ABSTRACT

AUTOMATIC PHYSICAL DATABASE DESIGN: RECOMMENDING MATERIALIZED VIEWS

by
Wugang Xu

This work discusses physical database design while focusing on the problem of selecting materialized views for improving the performance of a database system. We first address the satisfiability and implication problems for mixed arithmetic constraints. The results are used to support the construction of a search space for view selection problems. We proposed an approach for constructing a search space based on identifying maximum commonalities among queries and on rewriting queries using views. These commonalities are used to define candidate views for materialization from which an optimal or near-optimal set can be chosen as a solution to the view selection problem. Using a search space constructed this way, we address a specific instance of the view selection problem that aims at minimizing the view maintenance cost of multiple materialized views using multi-query optimization techniques. Further, we study this same problem in the context of a commercial database management system in the presence of memory and time restrictions. We also suggest a heuristic approach for maintaining the views while guaranteeing that the restrictions are satisfied. Finally, we consider a dynamic version of the view selection problem where the workload is a sequence of query and update statements. In this case, the views can be created (materialized) and dropped during the execution of the workload. We have implemented our approaches to the dynamic view selection problem and performed extensive experimental testing. Our experiments show that our approaches perform in most cases better than previous ones in terms of effectiveness and efficiency.

AUTOMATIC PHYSICAL DATABASE DESIGN: RECOMMENDING MATERIALIZED VIEWS

by
Wugang Xu

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Electrical Engineering

Department of Electrical and Computer Engineering

August 2007

APPROVAL PAGE

AUTOMATIC PHYSICAL DATABASE DESIGN: RECOMMENDING MATERIALIZED VIEWS

Wugang Xu

Dimitri Theodoratos, Ph.D, Dissertation Advisor — Date
Associate Professor, Department of Computer Science, NJIT

Vijay Atluri, Ph.D, Committee Member — Date
Professor, Management Science and Information Systems Department, Rutgers University

Narain Gehani, Ph.D, Committee Member — Date
Professor, Department of Computer Science, NJIT

James Geller, Ph.D, Committee Member — Date
Professor, Department of Computer Science, NJIT

Vincent Oria, Ph.D, Committee Member — Date
Associate Professor, Department of Computer Science, NJIT

BIOGRAPHICAL SKETCH

Author:	Wugang Xu
Degree:	Doctor of Philosophy
Date:	August 2007

Undergraduate and Graduate Education:

- PhD in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2007
- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2002
- Master of Science in Computing Acoustics,
  Institute of Acoustics, Chinese Academy of Sciences, Beijing, China, Jun 2000
- Bachelor of Science in Physics,
  Peking University, Beijing, China, July 1997

Major: Computer Science

Presentations and Publications:

Wugang Xu, Dimitri Theodoratos, "A Dynamic View Selection Problem for Sequences of Query and Update Statements" *In Preparation for Journal Submission.*

Wugang Xu and Dimitri Theodoratos and Calisto Zuzarte "View Maintenance Using Closest Multi-Query Optimization Techniques," *Ready for Journal Submission.*

Dimitri Theodoratos and Wugang Xu and Calisto Zuzarte "Mixed Arithmetic Constraint in Databases: Satisfiability, Implication, and Attribute Range," *Submitted to the Journal of Intelligent Information Systems.*

Wugang Xu and Dimitri Theodoratos and Calisto Zuzarte "A Dynamic View Materialization Scheme for Sequences of Query and Update Statements," *In Proceeding of the 9th International Conference on Data Warehousing and Knowledge Discovery*, Regensburg, Germany, to Appear.

Wugang Xu and Calisto Zuzarte and Dimitri Theodoratos, "Computing Closest Common Subexpressions for View Selection Problems," *In Proceedings of ACM 9th International Workshop on Data Warehousing and OLAP*, Arlington, Virginia, USA, 2006, pp. 75-82.

Wugang Xu and Calisto Zuzarte and Dimitri Theodoratos, "Preprocessing for Fast Refreshing Materialized Views in DB2," *In Proceedings of the 8th International Conference on Data Warehousing and Knowledge Discovery*, Krakow, Poland, 2006, pp. 55-64.

Dimitri Theodoratos, Wugang Xu, "Constructing Search Spaces for Materialized View Selection," *In Proceedings of ACM 7th International Workshop on Data Warehousing and OLAP*, Washington, DC, USA, 2004, pp. 112-121.

To my wife, Zhou Lin and my parents, Xu Lingyun and Ren Meifeng

ACKNOWLEDGMENT

I give my special thanks to my advisor, Dr. Dimitri Theodoratos for his valuable ideas and his untiring guidance. I greatly appreciate his generosity with time, his inspiring suggestions and encouragement.

I am grateful to Dr. James Geller for agreeing to be on my committee. He gave me much advice on my dissertation.

I am very thankful to Dr. Vijay Atluri, Dr. Narain Gehani and Dr. Vincent Oria for agreeing to be on my committee and for their support during my research at NJIT.

I thank all the faculty in the Department of Computer Sciences at NJIT for imparting their knowledge and for the many research discussions with them.

I thank IBM Canada Ltd. for providing me a fellowship and an opportunity to work with their research and development staff. Sincere thanks to Dr. Calisto Zuzarte for his support and advice throughout my stay at IBM Canada. Thanks to Dr. Kelly Lyons and all other staff of CAS in IBM Canada for their generous hosting and support.

Thanks to my best friends, Cheng Shouxian and Xu Jun, who have accompanied me throughout my graduate school and helped me both in life and in studies. Finally, I take this opportunity to express my deep gratitude to my beloved wife, Zhou Lin, for her love and for her faith in me. Without her, I would not have been able to accomplish this dissertation and degree.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

LIST OF FIGURES

LIST OF FIGURES
(Continued)

CHAPTER 1

INTRODUCTION

1.1 Motivation

Nowadays, databases are used very frequently in software applications. Enterprises store in their databases not only transactional data for a single department, but also historical data for the whole enterprise. This need for storing large amount of data led to the introduction of data warehouses [141, 23, 74, 150]. At the same time, the queries are getting more complex (e.g., queries involving roll-up and drill down operations and top-k queries) [18, 129, 23, 58]. They include not only online transaction processing (OLTP) queries, but also online analytical processing (OLAP) queries. These latter queries are more complex in structure and usually take a longer time than OLTP queries to execute [88]. Both data warehousing and OLAP queries are essential elements of decision support applications [12], which have increasingly become the focus of the database industry [23].

Designing and tuning a decision support system in the presence of large data sets and complex queries is very challenging for database designers and administrators. The traditional techniques such as normalization [32, 115, 48], index selection [52] and query plan optimization [77] are not sufficient to satisfy these new chanllenge, even with the increasing computing power of the hardware. Many techniques have been proposed, such as data partitioning (both horizontal [19] and vertical [39]), parallel computing [107] and view materialization [24, 148]. Among all the suggested techniques, the materialization of views is the most important one, and has the most significant effect on the performance of the underlying database systems. Benchmarks and applications show that the use of materialized views can improve the response time of decision support queries by several orders of magnitude [151].

Creating and using materialized views generalizes that of indexes. Both indexes and materialized views are redundant structures that speed up query execution, compete for the same resources such as storage, and incur maintenance overhead in the presence of updates. On the other hand, a materialized view is much richer in structure than an index since it may be defined over multiple tables, and can involve selections, grouping and aggregations over multiple columns. In fact, an index can be logically considered a special case of a single-table projection-only materialized view. This richness of structure of materialized views makes the problem of selecting views for materialization significantly more complex than that of selecting indexes [8]. This work focuses on selecting materialized views.

The performance of a database depends heavily on its continuous tuning by the database administrators. With the use of view materialization techniques, the task of the database administrators becomes more complex. For this reason, most current commercial database management systems focus on providing tools to help database designers and administrators to design and tune databases by automating the selection of materialized views and indexes (e.g., Oracle [40], Microsoft SQL Server [25, 26, 8, 7], IBM DB2 [140, 153, 154]). These tools recommend indexes and materialized views in the presence of a storage constraint. However, M. P. Consens et al. [38] show that there is a large margin of improvement for these tools in recommending indexes and materialized views.

1.2 Background

Inn this section, we provide a brief introduction to the background of the view selection problems.

Platform configuration is the hardware and software configuration that affects the performance of a database system. Examples include the speed and number of CPUs, the access speed and size of memory and disks, the disk management schemes of the operating system, etc. A platform configuration may affect the selection of indexes and materialized views. For instance, the disk space may determine the space available for materialization which may affect the views and indexes that will be materialized. In the following chapters, it is assumed that the platform configuration is fixed when indexes and materialized views are recommended.

Logical database design refers to the logical schema of the database such as the relations, integrity constraints, normalization information, etc. A logical database design has a significant effect on the performance of the application queries and updates. For example, implementing a data warehouse as a snowflake [23] or cascaded-star [149] instead of a star [23] by normalizing dimensional relations may affect the overall performance of the database system. However, the logical database design is visible to the applications that use the database. Whenever a logical database design changes, the corresponding queries and update statements upon this database may also need to be rewritten to adapt to these changes. For this reason, the logical database design is rarely modified in an operational database. In the following chapters, it is assumed that the logical database design is fixed when indexes and materialized views are recommended.

Physical database design refers to physical access structures and table storage choices used in the database that can be understood by the state-of-the-art optimizers. For example, table clustering, bitmap indexes, horizontal partitioning and materialized views are physical design options. The work in this dissertation focuses on recommending materialized views. Therefore, the physical database design refers to the recommended indexes and materialized views used in a database. These indexes and views are usually the output of a general physical configuration recommender in a current database management system.

Workload refers to a collection of query and update statements issued against the underlying database. A workload can be considered as a set if the order of statements is of no significance. Otherwise, it is considered as a sequence of a specific order. Sometimes, only a partial order of the statements in a workload is needed. For example, in order to keep the semantic consistency of statements in a workload with mixed statements of queries and updates, a partial order is sufficient. For another example, in a workload of queries with different priorities, no order needs to be specified among queries with the same priority. In most contexts, a workload of a database can be decided by the applications that use the database or are collected by a monitoring tool of the database system.

Constraint is a condition that has to be satisfied by the database system. Many constraints can be characterized as either space constraints or time constraints. An example of a space constraint is the following: the size of the recommended indexes and materialized views should not exceed the size of the space available for materialization. An example of a time constraint is the following: the time needed to update the recommended indexes and materialized view should not exceed a given time window (e.g., the time at night, when the database is not used). Sometimes a constraint may be a combination of both a space constraint and a time constraint. There are also constraints that are not related to time or space. Such an example is a constraint that requires all the queries to be answerable from the selected materialized views without accessing the base relations.

Optimization goal is the minimization (or maximization) of a function by selecting a set of indexes and/or materialized views. Usually, the function is a time and/or space measurement. For example, an optimization goal may be the minimization of the total evaluation cost of a workload. Another optimization goal may be the minimization of the total size of the selected physical structures.

1.3 Problem Formulation

The problem of recommending materialized views can be defined as follows using the concepts introduced above:

Given a platform configuration, a logical database design, a workload of statements, a constraint, and an optimization goal, find a physical database design such that the constraint is satisfied and the optimization goal is achieved.

This problem is also referred to as a "view selection problem." There are many variations of this problem based on different logical database designs, workloads, constraints and optimization goals. For instance, the data warehouse design problem is a variation of the view selection problem where the workload is a set of queries, the constraint is a restriction on the total size of the materialized views, and the optimization goal is the minimization of the combination of the query evaluation and view maintenance cost.

If the selected physical structures can be only created before the execution of the statements in the workload, the view selection problem is called *static view selection problem*. If the selected physical structures can be created and dropped during the execution of statements in the workload, the view selection problem is called *dynamic view selection problem*. For the static view selection problem, the order of statements in the workload is of no importance. In contrast, this order affects several versions of the dynamic view selection problem.

1.4 Outline

Chapter 2 addresses the issues of maintaining materialized views and answering queries using materialized views. Chapter 3 is a review on the view selection problem. In Chapter 4, preliminary theoretical results on arithmetic constraints are provided. These results are used in the following chapters. In Chapter 5, we suggest a method for constructing search spaces for general versions of the view selection problems. An extension of this method is presented in Chapter 6. Chapter 7 addresses an instance of the view selection problem in which a set of views is selected in order to maintain multiple materialized views using multi-query optimization techniques. Chapter 8 discusses how the approach of maintaining multiple views using multi-query optimization techniques can be applied to a commercial database management system. Chapter 9 considers a dynamic view selection problem with space constraints where the workload is a sequence of queries and update statements. Chapter 10 concludes the whole work.

CHAPTER 2

VIEW MATERIALIZATION

In order to use materialized views for improving the performance of a database system, two central issues need to be addressed: (1) The set of materialized views must be synchronized with the updates of base tables upon which these views are defined [67, 23, 143]; (2) The queries in the workload must be rewritten using materialized views [96, 126]. In this chapter, previous work on the problem of view maintenance and the problem of rewriting queries using views is reviewed.

2.1 View Maintenance

A materialized view gets "dirty" whenever the underlying base tables are modified. The process of updating a materialized view in response to changes to the underlying data is called *view maintenance* or *view refreshing*.

There are different view refreshing policies. The view refreshing may occur after every single update at any data source, or it can be done periodically, e.g., daily or monthly. View refreshing can also be performed on demand, according to the needs of the end users, or after the occurrence of triggering events. Obviously, "when to refresh" is a trade-off between consistency and time, the size of the window of opportunity, the capacity of the OLTP systems that may be impaired by the propagation of data, etc. View refreshing can be applied in an on-line or off-line mode. On-line refreshing allows the applications to continue functioning while the refreshing is performed. However, the integrity of the data should be guaranteed. If an update is propagated from a base table to a materialized view and aggregates have been built on this materialized view, the refreshing process should include updating both the materialized view and its aggregates in one ACID transaction. Otherwise any user query that interleaves with the refreshing process and performs drilldowns/roll-ups on the materialized view and its aggregates might perceive an inconsistency. As mentioned earlier, application dependent correctness criteria (e.g., the significance of inconsistency) might need to be taken into account. For off-line refreshing, all applications should be stopped when the refreshing is performed.

View maintenance techniques fall into two categories: re-computation from scratch and incremental view maintenance. The former technique answers the queries on which the materialized views are defined and then inserts the results into the view tables. To apply this view maintenance technique on a database system, one needs the definition of materialized views and the content of the base tables after the updates [94]. The latter technique uses the heuristic of inertia (only part of the view changes in response to changes in base relations). Thus, one can compute only the changes in the materialized views and combine them with the old materialized views. To apply this technique, more information is usually required in addition to the definition of the materialized views. For example, the original contents of the materialized views are necessary. Depending on the types of updates to the base tables, some other information such as the updates to the base tables is also required [67].

2.1.1 View Maintenance by Re-computation

In most cases it is less cost-effective to re-compute materialized views from scratch than to maintain views incrementally, especially if the source data is less volatile in nature. However there are some cases when the re-computation approach is cheaper. For example, if an entire base table is deleted, it may be cheaper to re-compute a view that depends on the deleted table (if the new view will quickly evaluate to an empty table) than to compute the changes to the view. Generally the re-computation approach is preferable when changes to base tables are relatively large compared to the original contents of the base tables [67]. For legacy systems or flat files or for the initial materialization of views, this re-computation approach might be the only choice.

