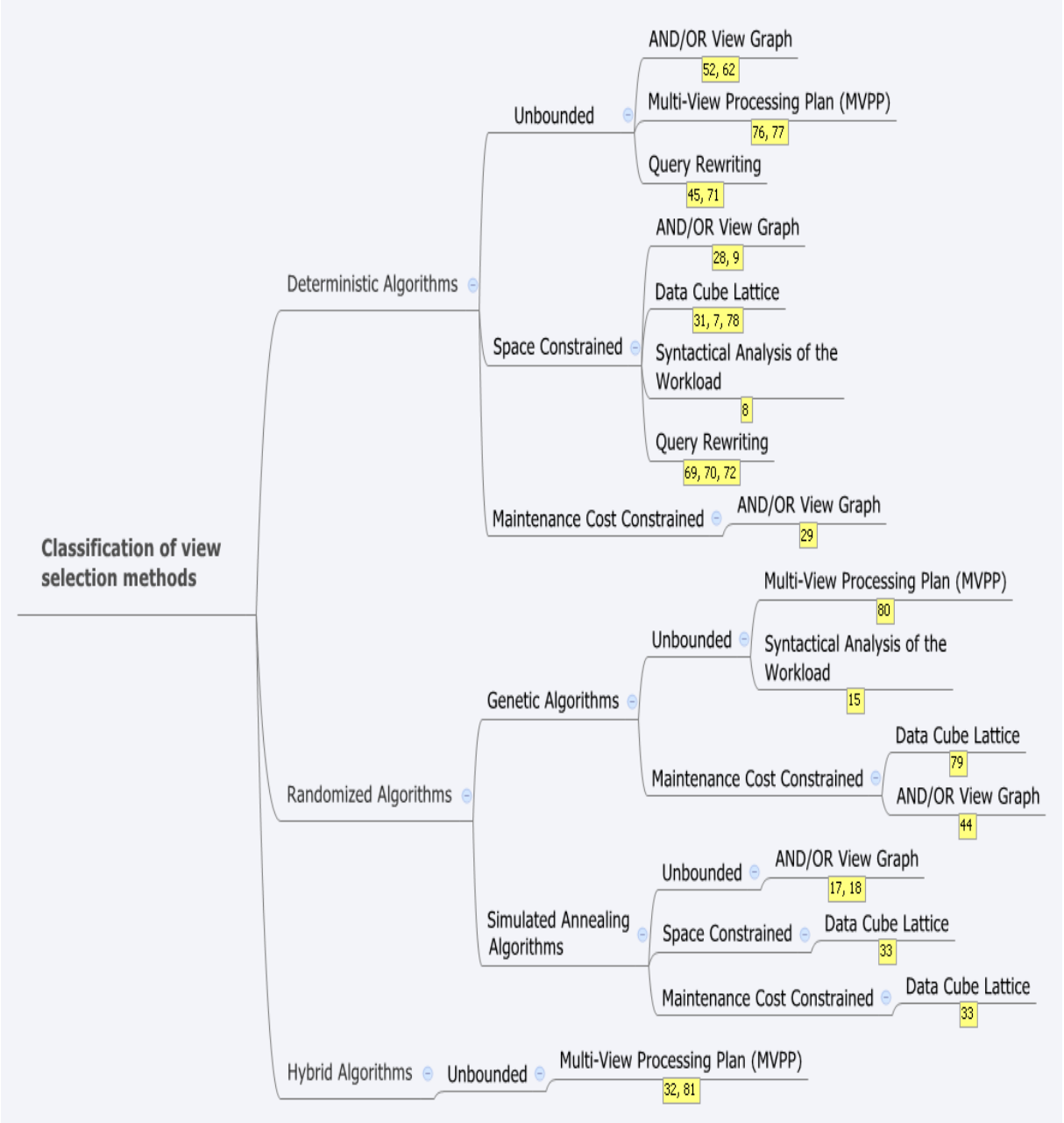


Figure 3.10: A Classification of view selection methods.

cost savings with modest storage space requirements. Nevertheless, an exhaustive search is impractical for many real world problems since it cannot compute the optimal solution in a reasonable time. In order to avoid having to exhaustively traverse the solution space in search of the optimal solution, many view selection methods use a form of greedy strategy. In what follows, we provide a review of such methods.

The authors in [31] have investigated the problem of choosing which set of views to materialize in the special case of data cubes. They present and analyze several view selection algorithms when there are queries with only aggregate functions involved for decision support applications i.e., OLAP-style queries. The view selection has been modeled using the data cube lattice framework. Using this framework, they provide polynomial-time greedy algorithms to select the right set of views to materialize that minimizes the query cost subject to a space constraint. One of the negative points of this approach is that the view maintenance cost has not been taken into account which can efficiently change the materialized view selection.

The work in [75] 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 views to be materialized so that the sum cost of processing the queries of workload and maintaining the materialized views is minimized. Besides, they presented a framework which is the Multi-View Processing Plan (MVPP) that can provide a feasible solution based on individual query plans. They also map the materialized view selection problem as O-l integer programming problem, whose solution can guarantee an optimal solution. The weak point of this approach is that they have used a very simple cost model for updating the view which considers the cost used for constructing this view. They assume that re-computing (from scratch) is used whenever an update of an involved base relation occurs. As a result, the maintenance cost for the selected view set is not very realistic. 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 [28]. Their work aims to find a set of views to materialize under a storage space constraint, which have the best balance between view maintenance cost and query cost. They provide a near-optimal exponential time greedy algorithm for the most general case of AND-OR view graph, where for each view (or query), they consider all its possible execution strategies. The authors design also a near-optimal polynomial time greedy algorithms for some special cases of the general data warehouse scenario: (i) AND view graph where each query or view has a unique evaluation and (ii) OR view graph, in which any view can be computed from

any one of its related views i.e., data cube. This approach was extended in [29] to study the view selection under a maintenance cost constraint instead of the storage space constraint. They refer to this problem as the maintenance-cost view-selection problem. This is more difficult than the view selection problem with a storage space constraint because of the non-monotonic behavior of the benefit function per unit of maintenance cost (as explained in the previous section). To solve this problem, the authors use the notion of inverted tree set to develop a greedy heuristic algorithm, which delivers a near-optimal solution for the special case of OR view graphs. For the general case of AND-OR view graphs, they design an A* heuristic that provides an optimal solution but it takes a considerable amount of time as it is exponential in the size of the input graph.

The authors in [61] demonstrate that using multi-query optimization techniques is practical and provides significant benefit in view selection setting. The benefits of multi-query optimization were also demonstrated on a real database system. The main interest of such techniques relies in detecting common sub expressions between the different queries of workload. This feature can be exploited for sharing updates and space storage. To find a solution to the view selection problem, a greedy heuristic has been designed which is based on the AND-OR DAG representation of queries and picks the set of views to materialize so that the cost of processing the queries is minimal. This approach also handles index selection and nested queries. This study was extended in [51] to consider how to optimize view maintenance cost (minimize the cost of maintenance). In addition to speed up the query workload by selecting materialized views, they have presented greedy algorithms which exploit common sub-expressions between view maintenance expressions to compute an efficient plan to the maintenance of the materialized views. In particular, it has been shown how to efficiently choose sub-expressions and indexes to be materialized temporarily or permanently (and maintained along with other materialized views) to faster view maintenance. However, the view selection problem has been studied without any resource constraint.

A pragmatic approach of the view selection problem that combines local with global optimization have been presented in [9]. Polynomial greedy algorithms have been designed to provide a solution based on the balance between query processing and maintenance cost. More precisely, the view selection problem has been solved in two phases. The first phase depends on local optimization by searching the views to materialize per level and per query which can preserve the data independence whenever adding a query to the view configuration or removing one from it. The first phase is based on the notion of level in the query tree. Indeed, each view of

the query tree is associated to a level. As result, for each query a set of views is pre-selected which is associated to the level that has the minimal sum of query processing and view maintenance cost (local benefit). Based on the pre-selected views, the second phase generates the set of views to be materialized which optimize the trade-off between the total query cost and the view maintenance cost (global benefit) subject to a space constraint.

The view selection has been studied in [45, 68, 69, 70, 71] under the condition that the input queries can be answered using exclusively the materialized views. This is done by formulating the view selection problem as a state space optimization problem and using a set of transformation rules to rewrite completely the input queries over the view selection (please see section 3.2.1). An exhaustive algorithm has been designed in [70] to select a set of materialized views while minimizing the combination of the query processing and view maintenance cost. In this study it was considered that there is no storage space restriction in the data warehouse. This work was extended in [45] by developing greedy algorithms that expand only a small fraction of the states produced by the exhaustive algorithm. The issue of selecting a set of views to be materialized has been investigated in [68, 69, 71] under a storage space constraint. However, their view selection algorithms are still in exponential time. A survey of work on answering queries using views can be found in [30].

The study in [8] presents another approach that is based on a syntactical analysis of the workload. This approach deals with the problem of selecting both view and indexes to be materialized in order to optimize the physical design of SQL databases by taking into account the interaction between indexes and materialized views. More specifically, this approach proceeds in three main steps. The first step analysis the workload and chooses subsets of base relations with a high impact on the costs of processing all the queries (please see section 3.2.1). Based on the base relations subsets, the second step identify syntactically relevant views and indexes that can potentially be materialized. The goal of candidate materialized view selection is to eliminate materialized views that are syntactically relevant for one or more queries in the workload but are never used in answering any query. Based on the result of the second step, the system runs a greedy enumeration algorithm to pick a set of views and indexes to materialize in order to determine the ideal physical design. The selection of the materialized views has been done under the condition that these views have to fit in the available storage space. The drawback of this approach is that it does not take into account the view maintenance cost. This feature is important to ensure the correctness of the index and view selection.

The works published in [7, 77] address the view selection problem in a distributed

data warehouse environment. An extension of the concept of a data cube lattice to capture the distributed semantics has been proposed (please see section 3.2.1). Moreover, they extend a greedy based selection algorithm for the distributed case. However, the cost model that they have used does not include the view maintenance cost. Furthermore, the network transmission costs (bandwidth network) are not considered which is very important in a distributed context. Indeed, the communication cost is computed as a function of the size of the query result.

The above methods take a deterministic approach either by exhaustive search or by some heuristics such as greedy. However, greedy algorithms may be unsatisfactory in term of the solution quality because the greedy nature of the algorithm may make it converge to poor local minima since initial solutions influence the solution greatly. As a result, many paradigms and have been developed to improve the solutions of the view selection problem, namely: randomized algorithms and hybrid algorithms which we describe in next subsection.

3.3.2 Randomized Algorithms Based Methods

Typical randomized algorithms are genetic [24] or use simulated annealing [40].

Genetic Algorithms Based Methods

A genetic algorithm has been proposed in [79] in conjunction with the Multi-View Processing Plan (MVPP) framework to deal with the selection of materialized views in a data warehouse. The materialized views have been selected according to their reduction in the combined query cost and view maintenance cost. They have shown that a genetic algorithm is particularly a suitable and feasible approach toward solving materialized view selection problem. 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.

The study in [44] focused on an efficient solution using genetic algorithm to the maintenance cost view selection problem. Indeed, a penalty function has been included in the fitness function to reduce the fitness each time the maintenance cost constraint is not satisfied. The problem of selecting a set of views to be materialized has been explored in the context of OR view graph where each view can be computed from any one of its related views. This approach minimizes the query

processing cost given varying upper bounds on the view maintenance cost, assuming unlimited amount of storage space because storage space is cheap and not regarded as a critical resource anymore. 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. They believe that a genetic algorithm can become an important tool for warehouse evolution, especially for those data warehouses that contain a large number of views and must accommodate frequent changes to the queries.

The common methods for dealing with constrained combinatorial optimization problems is to introduce a penalty function to the objective function in order to penalize the solutions violating the resource constraints. However, it is difficult to find a precise value to realize the right balance between the original objective function and the penalty function. A solution was provided in [78] to ensure this balance and keep improving the solution. Constraints are incorporated into the algorithm through a stochastic ranking procedure where no penalty functions are used.

The study presented in [15] which is based on a syntactical analysis of the workload deals with the distributed view selection. This approach consists of three main steps. The first one extends the base relations selection algorithm described in [8] for the distributed scenario. Based on the result of the first step and the similarity between queries, the second step generates the candidate views which are promising for materialization. In the third step a genetic algorithm is applied to select a set of materialized views and the nodes of the network on which they will be materialized that minimize the query processing and view maintenance cost. However, this approach does not take into account either the space constraint or the maintenance cost constraint.

Simulated Annealing Based Methods

A randomized approach for selecting a set of views that are able to answer the input queries has been developed in [67]. It is based on the simulated annealing process. In this approach, the views are selected to be materialized such that the combination of the query cost and the view maintenance cost is minimized.

The approach proposed in [33] have also studied the application of randomized search heuristics to address the view selection problem. Simulated annealing algorithms were adapted to find an appropriate set of views that minimizes querying cost and meet the resource constraints of the data warehouse. Specifically, the view

selection problem has been studied 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 randomized search in the context of dynamic view selection.

The use of simulated annealing algorithms has also been investigated in [17] in conjunction with the use of the Multi-View Processing Plan (MVPP) framework to decide which views to materialize for large data warehouse systems. In order to deal with larger sets of views and gain further improvements in solution quality, Parallel Simulated Annealing (PSA) has been explored in [18] for materialized view selection. By performing simulated annealing with multiple inputs over multiple computer nodes concurrently, PSA is able to improve the quality of obtained sets of materialized views. Moreover, PSA is able to perform view selection on MVPP having a much larger number of views, which reflects the real data warehousing environment. However, the view selection problem is solved without any bound neither on the storage space nor on the view maintenance cost.

Randomized algorithms can be applied to complex problems dealing with large or even unlimited search spaces. Thus, the use of randomized algorithms can be considered in solving large combinatorial problems. Indeed, they can be easily adapted to solve the view selection problem in a centralized context as well as in a distributed setting. They have also provided significant improvements over existing methods i.e., deterministic methods for both the quality of the solutions and the time allocated for view selection. However, their successes to provide *good* quality solutions often depend on the set-up of the algorithms as well as the extremely difficult fine-tuning of the parameters of the algorithms that must be performed during many test runs. Furthermore, there is no guarantee of performance. Indeed, randomized algorithms may tend to get stuck at a poor local optimum fairly early because of their probabilistic behavior.

3.3.3 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 [80] 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.

An evolutionary search is also described in [32] which use a Genetic Local Search (GLS) algorithm to solve the view selection problem. GLS is a hybrid heuristic that combines the advantages of population based algorithm i.e., genetic algorithm and local optimization. Local search iteratively moves from one solution to a better one on its neighborhood until a local minimum is reached. While it quickly finds good solutions in small regions of the search space, the genetic operators such as selection, crossover and mutation are suitable for exploring the whole search space in order to identify interesting regions.

Hybrid methods have been developed in order to achieve furthers improvement in the solution quality i.e., the quality of the obtained set of materialized views, in terms of cost saving. However, the drawback of such methods is that they are more time consuming and may be impractical due to their excessive computation time.

3.4 Static View Selection vs. Dynamic View Selection

A static view selection approach is based on a given workload and chooses accordingly the set of views to materialize. Whereas, in a dynamic view selection approach, the view selection is applied as a query arrives. Therefore, the workload is built incrementally and changes over time. Because the view selection has to be in synchronization with the workload, any change to the workload should be reflected to the view selection as well. Indeed, in a system of a dynamic nature [11], the set of materialized views can be changed over time and replaced with more beneficial views in case of changing the query workload. Dynamic view indexing has also been considered in [63] which can be seen as a special case of the materialized views.

The principle of the dynamic system described in [38] is monitoring constantly the incoming queries and considers the materialization at the final result of a query. This approach deals with multidimensional data warehouse and the replacement of the view depends either on space constraint or maintenance cost constraint. Unlike other dynamic approaches, in the one described in [81] dynamicity is applied to the view data. This approach aims at materializing the most frequently accessed tuples

of the view rather than materializing all tuples of the view in order to reduce view maintenance cost and storage space requirements. The set of materialized tuples can be changed dynamically as the queries change, either manually or automatically by an internal cache manager using a feedback loop. However, the task of monitoring constantly the query pattern and periodically recalibrating the materialized views is rather complicated and time consuming especially in large data warehouse where many users with different profiles submit their queries.

The work presented in [10] has designed an approach for dynamically selecting an effective set of views to be materialized and place them in key points in the P2P system so as to achieve the best combination of *good* query performance and low view maintenance cost. Moreover, as the system is dynamic, their approach continuously monitors the incoming query and adjusts the system configuration by removing materialized views in order to replace the less beneficial views with more beneficial ones.

A dynamic view selection is often referred to as view caching. With caching, the cache is initially empty and data are inserted or deleted from the cache during the query processing. Materialization could be performed even if no queries have been processed and materialized views have to be updated in response of changes on the base relations. A detailed comparison of these two techniques is given in [36].

Traditional caching approaches aim at caching the results of queries. Another alternative is to cache only a part of a view. Indeed, a chunk based scheme has been introduced in [19] for fine granularity caching. Chunk based caching allows caching of only few, frequently used tuples of views. To facilitate the computation of chunks required by a query but not found in the cache, a new organization for base relations has been proposed which they called a chunked file. Caching has been adopted in data warehousing [62], distributed databases [37] and peer to peer systems [34].

The design of an intelligent data warehouse cache manager has been proposed in [62] called watchman which aims at minimizing the query cost. The cache manager employs cache replacement and cache admission algorithms. These algorithms explicitly consider retrieved set sizes and processing costs of the associated queries in order to improve the query performance.