The view maintenance using the re-computation approach can be optimized using multi-query optimization techniques. The cost to maintain a view using re-computation includes the cost of answering the query on which the view is defined and the cost of inserting the results into the view table. The writing cost is fixed for a given instance of the database and the changes to base tables. Then, the problem of maintaining a set of views is reduced to the problem of answering a set of queries. W. Lehner et al. [94] propose an algorithm to refresh a set of materialized views using multi-query optimization techniques. Similar work can be found in [106, 144]

2.1.2 Incremental View Maintenance

There is an extensive body of work on incremental view maintenance. A. Gupta et al. [67] classify this problem based on the following dimensions:

- Information Dimension: what is the amount of information available for maintenance?
- Modification Dimension: what modifications to the base tables can be handled by the view maintenance algorithms?
- Language Dimension: what language is used to specify the definition of the materialized views?
- Instance Dimension: does the view maintenance algorithm work for all instances or only for some instances of the database?

When all base tables and materialized views are available during the maintenance process, A. Gupta et al. [68] propose a counting algorithm for maintaining non-recursive views. The views may or may not have duplicates and they may be defined using UNION, negation and aggregation. The basic idea in the counting algorithm is to keep a count of the number of derivations for each view tuple as extra information in the view. Other approaches using counting algorithms include [124, 16, 119]. R. Paige [109] proposes an algebraic differencing algorithm. X. Qian et al. [114] apply it in view maintenance. They differentiate algebra expressions to derive the relational expressions that compute the changes to an SPJ view without doing redundant computation. T. Griffin et al. [59] extend the algebraic differencing algorithm to a multi-set algebra with aggregations and multi-set difference. S. Ceri et al. [20] propose an algorithm that derives productions to maintain selected SQL views - those without duplicates, aggregation, and negation, and those where the view attributes functionally determine the key of the base relation that is updated.

Most of the work on maintaining recursive views has been in the context of Datalog. A. Gupta et al. [68] propose the DRed algorithm that applies to Datalog or SQL views, including views defined using recursion, UNION, and stratified negation and aggregation. However, SQL views with duplicate semantics can not be maintained using this algorithm. J. V. Harrison et al. [76] propose a propagation/filtration algorithm that is similar to the DRed algorithm [68] except that it propagates the changes made to the base relations on a relation-by-relation base. V. Kuchenhoff [91] propose an algorithm that derives rules to compute the difference between consecutive database states for a stratified recursive program. T. Urpi et al. [139] propose an algorithm that derives transition rules for stratified Datalog views showing how each modification to a relation translates into a modification to each derived relation, using existentially quantified subexpressions in Datalog rules. Other approaches relating to incremental maintenance of recursive views include [44, 43, 96, 142]

It is possible that only a subset of the underlying base tables involved in a view is available to maintain this view. This is referred to as view maintenance with partial information. Because a view may be maintained with partial information for some kinds of modifications, some of the previous papers focus first on checking whether the view can be maintained and then on how the view can be maintained. There is a lot of work on optimizing view maintenance by determining when a modification leaves a view unchanged [16, 15, 47, 99]. This is known as the problem of "irrelevant updates." All these papers provide checks to determine whether a particular modification will be irrelevant. No base relation information is needed for this check. In the presence of key and foreign key constraints, views that can be maintained using only the materialized views are called self-maintainable views [66]. Several results on the self-maintainability of SPJ and outer-join views for insertions, deletions and updates are presented in [66]. For insertions and deletions only, a database-instance specific self-maintenance algorithm for SPJ views is discussed in [136]. There are also cases where all materialized views and base relations except the modified relations are available for view maintenance. Such an example is a chronicle view. Techniques to specify and maintain such views efficiently are presented in [79].

2.2 Answering Queries Using Views

The problem of answering queries using views [73] (also known as rewriting queries using views) has received significant attention because of its relevance to a wide variety of data management problems: query optimization, maintenance of physical data independence, data integration and data warehouse design. Informally speaking, the problem is the following. Given a query Q over a database schema, and a set of view definitions $V_1, \ldots, V_n$ over the same schema, is it possible to answer the query Q using only the views $V_1, \ldots, V_n$? Alternatively, what is the maximal set of tuples in the answer of Q that can be obtained from the views? If both the views and the database relations are accessible, what is the cheapest query execution plan for answering Q?

A lot of approaches have been proposed to incorporate materialized views into query optimization. The focus of these algorithms is to judiciously decide when to use views to answer a query. The output of the algorithm is an execution plan for the query. The approaches differ depending on which phase of query optimization was modified to consider materialized views. O. G. Tsatalos et al. [137] describe the storage of the data using GMAPs (generalized multi-level access paths), expressed over the conceptual model of the database. S. Chaudhuri et al. [24] analyze the problem of optimizing queries using materialized views and provide a comprehensive solution. They consider select-project-join queries with bag semantics which may include arithmetic comparison predicates. J.

Goldstein et al. [54] describe an algorithm for rewriting queries using views This algorithm is implemented in the transformational optimizer of Microsoft SQL Server. M. Zaharioudakis et al. [151] describe how view rewriting is incorporated into the query rewriting phase of the IBM DB2 optimizer. This algorithm operates on the Query Graph Model (QGM) representation of a query [72], which decomposes the query into multiple QGM boxes, each corresponding to a select-project-join or grouping/aggregation block. A. Deutsch et al. [42] use a transformational approach to uniformly incorporate the use of materialized views, specialized indexes and semantic integrity constraints. All of these are represented as constraints. L. Popa et al. [111] describe an implementation of this framework and experiments that prove its feasibility. R. G. Bello et al. [14] describe a limited use of transformation rules to incorporate view rewriting algorithms into the Oracle 8i DBMS. Several papers consider the problem of answering queries using views in the presence of grouping and aggregation: a set of transformations in the query rewriting phase are considered in [65]. A semantic approach is suggested in [126]. The authors present conditions for a view to be usable for answering a query in the presence of grouping and aggregation, and a rewriting algorithm. The work described in [151] extends the results on grouping and aggregation to multi-block queries and to multi-dimensional aggregation functions such as cube, roll-up, and grouping sets [126]. Several approaches [36, 60, 61] consider the formal aspects of the problem of answering queries using views in the presence of grouping and aggregation. They present cases in which a rewriting algorithm is complete in the sense that it finds a rewriting if one exists.

The above work focuses on extending query optimizers to accommodate the use of views. They were designed to handle cases where the number of views is relatively small (i.e., comparable to the size of the database schema), and cases where an equivalent rewriting of the query is required. In contrast, in the context of data integration, a large number of views are considered since each data source is being described by one or more views. In addition, the view definitions contain many complex predicates, whose goal is to express fine-grained distinctions between the contents of different data sources. The algorithms of answering queries using views that were developed specifically for data integration are reviewed in the following paragraphs. These algorithms include the bucket algorithm developed in the Information Manifold system [98, 57], the inverse rules algorithm [113, 46] which was implemented in the Info Master System [45], and the MiniCon algorithm [112]. It should be noted that unlike the algorithms of answering queries using views for the query optimization problem, the output of these algorithms is not a query execution plan, but rather a query referring to the view relations. The goal of the bucket algorithm is to reformulate a user query that is posed on a mediated (virtual) schema into a query that refers directly to the available data sources. Both the queries and the sources are described by conjunctive queries that may include atoms of arithmetic comparison predicates. The inverse-rules algorithm is also developed in the context of a data integration system. The key idea underlying this algorithm is to construct a set of rules that invert the view definitions, i.e., rules that show how to compute tuples for the database relations from tuples of the views. The MiniCon algorithm addresses the limitations of the previous algorithms. The key idea underlying this algorithm is a change of perspective: instead of building rewritings by combining rewritings for each of the query subgoals or the database relations, the authors consider how each of the variables in the query can interact with the available views.

Some theoretical work on answering queries using views is shown below. A. Y. Levy et al. [96] show that when a query does not contain a comparison predicate and has n subgoals, there exists an equivalent conjunctive rewriting of the query using a view only if there is a rewriting with at most n subgoals. It is also shown that the problem of finding a rewriting is NP-hard for two independent reasons: (1) the number of possible ways to map a single view into the query, and (2) the number of ways to combine the mappings of different views into the query. C. Chekuri et al. [29] exploit the close connection between the containment and rewriting problems, and show several polynomial-time cases of the rewriting problems, corresponding to analogous cases for the problem of query containment. The question of finding all certain answers is considered in detail in [2, 57]. In their analysis they distinguish the case of the open-world assumption from that of the closed-world assumption. The open-world assumption is especially appropriate in data integration applications, where the views that describe sources may be incomplete. Under the open-world assumption, [2] shows that in many practical cases, finding all certain answers can be done in polynomial time. However, the problem becomes co-NP-hard (in the size of the view extensions) as soon as union in the language is allowed for defining views, or predicate $\neq$ is allowed in the query language. Under the closed-world assumption the situation is even worse. Even when both the views and the queries are defined by conjunctive queries without comparison predicates, the problem of finding all certain answers is already co-NP-hard. The closed-world assumption is appropriate for the context of query optimization and maintaining physical data independence, where views have actually been computed from existing database relations. The hardness of finding all certain answers provides an interesting perspective on formalisms for data integration. The result entails that when views are used to describe the contents of data sources, even if only conjunctive queries are used to describe the sources, the complexity of finding all the answers to a query from a set of sources is co-NP-hard. In contrast, using a formalism in which the relations of the mediated schema are described by views over the source relations, the complexity of finding all the answers is always polynomial. Hence, this result hints that the former formalism has a greater expressive power than formalism for data integration. It is also interesting to note that the work in [57] also considers the case where the views may either be incomplete, complete, or contain tuples that do not satisfy the view definition (referred to as incorrect tuples). It is shown there that without comparison predicates in the views or the query, when either all the views are complete or all of them may contain incorrect tuples, finding all certain answers can be done in polynomial time in the size of the view extensions. In other cases, the problem is co-NP-hard. The work in

[103] considers the query rewriting problem in cases where bounds exists on the soundness and/or completeness of the views. Finally, [105] considers the problem of relative query containment, i.e., whether the set of certain answers of a query Q_1 is always contained in the set of certain answers of a query Q_2. The paper shows that for the conjunctive queries and views with no comparison predicates, the problem is $\prod_2^p$-complete, and the problem is still decidable in the presence of access pattern limitations.

CHAPTER 3

THE VIEW SELECTION PROBLEM

In this chapter, we review previous work done on the view selection problem formally defined in Chapter 1. As stated in Chapter 1, there are a lot of variants for this problem. We start by providing a categorization of these variants based on a number of dimensions. Then, we discuss previous work. Finally, we identify some new variants of the problem which have not been addressed previously. Some of them are studied in the subsequent chapters.

The view selection problem can be the classified according to the following dimensions.

1. The type of access structures to select. There are four cases to consider:
   - Indexes on base tables.
   - Materialized views.
   - Indexes on base tables and materialized views.
   - Materialized views and indexes on base tables and materialized views.
2. The time of creation and deletion of physical access structures. Two cases should be considered:
   - Static case: all selected physical access structures are created at once during the execution of the workload, usually at the beginning of the execution. They may be dropped after the execution of statements in the workload. This is referred to as static view selection problem.
   - Dynamic case: each physical access structure is created or dropped at a specific point during the execution of statements in the workload. This is referred to as dynamic view selection problem.
3. The constraint. Different constraints have been considered for the view selection problem. The two most commonly considered constraints are space constraints and time constraints:
   - Space Constraint. The disk space to accommodate the selected physical access structures is often a constraint for the view selection problem.

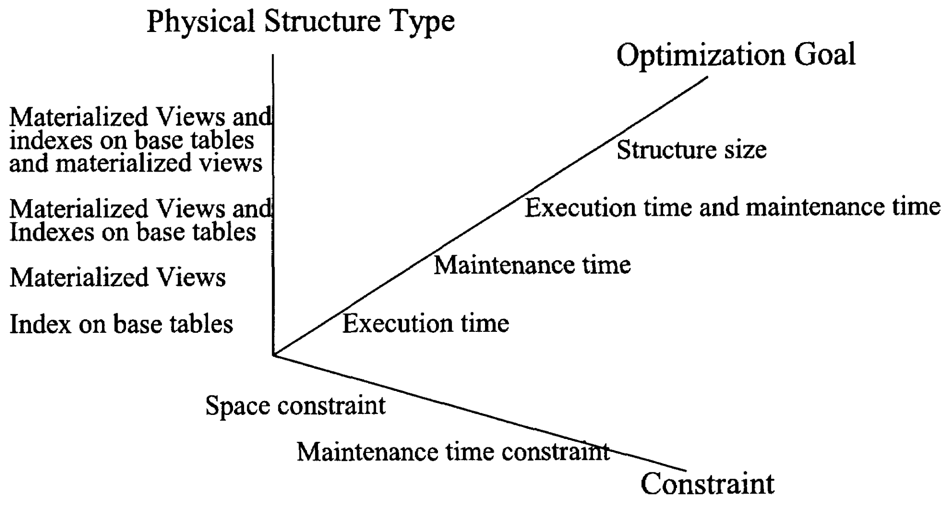

Physical Structure Type
Optimization Goal
Materialized Views and indexes on base tables and materialized views
Materialized Views and Indexes on base tables
Materialized Views
Index on base tables
Structure size
Execution time and maintenance time
Maintenance time
Execution time
Space constraint
Maintenance time constraint
Constraint

Figure 3.1 Space of view selection problems.

- Time Constraints. The time for maintaining the selected physical structures is usually constrained by the time window available for maintenance. For the updates, the system can adopt incremental or re-computation strategies.

4. The optimization goal. The most common optimization goal for the view selection problem is the minimization of the weighted execution time of the workload. Some other optimization goals have also been considered. The most commonly considered ones are the following:

   - Minimizing the total execution time of a workload.
   - Minimizing the total maintenance time of the selected physical access structures.
   - Minimizing a combination of the execution time of a workload and the maintenance time of the selected physical access structures.
   - Minimizing the total space occupied by the selected physical structures.

Most of the previous work addresses the static view selection problem. However, in some cases, considering the dynamic nature of the problem can lead to better solutions. This chapter first describes previous work on the static view selection problem. Then it reviews some previous work which addresses dynamic view selection problems.

The research space of the view selection problem is shown in Figure 3.1. Three dimensions of the view selection problem are considered in this figure, namely, the physical access structure type dimension, the constraint dimension and the optimization goal dimension.

Each point defined by the three dimensions refers to a possible instance of the view selection problem.

There are also some other possible dimensions which make the categorization of the view selection problem more complex: (1) The schema of the database (multi-dimensional schemas, star-schemas, normalized relational schemas, etc.); (2) The type of the workload (workload consisting only of queries, consisting of only of updates, or consisting of both); (3) The language used to express queries (select-project-join queries, general SQL queries with grouping aggregation, OLAP query languages, etc.). Figure 3.1 shows only three of the possible dimensions for simplicity. In this chapter, all the dimensions described above are considered in characterizing different view selection problems.

3.1 Static View Selection Problems

For the static view selection problem, the configuration of physical access structures is decided before the execution of the workload and never changes afterwards.

3.1.1 Index Selection

The problem of index selection has been studied long before decision support queries and data warehousing because of its significant effect on the performance of database systems. New index structures are still being proposed by the database community [11]. M. P. Consens et al. [38] show that a configuration with only single column indexes can outperform configurations with both indexes and materialized views recommended by most commercial DBMSs. This result shows not only the large improvement space for the current recommendation tools, but also the significant effect of indexes on the performance of a database.

There are three kinds of approaches for recommending indexes. The first one takes the semantic information such as key constraints, referential integrity constraints and rudimentary statistics (e.g., "small" versus "big" tables) and produces a configuration design.

This is the basic approach presented in textbooks [110]. This rule-based index selection approach does not take into account the workload, complex constraints, and the optimization goal. It defines indexes based on the schema definition and can reach in general a reasonable performance. However, the outcome of this approach may not be feasible because it violates space constraints. It may also perform poorly because it ignores the workload information.

The second approach for recommending indexes adopts an expert-system-like method where the knowledge of "good" design is coded in the form of rules used to create a configuration design. Such an approach can take into account workload information but does not exploit the system optimizer [25]. Without the ability to access the system optimizer during the recommendation of indexes, this approach does not estimate the benefit of indexes which is important in selecting indexes. Thus, it may select indexes that will never be considered by the system optimizer during the execution of the workload.

The third approach for recommending indexes uses the system optimizer's cost estimation to compare the "goodness" of the alternative candidate configurations of indexes [50, 25, 140]. The advantage of this approach is that the cost estimation for queries in a given workload under different configurations is made by the DBMS engine itself which is also used to find the optimal evaluation plan for each query. This makes the estimation relatively accurate and the corresponding recommended configurations more suitable for the underlying database and workload of queries. The disadvantage of this approach is that the system optimizer has to be called multiple times to estimate query evaluation costs under different configurations, which is expensive when there are many queries in the workload or when queries in the workload are complex or when there are many candidate indexes to consider.

Most current commercial DBMSs provide index recommendation tools based on the last approach. S. Chaudhuri et al. [25] propose a tool which can recommend indexes for Microsoft SQL Server. They consider the index selection problem where the number of indexes is constrained to be less than or equal to k. To reduce the number of calls to the system optimizer, they adopt several heuristics. The first heuristic deals with the selection of candidate indexes to be considered by the selection tool. It restricts the candidate indexes to those that belong to the best configurations of at least one query in the workload. Another heuristic deals with the cost estimation of queries in the workload. Based on the concept of atomic configuration [50], this heuristic does not need to call the system optimizer to estimate the cost for each query under each configuration. Instead, it derives the costs for some configurations from the cost estimation of some other configurations. The last heuristic deals with the generation of multi-column indexes. They adopt an iterative approach for taking into account multi-column indexes of increasing width. In the first iteration, they only consider single-column indexes. Based on the single-column indexes picked in the first iteration, they select a set of *admissible* two-column indexes. This set of two-column indexes, along with the "winning" single-column indexes, becomes the input for the second iteration. Along with the above heuristics, they propose a two-step $Greedy(m, k)$ algorithm. In the first step, the m best indexes are exhaustively selected from the candidate indexes. In the second step, the last $k - m$ indexes are selected in addition to the first m selected indexes in a greedy manner by selecting each time the locally best index. By adjusting the value of m from 0 to k, the database administrator can control the balance between the performance of this index selection tool and the quality of the selected indexes.

G. Valentin et al. [140] propose a tool which can recommend indexes for IBM DB2. Instead of constraining the number of selected indexes as in [25], they constrain the space available for accommodating selected indexes. In order to compute the candidate indexes, they propose an approach called SAEFIS enumeration algorithm which can provide wider indexes than those of [25]. To recommend indexes, they also design a two-step algorithm. In the first step, the candidate indexes are sorted in the order of their benefit/size ratio to the workload estimated by the optimizer. Then, they select indexes with the biggest benefit/size ratio, one at a time, until the space constraint is reached. In the second step, they adopt a random try-and-replace strategy to improve the quality of selected indexes until a given time constraint is reached. Unlike the work of [50, 25] where the index selection tools are external to the DBMS engine, they use the system optimizer to estimate the cost of the queries in the workload; this tool is internal to the DBMS engine itself. With this strategy, not only do they leverage the existing optimizer and thus avoid maintaining two distinct optimizers, but they also reduce the number of calls to the system optimizer. For example, if there is only one query in the workload, this tool needs only one optimizer invocation while the previous one [25] needs multiple invocations depending on how many indexes there are in the candidate index set.

3.1.2 Selection of Materialized Views and Indexes

Space Constraint. As mentioned in Chapter 1, the use of materialized views can be considered as an extension to that of indexes. Although indexes have been studied since the beginning of relational databases, view materialization become popular mainly because of data warehousing and OLAP in the last decade. The problem of selecting views for materialization ia initially studied in the area of multi-dimensional databases (MDDBs) with OLAP queries.

V. Harinarayan et al. [75] address the view selection problem for MDDB where all queries in the workload and all candidate views are data cubes [58]. They consider the view selection problem with the space constraints and an optimization goal of minimizing the total execution time of a given workload. As pioneers in this area, they made the following contributions:

(a) They propose a lattice framework to describe the hierarchies of MDDB schemas and the dependencies between data cubes.

(b) They prove the NP-hardness of the view selection problem and propose a greedy heuristic with a guaranteed lower bound on the optimality.

The lattice is defined as a DAG with the vertices to be all cubes and edges showing whether one vertex cube can be used to answer the other one. It models not only the hierarchies of each dimension, but also the inter dependence of all vertices, which represents all the queries and candidate views. With the lattice framework, the authors provide the following algorithm for selecting views:

This algorithm is for the view selection problem that is constrained by the number of selected views. The time complexity for this algorithm is kn^2 where k is the number of views to be selected and n is the total number of data cubes in the lattice. [75] proves that this greedy algorithm finds a solution with a guaranteed lower bound of 63% of the optimal solution. For a space constraint, the lower bound of the algorithm is reduced to $0.63 - f$ where f is the ratio of the biggest views in size to that of the total size of the search space.

For this greedy algorithm proposed by Haranrayan and Ullman, H. Karloff and M. Mihail [87] prove that the view selection problem is inapproximable. Hence, studies of this problem should focus on special cases of practical significance, such as hyper cubes and on experimental comparison of heuristics.

Jingni Li et al. [101] consider the same view selection problem using the same lattice framework as [75]. Instead of using a greedy algorithm, they present a transformation

Algorithm 1 Harinarayan et al. Greedy Algorithm

```
Input: A lattice of queries and views
Output: A set of selected views
1: $S = \{Top\ View\}$ (of the lattice)
2: for $i = 1$ to $k$ do
3:     select the view $v$ from the remaining views which has the biggest benefit with respect
       to the set of selected views
4:     $S = S \cup \{v\}$
5: end for
6: Result $S$ is the greedy selection of views
```

of this problem to an Integer Programming (IP) problem and then solve it with Integer Programming techniques. Their experimental results show that this algorithm produces a better view set than [75].

A. Shukla et al. [125] propose another heuristic called pick by size (PBS). It picks views in increasing size until the space limit is reached. If the queries are not uniformly distributed, the sorting key is the size of the view divided by the number of queries that are best evaluated by it. This algorithm selects the same set of views as the one above [75] in time complexity of $O(n \log n)$ given that the lattice is size restricted, i.e., the size of each view is at least $k + 1$ times larger than the size of its largest child (where k is the number of its children). For non-uniform query distributions, the weighted view sizes have to follow the same rule. If this rule does not apply, the method does not give a quality guarantee for the picked selection.

H. Gupta [70] extends the lattice framework to represent both views and indexes on views. With this extension, this paper proposes algorithms to select not only materialized views, but also indexes on materialized views. Two algorithms, r-greedy algorithm and Inner-Level greedy algorithm are proposed. The former algorithm increases the performance guarantee when increasing the time complexity of the algorithm (increase r). For example, when r increases from 1 to 2, 3 and 4, the performance guarantee increases from 0 to 0.39, 0.49 and 0.53, respectively. The increase diminishes exponentially when r increases until the performance guarantee is 0.63. Because there is a knee for the increase of performance at $r = 4$, usually 4-greedy algorithm is selected. Unlike the r-greedy algorithm which presents a different optimality guarantee for different values of r, the Inner-Level greedy algorithm presents a fixed performance guarantee of 0.49.

While these algorithms have a polynomial time complexity in the number of candidate views, it does not scale well with the complexity of the database. This is because the number of candidate views is exponential in the number of attributes in dimension tables. The condition is even worse when both views and indexes are considered for selection. M.

Baralis et al. [13] extend the lattice framework with a formal definition of it in the context of the view selection problem. More importantly, they provide approaches to reduce the search space of candidate views to views that are related to the queries in the workload before the view selection algorithm. Qiu and Ling [122] further investigate this issue. They propose two methods, called the functional dependency filter and the size filter, to eliminate redundant or insignificant views.

Considering the scaling problem of the lattice framework, Z. Chen [31] et al. propose a totally new heuristic for the view selection problem which does not construct the lattice at all, but selects views by merging queries in the workload in a bottom-up way. This algorithm scales quite well on the number of group-by attributes. Empirical results show that it generates results of good quality.

H. Gupta [69] generalizes the view selection problem from multi-dimensional data cubes to general relational databases. Instead of constructing the search space of views using the lattice framework, he assumes the existence of AND/OR graphs of candidate views (or indexes) as input for the view selection problem. For the special cases of AND graphs and OR graphs, he proposes greedy algorithms with polynomial time complexity. He also proves the existence of a lower bound for the optimality of these algorithms. For the general AND/OR graph, he proposes an AO-greedy algorithm and a multi-level greedy algorithm with the following theoretical results:

For a query-view graph without updates and a quantity S as the space constraint, the AO-greedy algorithm produces a solution M that uses at most $2S$ units of space. The absolute benefit of M is at least $(1 - 1/e)$ times the optimal benefit achievable using as much space as that used by M.

For a query-view graph G and a quantity S as the space constraint, the r-level greedy algorithm delivers a solution M that uses at most $2S$ units of space. The benefit of M is at least $1 - (\frac{1}{e})^{0.63^r}$ of the optimal benefit achievable using as much space as that used by M assuming that no view occupies more than S units of space. The r-level greedy algorithm takes $O((kn)^{2r})$ time excluding the time taken at the final level. Here k is the maximum number of views that can fit in S units of space.

The view selection problem with space constraints has been supported by most modern commercial database systems [8, 154]. Microsoft SQL Server [8] utilizes a pruning technique to select a set of candidate views that is syntactically relevant to the workload of queries. It first finds the table set upon which the potentially valuable views are defined. Then it exploits the query optimizer to find the most valuable view configuration for each single query. Lastly, it gets more views by merging existing views. Once the candidate view set is decided upon, it utilizes the $Greedy(m, k)$ algorithm similar to that used in [25] to find the set of views to materialize. IBM DB2 [154] uses a method that generates a candidate view set. It employs a multi-query optimization technique which generates common sub-expressions among queries. Then it adds to the view set indexes on tables and on the candidate views. Based on a knapsack-like algorithm it selects the views and indexes that bring most benefit per unit of space under the constraint that indexes on views bring no benefit at all if the corresponding view is not selected. Further, it utilizes a swap algorithm to tune the selection of views/indexes for a better performance. Lastly it filters out those views/indexes that are never used during the execution of the workload because a better plan can be obtained based on other selected views/indexes.

Most of the current approaches to the view selection problems choose their view set in a bottom-up way. This means they start with an empty view set and add into it some selected views or indexes based on some greedy heuristic until the space constraint is reached. To handle large workloads and multiple kinds of physical structures, these techniques have become increasingly complex: they exhibit many special cases, shortcuts, and heuristics that make it very difficult to analyze and extract properties. N. Bruno et al. [17] propose a totally different framework for the space constrained view selection problem. This new framework first finds an optimal physical structure set for the view selection problem without space constraint. Then, it relaxes the space of the optimal set by deleting or replacing some structures in the set until the space constraint is satisfied. The experimental results show that this approach could result in comparable (and, in many cases, considerably better) recommendations than state-of-the-art commercial alternatives. More importantly, with this relaxation-based approach, a curve of estimated cost of the workload to the size of selected physical structures is easily obtained as a by-product in one execution of the algorithm. This curve is quite important for the system administrator to decide whether to increase the disk capacity for the selected view to achieve a significant improvement on the performance of the workload.

Update Constraint Though the papers mentioned above have offered significant insights into the nature of the view selection problem, the constraints considered therein make the results less applicable in practice because disks are very cheap in real life. In practice, the real constraining factor that prevents materializing everything at the warehouse is the maintenance time incurred in keeping the materialized views up to date. Usually, changes to the source data are queued and propagated periodically to the warehouse views in a large batch update transaction. The update transaction is usually done overnight so that the warehouse is available for querying and analysis during the day time. Hence there is a constraint on the time that can be allotted to the maintenance of materialized views.

How materialized views are maintained is also important for the view selection problem under a maintenance constraint. Different maintenance techniques suggest different ways for tackling the view selection problem. There is a lot of work on the view maintenance problem [94, 51, 144]. Incremental update of views is discussed in [51]. Complete refreshing using multi-query optimization techniques is discussed in [94, 144], which itself is an instance of the view selection problem.

The view selection problem under a maintenance constraint is more difficult than the view selection problem with a space constraint. In the case of a space constraint, as the benefit of a view never increases with the materialization of other views, the benefit per unit space of a non-selected view always decreases monotonically with the selection of other views. This is defined formally as the monotonicity property of the benefit function in [69].

H. Gupta et al. consider the view selection problem under a maintenance constraint [71]. Because of the difficulty, only the case of OR view graphs is considered. This paper proposes an algorithm (Inverted Tree Greedy Algorithm) for an OR view graph which can give a guarantee of 63% of the optimal solution with up to two times of the maintenance cost constraint. Although the algorithm has an exponential worst case time complexity for general OR view graphs, it has a polynomial time complexity for view graphs of the format of balanced binary trees. It also runs efficiently when the view graph is sparse. The result of this algorithm is compared to that of an $A*$ algorithm, which can give an optimal solution for general AND/OR view graphs. The latter algorithm is much slower than the former one for OR view graphs and it requires exponential space.

Because the essential difficulty of the view selection problem under a maintenance time constraint, most approaches consider a variation of the problem which considers maintenance statements into the workload weighted by the relative frequency of the update statements [154, 9].

K. A. Ross et al. [118] consider the view selection problem with view maintenance statements combined into the workload. Instead of using the view maintenance time as a constraint, they include it into the optimization goal. Thus, the problem involves selecting a set of additional views to materialize during the maintenance of a given set of views in order to minimize the total maintenance time. H. Mistry et al. [106] consider a similar problem. However, instead of materializing all selected views permanently, they choose transient materialized views which are materialized only during the maintenance of the given views.

Minimizing the Total Size of Views R. Chirkova et al. [33] consider the view selection problem in a distributed database system. The bottleneck in this case is the transmission time of data through the network. This motivates the following problem: given a set of queries (a query workload) and a fixed database, define and compute a set of intermediate results (views) such that these views can be used to compute the answer to each query in the workload. In addition, the total size of selected views should be minimized on the given database. This is also an instance of the view selection problem where the constraint is the answerability of the queries using exclusively the selected views and the optimization goal is the minimization of the total size of selected views. The authors prove that the problem is NP-hard and they propose two strategies to tackle it. The first strategy is based on an exhaustive algorithm. Using pruning techniques, this algorithm can get a view set of good quality with a reasonable performance. The second strategy uses both bottom-up and top-down heuristics.