The approach in [37], called cache investment has been proposed for integrating query optimization and data placement. The main goal of this study is to place copies of data closest to where they will most likely be accessed. Caching is performed here in terms of partitions of base relations. This approach is ideal for a client-server caching architecture, in which queries are submitted, data is cached, and results are displayed at client workstations while the primary copies of data

reside on the server (server machines). The authors demonstrated that there is circular dependency between caching and query plan optimization which has significant performance implications for advanced distributed database systems. The role of a cache investment policy is to determine which data items should be cached at the client. However, the weak point of this approach is that they assume the client-server architecture without any cooperation between the clients in order to invest their cache together. Moreover, base relation caching would be space wasted and high communication cost would be paid in order to transfer the entire base relation from its origin to the client.

The caching system presented in [34] addresses the problem of PeerOLAP architecture where a large number of peers access sporadically a number of separate data warehouses for processing on-line analytical queries. The peers act as large distributed caches and offer their resources aiming at achieving lower query processing cost. When any query arrives at a peer, it is decomposed into chunks. If a query cannot be answered locally by using the cache contents e.g., chunks of the computer node where it is issued, it is propagated through the network until a peer that has cached the answer is found. Due to the space constraint, PeerOLAP provides a replacement algorithm to control the local cache at each peer. Last access time is the replacement criteria used for the replacement of the less beneficial chunks with more beneficial chunks. The authors specified different degree of collaboration between the peers by introducing view placement policies. The cooperation between the peers is achieved within a cluster. Intuitively, peers with similar query patterns should be neighbors (belong to the same cluster). Each peer implements a mechanism which constantly evaluates the current neighbors and drops or adds peers to the neighbor list, in order to achieve lower query cost. However, the network transmission cost has not been taken take into account when designing the different clusters and this may be considered as a drawback when the communication cost between two peer neighbors is too high.

3.5 Summary and Observations

In this section, we survey our review of the different approaches related to the view selection and our observations, which will be useful to introduce our approach. One line of past research explores the view selection in relational databases and data warehouses when all the queries are assumed to be known and given in advance. The view selection problem has also been studied in a distributed setting consisting of many computer nodes, where each node issues many different queries and updates

at different rates. A separate line of research has studied the dynamic view selection, where views are selected continuously, to respond to the changes in the query workload over time. In this work, we focus only on static view selection.

The view selection methods we discussed have been classified based on the main view selection dimensions that we have identified and which we summarize in figure 3.11. Specifically, we classify them based on what kind of heuristic algorithms they design to deal with the view selection issue pointing which frameworks they use to obtain the candidate views and resource constraints they consider during the view selection process.

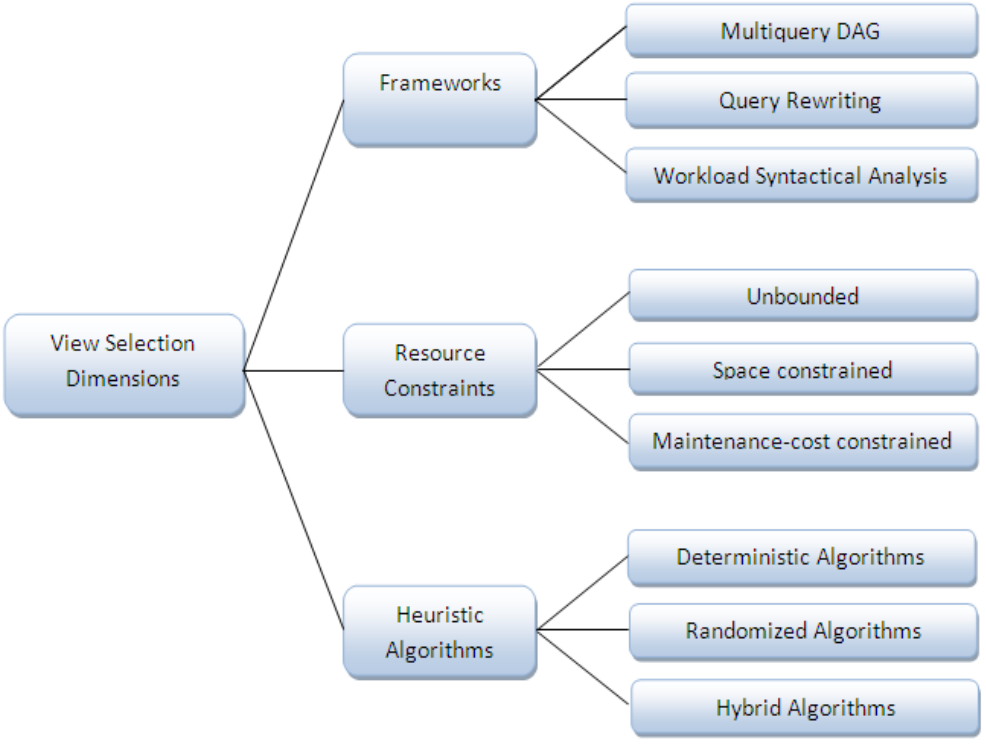


Figure 3.11: View Selection Dimensions.

The principal dimensions that are the basis in the classification of view selection methods can be divided into three main categories. The first category of frameworks is based on identifying the candidate views. More precisely, techniques based on muliquery DAG, query rewriting or syntactical analysis of the workload have been used to obtain the candidate views. In our work, workload analyzing and query rewriting techniques has been substituted by using multi-query optimization techniques which can significantly save a lot of work and cost as well. By using multi-query optimization, we can find the ideal search space just by constructing the DAG representation of the query workload which can recognize possibilities of shared computation, updates and storage space. Particularly, we have used in our

approach the AND-OR DAG representation (the AND-OR view graph). Our motivation to use this representation is explained in the next chapter.

The second main category focuses on the resource constraints which are incorporated into the view selection algorithms. In the unbounded model, the selection of materialized views has been done without considering any resource constraint. The space constrained model specifies the availability of the storage space in a database or a data warehouse, whereas the maintenance-cost constrained model specifies how long the materialized views must be updated (because changes to the source data result in recomputing these views). In our work, we studied the problem under resource constraints since they have been considered as a crucial condition to select the right set of materialized views. In our approach, the view selection has been explored and tested under various cases where: (i) only the storage space constraint is considered (i) the limiting factor is the view maintenance cost and (ii) both view maintenance cost and storage space constraints exist.

The third main category describes the kind of heuristic algorithms characterizing the view selection methods. As mentioned above, the best-known heuristic algorithms proposed in literature to solve the view selection problem, are: deterministic algorithms, randomized algorithms and hybrid algorithms. Analysis of state of the art of view selection methods has shown that deterministic methods such as greedy methods encounter significant problems with respect to performance (solution quality) when the problem size grows above a certain limit. In order to deal with larger search space and achieve further improvement in solution quality, randomized methods have been designed. Such methods can find a reasonable solution within a relatively short period of time by trading executing time for quality. Among the randomized algorithms, the most well-known algorithms are simulated annealing and genetic algorithms. The main difference between the simulated annealing algorithm and the genetic algorithm is that the latter uses a multi-directional search which allows efficiently searching the space and finding a point near the global optimum. Hybrid algorithms which combine the strategies of pure deterministic and pure randomized algorithms have also been designed to further improve the solution quality. However, we have observed that they often require longer computation time and may be impractical due to their excessive computation time.

Consequently, we can deduce that genetic algorithms provides a good balance between the computing cost that an algorithm takes for finding a solution to the view selection problem and the gain to be realized in solution quality. However, there is no guarantee of performance because the probabilistic behavior of genetic algorithms does not insure to find the optimal solution. To this aim, we have proposed a novel

approach that is based on constraint programming which is known to be an efficient method for solving NP-complete problems. We have also observed that randomized methods do not work well for the optimization problem with constraints. Because of their random characteristic, some solutions can be infeasible with respect to the resource constraints. In contrast with randomized methods, our approach may be seen as a constraint handling technique that can deal with resource constraints effectively. Besides, the success of the randomized methods often depends on the set-up of the algorithms as well as the extremely difficult fine-tuning of algorithms that must be performed during many test runs. In our constraint programming based approach, the user only has to specify the problem itself instead of specifying how to solve a problem.

Analysis of state of the art of view selection has also shown that there is very few work on view selection in distributed databases and data warehouses since the view selection problem becomes more challenging in such environments. As mentioned before, it includes another issue which is to decide on which computer nodes the selected views should be materialized. Furthermore, resource constraints such as CPU, IO, and network bandwidth have to be taken into consideration for each computer node. The view selection problem in a distributed context may also be constrained by storage space capacities per computer node and maximum view maintenance cost. In our approach, all these resource constraints will easily be modeled and handled.

3.6 Conclusion

In this chapter we have provided a broad overview of the current state of the art of view selection. We have introduced the main dimensions which are the basis in the classification of view selection methods. Based on this classification, we have reviewed existing view selection methods by identifying their respective potentials and limitations. We also provided an insight on dynamic view selection methods. Finally, we summarized our review of related work and our observations. We pointed out that none of the mentioned approaches meet all the requirements of view selection problem. Therefore, our goal in this thesis is to provide a novel approach for view selection problem that satisfies all these requirements. Our approach that we have designed in this study use constraint programming techniques to address the view selection problem in a centralized context (see chapter 4) as well as in a distributed environment (see chapter 5).

Chapter 4

A Declarative Approach to View Selection Modeling

View selection is important in many data-intensive systems e.g., commercial database and data warehousing systems. Given a database (or 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 selected views are referred to as materialized views and the problem of choosing which views to materialize is known as the view selection problem. This is one of the most challenging problems in many applications such as query processing and data warehousing and it is known to be a NP-complete problem.

In this chapter, we propose a declarative approach that involves a constraint programming technique which is known to be efficient for the resolution of NP-complete problems and a powerful method for modeling and solving combinatorial optimization 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 (CSP) 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 content of this chapter is mainly based on our material published in [50] and has the following contributions.

1. We make use of the concept of the AND-OR view graph to exhibit common sub-expressions between queries of workload which can be exploited for sharing computation, updates and storage space.

2. We provide the constraint satisfaction model that we have proposed to the view selection problem. Then, a constraint programming solver can be applied to set up the search space by identifying a set of views that minimizes the total query cost. We also provide an insight on how constraint programming can be applied to select materialized views.

3. We define heuristic search strategies within the constraint solver in order to reduce the solution space and hence the execution time that our approach incurs to find the set of materialized views. Then, we show the effectiveness of our heuristic based search strategy which improves in several magnitude the solution provided by the default one.

4. We have implemented our approach and compared it with the genetic algorithm which is known to provide the best trade-off between the execution time and the gain to be realized in solution quality. We demonstrate through many different experiments that our approach provides better performance resulting from evaluating the solution quality in terms of cost saving.

The rest of this chapter is organized as follows. Section 4.1 defines the problem that we address in the context of view selection in a centralized environment. In section 4.2, we present the framework that we have used for representing views to materialize in order to exhibit common sub-expressions between the different queries of workload. Section 4.3 describes how to model the view selection problem as a constraint satisfaction problem as well as the heuristic search strategies that we have designed for optimization purpose. Section 4.4 gives a performance analysis comparing our approach with the genetic algorithm which is known to optimize the balance between quality of solutions in terms of cost saving and execution time. The chapter ends with a summary in section 4.5.

4.1 Problem Definition

In this chapter, we are targeting one of the most challenging problems in data warehousing systems [74]: deciding which views to materialize in the warehouse to obtain the optimal query performance [31]. This problem has also been investigated in commercial database systems to facilitate efficient query processing [8]. We refer to this problem as the view selection problem.

As mentioned in chapter 2, a materialized view is a view whose content is computed and stored. In most cases it is cheaper to read the contents of a materialized

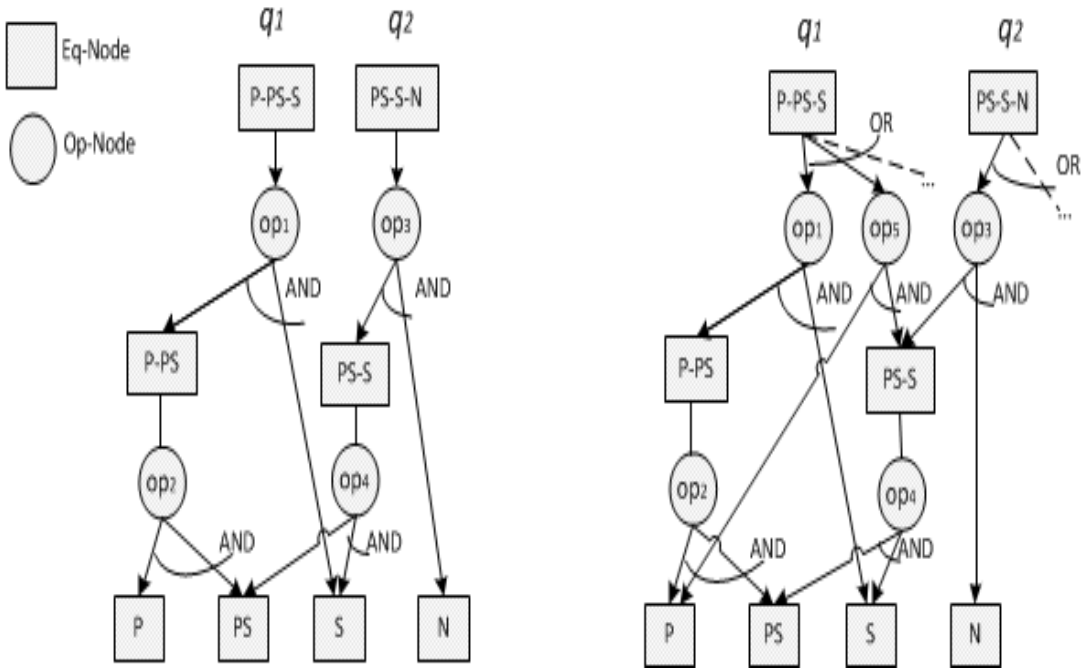
view than to compute its content by executing the query defining the view. Materialized views are used to speed up the query processing as they can be accessed quickly. However, whenever a base relation is changed the materialized views built on it have to be updated in order to compute up-to-date query results. The process of updating materialized views is known as view maintenance. Besides, materialized views require storage space. Materializing all the input queries can achieve the lowest query cost but the highest view maintenance cost which can cause overhead to the system. Besides, the query result can be too large to fit in the available storage space. 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.

More precisely, the view selection problem can be defined as follows: Given a query workload with an associated frequency for each query on a given database (or data warehouse) schema and a limited amount of resource, e.g., storage space and/or view maintenance cost, select a set of views to materialize so that the cost of evaluating the query workload is minimal. The search space for the optimal solution to the view selection problem grows exponentially as the problem size increases. Indeed, the number of possible view combinations to materialize grows exponentially with the number of queries and with the numbers of columns, join predicates, grouping clauses and the base relations referenced in each query of the workload.

In this chapter, we propose a novel 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 (CSP). As mentioned before, our motivation to use constraint programming in solving the view selection problem is that it is known to be a powerful approach for modeling and solving combinatorial problems [73]. Furthermore, constraint programming is an effective paradigm for the resolution of NP-complete problems [58].

4.2 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 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



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

Figure 4.1: DAG representation of two queries q_1 and q_2

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 4.1. 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 4.1a), 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 4.1b. 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 4.1b by dashed lines.

In this work, we use the AND-OR view graph to compactly represent alternative query plans and exhibit common sub-expression. Our motivation to use this representation rather than the AND view graph 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 ⋈ PS) ⋈ S) and the plan ((PS ⋈ S) ⋈ N), as it is shown in figure 4.1a. These execution plans represent the optimal plans for q_1 and q_2. However, if we choose the alternative plan (P ⋈ (PS ⋈ S)) to compute the view P-PS-S, the view PS-S becomes a common sub-expression (see figure 4.1b). 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 PS-S may also be used for sharing updates and hence reducing the view maintenance cost.

4.3 Our view selection approach

After a short introduction in chapter 2 to constraint programming and Constraint Satisfaction Problems (CSPs), 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. Finally, we provide an insight on how constraint programming can be applied to select materialized views.

4.3.1 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 table 4.1, we define all the symbols as well as the variables that we have used in our constraint satisfaction model.

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.

$$minimize \quad \sum_{v_i \in Q(G)} \Big(f_q(v_i) * Qc(v_i) \Big) \tag{4.1}$$

$$subject\ to \quad \sum_{v_i \in V(G)} \Big(Mat(v_i) * size(v_i) \Big) \leq Sp_{max} \tag{4.2}$$

$$\sum_{v_i \in V(G)} \Big(Mat(v_i) * f_u(v_i) * Mc(v_i) \Big) \leq U_{max} \tag{4.3}$$

Symbols of our constraint satisfaction model	
G	The AND-OR view graph described in the previous section.
$Q(G)$	The queries of the workload which corresponds to the root nodes in the AND-OR view graph G.
$V(G)$	The set of views in G which are candidates to materialization.
U	The set of updates in response to changes of the base relations.
$\delta(v_i, u)$	The differential result of view v_i with respect to update u.
f_q	The frequency or importance of the associated query.
f_u	The frequency of propagating the changes of each associated base relation to the materialized views.
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, $dom_{Mat(v_i)}$ ={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 $\mathbb{R}^*$ such as $dom_{Qc(vi)} \subset \mathbb{R}^*$.
$Mc(v_i)$	The maintenance cost corresponding to a view v_i, where $dom_{Qc(vi)} \subset \mathbb{R}^*$.

Table 4.1: Symbols and CSP variables.

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 (4.2) and (4.3) state that the views are selected to be materialized under a limited amount of resources. Constraint (4.2) ensures that the total space occupied by the materialized views is less than or equal to the maximum storage space capacity. Constraint (4.3) 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 formulae described in [61, 51] to compute these two costs. Note that the query and maintenance costs corresponding to a base relation are equal to zero.

The query cost and view maintenance cost may be formulated as follows.