3.2 Dynamic Management of Physical Structures

Materialized views represent a set of redundant entities in a data warehouse that are used to accelerate On-Line Analytical Processing (OLAP) queries. The problem is to select an appropriate set of views such that the total execution time of the workload can be minimized. The amount of redundancy added is controlled by the data warehouse administrator who specifies the space to be allocated for the materialized data. Given this space constraint and, if available, a description of the workload, these algorithms suggest a set of views to materialize for improving the query performance.

This static selection of views, however, contradicts the dynamic nature of decision support analysis applications. In particular for ad-hoc queries, where the expert user is looking for interesting trends in the data repository, the query pattern is difficult to predict. In addition, as the data and the trends are changing over time, a static selection of views might very quickly become outdated. This means that the administrators should monitor the query pattern and periodically re-calibrate the materialized views by rerunning these algorithms. This task for a large data warehouse where many users with different profiles submit their queries is rather complicated and time consuming.

For the static view selection problem, the input of a workload is always a set of queries which may also include update statements. But for a dynamic view selection problem, the input workload can take several different formats. This affects the problem significantly. For example, if the input is an unknown sequence of queries, then the view selection problem is a cache problem [41, 90]. If the input is a specific sequence of queries, then the problem is a configuration transformation problem [9]. If the workload is a set of queries, then the problem is a space constrained multi-query optimization problem.

If the input is unknown, there is no prediction on what structures will be used by future queries. There are two critical problems here. The first is how to answer queries using cached structures, either partially or completely. The other is which structures to cache for the use of future queries. P. M. Deshpande et al. consider the first problem by introducing "chunks" [41] in a multi-dimensional database schema. In that paper, they propose a chunk based scheme that addresses these problems by dividing the multi-dimensional query space uniformly into chunks and caching these chunks. The results of a query are contained in an integral number of chunks, which form a "bounding envelop" about the query result. (If the query boundaries do not match the chunk boundaries, the chunks in this bounding envelop may contain tuples that are not in the answer of the query.) Since chunks are at a lower granularity than query level caching, they can be reused to compute the partial result of an incoming query. By using a fast mapping between query constants and chunk numbers, the set of chunks needed for completely answering a query can be determined. The authors partition this set of chunks into two disjoint sets such that the chunks in one partition can be found in the cache while the others are computed using the back end relational engine. A consequence of partitioning the chunks is the intended query decomposition. Since query results are an integral number of chunks, unlike query level caching, the replacement policy can take advantage of the "hotness" of the chunks.

Yannis Kotidis et al. [90] study a dynamic view selection problem for data warehousing. They present "DynaMat", a system that dynamically materializes information at multiple levels of granularity in order to match the demand (workload). Both the maintenance constraint and the space constraint are taken into account in their approaches. DynaMat unifies the view selection and the view maintenance problems under a single framework using a novel "goodness" measure for the materialized views. It constantly monitors incoming queries and materializes the best set of views subject to the space constraint. During updates, DynaMat reconciles the current materialized view selection and refreshes the most beneficial subset of it within a given maintenance window. The dynamic view management of Dynamat is separated into two phases: the online phase and update phase. During the on-line phase, the goal of the system is to answer as many queries as possible from the pool, because most of them will be answered much faster from caches than from the base relations. At the same time, DynaMat will quickly adapt to new query patterns and efficiently utilize the system resources. The second phase of DynaMat is the update phase, during which updates received from the data sources get stored in the warehouse and materialized results in the Pool get refreshed. With a dynamic management of selected physical structures, Dynamat can not only outperform the optimal solution of the static view management, but also quickly adapts to changes in the workload or in the constraint or in both.

S. Agrawal et al. [9] formally define the problem of recommending physical structures for workloads that are sequences of statements and suggest a dynamic management of the selected physical structures. By exploring the information in the order of the statements, the system can create and drop physical structures at different points of the execution of the sequence so that the total benefit of these structures is maximized. The paper first introduces an exhaustive approach of an exponential time complexity to the number of candidate physical structures. It also provides a cost-based pruning technique which can reduce the search space significantly while maintaining the optimality of the solution. By introducing the concept of disjoint sequence, it suggests a heuristic approach that applies splitting and merging techniques. This approach first finds an optimal solution to each disjoint sequence of statements and then merges those solutions to get the solution for the original sequence of statements. After merging, if the configurations satisfy the space constraint at each point of the sequence, the result is optimal. Otherwise, another heuristic is applied to the solution to remove or replace physical structures until the sequence satisfies the space constraint at each point. The resulting solution may be sub optimal. Although this paper focuses on indexes, the results hold for materialized views and indexes on materialized views.

3.3 Strategies for the View Selection Problem

As described above, most of the previous work on the view selection problem focuses on greedy algorithms. The advantage of this strategy is that it usually outputs a result of good quality. But because the complexity of the problem itself [87], there is no optimal lower bound guarantee for the algorithms proposed in the general view selection problem. Most of the algorithms with greedy strategy have poor performance. Although this is not a problem for many instances of the static view selection problem, it is a critical issue for some dynamic instances of the view selection problem where the time window for the selection of views is much more restrictive than that of static instances. For the static view selection problem, it is also important to improve the performance of the algorithms for view selection when the size of the instance is large. For this reason some faster strategies based on restrictive genetic algorithms and randomized algorithms are proposed in the literature.

The genetic algorithm (GA), first proposed by Holland in 1975, is an approach that mimics biological processes for evolving optimal or near optimal solutions to problems.

Horng et al. [82] propose a genetic algorithm to select the appropriate set of views for minimizing the query and view maintenance cost. The input to the problem is an AND/OR DAG. The genetic algorithm might generate some unfeasible solutions such as a set of views that exceeds the given space constraint. In this paper, a penalty function is used to punish the fitness of these unfeasible solutions. W. Y. Lin and I. C. Kuo extend the work of [82] by considering as input to the problem a lattice of data cubes instead of an AND/OR DAG of views. To deal with unfeasible solutions, they use a greedy repair method to correct them. Among the papers on view selection problems that use genetic algorithms, [152] recommends an optimal set of processing plans for multiple queries instead of recommending a set of views and indexes.

As genetic algorithms, randomized algorithms are another strategy for optimization problems which do not require the exhaustive check of the states in the search space. Randomized algorithms have long been used in query optimizations [128, 78, 53]. In traditional applications, however, the demand for query optimization using randomized search is limited, since queries that involve more than 10 joins are seldom used. In contrast, in OLAP applications it is common to have $10-15$ dimensions, resulting to $2^{10}-2^{15}$ views. Therefore, a randomized search for sub-optimal view sets becomes the only practical optimization method [84].

The search space of an optimization problem can be thought of as an undirected graph, where nodes correspond to candidate solutions. Each node (also called a state) has a cost assigned to it and the aim of the optimization process is to find the one with the minimum cost. For the view selection problem, each state corresponds to a set of views which meet the space or time constraint. An edge of the graph defines a move (or transition) from one state to another after applying a simple transformation. The move is called downhill if the destination state has a lower cost than the starting one. In the opposite case, it is called uphill. A state from which no moves are downhill is called local minimum. The local minimum with the minimum cost is called the global minimum. The graph should be connected, i.e., there should be at least one path of transitions between any two states. Randomized algorithms start from a random initial state and apply sequences of transformations trying to detect the global minimum [84].

P. Kalnis et al. [84] propose four randomized algorithms in view selection problems considering either space constraint or maintenance constraint. Iterative improvement (II) starts from a random initial state (called seed), performs a random series of moves and accepts only downhill ones, until a local minimum is detected. This process is repeated until a time limit is reached, each time with a different seed. Simulated annealing (SA) follows a procedure similar to II; however, it also accepts uphill moves with some probability. This probability is gradually decreased with time and finally the algorithm accepts only downhill moves leading to a good local minimum. Random sampling (RA) selects states randomly and returns the state with the minimum cost from those visited given a limited time limit. Two-phase optimization (2PO) is a hybrid randomized algorithm. It combines II and SA and can be more efficient than both algorithms. This method uses II to quickly locate an area where many local minimums exist, and then applies SA with a small initial temperature to search for the global minimum in this area. P. Kalnis et al. [84] compare the systematic algorithms [75, 125] and randomized algorithms under both space constraints and maintenance time constraint. Among the four randomized algorithms, the 2PO algorithm always performs better than other random algorithms. For the view selection problems under space constraints, the systematic algorithms get better results than randomized algorithms, but also require much more time. For view selection problem under maintenance constraint, both quality and performance of the randomized algorithm outperform that of systematic algorithms.

Michael Lawrence et al. [104] consider using randomized algorithms for a dynamic view selection problem. The distribution of queries in the workload changes over time; thus the selected view set must be updated online to better serve the incoming queries. Considering the better quality of a systematic algorithm and the better performance of a randomized algorithm, the author propose a hybrid algorithm which combines these two strategies. They use a systematic algorithm to compute an initial set of views statically when a bigger time window is available for the initialization. Then they utilize randomized algorithms to update the view set dynamically due to changes of the distribution of queries when the time window for the view set change is much more restrictive. This combination of systematic and randomized strategy is not new. In [69, 154], a randomized replacement approach is used to improve the quality of a selected view set after it has been computed using a systematic algorithm.

3.4 Applications

Among many applications of view selection problems, the following main class of applications is considered in this work, based on different dimensions.

- Data warehouse
- Multi-query optimization
- View maintenance

D. Theodoratos et al. [133] summarize the data warehouse design problem as follows: In the data warehousing approach to the integration of data from multiple information sources, selected information is extracted in advance and stored in a repository. A data warehouse (DW) can therefore be seen as a set of materialized views defined over the sources. When a query is posed, it is evaluated locally, using the materialized views, without accessing the original information sources. The applications using DWs require high query performance. This requirement is in conflict with the need to maintain in the DW updated information. The DW configuration problem is the problem of selecting a set of views to materialize in the DW that answers all the queries of interest while minimizing the total query evaluation and view maintenance cost. With the trend that databases are getting bigger in size and queries getting more complex, most DBMSs start supporting data warehouse features in regular database systems. Thus recommending physical structures for a database system is essentially a data warehouse problem.

Multi-query optimization techniques were considered too expensive in the past when the database was not very big and queries were not very complex. The most important difference between single query optimization and multi-query optimization techniques is the utilization of common subexpressions among multiple queries. Among all the works on multi-query optimization, the most important problem is which common subexpression to use and how to rewrite the original queries using selected common subexpressions (views) [121, 102, 85, 120]. No matter whether the selected common structures are stored in memory or disk space, the multi-query optimization problem is a special application of view selection problems. But in a multi-query optimization problem, it is not only to select common subexpressions, but also to recommend the globally optimal evaluation plan for execution of the multiple queries.

View maintenance problems are a special case of multi-query optimization problem where all the queries are update statements for a pre-selected set of views [16, 67, 118, 94, 106, 144]. Basically two main diagonal techniques to a view maintenance problems are proposed: incremental maintenance and maintenance using multi-query optimizations. This view maintenance problem is a complementary problem for view selection problem. All selected materialized views and indexes have to be updated to keep consistency with the underlying base tables. How views are maintained has a significant effect on the view selection problem. This is especially true when maintenance time constraints are considered or update statements are combined into the query workload.

3.5 Other Problems

Although the view selection problem has been studied for some time now, there are still a lot of open issues to address.

The construction of a search space for the view selection problem is a difficult issue. The reason is that common subexpression among queries need to be identified as candidate views for materialization, and the queries need to be rewritten using these views. For the special case of a multi-dimensional database with cube queries, the search space can be represented in a straightforward way using a lattice framework [75, 13]. The rewriting of the queries using views is also trivial in this case. However, in the general case of a relational database with complex queries involving grouping, aggregation and nesting, the construction of the search space and its use by view selection algorithms is an open problem.

Maintaining views using multi-query optimization techniques is a special instance of the view selection problem. In this case, an additional set of views is selected to be temporally materialized in order to accelerate the view maintenance process. Besides the selected views, a global maintenance plan that involves these views should be provided as an output. In practice, memory and time restrictions make this problem more complicated.

A number of problems can be modeled as view selection problems where the input statements involve both queries and updates and the order of these statements is of importance. These versions of the view selection problems are more complex since the input update statements may update the views that are temporarily or permanently materialized and this cost should be taken into account by the view selection algorithms.

CHAPTER 4

MIXED ARITHMETIC CONSTRAINTS IN DATABASES

Important issues in databases involve the satisfiability, implication and equivalence problems for arithmetic constraints with inequalities. These problems have been studied in the area of databases when the domain of the variables (attributes) is exclusively the integers, or exclusively the reals. Various issues in databases, including optimizing queries with aggregate functions require solving the previous problems for arithmetic constrains that involve simultaneously both types of variables (mixed arithmetic constraints). Furthermore, they need efficient algorithms for computing the maximal range of variables in mixed arithmetic constraints.

All these problems for conjunctions of mixed arithmetic constraints are addressed in this chapter. The arithmetic constraint are of the form $X\theta Y + c$ or $X\theta\, c$, where X,Y are variables of any of the two types, $\theta \in \{<, \leq, =, \neq, >, \geq\}$, and c is a real constant. It is shown that, as expected, when $\neq$ is involved in the constraints, the corresponding problems are NP-hard. Using a graph structure for constraints having two types of nodes to account for integer and real variables, necessary and sufficient conditions for unsatisfiability are derived. Finally, efficient polynomial algorithms are designed for these problems when $\neq$ is not allowed to occur in the constraints and their complexity are studied.

4.1 Introduction

Many important problems in various database areas require solving the satisfiability, implication, and equivalence problems for arithmetic constraints. Arithmetic constraints are boolean expressions of inequalities of the form $X\,\theta\, Y + c$, or $X\,\theta\, c$, where X and Y are variables (attributes), c is a constant, and $\theta \in \{<, \leq, =, \neq, >, \geq\}$.

Checking query equivalence and finding equivalent rewritings of queries [37, 35, 3], checking containment of queries [5, 34], answering queries using (partially or completely) views [92, 146, 96, 93, 4], optimizing query evaluation using materialized views [137, 24, 6], processing queries in a distributed environment over horizontally fragmented databases [108], optimizing multiple queries concurrently through the detection of common subexpressions [80, 121, 123, 120, 106], maintaining materialized views [15, 67, 106, 144], selecting views for materialization in a warehouse [133, 106, 135], enforcing consistency of rules and triggers [83], evaluating queries in constraint query languages or in the presence of constraints [86, 100] are only some of the traditional or more recent database areas where the aforementioned problems require an efficient solution.