Query cost

$$Qc(v_i) = \begin{cases} CCost(v_i) \;\; if \;\; Mat(v_i) = 0 \\ RCost(v_i) \; otherwise \end{cases} \tag{4.4}$$

$$\begin{aligned} &where \\ &CCost(v_i) = argmin_{op_j \in child(v_i)} \Big(cost(op_j) + \sum_{v_k \in child(op_j)} Qc(v_k) \Big) \end{aligned} \tag{4.5}$$

Constraint (4.4) 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 (4.5) ensures that the minimum cost path is selected for computing a given view. Each minimum cost path includes 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

$$Mc(v_i) = \begin{cases} 0 \;\; if \;\; Mat(v_i) = 0 \\ \sum_{u \in U(v_i)} Mcost(v_i, u) \; otherwise \end{cases} \tag{4.6}$$

$$\begin{aligned} &where \\ &Mcost(v_i, u) = argmin_{op_j \in child(v_i)} \Big(cost(op_j, u) \\ &\qquad + \sum_{v_k \in child(op_j)} UCost(v_k, u) \Big) \end{aligned} \tag{4.7}$$

$$UCost(v_k, u) = \begin{cases} Mcost(v_k, u) \;\; if \;\; Mat(v_k) = 0 \\ \delta(v_k, u) \; otherwise \end{cases} \tag{4.8}$$

Constraint (4.6) 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 (4.7) and (4.8) 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.

As mentioned in chapter 2, we have used the constraint solver CHOCO [2] in our work for modeling and solving the view selection problem as a CSP. Note that a complete API is provided to allow the user of CHOCO to state the problem in its constraint language in a natural and declarative way. In the Appendix A, we provide an insight on how to create the constraint satisfaction model by using the large Javadoc API provided by the CHOCO constraint solver.

4.3.2 Search strategy

A key ingredient of any constraint satisfaction approach is an efficient search strategy. As mentioned in chapter 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 a series of branching strategies that defines 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. As mentioned in chapter 2, the most common branching strategies in the constraint solver are based on the assignment of a selected variable to one or several selected values. 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.

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:

$$val = \min(dom_{Mat(v_i)})$$

In the next branch, the value val is removed from the variable domain $dom_{Mat(v_i)}$.

Our own search strategy

As mentioned in chapter 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 incurs 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 selection heuristic. 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. 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 for 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. To 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**
    $MV = MV \cup \{Mat(v_s)\}$
  end for
end for

Finally, the variable selector will choose the materialization variable 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: $max(dom_{Mat(v_i)})$. 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.

4.3.3 Solving the view selection problem in a centralized context with constraint programming

As mentioned in chapter 2, most algorithms for solving constraint satisfaction problems usually employ a search procedure and constraint propagation: when the search fixes the value of a variable, 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 through an example how the constraint programming can be applied to select materialized views: Assume that we have 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, $dom_{Mat(v_i)}$ ={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

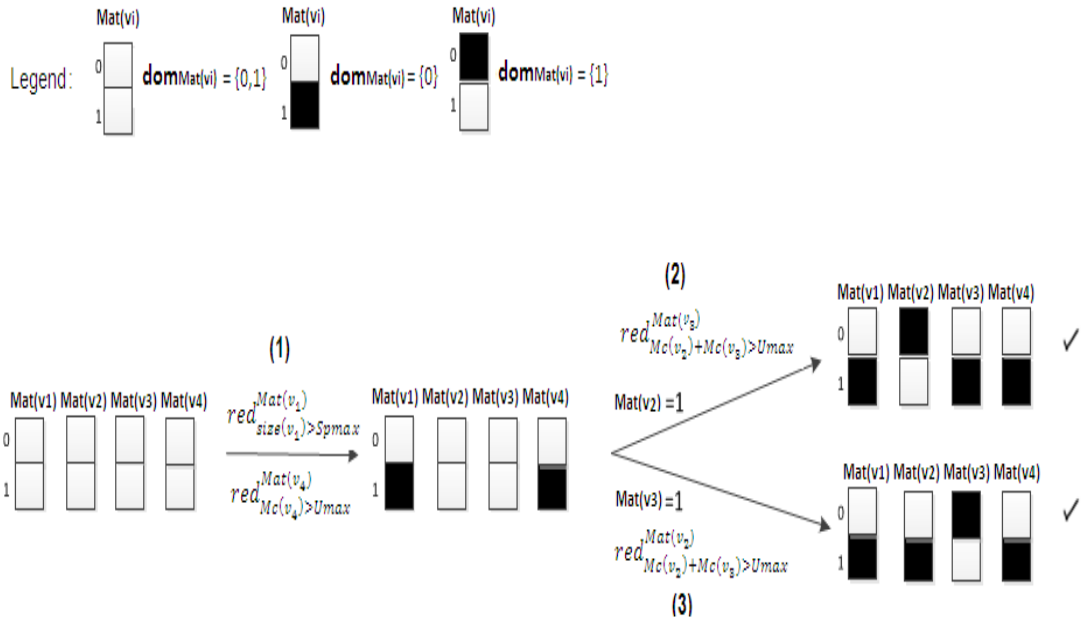


Figure 4.2: Search tree using constraint propagation to select materialized views.

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, $dom_{Mat(v_1)}$=$dom_{Mat(v_2)}$=$dom_{Mat(v_3)}$ =$dom_{Mat(v_4)}$={0,1}, are represented by four columns of white squares as shown in figure 4.2. 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 $dom_{Mat(v_1)}$ and $dom_{Mat(v_4)}$. 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 $dom_{Mat_{v_3}}$. 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 $dom_{Mat(v_2)}$. This enumeration stage leads in our example to two solutions. These solutions are of various quality or cost.

4.4 Performance Evaluation

In this section, we evaluate the performance of our approach through experimentations over the database schema of the TPC-H benchmark [5]. 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 4.2) 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 4.4.1, we describe our experimental setup, and the randomized method used for comparison. In Section 4.4.2, we study the impact of variable and value selection heuristics on the search space explored by our approach. In section 4.4.3, 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 4.4.4, we study the benefit of using materialized views to improve query performance. Finally, we summarize the performance results in Section 4.4.5.

4.4.1 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 the best 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 chapter 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 [2] (knowing that the constraint solvers are structured around annual competitions [43]). For the genetic algorithm, we have implemented the one presented in [15] 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

$$Q_s = \frac{WM - \sum_{v_i \in Q(G)} (f_q(v_i) * Qc(v_i))}{WM - ALLM} \tag{4.9}$$

   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 above, $Qc(v_i)$ 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 [33], Sp_{max} is computed as a function of the size of the associated query workload.

$$Sp_{max} = \alpha * Sp_{AllM} \tag{4.10}$$

   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 [33], U_{max} is calculated as a function of the total maintenance cost when all the queries are materialized.

$$U_{max} = \beta * Mc_{AllM} \tag{4.11}$$

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 4.4.4 to measure the running time of the query workload given a set of materialized view. 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 and the genetic algorithm.

Timeout (sec)	Solution Quality	
	customizing search	default search
0.25	0.523	No solutions
0.5	0.662	No solutions
1	0.705	No solutions
2	0.798	No solutions
4	0.809	No solutions
8	0.827	No solutions
16	**0.836**	No solutions
32		No solutions
64		No solutions
128		0.296
256		0.662
512		0.798
1020		0.827
2048		**0.836**

Table 4.2: Impact of heuristics on the search

4.4.2 Impact of variable and value selection heuristics

Here, we study the impact of variable and value selection heuristics that we have presented in section 4.3.2, 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

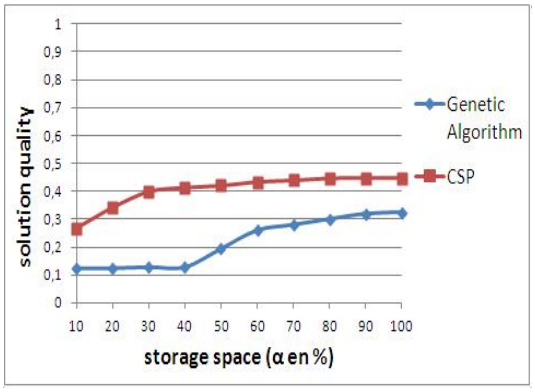
selection heuristics that we have defined in Section 4.3.2 are implemented in the search strategy. As mentioned in chapter 2, the constraint solver (CHOCO Solver) can find a set of feasible solutions in which all the constraints are satisfied before reaching the optimal solution. In this case, we use *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 table 4.2. The bold number **0.836** represents the quality of the optimal solution. *default search* denotes the default search strategy while *customizing search* requires the variable and value selection heuristics that we have defined in the search strategy. We can observe from table 4.2 that the time that a solver takes in the presence of *customizing 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 *customizing search* in the constraint satisfaction model.

4.4.3 Solution quality: Our approach versus Genetic algorithm

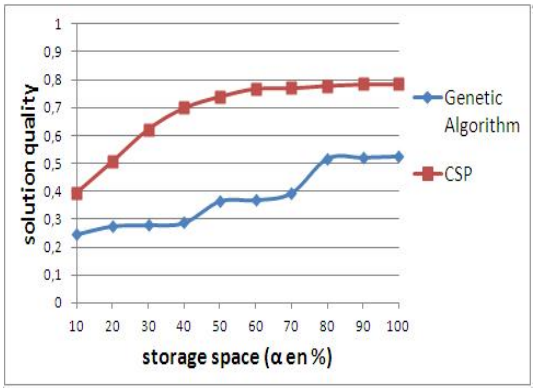
In this section, we examined the effectiveness of our approach by measuring the gain in solution quality obtained by using our approach versus the genetic algorithm. First, we compare the performance of our approach and the genetic algorithm for various values of storage space and maintenance cost limits and then we present the results 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.

Resource constraints

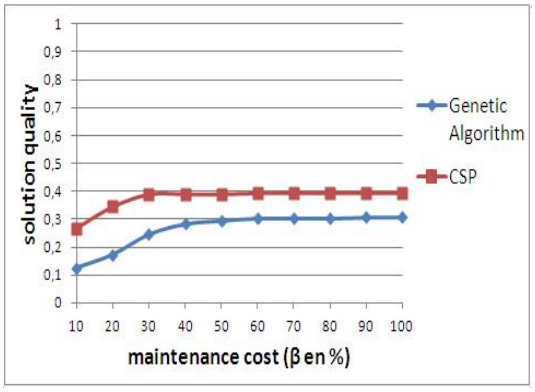
In this experiment, we first examine the impact of space and maintenance cost constraints on solution quality. For this evaluation, we consider a workload of 50



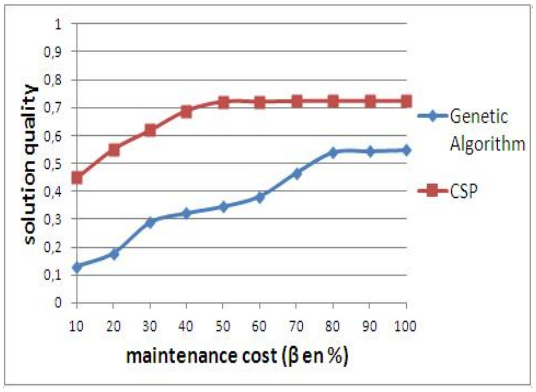
(a) β=10%



(b) β=30%



(c) α=10%



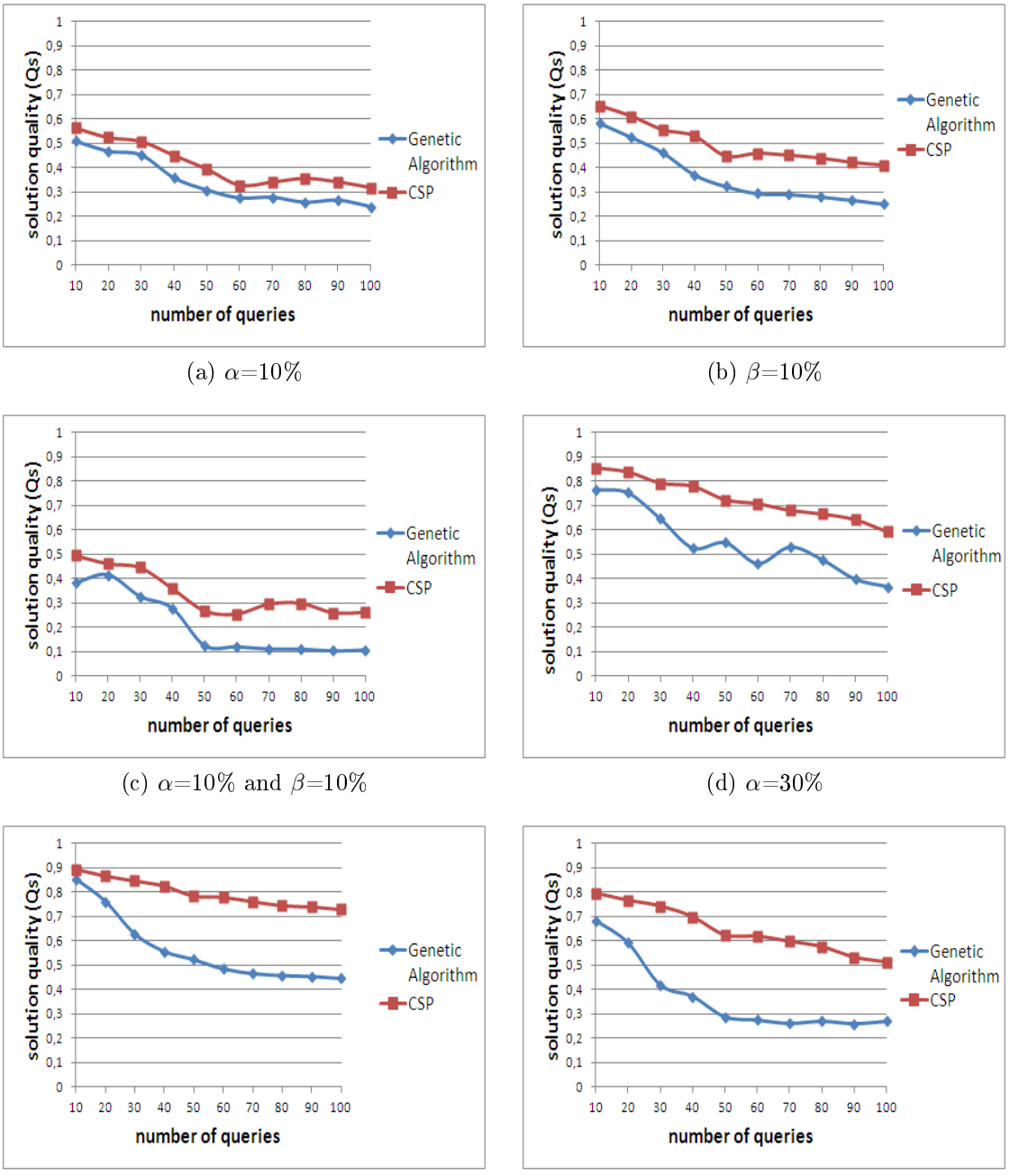
(d) α=30%

Figure 4.3: Solution quality while varying the space or the maintenance cost constraint

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 4.3.

Figure 4.3a and Figure 4.3b 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 4.3c and figure 4.3d 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 4.3a and figure 4.3b) or β (see figure 4.3c and figure 4.3d) 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 4.3 that our approach provides better solution quality



(a) α=10%

(b) β=10%

(c) α=10% and β=10%

(d) α=30%

(e) β=30%

(f) α=30% and β=30%

Figure 4.4: Solution quality on large workloads under different resource constraints

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 4.3b while $Q_s \approx 0.7$ when β=100% and α=30% in figure 4.3d). The reason is the maintenance cost of a 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 [29].

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.

Large query workload

Let us now evaluate the performance of our approach and the one of genetic algorithm on larger query workload. To 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 4.4a and figure 4.4d); (ii) the limiting factor is the view maintenance cost(see figure 4.4b and figure 4.4e); and (iii) both maintenance cost and space constraints exists (see figure 4.4c and figure 4.4f). 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. Another remark based on figure 4.4 is that in our 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 4.4a) and 16% (in figure 4.4b) more than the genetic algorithm. While this gain is up to 18% in figure 4.4c. This is because the idea of constraint programming is to solve problems by stating constraints and the search space is reduced when there is more constraints. This result is similar to the case where we relax the constraints (see figures 4.4d, 4.4e and 4.4f).

Query and update distributions

We now study the behavior of view selection methods while varying the query and update frequencies. To this purpose, we generated different query and update distribution to simulate various workloads (see table 4.3). 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.

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 lower 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 4.3: Distribution of query and update frequencies

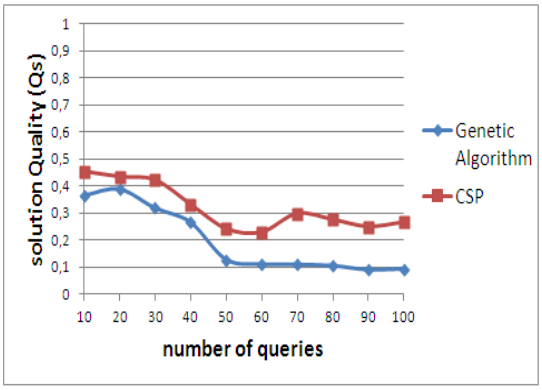
Figures 4.5 illustrates the quality of the solutions produced by the two methods for different query distributions (q_{random}, $q_{uniform}$, $q_{gaussian}$). In the first combination, α and β were set to 10% (see figures 4.5a, 4.5b and 4.5c). While, for the other combination, α and β were set to 30% (see figures 4.5d, 4.5e and 4.5f). 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 4.6).

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 4.5 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.