Because of their importance for databases, these problems have been extensively studied in the past [116, 89, 138, 15, 127, 63, 62]. All these contributions consider that the variables in the arithmetic constraints range exclusively over the integers or exclusively over the reals. In this chapter, these problems by considering, in contrast, arithmetic constraints where some variables range over the integers, and some variables range over the reals. Such constraints are called *mixed arithmetic constraints*

Motivation. Mixed arithmetic constraints can be met in multiple areas in Databases. In relational databases, the domain of some variables (attributes) in a selection condition can be the integers, while the domain of other variables in the same condition can be the reals. For instance, in SQL, the domain of two attributes can be defined to be `INTEGER` and `REAL`, and both attributes can occur in the condition of the `WHERE` clause of a query. Thus, the `WHERE` clause condition of an SQL query can be a mixed arithmetic constraint. Establishing the unsatisfiability of a `WHERE` clause condition, guarantees that the query will have empty answers no matter what the instances of the relations in the FROM clause are. This information can be obtained without evaluating the query, and therefore, it is very useful to query optimizers.

Example 1. Suppose that the domain of the attributes X and Y of the relations R and S, respectively, is INTEGER and that the domain of attribute Z of relation T is REAL. Consider the simple SQL query Q:

```
SELECT *
FROM R, S, T
WHERE R.X < S.Y - 5.5 AND T.Z + 2.4 >= S.Y
  AND T.Z < R.X + 3.5;
```

The WHERE clause condition of this query is a mixed arithmetic constraint. Using the results in this work one can efficiently check that this constraint is unsatisfiable (i.e. no assignment of an integer value to X and Y and a real value to Z makes this constraint true). The answer of Q is empty for any instance of R, S and T. Note however that the constraint is satisfiable if the domain of all three attributes X, Y, and Z is REAL.

Deciding about the usability of views in answering queries, requires deciding about the implication of constraints in the WHERE clauses of queries and views [96, 137, 24, 73, 4]. When the domain of some attributes is the reals and that of other attributes is the integers, and the goal is to choose optimal query evaluation plans that use materialized views, one is faced with the implication problem of mixed arithmetic constraints.

In recent years, the increased interest of the industry for Data Warehousing and On-Line Analytical Processing (OLAP) applications [23] has stimulated the research on issues related to queries (and views) with grouping and aggregation operations [27, 145, 65, 126, 28, 22, 60, 34]. Such queries in SQL involve aggregate functions (e.g., COUNT, MIN, AVG) in the conditions of their HAVING clauses, or aggregated attributes (coming from grouping/aggregation views) in the conditions of their WHERE clauses. Depending on the case, the SQL aggregate functions return real or integer values. In order to deal with optimization issues of grouping/aggregation queries, the satisfiability problem for arithmetic constraints involving aggregate functions (*aggregation arithmetic constraints*) need to be solved. The same holds for the implication and equivalence problems but these latter ones, as is well known, can be reduced to the satisfiability problem. The satisfiability problem for different classes of aggregation arithmetic constraints is of high complexity or even undecidable [117, 97]. However, mixed arithmetic constraints can be used to efficiently derive sound unsatisfiability results for aggregation arithmetic constraints:

Example 2. Suppose that the domain of the attributes X, Y and Z is REAL. Consider the following condition A from the HAVING clause of an SQL query:

```
COUNT(X) + 2 <= SUM(Y) AND AVG(Z) > SUM(Y) AND
AVG(Z) <= COUNT}(X)
```

This is an aggregation arithmetic constraint. Notice that the term COUNT(X) takes (not negative) integer values while the terms SUM(Y) and AVG(Z) take real values. If COUNT(X) is replaced by an (aggregated) variable X' ranging over the integers, and SUM(Y) and AVG(Z) by the (aggregated) variables Y' and Z', respectively, ranging over the reals, a mixed arithmetic constraint C is obtained: $X' + 2 \leq Y' \wedge Z' > Y' \wedge Z' \leq X'$. Using the results of this work, one can efficiently derive that C is unsatisfiable. Clearly, the unsatisfiability of condition C implies the unsatisfiability of condition A.

A further approach for obtaining sound unsatisfiability results for an aggregation arithmetic constraint A uses the range constraints on the aggregation terms in A that can be derived from A. Examples of such range constraints are COUNT$(X) \leq 3.5$ or AVG$(Y) > 2$. Range constraints specify a range of acceptable values for the corresponding aggregation term. Ross et al. [117] provide a polynomial algorithm that directly checks for satisfiability a conjunction C of range constraints on aggregation terms involving a single variable (attribute). For instance, this algorithm decides that the constraint MIN$(X) > 0 \wedge$ COUNT$(X) \leq 10 \wedge$ MAX$(X) \leq 1000 \wedge$ SUM$(X) > 10000$ is unsatisfiable. The conjunction of the derived range constraints on the aggregation terms of A can be provided as input to this algorithm to decide a potential unsatisfiability of A.

Distinction of the problems addressed from other known problems. A question that arises is whether these problems for mixed arithmetic constraints are subsumed by the corresponding problems for arithmetic constraints where the variables range exclusively over the integers, or exclusively over the reals. The next example shows that *they are not.*

Example 3. Consider the arithmetic constraint $C = X < Y + 3 \wedge X > Y + 2.5$. Clearly, if both variables X and Y range over the integers, there is no assignment of integer values to X and Y that makes C true. That is, C is unsatisfiable. In contrast, if at least one of the variables X and Y range over the reals then C becomes satisfiable. For instance, the assignment $X \leftarrow 2.6$, $Y \leftarrow 0$ makes C true.

Consider also the arithmetic constraint $C' = X \leq 0.3 \wedge X \geq -0.3 \wedge Y > X + 0.4 \wedge Y \leq X + 0.6$. If both variables X and Y range over the reals then C' is satisfiable. For instance, the assignment $X \leftarrow 0$, $Y \leftarrow 0.5$ satisfies C'. In contrast, if Y is a variable that ranges over the integers, it is not difficult to see that C' is unsatisfiable. Indeed, C' restricts the values of X in the interval $[-0.3, 0.3]$, and the values of Y in the interval $(0.1, 0.9]$. Clearly, there is no integer between 0.1 and 0.9.

The problem. From the previous discussion it becomes clear that, in order to deal with traditional or more recent important problems in databases, the satisfiability, implication, and equivalence problems for mixed arithmetic constraints, and the problem of computing ranges for variables in such constraints need to be solved. These problems have not been addressed previously for mixed arithmetic constraints.

Contribution and Outline The main contributions of this work are the following:

- Mixed arithmetic constraints are formally introduced. For the class of conjunctive mixed arithmetic constraints, the following three problems are addressed: the satisfiability problem, the implication problem, and the problem of computing ranges of variables in a constraint.

- It is shown in this chapter that when $\neq$ is allowed to occur in the constraints, the problem of computing ranges of variables in a constraint is NP-hard. In this case, the NP-hardness of the satisfiability and implication problems is a direct consequence of the NP-hardness of the corresponding problems when variables range exclusively over the integers. Further, we focus on the case where $\neq$ is not allowed to occur in the constraints.
- A directed graph structure for representing a conjunctive mixed arithmetic constraint is defined. This graph has two types of nodes to account for variables ranging over the integers, and for variables ranging over the reals.
- By exploiting the graph structure, sound and complete conditions for all three problems are derived.
- Polynomial algorithms are provided for checking satisfiability, implication, and for computing ranges of variables, and the corresponding complexities are studied.
- Besides their utility in traditional database issues, these results are essential in checking the satisfiability of arithmetic constraints involving aggregate functions.

4.2 Related Work

As a general remark, the complexity of the satisfiability, implication and equivalence problems increases when $\neq$ is involved in the arithmetic constraints, and when the domain of the variables is restricted to integers. Rosenkrantz et al. [116] study the satisfiability and equivalence problems of conjunctions of inequalities of the form $X\theta Y + c$ or $X\theta c$, involving $\neq$, when the variables range over the integers. They show that these problems are NP-hard and provide polynomial algorithms for the case where $\neq$ is not allowed in the constraints. Klug [89] provides a polynomial algorithm for the implication problem of conjunctions of inequalities of the form $X\theta Y$ or $X\theta c$, excluding $\neq$, when the domain of the variables is the rationals (these results apply equally to any dense totally ordered domain e.g., the reals). Ullman [138, subsection14.2] gives a polynomial algorithm for implication checking of conjunctions of inequalities of the form $X\theta Y$ only, involving $\neq$, when the variables range over the integers. This algorithm applies as well to the case where the variables range over an infinite totally ordered dense domain. Blakeley et al. [15] improve the satisfiability checking algorithm of [116] for the case where $\neq$ is not allowed in the inequalities, under the assumption that the integer domain of each variable is finite. Sun et al. [127] studied the following implication problem: decide whether C implies C', where C, C' are conjunctions and/or disjunctions of inequalities involving $\neq$, and the variables range over the integers. They prove that when C and C' are conjunctions of inequalities, and $\neq$ is allowed in C, the implication problem is NP-hard. They also provide a polynomial algorithm for this case when $\neq$ is not allowed to occur in C (and allowed to occur in C'). Guo et al. [63] addresses the satisfiability and implication problems of conjunctions of inequalities of the form $X \theta Y$ or $X \theta c$, involving $\neq$, when the variables range exclusively over the integers or exclusively over the reals. They provide polynomial algorithms, and improve the complexity of previous algorithms for the case where the domain of the variables is the integers and $\neq$ is excluded, and for the case where the domain of the variables is the reals and $\neq$ is allowed. Guo et al. [64] gives polynomial algorithms for the satisfiability and implication problems of conjunctions of inequalities of the form $X \theta Y + c$ or $X \theta c$, involving $\neq$, when the variables range over the reals. Finally, in [86], different complexity results are provided for the implication problem of conjunctions of constraints of the form $p(X_1, \ldots, X_n) \, \theta \, 0$ where p is a polynomial on variables $X_1, \ldots, X_n$ and the variables range over the reals.

None of the previous approaches consider the case of conjunctions of inequalities of the form $X \theta Y + c$ or $X \theta c$, where some variables range over the reals, and some variables range over the integers, as in here.

4.3 Mixed Arithmetic Constraints

In this section, mixed arithmetic constraints are formally defined. And the class of mixed arithmetic constraints considered in this chapter is specified.

Syntax. Two types of variables are considered. Variables that range over the reals (*real variables*) and variables that range over the integers (*integer variables*).

Atomic mixed arithmetic constraints are expressions of the form $X\,\theta\,Y + c$, or $X\,\theta\,c$, where X and Y are integer or real variables, c is a real constant, and θ is a comparison predicate in $\{<, \leq, =, \neq, >, \geq\}$. More complex *mixed arithmetic constraints* are built from atomic arithmetic constraints using conjunction, disjunction, and complementation, as usual. Mixed arithmetic constraints generalize arithmetic constraints where all the variables range exclusively over the reals or exclusively over the integers. For simplicity, mixed arithmetic constraints are also called in the following simply constraints.

Semantics. An assignment ϕ to the variables of a constraint is also defined as usual. An assignment is *well typed* if it assigns real values to real variables, and integer values to integer variables. A well-typed assignment *satisfies* a constraint C if and only if the expression $C\phi$ holds. If there is such an assignment, C is *satisfiable*. Constraint C is *valid* if it is satisfied by all well-typed assignments. A constraint C *implies* a constraint C' (denoted $C \models C'$) if every well typed assignment that satisfies C satisfies also C'. Two constraints C and C' are *equivalent* (denoted $C \equiv C'$) if $C \models C'$ and $C' \models C$. Since equivalence is defined in terms of implication, in the following, constraint equivalence checking is not explicitly addressed.

The class of constraints considered. A constraint C can be equivalently transformed to a disjunction of conjunctions of atomic constraints. Further, negations can be eliminated by appropriately changing the comparison predicate (e.g., $\neg(X \neq Y + c)$ can be replaced by $X = Y + c$, and $\neg(X < Y + c)$ can be replaced by $X \geq Y + c$). Then, in order to check C for satisfiability it suffices to check each disjunct (conjunction of atomic constraint) in the transformed C separately. In the worst case though, this transformation may cause the number of atomic constraints to grow exponentially. In the rest of this chapter, it is considered that *a constraint is a conjunction of atomic constraints*.

4.4 Mixed Arithmetic Constraint Graphs

In this section, a graph representation for mixed arithmetic constraints is introduced. This representation and related notions introduced here are used in the next sections for studying satisfiability and implication checking, and variable range computation.

Definition 1. An atomic constraint is in *standard from* if it is in the form $X \ \theta \ Y + c$, where:

(a) X, Y are distinct variables, or one of X, Y is a variable and the other is 0,

(b) $\theta \in \{<, \leq\}$, and

(c) c is a real constant.

A constraint is in *standard form* if it contains only atomic constraints in standard form.

For the needs of this section it is considered that constraints are in standard form.

Definition 2. Given a mixed arithmetic constraint C in standard form, the *constraint graph of C* is a labeled directed graph $\mathcal{G}_C$ defined as follows:

1. The nodes of $\mathcal{G}_C$ are partionned in two sets of nodes: integer nodes and real nodes.
2. For each integer (real) variable in C there is an integer (real) node in $\mathcal{G}_C$. There is also in $\mathcal{G}_C$ an integer node for the constant 0. In the following we may identify a node with the variable (or the constant 0) it represents.
3. For each atomic constraint $X \ \theta \ Y + c$ in C, there is an edge from node X to node Y labeled by the pair (θ, c).

The constraint graph $\mathcal{G}_C$ *represents* the constraint C. Since for each atomic constraint $X \ \theta \ Y + c$ in C, X and Y are distinct, there are no edge-loops in $\mathcal{G}_C$.

Example 4. Consider a constraint C which is a conjunction of the atomic constraints in the following set:

$\{X_1 < Y_1 + 3.5,\ Y_1 < X_2 + 2.1,\ X_2 \leq 0 - 2.1,\ 0 \leq Y_2 + 4,\ Y_2 \leq X_3 + 2.2, X_3 < Y_1 - 2,$
$X_1 < X_3 - 3.5,\ X_3 \leq Y_2 - 2.2,\ Y_2 \leq X_4 - 1\}.$

The X_is denote real variables, and the Y_is integer variables.

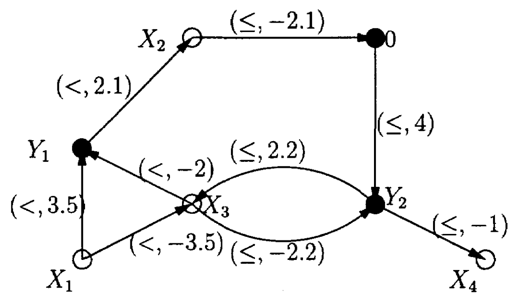


Figure 4.1 Constraint graph $\mathcal{G}_C$ representing constraint C.

Constraint C can be represented by the constraint graph $\mathcal{G}_C$ shown in Figure 4.1. Real nodes are represented by white circles while integer nodes are represented by filled black circles. This constraint graph is used as a running example in this work.

In the following the notions of path and cycle is used in a constraint graph. Paths and cycles here are supposed to be *elementary* that is, they do not meet the same node twice.

Integer nodes can be viewed as discontinuity points of the paths in $\mathcal{G}_C$. Therefore, integer paths are introduced below which are paths between integer nodes whose internal nodes are not integer nodes.

Definition 3. An *integer path* in $\mathcal{G}_C$, is a path from an integer node X to a (not necessarily distinct) integer node Y that does not include an integer node other than X and Y.

If X and Y coincide, the integer path is a cycle that includes a single integer node. If a cycle s in a constraint graph includes n, $n \geq 1$, integer nodes, it also includes exactly n integer paths and each of the edges of s are included in one of these integer paths. It is evident that if s does not include any integer node, it does not include any integer path.

For integer paths, the notion of compound labels is needed, as defined below. This definition is based on the following remark:

Remark 4.4.1. (a) Suppose that the constraints $X \leq Y + c_1$ and $Y \leq Z + c_2$ hold. Then, by transitivity, the constraint $X \leq Z + (c_1 + c_2)$ holds. Consider now the constraints $X \leq Y + c_1$ and $Y < Z + c_2$. These constraints differ from the previous ones in that one constraint involves a strict inequality. Then, by transitivity again, the constraint $X < Y + (c_1 + c_2)$ holds. This remark is valid for any type (real or integer) of variables X, Y, and Z.

(b) Suppose that the constraint $X \, \theta \, Y + c$, where $\theta \in \{<, \leq\}$, and X, Y are integer variables, holds.

(b1) If c in not an integer, the constraint $X \leq Y + \lfloor c \rfloor$ also holds. ($\lfloor x \rfloor$ denotes the floor function which rounds the value x down to the next integer.) For real variables X and Y, the constraint $X \leq Y + \lfloor c \rfloor$ is more restrictive than $X \, \theta \, Y + c$.

(b2) If c is an integer and θ is $<$, the constraint $X \leq Y + (c - 1)$ also holds. For real variables X and Y, the constraint $X \leq Y + (c - 1)$ is more restrictive than $X < Y + c$.

Remark (b) is valid even if one of X and Y is 0.

From the second remark it is also deduced that for integer variables it suffices to consider atomic arithmetic constraints in the standard form $X \, \theta \, Y + c$, where θ is $\leq$ and c is an integer. For instance, $X < Y + 3.4$ can be equivalently represented by $X \leq Y + 3$, while $X < Y + 3$ can be equivalently represented by $X \leq Y + 2$.

Definition 4. Let p be an integer path in $\mathcal{G}_C$. L_p is used to denote the multiset of labels of the edges in p.

The *compound label of p* is a label (θ, c) where:

1. θ is $\leq$.

2. $$c = \begin{cases} \sum_{(\theta_i, c_i) \in L_p} c_i - 1 & \text{if } \sum_{(\theta_i, c_i) \in L_p} c_i \text{ is an integer, and there is at least} \\ & \text{one label in } L_p \text{ whose first component is } < \\ \lfloor \sum_{(\theta_i, c_i) \in L_p} c_i \rfloor & \text{otherwise.} \end{cases}$$

Intuitively, the label (θ, c) of an integer path p from integer node X to integer node Y expresses that the constraint $X \, \theta \, Y + c$ is the most restrictive constraint involving X and Y that can be derived from the constraint represented by p.

Example 5. Consider the constraint graph $\mathcal{G}_C$ shown in Figure 4.1. The path $(Y_1, X_2, 0)$ is an integer path. Its compound label is $(\leq, -1)$. The path (Y_2, X_3, Y_1) is an integer path.

Its compound label is $(\leq, 0)$. The path (Y_2, X_3, Y_2) is an integer path that is also a cycle. Therefore, it includes a single integer node. Its compound label is $(\leq, 0)$.

The notion of compound labels is generalized below for paths that are not necessarily integer paths. These paths can also be cycles.

Definition 5. Let p be a path in $\mathcal{G}_C$. CL_p is used to denote the multiset of compound labels of the integer paths included in p, and by SL_p the multiset of labels of the edges contained in p that are not included in an integer path included in p.

The *compound label of p* is a label (θ, c) where:

$$1. \theta = \begin{cases} < & \text{if there is at least one label in } SL_p \text{ whose first component is } < \\ \leq & \text{otherwise.} \end{cases}$$

2. $c = \sum_{(\theta_i, c_i) \in SL_p} c_i + \sum_{(\leq, c_j) \in CL_p} c_j$.

Clearly, one of SL_p and CL_p can be an empty set. In this case only one of the sums contribute to c. If SL_p is empty, θ is necessarily $\leq$. This is the case if p is a cycle that includes at least one integer node. According to the previous definition, the compound label of a cycle is independent of the node considered as initial (and terminal) node of the corresponding path.

Example 6. Consider the constraint graph $\mathcal{G}_C$ depicted in Figure 4.1. The compound label of the path $(X_1, Y_1, X_2, 0, Y_2, X_4)$ is $(<, 5.5)$. The compound label of the path $(0, Y_2, X_4)$ is $(\leq, 3)$. The compound label of the path $(X_3, Y_1, X_2, 0, Y_2, X_3)$ (which is also a cycle) is $(\leq, 3)$.

A relation $\preceq$ can be defined on compound labels as follows:

Definition 6. Let (θ, c) and (θ', c') be two compound labels for paths. Then, $(\theta, c) \preceq (\theta', c')$ if and only if one of the following conditions hold:

(a) $c < c'$,

(b) $c = c'$ and θ' is $\leq$,

(c) $c = c'$ and θ is $<$.

If $(\theta, c) \preceq (\theta', c')$, and $(\theta, c) \neq (\theta', c')$, then $(\theta, c) \prec (\theta', c')$.

Clearly, $\preceq$ is a total ordering relation on the set of compound labels for paths. In the following, minimality of labels is meant with respect to $\preceq$.

4.5 Checking Satisfiability

In this section, the *satisfiability problem* of mixed arithmetic constraints, which is the problem of deciding whether a constraint C is satisfiable, is studied.

When $\neq$ is allowed to occur in a constraint, checking satisfiability is not polynomial. The following theorem is a direct consequence of previous results for arithmetic constraints involving only variables ranging over the reals.

Theorem 1. *Checking satisfiability of mixed arithmetic constraints is NP-hard.*

Proof: The satisfiability problem of arithmetic constraints that are conjunctions of atomic constraints of the form $X \neq Y$, and X, Y range over the integers is NP-hard [116]. Mixed arithmetic constraints include this class of arithmetic constraints.

In the rest of this section, the attention is restricted to constraints that do not involve unequality ($\neq$). An atomic constraint of the form $X \, \theta \, X + c$ is called *trivial atomic constraint*. A trivial atomic constraint is either unsatisfiable (e.g., $X \leq X - 3$), or valid (e.g., $X \geq X - 3$). It is not difficult to see that an unsatisfiable atomic constraint is a trivial atomic constraint. Clearly, a constraint that contains an unsatisfiable atomic formula is unsatisfiable. Such a constraint is called *trivially unsatisfiable*. It is also easy to see that a valid atomic constraint is a trivial atomic constraint. Clearly, a constraint is valid if and only if all its atomic constraints are valid. Therefore, validity and trivial unsatisfiability of C can be checked in linear time on the number of atomic constraints in C. If a constraint contains more than one atomic constraint, a valid atomic constraint in it can be removed to obtain an equivalent constraint. An atomic constraint that is not valid or trivially unsatisfiable, can be equivalently put in standard form. In the first step of algorithm 2, later in this section, it is shown how a constraint satisfying these restrictions can be put in standard form. Recall that X and Y in each atomic constraint $X \, \theta \, Y + c$ of a constraint in standard form are distinct. Unless stated differently, it is considered in the remainder of this section that a constraint is in standard form.

Now necessary and sufficient conditions are provided for the satisfiability of mixed arithmetic constraints.

Theorem 2. *Let C be a mixed arithmetic constraint in standard form that does not involve $\neq$. Then C is unsatisfiable if and only if there is a cycle in $\mathcal{G}_C$ whose compound label (θ, c) satisfies the condition $(\theta, c) \prec (\leq, 0)$.*

The proof follows the next example. It is worth noting that if the compound label (θ, c) of a cycle s in $\mathcal{G}_C$ satisfies the condition $(\theta, c) \prec (\leq, 0)$, and there are more than one integer nodes in s, the sum of the second components of the labels of the edges in s can be greater than 0. However, if there are only real nodes in s, or if there is exactly one integer node in s, the condition $(\theta, c) \prec (\leq, 0)$ can be satisfied only if this sum is less than or equal to 0.

Example 7. Consider the graph $\mathcal{G}_C$ of Figure 4.1 representing the constraint C of Example 4. Graph $\mathcal{G}_C$ contains exactly two cycles: $(Y_1, X_2, 0, Y_2, X_3, Y_1)$ and (Y_2, X_3, Y_2). Their compound labels are $(\leq, 3)$ and $(\leq, 0)$, respectively. Since $(\leq, 3) \not\prec (\leq, 0)$ and $(\leq, 0) \not\prec (\leq, 0)$, according to Theorem 2, constraint C is satisfiable.

Proof of Theorem 2: *Necessity.* It is to be proved that if there is a cycle in $\mathcal{G}_C$ whose compound label (θ, c) satisfies the condition $(\theta, c) \prec (\leq, 0)$, C is unsatisfiable. Let s be such a cycle in $\mathcal{G}_C$.

First, suppose that there are no integer nodes in s. Let the atomic constraints represented by s be $X_1 \, \theta_1 \, X_2 + c_1, \ \ldots, \ X_{n-1} \, \theta_{n-1} \, X_n + c_{n-1}, \ X_n \, \theta_n \, X_1 + c_n$, $n \geq 2$ (recall that since

C is in standard form a variable cannot appear more than once in an atomic constraint). By transitivity (see remark 4.4.1(a)), these atomic constraints imply the constraint $X_1\,\theta'\,X_1 + (c_1 + \ldots + c_n)$, where θ' is $<$ if one of the θ_i's, $i \in [1,n]$, is $<$, and is $\leq$ otherwise. Thus, the constraint $X_1\,\theta'\,X_1 + (c_1 + \ldots + c_n)$ is implied by the atomic constraints of C represented by s. By definition (since no integer node is included in s), $\theta = \theta'$ and $c = c_1 + \ldots + c_n$. Since $(\theta, c) \prec (\leq, 0)$, this constraint is unsatisfiable. Therefore, C is unsatisfiable.

Then, suppose that s includes one or more integer nodes. Consider an integer path p from integer node X_1 to integer node X_n in s. X_1 and X_n need not necessarily be distinct. However, if they coincide, other (real) nodes are also included in path p. X_1 and/or X_n can be node 0. Let $(\leq, c')$ be the compound label of the integer path p. Let also $X_1\,\theta_1\,X_2{+}c_1$, $\ldots$, $X_{n-1}\,\theta_{n-1}\,X_n{+}c_{n-1}$, $n \geq 2$, be the atomic constraints represented by p, where $X_2, \ldots, X_{n-1}$ are real variables. By transitivity (as above), these atomic constraints imply the following constraint: $X_1\,\theta_p\,X_n + (c_1 + \ldots + c_{n-1})$, where θ_p is $<$ if one of the θ_i's, $i \in [1, n-1]$, is $<$, and is $\leq$ otherwise. This constraint implies the constraint $X_1 \leq X_n + c'$, where $(\leq, c')$ is the compound label of the integer path p (see remark 4.4.1(b)). Thus, the constraint $X_1 \leq X_n + c'$ is implied by the atomic constraints represented by the integer path p in cycle s. Let $Y_1, \ldots, Y_m$, $m \geq 1$, be the integer nodes in s, in that order, $(\leq, c'_i)$ be the compound label of the integer path from Y_i to Y_{i+1} in s, $i \in [1, m-1]$, and $(\leq, c'_m)$ be the compound label of the integer path from Y_m to Y_1 in s. Then, the constraints $Y_1 \leq Y_2 + c_1$, $\ldots$, $Y_{m-1} \leq Y_m + c_{m-1}$, $Y_m \leq Y_1 + c_m$ are implied by the atomic constraints represented by cycle s. By transitivity, these atomic constraints imply the constraint $Y_1 \leq Y_1 + (c_1 + \ldots + c_m)$. Thus, the constraint $Y_1 \leq Y_1 + (c_1 + \ldots + c_m)$ is implied by the atomic constraints of C represented by s. By definition (since at least one integer node is included in s), θ is $\leq$ and $c = c_1 + \ldots + c_m$. Since $(\theta, c) \prec (\leq, 0)$, this constraint is unsatisfiable. Therefore, C is unsatisfiable.

Sufficiency. It is to be proved that if there is no cycle in $\mathcal{G}_C$ whose compound label (θ, c) satisfies the condition $(\theta, c) \prec (\leq, 0)$, there is a well-typed assignment that satisfies C (that is, C is satisfiable).

First, a procedure *assignment_construction* is provided that computes a well-typed assignment to the variables of a constraint C. This procedure associates intervals of values to the variables. The upper and lower bounds of an interval can be $+\infty$ and $-\infty$, respectively. The bound of an interval of a real variable can be open or closed. (It is of course open if it is $+/-\infty$.) Intervals of integer variables contain integer values and their bounds are closed (unless the bounds are $+/-\infty$).

Suppose that an interval of values is associated with each node of $\mathcal{G}_C$. Given a node Y in $\mathcal{G}_C$, let θ_u (θ_l) be $<$ if the upper (lower) bound of node Y is open, and $\leq$ otherwise; let c_u (c_l) be the upper (lower) bound value of Y.

Let b be the value of the closed upper bound of a node X. *Propagating the upper bound value b of X into $\mathcal{G}_C$*, refers to the following process:

For every node Y (distinct than X) in $\mathcal{G}_C$ such that there is a path from Y to X do:

(a) let (θ_p, c_p) be the minimal compound label of the paths from Y to X.

(b) let $\theta' = \theta_p$, and $c' = b + c_p$. If $(\theta_l, c_l) \preceq (\theta', c')$ and $(\theta', c') \prec (\theta_u, c_u)$ then

(C) let the upper bound of Y be open if θ' is $<$, and closed otherwise; let the upper bound value of Y be c'.

Let now b be the value of the closed lower bound of a node X. *Propagating the lower bound value b of X into $\mathcal{G}_C$*, refers to the following process:

For every node Y (distinct than X) in $\mathcal{G}_C$ such that there is a path from X to Y do:

(a) let (θ_p, c_p) be the minimal compound label of the paths from X to Y.

(b) let $\theta' = \theta_p$, and $c' = b - c_p$.

(c) if $(\theta_l, c_l) \prec (\theta', c')$ and $(\theta', c') \preceq (\theta_u, c_u)$ then let the lower bound of Y be open if θ' is $<$, and closed otherwise; let the lower bound value of Y be c'.

The procedure that computes a well-typed assignment to the variables of C follows:

Procedure *assignment_construction*

Initialization

- Associate the interval $[0, 0]$ with node 0, and the interval $(-\infty, +\infty)$ with the rest of the nodes in $\mathcal{G}_C$.
- Propagate the lower and the upper bound value 0 of node 0 into $\mathcal{G}_C$.

Main part

- While there is an integer node X whose upper and lower bound values are not equal do
  - Associate with X a closed interval $[b, b]$ such that b is greater than or equal to the old lower bound of X and less than or equal to the old upper bound of X. Recall that b must be an integer since X is an integer node.
  - Propagate the new lower and upper bound value b of X into $\mathcal{G}_C$.
- While there is a real node X whose upper bound is open do
  - Associate with X a closed interval $[b, b]$ such that b is greater than the old lower bound of X (or equal to the old lower bound if the old lower bound is closed) and less than the old upper bound of X.
  - Propagate the new lower and upper bound value b of X into $\mathcal{G}_C$.