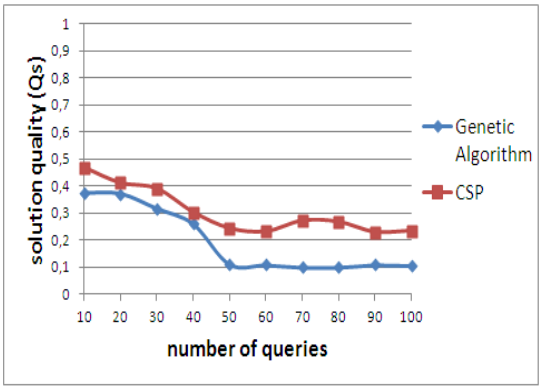
Query complexity

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, ($N_{Join_{Op}} \geq 4$).

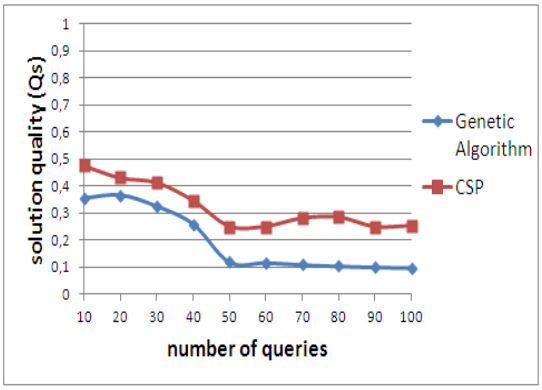
We run experiments with a workload of 50 queries and we measure the gain



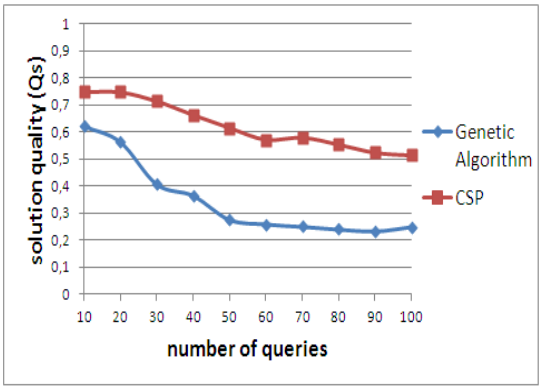
(a) q_{random}, α=10% and β=10%



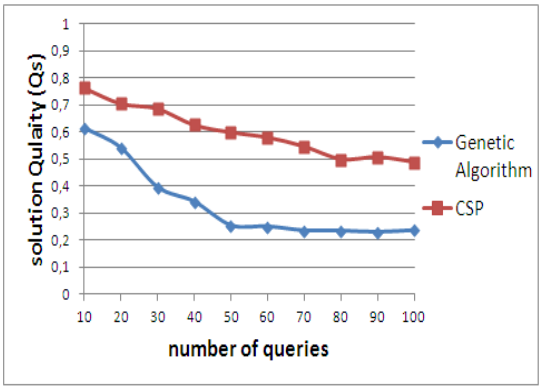
(b) $q_{uniform}$, α=10% and β=10%



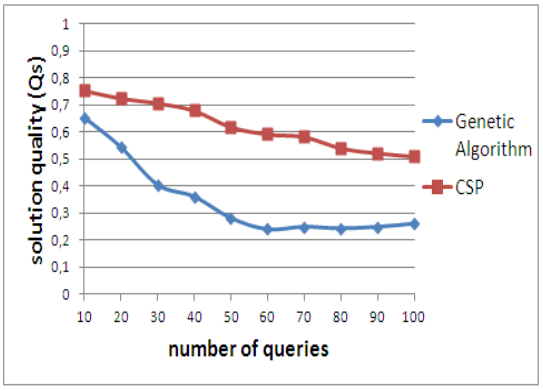
(c) $q_{gaussian}$, α=10% and β=10%



(d) q_{random}, α=30% and β=30%



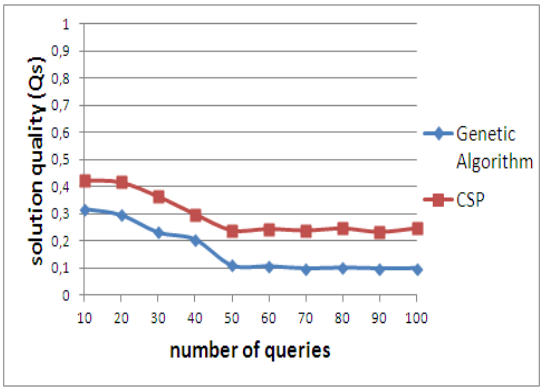
(e) $q_{uniform}$, α=30% and β=30%



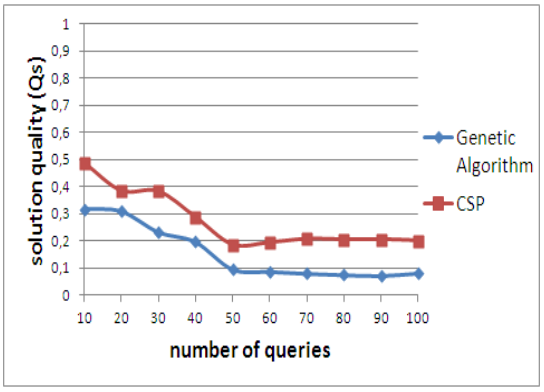
(f) $q_{gaussian}$, α=30% and β=30%

Figure 4.5: Solution quality for different query distributions

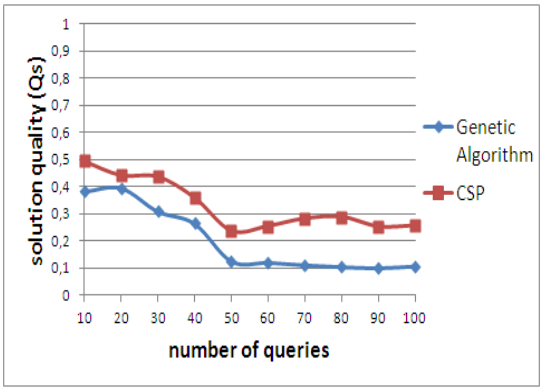
in solution quality according to the set of the obtained materialized views. The frequencies for access and update are at scale 1. Figure 4.7 shows the cost saving found by our approach and the genetic algorithm for both cases: (i) α and β was set to 10% (see figure 4.7a) and (ii) α and β was set to 30% (see figure 4.7b). 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



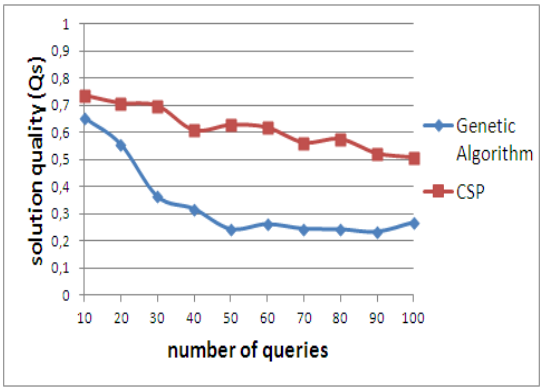
(a) u_{random}, α=10% and β=10%



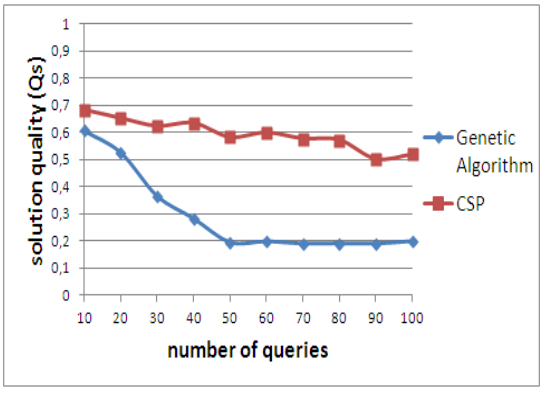
(b) $u_{uniform}$, α=10% and β=10%



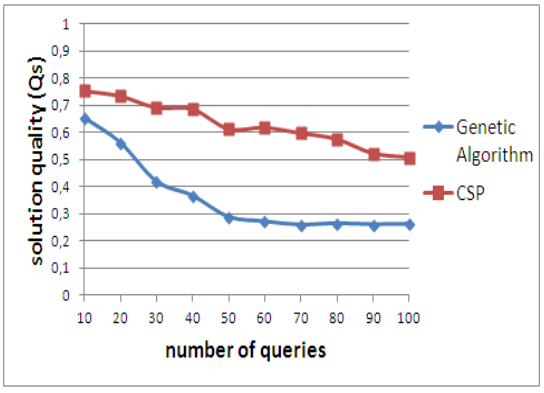
(c) $u_{gaussian}$, α=10% and β=10%



(d) u_{random}, α=30% and β=30%



(e) $u_{uniform}$, α=30% and β=30%



(f) $u_{gaussian}$, α=30% and β=30%

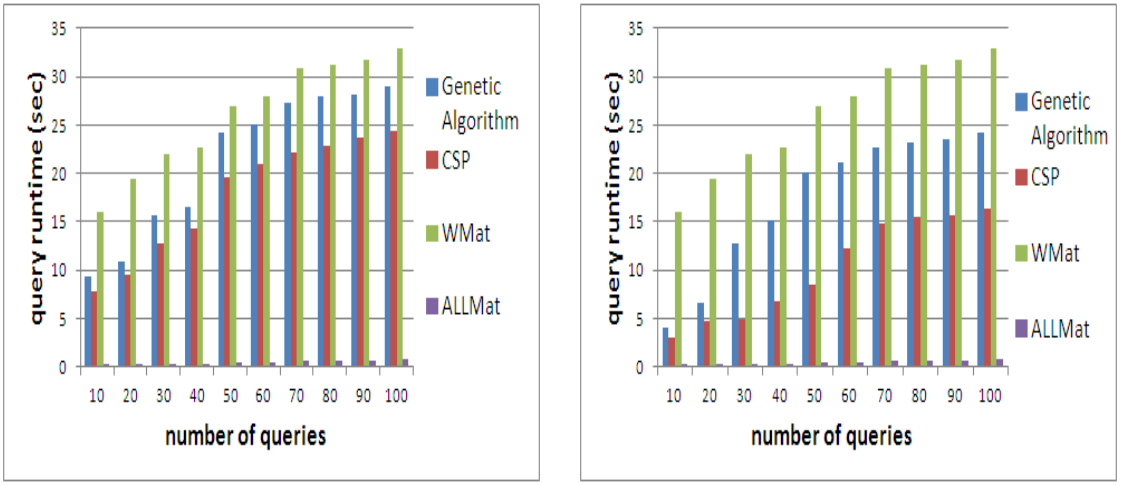
Figure 4.6: Solution quality for different update distributions

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 depicted in figure 4.7, 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%

Figure 4.7: query complexity on view selection performance



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

Figure 4.8: Query runtime using materialized views

4.4.4 query performance using materialized views

In this section, we study the benefit of using the 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. 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).

The results are shown in figure 4.8. The view selection has been decided under space and maintenance cost constraints: (i) α and β was set to 10% in figure 4.8a and (ii) α and β was set to 30% in figure 4.8b. 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 4.8b, when comparing the runtimes of the workload of 100 queries, our approach requires $\approx 16 seconds$ while genetic algorithm takes $\approx 24 seconds$.

4.4.5 Concluding remarks

Our experiments show that our approach achieves significant performance gains in comparison with 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 by generating different query and update distribution and query complexity. 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.

4.5 Conclusion

In this chapter, we proposed a declarative approach which involves constraint programming techniques. The originality of our approach is fourfold. First, our method provides a clear separation between the formulation (model) and the resolution

(solver) of the problems. The main interest of this separation is to propose to the user to model a problem without being interested in the way the problem is solved. Indeed, the user focuses only on the modeling part of the problem. Second, the model allows describing a problem in an easy and declarative way. It simply records the variables and the constraints defining the problem. A complete API is provided to be able to allow to the user to model a problem as constraint satisfaction problem in a natural and effortless way. Third, our approach is flexible and extensible, in that it can easily model and handle new constraints and define heuristic search strategies to reduce the solution space and hence the execution time. Fourth, the use of a constraint programming technique for view selection is new in this field.

We performed several experiments and comparison with the genetic algorithm. The latter is the most efficient algorithm proposed so far for deciding which views to materialize since it provides the best trade-off between quality of solutions and execution time. The experiment results showed that our approach provides better performance compared with the genetic algorithm resulting from evaluating the solution quality (i.e., the quality of the obtained set of materialized views) in terms of cost saving. Experiment results also showed the effectiveness of our heuristic based search strategy which reduce significantly the search space explored by the constraint solver and hence the execution time.

The natural next step (see next chapter), is to focus on extending our approach to address the view selection problem in a distributed setting.

Chapter 5

Modeling View Selection Under Multiple Resource Constraints in a Distributed Context

The view selection problem has received significant attention in past research but most of these studies presented solutions in the centralized context as it is shown in chapter 2. In a distributed environment the view selection problem becomes more challenging for multiple reasons. First, it includes another issue which is to place the materialized views at the appropriate computer nodes. Second, the number of possible views will grow exponentially with the number of computer nodes which make the solution space very large. Besides, distributed scenarios are composed of many heterogeneous nodes with different resource constraints such as CPU, IO and network bandwidth that have to be taken into consideration when attempting at solving the problem. The view selection problem in a distributed context may also be constrained by storage space capacities per computer node and maximum view maintenance cost.

To the best of our knowledge, no past work has addressed this problem under all these resource constraints. Our constraint programming based approach fills this gap. Indeed, all these resource constraints will easily be modeled and handled with the rich constraint programming language. Furthermore, the heuristic algorithms which have been designed to solve the view selection problem in a distributed scenario are deterministic algorithms. For example greedy algorithm [7] and genetic algorithm [15], a type of randomized algorithms. These heuristic algorithms may provide near optimal solutions but there is no guarantee to find the global optimum because of their greedy nature or their probabilistic behavior. We have demonstrated in the last chapter (see chapter 4) the benefit of using constraint

programming techniques for solving the view selection problem with reference to the centralized context. Indeed, our approach is able to provide a near optimal solution to the view selection problem during a given time interval. The quality of this solution may be improved over time until reaching the optimal solution. The major contributions of this chapter (also in [48, 49] as a preliminary version) can be summarized as follows.

1. We propose an extension of the concept of the AND-OR view graph in order to reflect the relation between views and communication network within the distributed scenario.

2. We describe how to model the view selection problem in a distributed context as a constraint satisfaction problem. Its resolution is performed automatically by the constraint solver such as the powerful version of CHOCO [2].

3. We address the view selection problem under multiple resource constraints. The limited resources are the total view maintenance cost and the storage space capacity for each computer node. Furthermore, we consider the IO and CPU costs for each computer node as well as the network bandwidth.

4. We have implemented our approach and compared it with a randomized method i.e., genetic algorithm [15] which has been designed for a distributed setting. We experimentally show that our approach provides better performance resulting from evaluating the quality of the solutions in terms of cost saving.

5. Since the solution space may be huge when we increase the number of computer nodes, we present a set of optimizations and pruning heuristics which reduce the search space and hence the execution time and memory needs.

The rest of this chapter is organized as follows. Section 5.1 defines the view selection problem in a distributed scenario and discusses the settings for the problem. In Section 5.2, we present the framework that we have designed specifically to a distributed setting. Section 5.3 describes how to model the view selection problem under multiple resource constraints in a distributed environment as a constraint satisfaction problem. In Section 5.4, it is provided our experimental evaluation. Section 5.5 presents the set of optimizations and heuristics that we propose to avoid traversing the whole search space. Finally, Section 5.6 contains concluding remarks.

5.1 Problem definition

The general problem of view selection in a centralized context is to select a set of views to be materialized that optimizes the cost of evaluating the query workload given a limited amount of resource e.g., storage space and/or view maintenance time. In a distributed scenario, multiple computer nodes with different resource constraints (i.e., CPU, IO, storage space capacity, network bandwidth, etc.) are connected to each other. Moreover, each computer node may share data and issue numerous queries against other computer nodes.

In this chapter, we have examined the problem of choosing a set of views and a set of computer nodes at which these views should be materialized so that the full query workload is answered with the lowest cost. For this purpose, we have extended the cost model with communication costs. The IO and CPU costs are the only factors considered in a centralized context. The communication cost component is equally important factor considered in a distributed environment. The communication duties are shared between the computer nodes. Recall from the chapter 2 that the communication cost is the time needed to transfer data e.g., transmitting views on the communication network. However, the optimization of communication costs makes the view selection problem more complex.

View materialization can significantly improve the query performance of relational databases and data warehouses. In this chapter, we consider materialized views to optimize complex scenarios where many heterogeneous sites (computer nodes) with different resource constraints query and update numerous base relations on different sites. Before introducing our distributed view selection approach, we present in what follows our motivation to investigate the distributed view selection.

Let us start with a simple example showing how the materialized views in distributed context can improve the query performance. We consider a distributed database scenario consisting of three sites s_1, s_2 and s_3 which are connected by a slow wide-area network. We assume that (s_1, s_2) and (s_1, s_3) are connected by an edge with the speed of 100 KB/sec, while the connection speed between (s_2, s_3) is 3 MB/sec. We consider three base relations r_1, r_2 and r_3 stored at s_1, s_2 and s_3 respectively, so we call them data origin, and two queries q_i and q_j; $q_i = r_1 \bowtie r_3$ is posed frequently at s_2, and $q_j = r_1 \bowtie r_2$ is posed frequently at s_3. Suppose that the query optimizer decides to execute all the queries at s_1 and the results size of q_i and q_j are respectively 1 MB and 2 MB. Let us assume that the cost to answer q_i each time at s_2 is 10 sec and to answer q_j at s_3 is 20 sec, in addition to the cost to compute both queries at s_1. This cost has to be paid every time these queries are posed. While, if the results of q_i and q_j are materialized at s_2 and s_3 respectively, in

this case, both queries have to be computed once at s_1 and the results are shipped to s_2 and s_3 to be materialized with the communication cost of 10 and 20 sec respectively. Then, subsequent queries could be performed with approximately zero cost (only the cost of reading the materialized results is considered).

Obviously, materialized views have significantly reduced the query response time. However, the result of the queries may be too large to fit in the available storage space at a specific site. Therefore, materializing all the queries (or views) is not always possible due to the storage space limitation. Furthermore, whenever a base relation is changed, the materialized views built on it have to be updated in order to compute up-to-date query results. Indeed, the cost of the view maintenance may offset the performance advantages provided by the view materialization. Assume in our example that the relation r_2 is frequently updated, in this case materializing q_j can be less attractive because the cost that we save for answering q_j is now involved for maintaining the materialized view. Thus, the view is considered as beneficial if and only if its materialization reduces significantly the query cost without increasing significantly the view maintenance cost. The communication cost should also be taken into account during the view selection. Assume in the previous example that q_i is frequently used at both s_2 and s_3; in this case materializing q_i only at one site e.g., s_1 can be beneficial enough for the both sites as the communication cost between these sites is not high. In this way, the available storage space at the site s_2 is saved for another beneficial materialized view.

In order to select the right set of materialized views, we have studied the view selection under multiple resource constraints. Resources may be storage space capacity per computer node and maximum view maintenance cost. We also consider the IO and CPU costs of each computer node as well as the network bandwidth.

5.2 Distributed AND-OR View Graph

In order to exhibit common sub-expressions between queries of workload, we have used the AND-OR view graph framework, as it is shown in the previous chapter (see chapter 4). Common sub-expressions can be exploited for sharing computation, updates and storage space.

Recall from the chapter 3 and chapter 4 that, the AND-OR view graph is a Directed Acyclic Graph (DAG) which can be seen as the union of all possible execution plans of each query. It is composed of two types of nodes: Operation nodes (Op-nodes) and Equivalence nodes (Eq-nodes). Each Op-node represents an algebraic expression (Select-Project-Join) with possible aggregate function. An Eq-node

represents a set of logical expressions that are equivalent (i.e., that yield the same result). The Op-nodes have only Eq-nodes as children and Eq-nodes have only Op-nodes as children. The root nodes are equivalence nodes representing the queries and the leaf nodes represent the base relations. Equivalence nodes correspond to the views that are candidates to materialization.

In a centralized context, query execution strategies can be well expressed with respect to the way the relational algebra operators have been performed. In a distributed context, the base relations may be stored at different sites. Hence the choice of ordering relational algebra operators is not enough to express execution strategies. It must be supplemented with the selection of the best sites to process data.

To illustrate this, let us consider the following query expressed in relational algebra: $r_1 \bowtie r_2 \bowtie r_3$. We consider only the join operation since it is probably the most important operation because it is both frequent and expensive. Besides, the permutations of the join order have the most important effect on performance of relational queries. Let assume that the base relations r_1, r_2 and r_3 are respectively stored in sites s_1, s_2 and s_3. We have also made the assumption that the sites s_1 and s_3 are not connected. This query can be executed in at least five different way. In what follows, we describe these strategies, where $r_i \rightarrow s_j$ means that the base relation r_i is transferred to site s_j.

1. $r_1 \rightarrow s_2$
   s_2 computes $r'_1 = r_1 \bowtie r_2$
   $r'_1 \rightarrow s_3$ s_3 computes $r'_1 \bowtie r_3$

2. $r_2 \rightarrow s_1$
   s_1 computes $r'_1 = r_1 \bowtie r_2$
   $r'_1 \rightarrow s_3$ s_3 computes $r'_1 \bowtie r_3$

3. $r_2 \rightarrow s_3$
   s_3 computes $r'_2 = r_2 \bowtie r_3$
   $r'_2 \rightarrow s_1$ s_1 computes $r'_2 \bowtie r_1$

4. $r_3 \rightarrow s_2$
   s_2 computes $r'_3 = r_3 \bowtie r_2$
   $r'_3 \rightarrow s_1$ s_1 computes $r'_3 \bowtie r_1$

5. $r_1 \rightarrow s_2$
   $r_3 \rightarrow s_2$

s_2 computes $r_1 \bowtie r_3 \bowtie r_2$

This example shows the importance of site selection and communication to describe for each query its execution strategies. For this purpose, we have extended the concept of the AND-OR view graph to deal with distributed settings. Therefore, we propose the distributed AND-OR view graph to reflect the relation between views and communication network in the distributed scenario.

Recall that in this work we consider selection-projection-join (SPJ) queries that may involve aggregation and a group by clause as well. Let us consider the query q defined in chapter 2. Query q finds the minimal supply cost for each country and each product having the brand name 'Renault'. The associated query is as follows:

Select **P.partkey, N.nationkey, N.name, Min(PS.supplycost)**
From **Part P, Supplier S, Nation N, PartSupp PS**
Where **P.brand = 'Renault'**
and **P.partkey = PS.partkey**
and **PS.suppkey = S. suppkey**
and **S.nationkey = N.nationkey**
Group by P.partkey, N.nationkey, N.name;

A sample distributed AND-OR view graph is shown in figure 5.1. For simplicity, we consider a network of only three sites s_1, s_2, s_3 and we illustrate a part of the query q by considering only join operations and one execution strategy. Indeed, in figure 5.1 we consider only the join between Part (P) and PartSupp (PS) and the join between PartSupp (PS) and Supplier (S). The execution strategy that we have presented in figure 5.1 is ((P join PS) join S). We suppose that the base relations are stored on different sites.

In order to represent the communication channels, every node is split into three sub-nodes, each of which denotes the view or the execution operation at one site. The communication edges between equivalence nodes of the same level (i.e., $(P-PS-S, s_1)$, $(P-PS-S, s_2)$ and $(P-PS-S, s_3)$), as shown in the dashed rectangle in figure 5.1, denote that a view can be answered from any other site if it is less expensive than computing this view from any children nodes. However, these edges are bidirectional creating cycles which no longer conforms to the characteristics of a DAG. In order to eliminate cycles, each sub-node (v_i, s_j), as illustrated in figure 5.2, has been artificially split into two nodes $(v_i, s_j)'$ and $(v_i, s_j)''$.

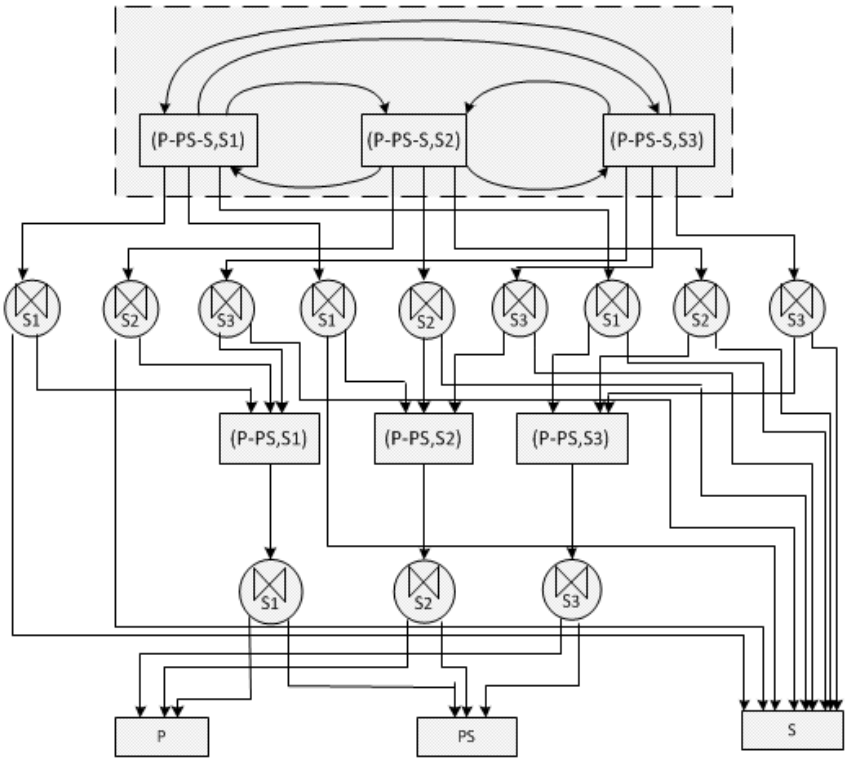


Figure 5.1: Distributed AND-OR view graph.

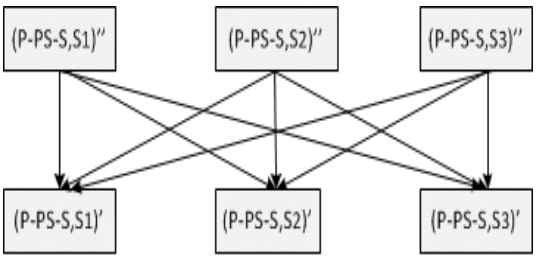


Figure 5.2: Modified Distributed AND-OR view graph.

Before defining the distributed AND-OR view graph formally, the definition of operation nodes Op-nodes and equivalence nodes Eq-nodes has to be extend by considering the distributed settings:

- In addition of representing the expression defined by the algebraic operation and its inputs. The Op-node (op_i, s_j) in the distributed AND-OR view graph has to represent the decision on which site s_j the operation op_i is evaluated.
- Let $Eq-nodes = Eq-nodes' \cup Eq-nodes''$ denote the set of all equivalence nodes. The definition is based on applying split to every equivalence node in order to eliminate cycles. In addition to represent a view that could be selected for materialization, each equivalence node (v_i, s_j) in the distributed AND-OR view graph has to denote at which site s_j this view v_i can be materialized.

Definition 5.1 (AND-OR View Graph) A graph G is called a distributed AND-OR view graph for the views (v_1,s_1), (v_1,s_2),..., (v_1,s_s); (v_2,s_1), (v_2,s_2),..., (v_2,s_s);...; (v_v,s_1), (v_v,s_2),..., (v_v,s_s) if for each view (v_i, s_j), there is an AND-OR-DAG representation (defined in chapter 4).

5.3 Our Distributed View Selection Approach

In this section, we introduce the constraint satisfaction model that we have proposed for the view selection problem in a distributed context. We then show through an example how constraint programming can be applied to select and place materialized views.

5.3.1 Modeling View Selection Problem in a Distributed Context as a Constraint Satisfaction Problem (CSP)

In this subsection, we describe how to model the view selection problem in a distributed scenario as a constraint satisfaction problem. Then, its resolution is performed automatically by the constraint solver. All the symbols as well as the variables that we have used in our CSP are defined in Table 5.1. The view selection in a distributed scenario can be formulated by the following constraint satisfaction model.

$$minimize \qquad \sum_{(v_i,s_j)\in Q(G)} \Big(f_q(v_i) * Qc(v_i, s_j) \Big) \tag{5.1}$$

$$subject\ to \qquad \forall s_j \in S \sum_{(v_i,s_j)\in V(G)} \Big(Mat(v_i,s_j) * size(v_i) \Big) \leq Sp_{max_j} \tag{5.2}$$

$$\sum_{(v_i,s_j)\in V(G)} \Big(Mat(v_i,s_j) * f_u(v_i) * Mc(v_i,s_j) \Big) \leq U_{max} \tag{5.3}$$

In our approach, the main objective is the minimization of the total query cost. The total query cost is computed by summing over the cost of processing each input query rewritten over the materialized views. Constraints (5.2) and (5.3) state that the views are selected to be materialized on a set of sites under a limited amount of resources. Constraint (5.2) ensures that for each site the total space occupied by the materialized views on it is less than its storage space capacity. Constraint (5.3) guarantees that the total maintenance cost of the set of materialized views is less than the maximum view maintenance cost.

In a distributed context, the query and maintenance costs may be formulated as follows.

Query cost

$$Qc(v_i,s_j) = \min_{s_k\in S} \left(Qc_{local}(v_i,s_k) + \frac{size(v_i)}{Bw(s_k,s_j)} \right) \tag{5.4}$$

$$Qc_{local}(v_i,s_j) = \begin{cases} ComputingCost(v_i,s_j) \quad if \quad Mat(v_i,s_j) = 0 \\ size(v_i) * IO_j \ otherwise \end{cases} \tag{5.5}$$

$$ComputingCost(v_i,s_j) = \min_{op_l\in child(v_i,s_j)} \Big(cost(op_l,s_j) + \sum_{(v_m,s_n)\in child(op_l)} \Big(Qc(v_m,s_n) + \tfrac{size(v_m)}{Bw(s_n,s_j)} \Big) \Big) \tag{5.6}$$

The query cost includes the local processing cost and the communication cost. The local processing cost reflects CPU and IO costs (see subsection 2.1). Constraint (5.4) guarantees that a view is answered from the site that can provide the answer with the lowest cost. Constraint (5.5) and (5.6) ensure that the minimum cost path is selected for computing a given view on a given site. Each minimum cost path is composed of all the cost of executing the operation nodes on the path and the

Symbols of CSP model	
G	The distributed AND-OR view graph.
$Q(G)$	The query workload.
$V(G)$	The set of candidate views
U	The set of updates.
$\delta(v_i, s_j, u)$	denotes the differential result of view v_i on s_j, with respect to update u.
f_q	The frequency of a query.
f_u	The update frequency of a query (or view).
S	The set of sites which represent the computer nodes.
Sp_{max_i}	The storage space capacity of the site s_i.
U_{max}	The maximum view maintenance cost.
$size(v_i)$	The size of v_i in terms of number of bytes.
$Bw(s_k, s_j)$	The bandwidth between s_j and s_k.

CSP variables and their domains	
$Mat(v_i, s_j)$	The materialization of the view v_i on site s_j. It is a binary variable ($dom_{Mat(v_i,s_j)}$ ={0,1}; 0: v_i is not materialized on s_j, 1: v_i is materialized on s_j).
$Qc(v_i, s_j)$	The query cost corresponding to the view v_i if it is computed or materialized on site s_j.
$Mc(v_i, s_j)$	The maintenance cost corresponding to the view v_i if it is updated on site s_j.
The costs are defined in terms of time (see subsection 2.1). Their domain is a finite subset of $\mathbb{R}_+^*$($dom_{Qc(v_i,s_j)} \subset \mathbb{R}_+^*$ and $dom_{Mc(v_i,s_j)} \subset \mathbb{R}_+^*$).	

Table 5.1: Symbols and CSP variables in a distributed setting.

query cost corresponding to the related views or bases relations. The reading cost is considered if the view has been materialized.

View maintenance cost

$$Mc(v_i, s_j) = \begin{cases} 0 \;\; if \;\; Mat(v_i, s_j) = 0 \\ \sum_{u \in U(v_i, s_j)} \left(\min_{s_k \in S} \left(Mcost(v_i, s_k, u) + \frac{size(v_i)}{Bw(s_k, s_j)} \right) \right) otherwise \end{cases} \tag{5.7}$$

$$\begin{aligned} &Mcost(v_i, s_j, u) = \min_{op_l \in child(v_i, s_j)} \Big(cost(op_l, s_j, u) + \\ &\sum_{(v_m, s_n) \in child(op_l)} \left(UpdatingCost(v_m, s_n, u) + \frac{size(v_m)}{Bw(s_n, s_j)} \right) \Big) \end{aligned} \tag{5.8}$$

$$UpdatingCost(v_m, s_n, u) = \begin{cases} Mcost(v_m, s_n, u) \;\; if \;\; Mat(v_m, s_n) = 0 \\ \delta(v_m, s_n, u) \; otherwise \end{cases} \tag{5.9}$$

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. Constraint (5.7) guarantees that a view with respect to the updates of the underlying base relations is updated from the site that can provide the differential results with the lowest cost. Constraints (5.8) and (5.9) insure that the best plan with the minimum cost is 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 update on the path and the maintenance cost corresponding to the related views.

Once the user specify the problem as a constraint satisfaction problem, one need is to translate it to a more CHOCO like model that is expressed in the CHOCO API in order to be performed by the solver (recall that we have used the CHOCO solver [2] for the formulation and the resolution of the view selection problem in a distributed context). In the Appendix A, we provide an overview on how we use CHOCO for modeling and solving the view selection problem in a distributed context.