- Return an assignment that assigns to every variable (node) in $\mathcal{G}_C$ its upper bound value.

An application of procedure *assignment_construction* is shown in example 8 below. Notice that during the execution of procedure *assignment_construction*, once a closed interval is associated with a (real or integer) node, it is not modified afterwards. Also, the computed bound values of the intervals of integer nodes are integers since: (a) integer nodes are treated first by procedure *assignment_construction*, and (b) integer bound values are associated with integer nodes while the second component of the compound label of integer paths is an integer. Therefore, the returned assignment is well-typed.

Clearly, procedure *assignment_construction* terminates, and if there is no cycle in $\mathcal{G}_C$ whose compound label (θ, c) satisfies the condition $(\theta, c) \prec (\leq, 0)$, the computed well-typed assignment satisfies all the atomic constraints in C.

The following example outlines the computation of a well-typed assignment by procedure *assignment_construction*.

Example 8. Suppose that the constraint graph $\mathcal{G}_C$ of Figure 4.1 is provided as input to procedure *assignment_construction*.

In the initialization phase, first the interval $[0, 0]$ is associated with node 0, and the interval $(-\infty, +\infty)$ is associated with the rest of the nodes in $\mathcal{G}_C$. Then the lower and the upper bound value 0 of node 0 are propagated into $\mathcal{G}_C$, associating intervals with nodes as follows: $0 : [0, 0], Y_1 : [-4, -1]$, $Y_2 : [-4, -1]$, $X_1 : (-\infty, -6.5), X_2 : (-6.1, -2.1)$, $X_3 : [-6.2, -3)$, $X_4 : [-3, +\infty)$.

In the main part of the procedure, suppose that the integer node Y_1 is considered first, that it is associated with the interval $[-4, -4]$, and that the upper and lower bound value -4 of Y_1 is propagated into $\mathcal{G}_C$. The following new associations of nodes with intervals are produced: $Y_1 : [-4, -4], Y_2 : [-4, -4]$, $X_1 : (-\infty, -9.5)$, $X_3 : [-6.2, -6)$.

Then, performing each of the associations $X_4 : [-3, -3]$, $X_3 : [-6.2, -6.2], X_1 : [-10, -10]$, and $X_2 : [-3.5, -3.5]$, the corresponding propagations of the lower and upper bound values into $\mathcal{G}_C$ does not alter the rest of the associations.

One can check that the returned well-typed assignment $Y_1 \leftarrow -4$, $Y_2 \leftarrow -4, X_1 \leftarrow -10, X_2 \leftarrow -3.5$, $X_3 \leftarrow -6.2$, $X_4 \leftarrow -3$, assigning to each variable its computed upper bound value, satisfies constraint C.

An algorithm is provided below for checking the satisfiability of a mixed arithmetic constraint that does not involve $\neq$. This constraint need not necessarily be in standard form. Let N_C denote the number of atomic constraints (conjuncts) in C, and N_N denote the number of distinct variables in C.. Then, the following theorem characterizes the complexity of algorithm 2.

Theorem 3. *The satisfiability problem of mixed arithmetic constraints that do not involve $\neq$ can be solved in $\mathcal{O}(N_C N_N)$ time.*

Proof: Steps 1 and 2 of Algorithm 2 can be done in one scan of C which takes $\mathcal{O}(N_C)$ time. The number of conjuncts and variables in the resulting constraint C_s is also bounded by $\mathcal{O}(N_C)$ and N_N, respectively. The construction of $\mathcal{G}_{C_s}$ can be done, in step 3, in $\mathcal{O}(N_C)$ time. Strongly-connected components in $\mathcal{G}_{C_s}$ can be computed in $\mathcal{O}(N_C + N_N)$ time. The cycles in $\mathcal{G}_{C_s}$ and their compound labels can be computed using the Bellman-Ford algorithm for detecting negative-weighted cycles in a strongly connected component $\mathcal{O}(N_C N_N)$ time. As usual, it is assumed that accessing a node in the adjacency structure used to represent the graph $\mathcal{G}_{C_s}$ takes constant time. Therefore, the complexity of Algorthm 2 is $\mathcal{O}(N_C)$.

The results of this section generalize to the mixed arithmetic constraint case previous results on the satisfiability of arithmetic constraints where all the variables range exclusively over the integers [116, 15, 63, 64] or exclusively over the reals [63, 64].

Algorithm 2 Satisfiability checking

```
Input: A mixed arithmetic constraint $C$ that does not involve $\neq$
Output: A decision on whether $C$ is satisfiable or not
    {* Step 1: Remove trivial atomic constraints *}
 1: Remove from $C$ all the valid atomic constraints of the form $X \,\theta X + c$
 2: if No atomic condition is left in $C$ then
 3:     return $C$ is unsatisfiable
 4: end if
    {* Step 2: Put $C$ in standard form. *}
    {Replace all the occurrences of atomic constraints in the resulting constraint as follows:}
 5: Replace an atomic constraint of the from $X = Y + c$ by a conjunction of two atomic constraints $X \leq Y + c \wedge Y \leq X - c$
 6: Replace an atomic constraint of the form $X \,\theta\, Y + c$, where $\theta$ is $>$ or $\geq$, by $Y \,\theta'\, X - c$, where $\theta'$ is $<$ or $\leq$, respectively
 7: Replace an atomic constraint of the form $X = c$ by a conjunction of two atomic constraints $X \leq 0 + c \wedge 0 \leq X - c$
 8: Replace an atomic constraint of the form $X \,\theta\, c$, where $\theta \in \{<, \leq\}$, by $X \,\theta\, 0 + c$
 9: Replace an atomic constraint of the form $X \,\theta\, c$, where $\theta$ is $>$ or $\geq$, by $0 \,\theta'\, X - c$, where $\theta'$ is $<$ or $\leq$, respectively
    {* Step 3: Construct $\mathcal{G}_{C_s}$. *}
10: Construct the constraint graph $\mathcal{G}_{C_s}$ representing $C_s$
    {* Step 4: Check satisfiability. *}
11: Detect all the cycles in $\mathcal{G}_{C_s}$ and compute their compound labels
12: return "$C$ is unsatisfiable" if a cycle is found whose compound label $(c, \theta)$ satisfies the condition $(c, \theta) \prec (0, \leq)$; otherwise, return "$C$ is satisfiable"
```

4.6 Checking Implications

In this section, the *implication problem* of mixed arithmetic constraints is studied, which is the problem of deciding, given two constraints C and C', whether $C \models C'$.

As with satisfiability checking, when $\neq$ is allowed to occur in a constraint, checking implication is not polynomial. The following theorem is a direct consequence of results for arithmetic constraints involving only variables ranging over the integers.

Theorem 4. *Checking implication of mixed arithmetic constraints is NP-hard.*

Proof: The implication problem of arithmetic constraints that are conjunctions of atomic constraints of the form $X \neq Y$, and X, Y range over the integers is NP-hard [127]. Mixed arithmetic constraints include this class of arithmetic constraints.

The rest of this section deals with constraints that do not involve $\neq$.

The implication problem can be solved by solving a number of corresponding satisfiability problems: if $C' = a_1 \wedge \ldots \wedge a_n$, where the a_is are atomic constraints, then $C \models C'$ if and only if the constraints $C \wedge \neg a_1, \ldots, C \wedge \neg a_n$ are unsatisfiable. If some a_i is an atomic constraint involving equality (e.g., $X = Y+c$) then $\neg a_i$ is equivalent to an atomic constraint involving unequality (e.g., $X \neq Y + c$). $\neg a_i$ can be equivalently rewritten as a disjunction of two atomic constraints a_{i1} and a_{i2} involving strict inequalities (e.g., $X < Y + c$ and $X > Y + c$). Thus, $C \wedge \neg a_i$ is unsatisfiable if and only if $C \wedge a_{i1}$ and $C \wedge a_{i2}$ are both unsatisfiable. As a consequence, $C \models C'$ can be decided by checking for satisfiability at most $2 * N_{C'}$ constraints, each involving $N_C + 1$ atomic constraints. Recall that N_C denotes the number of atomic constraints in C, while $N_{C'}$ is the number of atomic constraints in C'. Since these constraints do not involve $\neq$, Algorithm 2 can be used. The time complexity of this process is $\mathcal{O}(N_C * N_{C'})$.

Below an algorithm is provided with a different complexity that solves the implication problem without transforming it into the satisfiability problem.

In the remainder of this section, unless differently stated, it is assumed that C and C' are in standard form. A constraint can be put in standard form if it is not valid or trivially unsatisfiable. It is also assumed that C is satisfiable. These assumptions imply that any variable appearing in C' appears also in C: if C' involves a variable that does appear in C (and C and C' are in standard form) then, clearly, $C \models C'$ if and only if C is unsatisfiable. As mentioned in the previous section, validity and satisfiability of mixed arithmetic constraints can be checked efficiently. Therefore, there is no loss of generality with these assumptions.

Now the necessary and sufficient conditions are provided for the implication of mixed arithmetic constraints.

Theorem 5. *Let C and C' be mixed arithmetic constraints in standard form that do not involve $\neq$. Suppose that C is satisfiable. Then, $C \models C'$ if and only if, for every atomic constraint $X\ \theta\ Y + c$ in C', there is a path from node X to node Y in $\mathcal{G}_C$ whose compound label (θ', c') satisfies the condition $(\theta', c') \preceq (\theta, c)$.*

The proof follows the next example.

Example 9. Consider the constraint graph $\mathcal{G}_C$ of Figure 4.1 representing constraint C of Example 4. Let C' be the constraint in standard form: $X_1 \leq X_4 - 4\ \wedge\ 0 < X_4 + 4$. The compound label $(<, -6.7)$ of the path (X_1, X_3, Y_2, X_4) from X_1 to X_4 in $\mathcal{G}_C$ satisfies the condition $(<, -6.7) \preceq (\leq, -4)$. Also, the compound label $(\leq, 3)$ of the path $(0, Y_2, X_4)$ from 0 to X_4 in $\mathcal{G}_C$ satisfies the condition $(\leq, 3) \preceq (\leq, 4)$. Therefore, according to Theorem 5, $C \models C'$.

Let C'' be the constraint in standard form $Y_1 < Y_2 + 0 \wedge Y_1 < 0 + 3.1$. $(Y_1, X_2, 0, Y_2)$ is the only path from Y_1 to Y_2 in $\mathcal{G}_C$. Its compound label $(\leq, 3)$ satisfies the condition $(\leq, 3) \not\preceq (<, 0)$. Therefore, according to Theorem 5, $C \not\models C''$.

Proof of Theorem 5: *Necessity.* Suppose that for every atomic constraint $X\ \theta\ Y + c$ in C', there is a path from node X to node Y in $\mathcal{G}_C$ whose compound label (θ', c') satisfies the condition $(\theta', c') \preceq (\theta, c)$. Let $X\ \theta\ Y + c$ be an atomic constraint in C'. Let p be a path from X to Y in $\mathcal{G}_C$ whose compound label (θ', c') satisfies the condition $(\theta', c') \preceq$

(θ, c). Let $X_1, \ldots, X_n$, $n \geq 1$, be the integer nodes in p from X to Y in that order (the general case is considered where at least one integer node is included in path p between nodes X and Y; the case where no integer node is included in p can be easily deduced from the general one). Each of nodes X and Y can be an integer or a real node. Let (θ_0, c_0) and (θ_n, c_n) be the compound labels of the paths from X to X_1 and from X_n to Y in p, respectively, and $(\leq, c_i)$, $i \in [1, n-1]$ be the compound labels of the integer paths from X_i to X_{i+1}. Then, as shown in the proof of Theorem 2, the atomic constraints $X\,\theta_0\,X_1 + c_0$, $X_1 \leq X_2 + c_1$, $\ldots$, $X_{n-1} \leq X_n + c_{n-1}$, $X_n\,\theta_n\,Y + c_n$ are implied by the atomic constraints represented by p. By transitivity, these atomic constraints imply the constraint $X\,\theta_p\,Y + (c_0 + \ldots + c_n)$, where θ_p is $<$, if one of θ_0, θ_n is $<$, and is $\leq$ otherwise. By definition, $\theta' = \theta_p$, and $c' = c_0 + \ldots + c_n$. Thus, the atomic constraints represented by p imply the constraint $X\,\theta'\,Y + c'$. Since $(\theta', c') \preceq (\theta, c)$, $X\,\theta'\,Y + c' \models X\,\theta\,Y + c$. Therefore, $C \models X\,\theta\,Y + c$. Consequently, C implies all the atomic constraints in C', and therefore, their conjunction (i.e. $C \models C'$).

Sufficiency. Suppose that $C \models C'$. Let $X\,\theta\,Y + c$ be an atomic constraint in C'. Then, the constraint $C \wedge \neg(X\,\theta\,Y + c)$ is unsatisfiable. The constraint $\neg(X\,\theta\,Y + c)$ is equivalent to the atomic constraint $Y\,\theta''\,X + c''$ where θ'' is $<$, if θ is $\leq$, and is $\leq$ otherwise, and $c'' = -c$. Let C'' denote the constraint $C \wedge Y\,\theta''\,X + c''$. The constraint graph $\mathcal{G}_{C''}$ is obtained by the constraint graph $\mathcal{G}_C$ by adding a directed edge e from Y to X labeled as (θ'', c''). Since C'' is unsatisfiable, by Theorem 2, there is a cycle s in $\mathcal{G}_C$ whose compound label (θ_s, c_s) satisfies the condition $(\theta_s, c_s) \prec (\leq, 0)$. Since C is satisfiable, by Theorem 2, there is no such cycle in $\mathcal{G}_C$. Therefore, cycle s includes edge e. Let p be the path in s from X to Y, and (θ_p, c_p) be its compound label.

First, suppose that s does not include any integer node. By definition, θ_s is $<$ if one of θ_p, θ'' is $<$, and is $\leq$ otherwise, and $c_s = c_p + c''$. Two cases are considered as follows:

(a) θ_s is $<$.
Since $(\theta_s, c_s) \prec (0, \leq)$, $c_p + c'' \leq 0$. Then, $c_p \leq -c''$, i.e. $c_p \leq c$.

If θ'' is $<$, then θ is $\leq$. Therefore, $(\theta_p, c_p) \leq (\theta, c)$.
If θ'' is $\leq$, then θ is $<$, and θ_p is $<$. Therefore, $(\theta_p, c_p) \leq (\theta, c)$.

(b) θ_s is $\leq$.
Since $(\theta_s, c_s) \prec (0, \leq)$, $c_p + c'' < 0$. Then, $c_p < -c''$, i.e. $c_p < c$. Therefore, $(\theta_p, c_p) \leq (\theta, c)$.