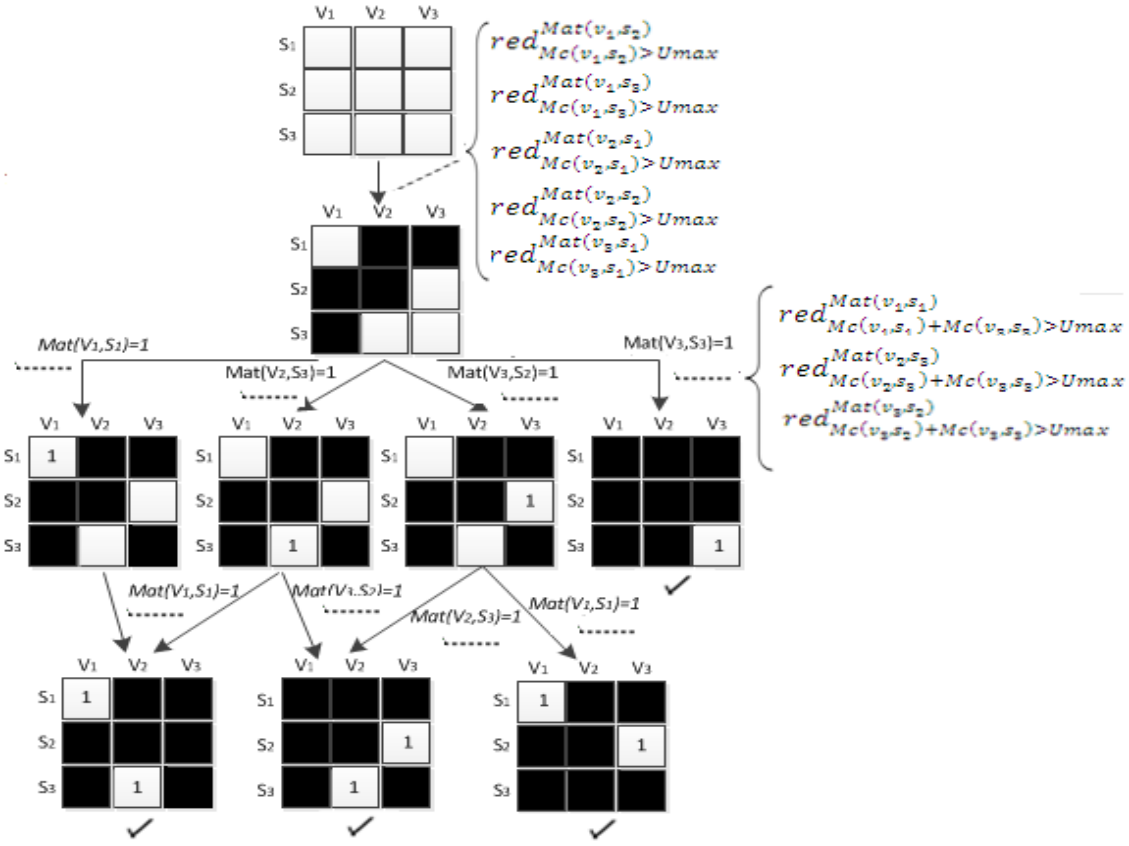


Figure 5.3: Search tree using constraint propagation to select and place materialized views.

5.3.2 Solving the view selection problem in a distributed context with constraint programming

As mentioned in chapter 2, the basic principles of constraint programming are constraint propagation and search. With reference to the last chapter, we illustrated through an example how constraint propagation and search can be applied to decide which views to materialize. In this section, we show how these techniques can be used to solve the view selection problem in a distributed context. For this purpose, let us consider nine variables $Mat(v_1, s_1)$, $Mat(v_1, s_2)$, $Mat(v_1, s_3)$, $Mat(v_2, s_1)$, $Mat(v_2, s_2)$, $Mat(v_2, s_3)$, $Mat(v_3, s_1)$, $Mat(v_3, s_2)$ and $Mat(v_3, s_3)$ where $Mat(v_i, s_j)$ denotes for each view v_i if it is materialized or not materialized on site s_j. It is a binary variable, $dom_{Mat(v_i,s_j)} = \{0,1\}$ (0: v_i is not materialized on site s_j, 1: v_i is materialized on s_j). The problem is to select a set of views and a set of sites at which these views should be materialized under a maintenance cost constraint which guarantees that the total maintenance cost of the set of materialized

views is less than U_{max}. Note that U_{max}=12sec, $Mc(v_1, s_1)$=8sec, $Mc(v_1, s_2)$=14sec, $Mc(v_1, s_3)$=18sec, $Mc(v_2, s_1)$=16sec, $Mc(v_2, s_2)$=15sec, $Mc(v_2, s_3)$=3sec, $Mc(v_3, s_1)$= 13sec, $Mc(v_3, s_2)$=3sec and $Mc(v_3, s_3)$=10sec; where $Mc(v_i, s_j)$ denotes the cost of maintaining the view v_i on site s_j).

Figure 5.3 shows the domain reduction of these variables. At the beginning, the initial variable domains are represented by three columns of white squares meaning that every view can be materialized on any site. Considering the maintenance cost constraint, it appears that $Mat(v_1, s_2)$, $Mat(v_1, s_3)$, $Mat(v_2, s_1)$, $Mat(v_2, s_2)$ and $Mat(v_3, s_1)$ cannot take the value 1 because otherwise the total maintenance cost will be greater than U_{max}. $red^{Mat(v_1,s_2)}_{Mc(v_1,s_2)>U_{max}}$, $red^{Mat(v_1,s_3)}_{Mc(v_1,s_3)>U_{max}}$, $red^{Mat(v_2,s_1)}_{Mc(v_2,s_1)>U_{max}}$, $red^{Mat(v_2,s_2)}_{Mc(v_2,s_2)>U_{max}}$ and $red^{Mat(v_3,s_1)}_{Mc(v_3,s_1)>U_{max}}$ filters respectively the value 1 (the inconsistent value) from the domain of $Mat(v_1, s_2)$, $Mat(v_1, s_3)$, $Mat(v_2, s_1)$, $Mat(v_2, s_2)$ and $Mat(v_3, s_1)$. The deleted values are marked with a black square.

After this stage some variable domains are not reduced to singletons, the solver takes one of these variables and tries to assign it each of the possible values in turn i.e., $Mat(v_3, s_3)$=1. This enumeration stage triggers more reductions i.e., $red^{Mat(v_1,s_1)}_{Mc(v_1,s_1)+Mc(v_3,s_3)>U_{max}}$, $red^{Mat(v_2,s_3)}_{Mc(v_2,s_3)+Mc(v_3,s_3)>U_{max}}$ and $red^{Mat(v_3,s_2)}_{Mc(v_3,s_2)+Mc(v_3,s_3)>U_{max}}$. The reduction of the domain of the other variables is just indicated by dashed lines. This leads in our example to four solutions. These solutions are of various quality or cost.

5.4 Experimental Evaluation

In this section, we demonstrate the performance of our approach and a randomized method i.e., genetic algorithm which has been designed for a distributed setting [15]. 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.

5.4.1 Experiment Settings

For our experiments, we implemented a simulated distributed environment including a network of a set of sites (computer nodes). We assume that the different sites are divided into clusters so that there is a high probability that the sites which belong to the same cluster have similar query workloads. In our approach, for each cluster all the queries of the different workloads are merged into the same graph (see section 3) in order to detect the overlapping and capture the dependencies among them.

Then, our method decides which views have to be selected and determine where these views should be materialized so that the full query workload is answered with the lowest cost under multiple resource constraints. The query workload are defined over the database schema of the TPC-H benchmark [5]. We then randomly assigned values to the frequencies for access and update based on a uniform distribution. In order to solve the view selection problem in a distributed context as a constraint satisfaction problem, we have used the latest powerful version of CHOCO [2]. For the randomized method, we have implemented the genetic algorithm presented in [15] by incorporating space and maintenance cost constraints into the algorithm. 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 satisfy space and maintenance cost constraints are most likely selected for materialization. In the experimental results, the solution quality denoted by Q_s is computed as follows.

$$Q_s = 1 - \frac{\sum_{(v_i,s_j)\in Q(G)}(f_q(v_i) * Qc(v_i,s_j))}{WM} \tag{5.10}$$

Where WM is the total query cost obtained using the "WithoutMat" approach which does not materialize views and always recomputes queries. The "WithoutMat" approach is used as a benchmark for our normalized results. Recall that $Qc(v_i, s_j)$ is the query cost corresponding to the view v_i on site s_j and $f_q(v_i)$ is the frequency of the view v_i corresponding to a single query.

In our approach, the view selection problem in a distributed environment is constrained by storage capacities $Sp_{max} = \{Sp_{max_i}, Sp_{max_j}, .., Sp_{max_n}\}$ where each site s_i has an associated storage space capacity Sp_{max_i} and maximum view maintenance cost U_{max}. Similar to [33] the storage space and maintenance cost limits are computed respectively as a function of the size (see equation 11) and total maintenance cost (see equation 12) of the query workload.

$$Sp_{max_i} = \alpha * Sp_i(AllM) \tag{5.11}$$

$$U_{max} = \beta * Mc(AllM) \tag{5.12}$$

Where $AllM$ is the "AllMat" approach which materializes the result of each query of the workload; α and β are constant. In our experiments, the storage space limit is per site and computed as a function of the size of the associated query workload. The view maintenance cost limit is calculated as a function of the total maintenance cost when all the queries are materialized.

Our approach to solve the view selection problem in a distributed setting is able to provide optimal solutions. However, computing optimal solutions may be very expensive because of the great number of comparisons between all possible subsets of views which are candidate to materialization. In this case, we use *timeout* condition to limit the search by considering that some solutions should not be explored. As mentioned in section 2.2, the constraint solver can find a set of feasible solutions in which all the constraints are satisfied before reaching the optimal solution. In the next experiments, the constraint solver performed a search until reaching the *timeout* condition. Indeed, our approach is able to provide a feasible solution at any time. The *timeout* condition was set to the time required by the genetic algorithm to solve the problem. This means that the constraint solver was left to run until the convergence of the genetic algorithm in the following experiments.

5.4.2 Experiment Results

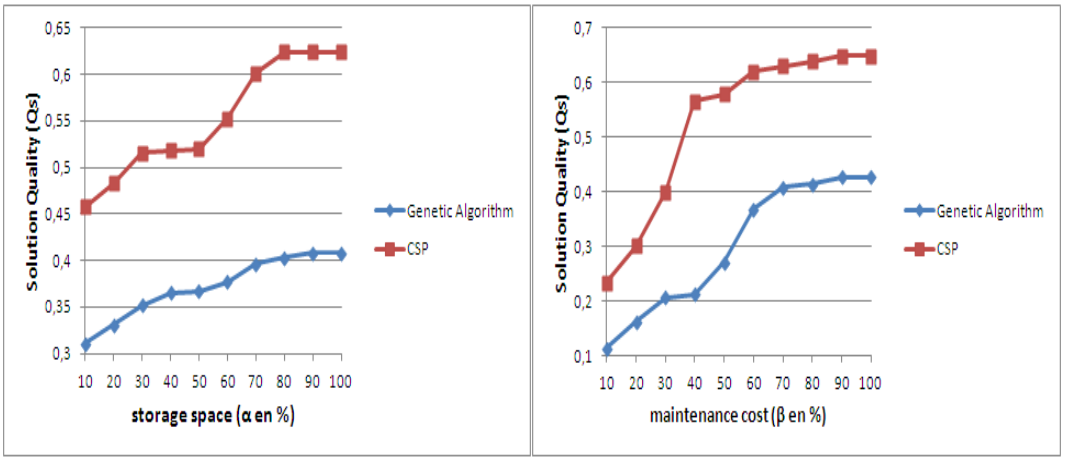
We examined the effectiveness of our approach within three experiments. The first one compares the performance of our approach and the genetic algorithm for various values of storage space and maintenance cost limits. The second experiment evaluates the view selection methods with respect to different sizes of the distributed AND-OR view graph in terms of number of views (equivalence nodes). Finally, the last experiment evaluates our approach and the genetic algorithm with different network sizes in terms of the number of sites per cluster.

Performances under resource constraints.

In this experiment, we examine the impact of space and maintenance cost constraints on solution quality. For this evaluation, each cluster includes 8 sites with different constraints of CPU, IO and network bandwidth and each site has an associated query workload. The values of α and β which define respectively the storage space capacities and the view maintenance cost limit are varied from 10% to 100%. All the results are shown in figure 5.4.

Figure 5.4 (a) investigates the influence of space constraint on solution quality for each value of α where β was set to 60%. We note that the quality of the solutions produced by our approach and genetic algorithm improves when α increases, since there is storage space available for more views to be materialized. However, when $\alpha >= 80\%$ there is no improvement in the solution quality because the maintenance cost constraint becomes the significant factor.

Figure 5.4 (b) examines the impact of maintenance cost constraint on solution



(a) Solution quality while varying the space constraint

(b) Solution quality while varying the maintenance cost constraint

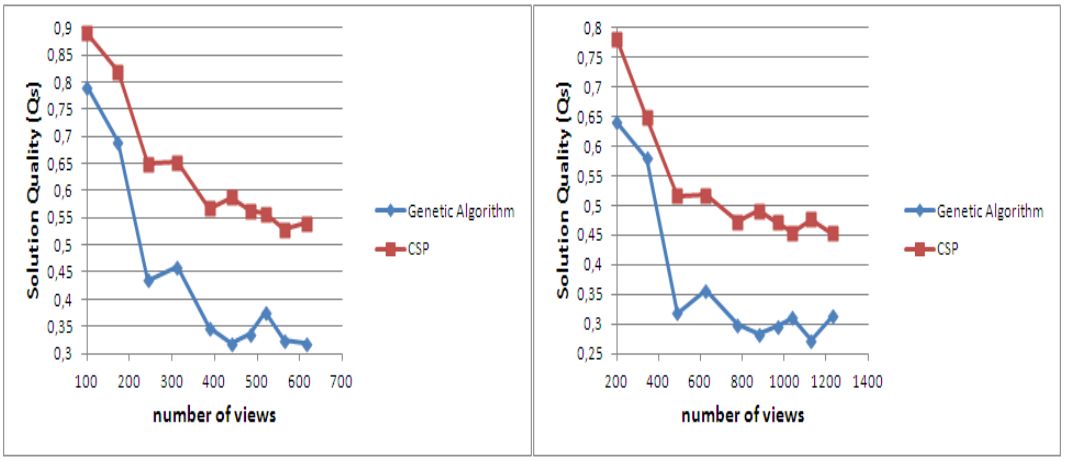
Figure 5.4: Evaluating the performance under resource constraints.

quality for each value of β where α was set to 80%. We can observe similarly to figure 5.4 (a) that we have better solutions when β increases since there is time to update the materialized views. The performance stabilizes when β>=90% because the space constraint becomes the significant factor. We note from these experiments that our approach outperforms the genetic algorithm in the case where the resource constraints become very tight as well as in the case where we relax them. Indeed, for different values of α and β we can see that our approach generates solutions with cost saving more than 2 times more than the genetic algorithm.

Performance according to the number of views.

Let us now evaluate the performance of our approach and the one of genetic algorithm while varying the size of the search space. Recall that the size of the search space is estimated according to the number of views (equivalence nodes) in the distributed AND-OR view graph described in section 3.

Figure 5.5 illustrates the quality of the solutions produced by the two methods in a distributed environment. The number of sites per cluster is 4 sites in figure 5.5 (a) and 8 sites in figure 5.5 (b). The queries of the workload are randomly distributed over the network so that each site has an associated query workload. For instance, in figure 5.5 (b), the number of views in the distributed AND-OR view graph ranges from 200 to 1232 views. For each site, α was set to 40%. For the maintenance cost constraint, β was set to 60%. The experiment results depicted in figure 5.5 (a) and 5.5 (b) show that our approach provides the lowest query cost while varying



(a) Number of sites=4 (b) Number of sites=8

Figure 5.5: Evaluating the performance over different number of views.

the number of views. In fact, the cost saving is up to 27% more than the genetic algorithm. Therefore, our approach provides better performances compared with the genetic algorithm in terms of the solution quality.

Performance according to the number of sites.

In order to evaluate the performance of view selection methods according to the number of sites, we conducted experiments with clusters of different sizes. For each cluster, we considered different number of sites with different constraints of CPU, IO and network bandwidth. The number of sites per cluster varies from 2 to 20. For each site, α was set to 40% and for the maintenance cost constraint, β was set to 60%. The experiment results are shown in figure 5.6. As in the previous experiments, we observe that our approach provides an improvement in the quality of the obtained set of materialized views in terms of cost saving compared with the genetic algorithm. Indeed, the cost saving is up to 15% more than the genetic algorithm.

5.4.3 Experiment Conclusion

We note from the above experiments that our approach outperforms the genetic algorithm in all cases. More specifically, our approach provides the better solution quality in terms of cost saving where we consider (i) various values of storage space and maintenance cost limits; (ii) diverse number of views; and (ii) different network sizes. In our approach, we considered multiple execution plans for each query rather

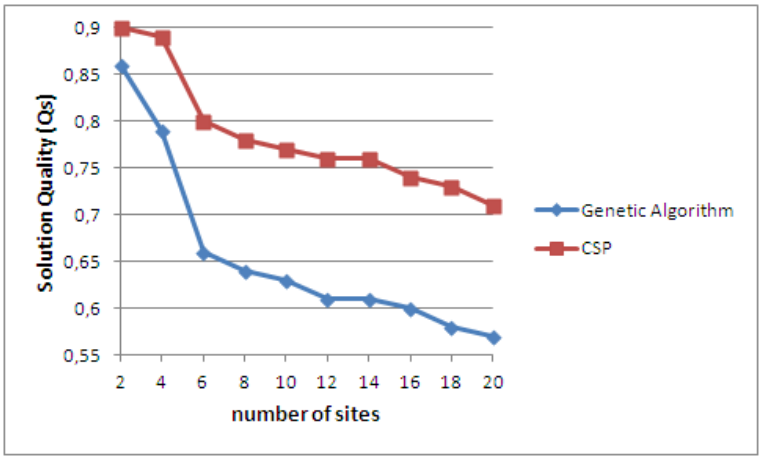


Figure 5.6: Evaluating the performance over different number of sites.

than a unique execution plan in order to find the global optimal plan. Besides, every view can be materialized on any site of the network. Consequently, the search space of candidate views may be very. Indeed, the number of candidate views grows exponentially with the number of queries of workload as well as the number of sites of network. In order to solve the view selection problem in a large scale distributed environment (knowing that in our experiments we consider 20 sites per cluster), our approach must be extended with a set of search strategies to reduce the solution space. In the next section, we discuss a set of optimization and pruning heuristics that may be used in our future work to trade off completeness for efficiency of the search.

5.5 Optimization and heuristics

The complexity of our approach may be very high since the number of possible views to materialize grows exponentially with the number of sites and queries. In order to solve the view selection problem in a distributed environment within reasonable times, special heuristics may be designed to reduce the search space by considering only efficient execution plans and discarding those which are very costly.

An important aspect of query optimization is join ordering [54, 65], since permutations of the joins within the query may lead to improvements of orders of magnitude. Join ordering in a distributed context is even more important since joins between fragments may increase the communication cost which is considered to be the dominant factor in a distributed context. To optimize the communication cost, one need is to find an execution strategy for a query that selects the best or-

dering of operators in the query. Consider the query $r_1 \bowtie r_2 \bowtie r_3$ of Section 2, to select one of the possible execution strategies by using join ordering, the following sizes must be known: $size(r_1), size(r_2), size(r_3), size(r_1 \bowtie r_2)$ and $size(r_2 \bowtie r_3)$. Thus, estimating the size of join results is mandatory. We assume that the cardinality of the resulting join is the product of cardinalities. Then the base relations and the resulting relations are ordered by increasing sizes and the order of execution is given by this order. For instance the order (r_1, r_2, r_3) could use the strategy 1, while the order (r_3, r_2, r_1) could use the strategy 4. Another solution is to estimate the communication cost of all the alternatives strategies and choose the best one. This means that the execution strategy which optimizes the communication cost is selected. Another basic technique for optimizing a sequence of distributed join operators is through the semijoin operator [66]. It replaces joins by combinations of semijoins. The main value of the semijoin in a distributed environment is to reduce the size of the join operands and then the communication cost.

When distributed query optimization is used, either a single site or several sites may participate to the selection of the strategy to be applied for answering the query. However, the number of possible execution strategies is in fact exponential in the number of sites in the network. Indeed, in the distributed AND-OR view graph that we have designed to represent the communication channels, every node is split into sub-nodes, each of which denotes a possible execution strategy. A practical solution for the distributed view selection must design heuristics which select the most promising sites on which the views may be computed or materialized in order to avoid having to pay an important cost of communication. For example, assume that the site s_1 and s_2 are the only issuer of the query $q = r_1 \bowtie r_2$, where the base relations r_1 and r_2 are stored respectively in the sites s_3 and s_4. Naturally, the query has to be computed by involving the sites s_1, s_2, s_3 or s_4, if the goal is to minimize the communication cost. The neighbor sites which are connected to the query issuers with a high bandwidth may also be considered. However, the sites which are not able to communicate with the query issues by direct connection or with a high bandwidth may be discarded in the elaboration of the best execution strategy. Consequently the number of possible execution strategies will be significantly reduced.

With reference to the last chapter (chapter 4), defining heuristic search strategies within the constraint solver allows to reduce the solution space. Indeed, a well-suited strategy can reduce the number of expanded nodes (Recall from chapter 2 that the search in a constraint programming approach is organized as an enumeration tree, where each node corresponds to a subspace of the search) and hence the execution time required to find the set of materialized views. The most common branching

strategies in the constraint solver are based on the assignment of a selected variable to one or several selected values. In what follows, we describe the variable and value selection heuristics that may be defined in the search space to reduce the time and memory space.

Variable and Value Selection Heuristic. Our objective is to guide the search close to the optimal solution which leads the constraint solver to find near optimal solutions fast. In our work, the best solution is the one that minimizes the query cost subject to space and maintenance cost constraints. In our model, the query cost reflects the local processing cost and the communication cost. As mentioned before, the communication cost is considered to be the dominant factor in a distributed environment. For this purpose, we apply the following heuristic: A view is preferably to be placed (materialized) closest to where it is more frequently accessed. In the pseudo-code below, we describe our heuristic.

for *each* (v_i, s_j) *in* $V(G)$ **do**
  if $size(v_i) \leq Sp_{max_j}$ *and* $Mc(v_i, s_j) \leq U_{Max}$ **then**
    // compute the benefit of materializing the view v_i *on site* s_j
    $Benefit(v_i, s_j) = TCost(v_i, s_{origin} \rightarrow s_q) - TCost(v_i, s_j \rightarrow s_q)$
  end if
end for

Note that $TCost(v_i, s_{origin} \rightarrow s_q)$ is the cost of answering v_i at s_q by using the data origin and $TCost(v_i, s_j \rightarrow s_q)$ is the cost of answering v_i at s_q by using the data from the site s_j. This cost includes the total cost of reading the required data and performing the necessary computations and the communication costs for transferring the data.

The variable selector has to start by instantiating the variables $Mat(v_i, s_j)$ with the highest benefit. For this purpose, we sort the views in $V(G)$ according to their $Benefit$ in descending order (as it is presented below). We iterate over the sorted set starting with the views which have the highest benefit and we store them according to this order in the variable MVS.

//sort according to the Benefit in descending order
$VSSort = SortViewsSites(Benefit)$
for *each* (v_i, s_j) *in* $VSSort$ **do**
  $MVS = MVS \cup \{Mat_{v_i,s_j}\}$
end for

Then, the variable selector will choose the materialization variable Mat_{v_i,s_j} in the order they appear in MVS. Once the variable has been selected, the value selector will assign the variable to its highest value: $max(d_{Mat_{v_i,s_j}})$. As this way, the view v_i is considered as materialized on site s_j. By defining these heuristics in the search strategy, we expect that time and memory that the constraint solver incurs to find near optimal solutions and the optimal solution will be significantly reduced since a large number of nodes in the search tree will be discarded.

5.6 Conclusion

In this chapter we have extended our constraint satisfaction based approach to address the view selection problem under multiple resource constraints in a distributed environment. Furthermore, we have introduced the distributed AND-OR view graph to reflect the relation between views and communication network.

We have performed several experiments over TPC-H queries and comparison with a genetic algorithm which has been designed for a distributed setting. The first experimental results have shown that our approach provides better performance where the space and maintenance cost constraints become very tight as well as in the case where we relax them or when the number of views is high. Besides, our approach provides better solution quality in terms of cost saving when we consider diverse number of sites.

However, the complexity of our approach is high since the number of possible views to be materialized grows exponentially with the number of queries and sites. This highlights the need for optimizations and efficient heuristic. For this purpose, we have provide an overview of a set of pruning heuristics that we can use for reducing the search space and hence solving the view selection in a large scale distributed environments within reasonable execution times.

Because our approach is flexible and extensible, these optimizations and heuristics will be easily handled by our approach. More precisely, they will be modeled as new constraints in the CHOCO model or as search strategies in the CHOCH solver. We expect that this will allow our approach to scale well with the query workload and the network sizes.

Next chapter summarizes the main contributions of this thesis, enumerates the related issues not covered by our approach and provides some perspectives in the view selection context.

Chapter 6

Conclusion

This chapter concludes the dissertation by presenting three aspects of our research work. We first summarize our main contributions to address the view selection problem. Then, we outline the limitations and related issues not covered by our approach. Finally, we discuss future directions of research in the view selection field.

6.1 Main Contributions

This work has addressed the problem of choosing an appropriate set of views to materialize. This problem is crucial in commercial database systems to facilitate efficient query processing. Furthermore, this is the most studied problem in data warehousing systems to obtain the optimal query performance. Another application of the view selection issue is selecting views to materialize in distributed database and data warehouse architectures. Our main contributions are as follows.

View selection and constraint programming framework. We defined a framework in chapter 2 that may be helpful to discuss physical database design while focusing on the problem of selecting materialized views for improving the performance of a database system. First, we introduced the main notions and provided the basic contents related to the view selection context so that a reader can have a rather precise idea of the whole context and its potential. Then, as our work is based on constraint programming techniques, we extended our framework to capture the full breadth and depth of the constraint programming field. We also described our motivation to use constraint programming techniques to address the view selection problem. Finally, we showed the basics of modeling and solving with constraint solvers such as CHOCO [2].

State of the art of view selection methods. We reviewed in chapter 3 the view selection methods that have been proposed to address the view selection problem. The view selection dimensions have been highlighted in detail as the basic for classifying the view selection approaches. Consequently, we have discussed three main class of view selection methods based on what kind of heuristic algorithms they use: deterministic Algorithms Based Methods, randomized algorithms based methods and hybrid Algorithms Based Methods. Based on this classification, we survey and review the existing view selection methods by identifying their respective potentials and limitations. Further, we gave an overview of the dynamic view selection methods. Finally, we summarized our review of related work and our observations, which has been useful to introduce our approach.

A survey of view selection methods was published in [47].

Best balance between solution quality and execution time. We proposed in chapter 4 a novel approach that is based on a constraint programming technique. The latter is known to be efficient for the resolution of NP-complete problems and a powerful method for modeling and solving combinatorial optimization problems. The originality of our approach is that it provided a clear separation between formulation and resolution of the problem. For this purpose, the view selection problem has been modeled as a constraint satisfaction problem in an easy and declarative way. Then, its resolution was performed automatically by the constraint solver. Through performance experimentation, we showed the effectiveness of our approach. Indeed, our results demonstrate that our approach provides the best trade-off between the solution quality in terms of cost saving and execution time, compared to the most efficient method so far.

The initial version of this work appeared in [50] and an improvement proposal is going to be submitted.

Distributed view selection under multiple resource constraints. In a distributed environment the view selection problem becomes more challenging. Indeed, it includes another issue which is to decide on which computer nodes the selected views should be materialized. Furthermore, resource constraints such as CPU, IO and network bandwidth have to be taken into consideration. The view selection problem in a distributed context may also be constrained by storage space capacities per computer node and maximum view maintenance cost. Since our approach is flexible and extensible in that it can easily model and handle new constraints, we have extended the constraint satisfaction model to solve the problem under all these resource constraints. Experiment results showed that the performance of our approach remains significant when the view selection is studied in a distributed

setting. However, the complexity of our approach is high since the number of possible views to be materialized in a distributed context grows exponentially with the number of computer nodes. For this purpose, we provided an overview of a set of optimizations and pruning heuristics that we can use for reducing the search space. Solving the view selection in a large scale distributed environments is an important challenge which might included in our work.

This work was published in [48, 49].

6.2 On going work

This section covers the issues not handled by our proposed approach and discusses some possible extensions of our current work.

Reducing the search space of candidate views. In this work, we considered for each query all possible execution plans which represent its execution strategies. Then, we used the AND-OR view graph to compactly represent alternative query plans and exhibit common sub-expression. For distributed scenarios, we extended the AND-OR view graph to reflect the relation between views and communication network. However, the AND-OR view graph may become very large since the number of possible views to materialize grows exponentially with the number of computer nodes and queries, and with the number of join predicates, grouping clauses and base relations referenced in each query. Due to the huge solution space, special and efficient heuristics have to be designed in order to reduce the search space of candidate views to materialization. In the last chapter (see chapter 5), we discussed a set of pruning heuristics such join ordering and site selection that it would be interesting to use to restrict the solution space of the view selection problem for complex scenarios.

Improving the search strategy. A practical solution to the view selection problem for complex scenarios requires an efficient search strategy to be defined within the constraint solver in order to efficiently search the solution space. Indeed, a well-suited search strategy can reduce the number of expanded nodes in the enumeration tree and hence the time that the constraint solver incurs to find solutions to the view selection problem. A search strategy in CHOCO is a composition of branching strategies. The most common branching strategies are based on the assignment of a selected variable to one or several selected values. In the previous chapter (see chapter 5), we described the variable and value selection heuristics that may be defined in the search strategy. We believe that these heuristics will lead the constraint solver to finding a materialized view set fast without traversing the whole

search space.

Defining a limit search space Our work highlights the trade-off between view selection quality and time performance since our approach outperforms the genetic algorithm which is known to provide the best balance between the solution quality and the execution time. To allow a fair comparison and because our approach can provide a solution to the view selection problem at any time, the limit search space was set to the time required by the genetic algorithm to solve the view selection problem. It lacks the discussion between the limit search space, the time performance and the solution quality which requires more research work. One need is to define a limit on the depth of the search so that the view selection problem can be solved within reasonable execution time and the quality of the solutions found by the constraint solver remains high in terms of cost saving. One idea consists in using sampling and learning methods to compute the search time limit (for a given AND-OR view graph).

6.3 Open issues

In this section, we present some of directions of research in view selection field that we believe to be interesting to explore in future work.

Query performance in semantic web databases. The increasing interest in semantic web technology and their many applications has turned the query performance improving over semantic web databases to a challenging and crucial task. To this aim, recent research works [14, 20, 23] has explored materialized view selection in semantic web databases in order to facilitate efficient processing of RDF queries and updates.

The goals of view selection in semantic web databases and view selection in relational databases are similar; it is only that the data model is different since RDF data set is fundamentally a collection of tuples of the form $(subject, property, object)$, belonging conceptually to a single *triple* relation [4]. Therefore, given the similarities between these two studies, our approach may be applied the context of view selection for RDF data.

Dynamic view selection. In our proposals, the view selection problem has been addressed in the context of one fixed current workload. All queries are assumed to be known and given in advance and there is a frequency of occurrence associated with each query. In order to respond to the changes in the query workload over time, views need to be selected continuously. Consequently, administrating efforts such as monitoring and reconfiguration may be a part of our planned future work.

As mentioned above, one line of recent research has explored the problem of efficiently recommending a set of views to materialize in order to improve the RDF query performance [14, 20, 23]. However, they consider a static workload which contradicts the dynamic nature of the web. Indeed, any change to the workload should be reflected to the view selection as well. This issue will be the future aspect while studying the view selection in semantic web databases.

View selection in large scale networks. Analysis of state of the art of view selection has shown that there are very few works on view selection in distributed databases and data warehouses and no effective solution for peer to peer systems. Indeed, [25] seems to be the only paper which deals with the view selection problem in a peer to peer environment. In fact, it is provided a full definition of the problem but without providing any algorithm or detail on how to select an effective set of views to materialize and place them at appropriate peers. Thus, one of challenging directions of future work aims at addressing the view selection problem in a peer to peer environment. The view selection in the cloud computing, in which large amounts of data, content and knowledge are being spread over the service providers' infrastructures, is also an open issue. Recently, [52] is working on a novel cost models that complement the existing materialized view cost models with a monetary cost component that is primordial in the cloud.

Appendix A

Use of CHOCO for modeling and solving the view selection problem

A.1 Centralized Context

In what follows, we provide an insight on how to create the constraint satisfaction model by using the large Javadoc API provided by the CHOCO constraint solver. Note that in the following sample model, we have not considered the view maintenance.

First of all, we import the CHOCO class to use the API.

```
import choco.Choco;
```

Let's create a Model. More precisely, we create an instance of CPModel() for Constraint Programming Model.

```
// Build the model
CPModel m = new CPModel();
```

Then, we declare the variables of the problem

```
// Creation of HashMaps of Integer variables
Map<Noeud,IntegerVariable> Matv=new HashMap<Noeud,IntegerVariable>();
Map<Noeud,IntegerVariable> Qc=new HashMap<Noeud,IntegerVariable>();
for(Node n: G.getNodes()) {
  if (n.getType()==node.equivalence || n.getType()==node.source){
 //For each variable, we define the type and the boundaries of its domain
   IntegerVariable QcNode = makeIntVar(0,Integer.MAX_VALUE,"cp:bound");
   Qc.put(n,QcNode);
   IntegerVariable matNode = makeIntVar(0,1,"cp:decision");
```

```
    Matv.put(n,matNode);
   }
}
```

We have defined the variable using the makeIntVar method which creates a bounded domain. Now, we are going to state constraints ensuring that the minimum cost path (in the AND-OR view graph) is selected for computing a given view.

```
for(Node n: G.getNodes()) {
//Each base relation is stored and its query cost is zero.
  if(n.getType() == Node.source) {
   m.addConstraint(eq(matv.get(n),1));
   m.addConstraint(eq(Qc.get(n),0));
   }
  else if(n.getType() == Node.equivalence) {
   int cpt =0;
   for(Node op : n.getChildren()) {
    possibilites[cpt]=makeIntVar(0,Integer.MAX_VALUE,"cp:bound");
    IntegerVariable[] viewsOp = new IntegerVariable[op.getChildren().size()];
    int cpt2 =0;
    for(Node v : op.getChildren())
     viewsOp[cpt2++] = Qc.get(v);
    //Each minimum cost path includes the cost of executing the operation nodes on the path and the query cost corresponding to the related bases relations or views.
     m.addConstraint( eq( possibilites[cpt++], sum(cost(op),sum(viewsOp))));
      //The query cost corresponding to each given view 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.
     m.addConstraint( ifThenElse(eq(materialized.get(n),1),
                        eq(size(n),Qc.get(n)),
                        min(possibilites,Qc.get(n))));
   }
  }
}
```

Then, we add the constraint ensuring that the total space occupied by the materialized views is less than or equal to the maximum storage space capacity.

```
//Creation of an array of variables for the space constraint
IntegerVariable[] sizeViewtab =new IntegerVariable[nbEqNodes];
int j=0;
  for(Node n: G.getEquivalenceNodes()) {
   sizeViewtab[j]= makeIntVar(0,Integer.MAX_VALUE,"cp:bound");
   m.addConstraint(eq(sizeViewtab[j++],mult(matv.get(n),size(n))));
  }
//State the space constraint
m.addConstraint(leq(sum(sizeViewtab),Sp_max));
```

An optimal solution to the view selection problem is then a solution that minimizes the objective variable which is the total query cost.

```
//Creation of an array of variables to compute the objective variable totalQueryCost
IntegerVariable[] FreqcostQtab =new IntegerVariable[nbQueries];
int i=0;
 for(Node n: G.getRootNodes()) {
  FreqcostQtab[i]= makeIntVar(0,Integer.MAX_VALUE,"cp:bound");
  The query cost is weighted by the query frequency indicating the importance of the associated query.
  m.addConstraint(eq(FreqcostQtab[i++],mult(Qc.get(n),fq.get(n))));
 }
//The total query cost is computed by summing over the cost of processing each input query rewritten over the materialized views.
m.addConstraint(eq(sum(FreqcostQtab), totalQueryCost));
```

Now, we have defined the model. The next step is to solve it. For that, we build the solver. We create an instance of CPSolver() for Constraint Programming Solver. Then, the solver reads (translates) the model, defines the search strategy and solves the model.

```
//Build the solver
solver = new CPSolver();
//Read the model
solver.read(m);
//Defining the value selector which is applied to the materialization variable (the decision variable)
solver.setValIntSelector(new MaxVal());
```

```
//Defining the variable selector that choose the decision variables to be instantiated in the order they appear in the variable MV that we have defined above.
solver.setVarIntSelector(new StaticVarOrder(solveur.getVar(MV)));
//Solving the model by minimizing the objective variable
solver.minimize(solver.getVar(totalQueryCost));
```

A.2 Distribted Context

This section gives a general overview on how to create the constraint satisfaction model that we have designed to the view selection problem in a distributed context by using the CHOCO API. For simplicity, we have not considered the view maintenance in the following model.

First of all, we import the CHOCO class to use the API.

```
import choco.Choco;
```

Let's create a Model. More precisely, we create an instance of CPModel() for Constraint Programming Model.

```
// Build the model
CPModel m = new CPModel();
```

Then, we declare the variables of the problem

```
// Creation of HashMaps of Integer variables
Map<Noeud,IntegerVariable> Matv=new HashMap<Noeud,IntegerVariable>();
Map<Noeud,IntegerVariable> Qc_local=new HashMap<Noeud,IntegerVariable>();
Map<Noeud,IntegerVariable> Qc=new HashMap<Noeud,IntegerVariable>();
for(Node n: G.getNodes()) {
  if (n.getType()==node.equivalence || n.getType()==node.source){
 //For each variable, we define the type and the boundaries of its domain
  IntegerVariable QcNodeL = makeIntVar(0,Integer.MAX_VALUE,"cp:bound");
   Qc_local.put(n,QcNodeL);
   IntegerVariable QcNode = makeIntVar(0,Integer.MAX_VALUE,"cp:bound");
   Qc.put(n,QcNode);
   IntegerVariable matNode = makeIntVar(0,1,"cp:decision");
   Matv.put(n,matNode);
  }
}
```

We have defined the variable using the makeIntVar method which creates a bounded domain. Now, we are going to state constraints ensuring that a view is answered from the site that can provide the answer with the lowest cost. They also guarantee

that the minimum cost path is selected for computing a given view on a given site.

```
for(Node n: G.getNodes()) {
//Each base relation is stored and its query cost is zero.
  if(n.getType() == Node.source) {
   m.addConstraint(eq(matv.get(n),1));
   m.addConstraint(eq(Qc_local.get(n),0));
   m.addConstraint(eq(Qc.get(n),0));
   }
  else if(n.getType() == Node.equivalence) {
   int cpt =0;
   for(Node op : n.getChildren()) {
    possibilites[cpt]=makeIntVar(0,Integer.MAX_VALUE,"cp:bound");
    IntegerVariable[] viewsOp = new IntegerVariable[op.getChildren().size()];
    int cpt2 =0;
    for(Node v : op.getChildren()) {
     viewsOperation[cpt2]=makeIntVar(0,Integer.MAX_VALUE,"cp:bound");
      //The query cost includes the local processing cost and the communication
cost.
     m.addConstraint(eq(viewsOperation[cpt2++],plus(Qc.get(v),CCost)));
      }
    //Each minimum cost path includes the cost of executing the operation nodes
on the path and the query cost corresponding to the related bases relations or
views.
     m.addConstraint( eq( possibilites[cpt++], sum(cost(op),sum(viewsOp))));
      //The query cost corresponding to each given view 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.
     m.addConstraint( ifThenElse(eq(materialized.get(n),1),
                          eq(size(n),Qc_local.get(n)),
                          min(possibilites,Qc_local.get(n))));
    }
   }
   //Data can be shipped from another site. With the reference with figure 5.1,
let us assume that the view P-PS-S is materialized on site s_2 and s_1 requires this
view to answer to a given query. Hence, P-PS-S can be transferred to the site s_1
if the communication cost between these sites is not high.
```

```
    int cptSL=0;
   for(Node b : n.getBrothers()) {
    possSameLevel[cptSL]=makeIntVar(0,Integer.MAX_VALUE,"cp:bound");
     //The query cost includes the local processing cost and the communication
cost.
    m.addConstraint(eq( possSameLevel[cptSL++], plus(Qc_local.get(b),CCost)));
   }
   A view is answered from the site that can provide the answer with the lowest
cost.
   m.addConstraint(min(possSameLevel,Qc.get(n)));
}
```

Then, we add the constraint ensuring for each site that the total space occupied by the materialized views on it is less than its storage space capacity.

```
//Creation of an array of variables for the space constraint
IntegerVariable[][] sizeViewtab =new IntegerVariable[nbEqNodes][nbSites];
int j=0;
  for(Site s: G.getSites()) {
  int k=0;
   for(Node n: G.getEquivalenceNodes(s)) {
    sizeViewtab[j][k]= makeIntVar(0,Integer.MAX_VALUE,"cp:bound");
    m.addConstraint(eq(sizeViewtab[j++][k++],mult(matv.get(n),size(n))));
   }
//State the space constraint
m.addConstraint(leq(sum(sizeViewtab),Sp_max(s)));   }
```

The main objective of our approach is to minimize the total query cost (objective variable)

```
//Creation of an array of variables to compute the objective variable
totalQueryCost
IntegerVariable[] FreqcostQtab =new IntegerVariable[nbQ];
int i=0;
 for(Node n: G.getRootNodes()) {
  FreqcostQtab[i]= makeIntVar(0,Integer.MAX_VALUE,"cp:bound");
   The query cost is weighted by the query frequency indicating the importance
of the associated query.
  m.addConstraint(eq(FreqcostQtab[i++],mult(Qc.get(n),fq.get(n))));
 }
```

```
//The total query cost is computed by summing over the cost of processing each input query rewritten over the materialized views.
m.addConstraint(eq(sum(FreqcostQtab), totalQueryCost));
```

Now, we have defined the model. The next step is to solve it. For that, we build the solver. We create an instance of CPSolver() for Constraint Programming Solver. Then, the solver reads (translates) the model, defines the search strategy and solves the model.

```
//Build the solver
solver = new CPSolver();
//Read the model
solver.read(m);
//Solving the model by minimizing the objective variable
solver.minimize(solver.getVar(totalQueryCost));
```

Bibliography

[1] Choco: an open source java constraint programming library. http://www.emn.fr/z-info/choco-solver/uploads/pdf/choco-presentation.pdf.

[2] Choco, open-source software for constraint satisfaction problems. http://www.emn.fr/z-info/choco-solver.

[3] Choco solver documentation. http://choco.svn.sourceforge.net/viewvc/choco/trunk/src/site/resources/tex/documentation/choco-doc.pdf.

[4] Rdf semantic web standards. available at http://www.w3.org/rdf/.

[5] The TPC benchmark H (TPC-H). http://www.tpc.org/tpch/spec/tpch2.14.3.pdf.

[6] Materialized view selection in data warehousing: A survey. *Journal of Applied Sciences.*, pages 401–414, 2009.

[7] Andreas Bauer 0004 and Wolfgang Lehner. On solving the view selection problem in distributed data warehouse architectures. In *SSDBM*, pages 43–, Cambridge, Massachusetts, USA, 2003.

[8] 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.

[9] Xavier Baril and Zohra Bellahsene. Selection of materialized views: A cost-based approach. In *CAiSE*, pages 665–680, Klagenfurt, Austria, 2003.

[10] Zohra Bellahsene, Michelle Cart, and Nour Kadi. A cooperative approach to view selection and placement in p2p systems - (short paper). In *OTM Conferences (1)*, pages 515–522, Hersonissos, Crete, Greece, 2010.

[11] Zohra Bellahsene, Robbie Coenmans, and John Tranier. Matérialisation de vues dans les entrepôts de données. une approche dynamique. *Ingénierie des Systèmes d'Information*, 11(6):33–53, 2006.

[12] Sally C. Brailsford, Chris N. Potts, and Barbara M. Smith. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research*, 119(3):557–581, 1999.

[13] Yves Caseau and François Laburthe. Improved clp scheduling with task intervals. In *ICLP*, pages 369–383, Santa Margherita Ligure, Italy, 1994.

[14] Roger Castillo and Ulf Leser. Selecting materialized views for rdf data. In *Proceedings of the 10th international conference on Current trends in web engineering*, ICWE'10, pages 126–137, Berlin, Heidelberg, 2010. Springer-Verlag.

[15] 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.

[16] Rada Chirkova, Alon Y. Halevy, and Dan Suciu. A formal perspective on the view selection problem. *VLDB J.*, 11(3):216–237, 2002.

[17] Roozbeh Derakhshan, Frank K. H. A. Dehne, Othmar Korn, and Bela Stantic. Simulated annealing for materialized view selection in data warehousing environment. In *Databases and Applications*, pages 89–94, 2006.

[18] Roozbeh Derakhshan, Bela Stantic, Othmar Korn, and Frank K. H. A. Dehne. Parallel simulated annealing for materialized view selection in data warehousing environments. In *ICA3PP*, pages 121–132, island of Cyprus, 2008.

[19] Prasad Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F. Naughton. Caching multidimensional queries using chunks. In *SIGMOD Conference*, pages 259–270, Seattle, Washington, 1998.

[20] Vicky Dritsou, Panos Constantopoulos, Antonios Deligiannakis, and Yannis Kotidis. Optimizing query shortcuts in rdf databases. In *ESWC (2)*, pages 77–92, Heraklion, Gereece, 2011.

[21] 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.

[22] Mohammad Ghoniem, Hadrien Cambazard, Jean-Daniel Fekete, and Narendra Jussien. Peeking in solver strategies using explanations visualization of dynamic

graphs for constraint programming. In *Proceedings of the 2005 ACM symposium on Software visualization*, SoftVis '05, pages 27–36, New York, NY, USA, 2005. ACM.

[23] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. View selection in semantic web databases. *PVLDB*, 5(2):97–108, 2011.

[24] David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley, 1989.

[25] Steven D. Gribble, Alon Y. Halevy, Zachary G. Ives, Maya Rodrig, and Dan Suciu. What can database do for peer-to-peer? In *WebDB*, pages 31–36, Santa Barbara, California, USA, 2001.

[26] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[27] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD Conference*, pages 157–166, Washington, D.C., 1993.

[28] Himanshu Gupta. Selection of views to materialize in a data warehouse. In *ICDT*, pages 98–112, Delphi, Greece, 1997.

[29] Himanshu Gupta and Inderpal Singh Mumick. Selection of views to materialize under a maintenance cost constraint. In *ICDT*, pages 453–470, Jerusalem, Israel, 1999.

[30] Alon Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[31] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *SIGMOD Conference*, pages 205–216, Montreal, Canada, 1996.

[32] Jorng-Tzong Horng, Yu-Jan Chang, and Baw-Jhiune Liu. Applying evolutionary algorithms to materialized view selection in a data warehouse. *Soft Comput.*, 7(8):574–581, 2003.

[33] Panos Kalnis, Nikos Mamoulis, and Dimitris Papadias. View selection using randomized search. *Data Knowl. Eng.*, 42(1):89–111, 2002.

[34] Panos Kalnis, Wee Siong Ng, Beng Chin Ooi, Dimitris Papadias, and Kian-Lee Tan. An adaptive peer-to-peer network for distributed caching of olap results. In *SIGMOD Conference*, pages 25–36, Madison, Wisconsin, USA, 2002.

[35] Howard J. Karloff and Milena Mihail. On the complexity of the view-selection problem. In *PODS*, pages 167–173, Philadephia, Pennsylvania, USA, 1999.

[36] Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[37] Donald Kossmann, Michael J. Franklin, and Gerhard Drasch. Cache investment: Integrating query optimization and distributed data placement. *ACM TODS*, 25:2000, 2000.

[38] Yannis Kotidis and Nick Roussopoulos. Dynamat: A dynamic view management system for data warehouses. In *SIGMOD Conference*, pages 371–382, Philadephia, Pennsylvania, USA, 1999.

[39] Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

[40] P. J. M. Laarhoven and E. H. L. Aarts, editors. *Simulated annealing: theory and applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[41] 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.

[42] Alexandros Labrinidis, Qiong Luo, Jie Xu, and Wenwei Xue. Caching and materialization for web databases. *Found. Trends databases*, 2(3):169–266, March 2010.

[43] Christophe Lecoutre, Olivier Roussel, and Marc R. C. van Dongen. Promoting robust black-box solvers through competitions. *Constraints*, 15(3):317–326, 2010.

[44] 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.

[45] Spyros Ligoudistianos, Dimitri Theodoratos, and Timos K. Sellis. Experimental evaluation of data warehouse configuration algorithms. In *DEXA Workshop*, pages 218–223, Vienna, Austria, 1998.

[46] 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.

[47] Imene Mami and Zohra Bellahsene. A survey of view selection methods. *SIGMOD Record*, 41(1):20–29, 2012.

[48] 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.

[49] Imene Mami, Zohra Bellahsene, and Remi Coletta. A constraint satisfaction based approach to view selection in a distributed environment. Clermont-Ferrand, France, Bases de Donnees Avancees (BDA), 2012.

[50] Imene Mami, Remi Coletta, and Zhora Bellahsene. Modeling view selection as a constraint satisfaction problem. In *DEXA (2)*, pages 396–410, Toulouse, France, 2011.

[51] 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.

[52] Thi-Van-Anh Nguyen, Sandro Bimonte, Laurent d'Orazio, and Jérôme Darmont. Cost models for view materialization in the cloud. In *EDBT/ICDT Workshops*, pages 47–54, Berlin, Germany, 2012.

[53] Nils J. Nilsson. *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill Pub. Co., 1971.

[54] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, third edition.* Springer, 2011.

[55] Dallan Quass, Ashish Gupta, Inderpal Singh Mumick, and Jennifer Widom. Making views self-maintainable for data warehousing. In *PDIS*, pages 158–169, Miami Beach, Florida, USA, 1996.

[56] Luc De Raedt, Tias Guns, and Siegfried Nijssen. Constraint programming for itemset mining. In *KDD*, pages 204–212, Las Vegas, USA, 2008.

[57] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *SIGMOD Conference*, pages 447–458, Montreal, Canada, 1996.

[58] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[59] Nick Roussopoulos. The logical access path schema of a database. *IEEE Trans. Software Eng.*, 8(6):563–573, 1982.

[60] Nick Roussopoulos. View indexing in relational databases. *ACM Trans. Database Syst.*, 7(2):258–290, 1982.

[61] 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.

[62] Peter Scheuermann, Junho Shim, and Radek Vingralek. Watchman : A data warehouse intelligent cache manager. In *VLDB*, pages 51–62, Bombay, India, 1996.

[63] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. Colt: continuous on-line tuning. In *SIGMOD Conference*, pages 793–795, Chicago, Illinois, USA, 2006.

[64] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, It. A. Lorie, and T. G. Price. Access path selection in a relational database management system. pages 23–34, 1979.

[65] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB J.*, 6(3):191–208, 1997.

[66] Konrad Stocker, Donald Kossmann, Reinhard Braumandl, Alfons Kemper, and UniversitÃď t Passau. Integrating semi-join-reducers into state-of-the-art query processors. In *In Proc. of the Intl. Workshop on Research Issues in Data Engineering*, pages 124–131, 2001.

[67] Dimitri Theodoratos, Theodore Dalamagas, Alkis Simitsis, and Manos Stavropoulos. A randomized approach for the incremental design of an evolving data warehouse. In *ER*, pages 325–338, Yokohama, Japon, 2001.

[68] Dimitri Theodoratos, Spyros Ligoudistianos, and Timos K. Sellis. Designing the global data warehouse with spj views. In *CAiSE*, pages 180–194, Heidelberg, Germany, 1999.

[69] Dimitri Theodoratos, Spyros Ligoudistianos, and Timos K. Sellis. View selection for designing the global data warehouse. *Data Knowl. Eng.*, 39(3):219–240, 2001.

[70] Dimitri Theodoratos and Timos K. Sellis. Data warehouse configuration. In *VLDB*, pages 126–135, Athens, Greece, 1997.

[71] Dimitri Theodoratos and Timos K. Sellis. Data warehouse schema and instance design. In *ER*, pages 363–376, Singapore, 1998.

[72] Edward Tsang. Foundations of constraint satisfaction, 1993.

[73] Mark Wallace. Practical applications of constraint programming. *Constraints*, 1:139–168, 1996. 10.1007/BF00143881.

[74] Jennifer Widom. Research problems in data warehousing. In *CIKM*, pages 25–30, Baltimore, Maryland, USA, 1995.

[75] Jian Yang, Kamalakar Karlapalem, and Qing Li. Algorithms for materialized view design in data warehousing environment. In *VLDB*, pages 136–145, Athens, Greece, 1997.

[76] Jian Yang, Kamalakar Karlapalem, and Qing Li. A framework for designing materialized views in data warehousing environment. In *Baltimore, Maryland, USA*, pages 0–, 1997.

[77] Wei Ye, Ning Gu, Genxing Yang, and Zhenyu Liu. Extended derivation cube based view materialization selection in distributed data warehouse. In *WAIM*, pages 245–256, Hangzhou, China, 2005.

[78] Jeffrey Xu Yu, Xin Yao, Chi-Hon Choi, and Gang Gou. Materialized view selection as constrained evolutionary optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 33(4):458–467, 2003.

[79] Chuan Zhang and Jian Yang. Genetic algorithm for materialized view selection in data warehouse environments. In *DaWaK*, pages 116–125, Florence, Italy, 1999.

[80] 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.

[81] Jingren Zhou, Per-Åke Larson, Jonathan Goldstein, and Luping Ding. Dynamic materialized views. In *ICDE*, pages 526–535, Istanbul, Turkey, 2007.

[82] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *SIGMOD Conference*, pages 316–327, San Jose, California, 1995.