Then, suppose that s includes one or more integer nodes (node 0 can be one of them). Let p_0 be the integer path in s that includes edge e, and (θ_0, c_0) be its compound label. Let p_i, $i \in [1, n]$, $n \geq 0$ be the rest of the integer paths in s, and (θ_i'', c_i'') be their compound labels, respectively. $n = 0$ accounts for the case where there is only one integer node in s, and consequently, there is only one integer path in s. Let e_j, $j \in [1, m]$, $m \geq 0$ be the edges, other than edge e, in p_0, and (θ_j, c_j) be their labels, respectively. $m = 0$ accounts for the case where X and Y are integer nodes, and consequently, e is the only edge in the integer path p_0. By definition,

$c_p = c_1'' + \ldots + c_m'' + c_1 + \ldots + c_n$. (1)

By definition, θ_s is $\leq$, and $c_s = c_0 + c_1 + \ldots + c_n$. Since $(\theta_s, c_s) \prec (0, \leq)$, $c_0 + c_1 + \ldots + c_n < 0$. Since $c_0 + c_1 + \ldots + c_n$ is an integer, $c_0 + c_1 + \ldots + c_n \leq -1$. (2)

Two cases are considered as follows:

(a) One of θ'', θ_i'', $i \in [1, m]$, is $<$, and $c'' + c_1'' + \ldots + c_m''$ is an integer.
Then, by definition, $c_0 = c'' + c_1'' + \ldots + c_m'' - 1$. By (2), $c'' + c_1'' + \ldots + c_m'' + c_1 + \ldots + c_n \leq 0$. Therefore, $c_1'' + \ldots + c_m'' + c_1 + \ldots + c_n \leq -c''$, i.e. $c_1'' + \ldots + c_m'' + c_1 + \ldots + c_n \leq c$.
By (1), $c_p \leq c$.
If θ'' is $<$, then θ is $\leq$. Therefore, $(\theta_p, c_p) \leq (\theta, c)$.
If θ'' is $\leq$, then θ is $<$, and θ_p is $<$. Therefore, $(\theta_p, c_p) \leq (\theta, c)$.

(b) All of θ'', θ_i'', $i \in [1, m]$, are $\leq$, or $c'' + c_1'' + \ldots + c_m''$ is not an integer.
Then, by definition, $c_0 = \lfloor c'' + c_1'' + \ldots + c_m'' \rfloor$. By (2), $c'' + c_1'' + \ldots + c_m'' + c_1 + \ldots + c_n < 0$. Thus, $c_1'' + \ldots + c_m'' + c_1 + \ldots + c_n < -c''$. Using (1), $c_p < c$. Therefore, $(\theta_p, c_p) \leq (\theta, c)$.

Presented below is an algorithm for checking the implication of mixed arithmetic constraints that do not involve $\neq$. These constraints need not necessarily satisfy the previous restrictive assumptions of this section.

Algorithm 3 Implication Checking

```
Input: Two mixed arithmetic constraints $C$ and $C'$ that do not involve $\neq$
Output: A decision on whether $C \models C'$ or not
    {* Step 1: Remove trivial atomic constraints. *}
 1: If $C$ contains an unsatisfiable atomic constraint of the form $X\,\theta X + c$ then return $C \models C'$.
    Remove from $C'$ all the valid atomic constraints of the form $X\,\theta X + c$. If all the atomic constraints of $C'$ are valid constraints return $C \models C'$.
 2: If $C'$ contains an unsatisfiable atomic constraint of the form $X\,\theta X + c$ then return $C \not\models C'$.
    Remove from $C$ all the valid atomic constraints of the form $X\,\theta X + c$. If all the atomic constraints of $C$ are valid constraints return $C \not\models C'$.
    {* Step 2: Put $C$ and $C'$ in standard form. *}
 3: Put $C$ and $C'$ in standard form as in step 2 of Algorithm 2. Let $C_s$ and $C'_s$, respectively be the resulting constraints.
    {* Step 3: Construct $\mathcal{G}_{C_s}$. *}
 4: Construct the constraint graph $\mathcal{G}_{C_s}$ representing $C_s$.
    {* Step 4: Compute minimal compound labels. *}
 5: For each pair of nodes $(X, Y)$ in $\mathcal{G}_{C_s}$, compute the minimal compound label of the paths from $X$ to $Y$.
    {* Step 5: Check implication. *}
 6: If for each atomic constraint $X\,\theta\,Y + c$ in $C'_s$ there is a path from $X$ to $Y$ in $\mathcal{G}_{C_s}$ and the minimal compound label $(\theta', c')$ of the paths from $X$ to $Y$ satisfies the condition $(\theta', c') \preceq (\theta, c)$, return $C \models C'$.
    Otherwise return $C \not\models C'$.
```

Let N_C ($N_{C'}$) denote the number of atomic constraints, and N_N ($N_{N'}$) denote the number of distinct variables in C (C'). Then, the following theorem about the complexity of algorithm 3 can be stated.

Theorem 6. *The implication problem of mixed arithmetic constraints that do not involve $\neq$ can be solved in $\mathcal{O}(N_N^3 + N_{C'})$ time.*

Proof: Steps 1 and 2 of Algorithm 3 can be done in one scan of C and C' which takes $\mathcal{O}(N_C + N_{C'})$ time. The number of conjuncts in the resulting constraints C_s and C_s' is bounded by $\mathcal{O}(N_C)$ and $\mathcal{O}(N_{C'})$, respectively. Because both $\mathcal{G}_C$ and $\mathcal{G}_C'$ are connected graphs, N_N and $N_{N'}$ are bounded by $\mathcal{O}(N_C)$ and $\mathcal{O}(N_{C'})$, respectively. The test is step 3 can be done in $N_N + N_{N'}$ time. N_N is bounded by $\mathcal{O}(N_N)$, otherwise algorithm 3 returns $C \not\models C'$ in step 3. The construction of $\mathcal{G}_{C_s}$ can be done, in step 3, in $\mathcal{O}(N_C)$ time. The number of nodes in $\mathcal{G}_{C_s}$ is the number of distinct variables in C (plus 1 if the node 0 appears in $\mathcal{G}_{C_s}$). The minimal compound label for every pair of nodes in step 4 can be computed by the Floyd-Warshall algorithm in $\mathcal{O}(N_N^3)$ time. Thus, step 4 can be done in $\mathcal{O}(N_N^3)$. Since, n is bounded by N_C, step 4 is also bounded by $\mathcal{O}(N_C^2)$. Step 5 can be done in $\mathcal{O}(N_{C'})$ time since comparing each atomic constraint in C_s' takes constant time, and the number of atomic constraints in C_s' is bounded by $\mathcal{O}(N_{C'})$. As usual, it is assumed that accessing a node in the adjacency structure used to represent the graph $\mathcal{G}_{C_s}$ takes constant time. Therefore, the complexity of Algorithm 3 is $\mathcal{O}(N_C^3 + N_{C'})$.

The results of this section generalize to the mixed arithmetic constraint case previous results on the implication of arithmetic constraints where all the variables range exclusively over the integers [138, 127, 63, 64] or exclusively over the reals [89, 138, 63, 64].

4.7 Computing the Ranges of Variables in Constraints

The range of a variable X in a constraint C is a set of maximal intervals of values from the domain of X. Intuitively, these are the values X is restricted to by C. In this section, how to efficiently compute the range of a variable is studied. As the start, a formal definition is provided.

Definition 7. Given a constraint C, the *range of a variable X in C*, R_X^C, is the set of all the intervals I of values from the domain of X (that is, integer values, if X is an integer variable, and real values, otherwise) such that:

(a) For every $c \in I$, $C \wedge X = c$ is satisfiable. In other words, if any occurrence of X in C is replaced by a value $c \in I$, the resulting constraint is satisfiable.

(b) I is maximal with respect to set inclusion on the set of intervals satisfying condition (a).

Clearly, the range of a variable in a constraint C is empty if and only if C is unsatisfiable. The upper and lower bounds of an interval can be $+\infty$ and $-\infty$, respectively. For intervals of integer variables, convention similar to that adopted for the compound labels of integer paths is followed: the bound values are integers and the bounds are closed (unless they are $+/-\infty$). This is feasible after the remarks of Section 3. By the previous definition, the range of a variable in a valid constraint contains the unique interval $(-\infty, +\infty)$.

When $\neq$ is allowed to occur in a constraint C, the range of a (real or integer) variable in C can contain more than one interval.

Example 10. Consider the constraint, $X > 1 \wedge X \leq 5.1 \wedge X \neq 3$. Clearly, if X is a real variable, its range is the set of the two intervals $(1, 3)$ and $(3, 5.1]$. If X is an integer variable, its range is the set of the two intervals $[2, 2]$ and $[4, 5]$.

Consider also the constraint $X \geq 2 \wedge X \leq 10 \wedge X < Y + 3 \wedge Y < 2.5$. If X is a real and Y is an integer variable, it is not difficult to see that the ranges of X and Y are the intervals $[2, 5.5)$ and $(-\infty, 2]$, respectively.

Computing the range of a variable in a constraint that involves $\neq$ is not polynomial, as the next theorem shows.

Theorem 7. *Deciding whether the range of a variable in a constraint equals a given range is NP-hard.*

Proof: From Theorem 1, checking the satisfiability of a constraint C is NP-hard. Let the given range R be the empty set. Then, C is unsatisfiable if and only if the range of a variable in C equals R. Thus, checking satisfiability of a constraint is reducible to deciding whether the range of a variable in a constraint equals a given range.

In order to find efficient ways of computation of ranges of variables, in the rest of the section, the attention is restricted to constraints than do not involve $\neq$.

As the previous example indicates and the next lemma proves, when $\neq$ does not occur in a satisfiable constraint C, the range of a variable in C contains a unique interval.

Lemma 4.7.1. *The range of a variable in a satisfiable constraint that does not involve unequalities ($\neq$) contains a unique interval.*

Proof: Let C be a satisfiable constraint that does not involve $\neq$. Suppose that the range of a variable X in C includes more than one intervals. Since the intervals in R_X^C are maximal with respect to set inclusion (definition 7), they cannot have common values. If C is valid, R_X^C contains the unique interval $(-\infty, +\infty)$, a contradiction. Suppose then that C is not valid. Let I_1 and I_2 be two intervals in R_X^C such that each value in I_1 is less than a value I_2. Consider two values c_1 and c_2 such that $c_1 \in I_1$ and $c_2 \in I_2$. Since, by definition 7, I_1 and I_2 are maximal, there is a value c such that: (a) c is an integer if X is an integer variable, and is a real otherwise, (b) c does not belong to any interval in R_X^C, (c) $c_1 < c < c_2$. Let C_1, C_2, and C' denote the constraints $C \wedge X = c_1$, $C \wedge X = c_2$, and $C \wedge X = c$, respectively, in standard form (these constraints can be equivalently put in standard form since they are not trivially unsatisfiable or valid). The constraint graph $\mathcal{G}_{C_1}$ of C_1 is constructed by adding to $\mathcal{G}_C$ an edge from node X to node 0 labeled as $(\leq, c_1)$, and an edge from node 0 to node X labeled as $(\leq, -c_1)$. The constraint graphs $\mathcal{G}_{C_2}$ and $\mathcal{G}_{C'}$ are constructed analogously. Since $c_1 \in I_1$ and $c_2 \in I_2$, C_1 and C_2 are satisfiable. By theorem 2, there is no cycle in $\mathcal{G}_{C_1}$ or in $\mathcal{G}_{C_2}$ whose compound label (θ', c') satisfies the condition $(\theta', c') \preceq (\leq, 0)$. Therefore, there is no such cycle in $\mathcal{G}_{C'}$ neither. By theorem 2, C' is satisfiable, that is, c belongs to an interval in R_X^C. This is a contradiction.

Based on the previous lemma, in the remainder of this section, the range of a variable with the unique interval it contains is confounded.

A constraint of the form $X \theta\, c$, $\theta \in \{\leq, <, >, \geq\}$, is called *range constraint*. The next lemma provides necessary and sufficient conditions for a value to be the upper (or lower) bound of a range in terms of range constraint implication.

Lemma 4.7.2. Let X be a variable of a satisfiable constraint C that does not involve $\neq$. Then:

(a) c is the closed upper bound value of the range of X in C if and only if $C \models X \leq c$ and there is no value $c' \leq c$ such that $C \models X < c'$.

(b) c is the open upper bound value of the range of X in C if and only if $C \models X < c$ and there is no value $c' < c$ such that $C \models X \leq c'$. (This case applies only to a real variable X.)

(c) $+\infty$ is the upper bound of the range of X in C if and only if there is no value c such that $C \models X \leq c$.

Values c and c' above are chosen from the domain of X.

Analogous statements hold for the lower bound of the range of X in C.

Proof: (a) *Necessity.* Suppose that $C \models X \leq c$, and there is no value $c' \leq c$ such that $C \models X < c'$. Then, since C is satisfiable, $C \wedge X = c$ is satisfiable. Since $C \models X \leq c$, there is no value $c'' > c$ such that $C \wedge X = c''$ is satisfiable. Therefore, c is the closed upper bound value of the range of X in C.

Sufficiency. Suppose that c is the closed upper bound of the range of X in C. Then,

$C \wedge X = c$ is satisfiable, and $C \models X \leq c$. Since, $C \wedge X = c$ is satisfiable, there is no value $c' \leq c$ such that $C \models X < c$.

(b) *Necessity.* Suppose that $C \models X < c$, and there is no value $c' < c$ such that $C \models X \leq c'$. Then, since C is satisfiable, there is a value c'' such that $c'' < c$ and $C \wedge X = c''$ is satisfiable. Since $C \models X < c$, there is no value $c'' \geq c$ such that $C \wedge X = c''$ is satisfiable. Therefore, c is the open upper bound value of the range of X in C.
Sufficiency. Suppose that c is the open upper bound value of the range of X in C. Then, $C \models X < c$. Let c' be a value such that $c' < c$ and $C \models X < c$. Since X is a real variable, there is a value c'' such that $c' < c'' < c$, and $C \wedge X = c''$ is satisfiable. This is a contradiction. Therefore, there is no value $c' < c$ such that $C \models X \leq c'$.

(c) *Necessity.* Suppose that there is no value c such that $C \models X \leq c$. Since C is satisfiable, there is a value c' such that $C \wedge X = c'$ is satisfiable. Since for no value c, $C \models X \leq c$, for every value $c' \geq c$ such that $C \wedge X = c'$ is satisfiable, there is a value $c'' > c'$ such that $C \wedge X = c''$ is satisfiable. Therefore, $+\infty$ is the upper bound of the range of X in C.
Sufficiency. Suppose that $+\infty$ is the upper bound of the range of X in C. Then, clearly there is no value c such that $C \models X \leq c$.

The necessary and sufficient conditions in terms of range constraint implication of lemma 4.7.2 are translated to conditions over a constraint graph in the following theorem. This form facilitates their algorithmic use later on in this section.

A constraint in the following theorem is considered to be in standard form. A satisfiable constraint can be put in standard form if it is not valid. Validity can be checked in linear time (Section 5). Obviously, if C is valid, the range of every variable occurring in it is $(-\infty, +\infty)$.

Theorem 8. Let X be a variable of a satisfiable constraint C in standard form that does not involve $\neq$. Then:

(a) c is the closed upper bound value of the range of X in C if and only if the minimal compound label of the paths from node X to node 0 in $\mathcal{G}_C$ is $(c, \leq)$.

(b) c is the open upper bound value of the range of X in C if and only if the minimal compound label of the paths from node X to node 0 in $\mathcal{G}_C$ is $(c, <)$.(This case applies only to a real variable X.)

(c) $+\infty$ is the upper bound of the range of X in C if and only if there is no path from node X to node 0 in $\mathcal{G}_C$.

Analogous statements hold for the lower bound of the range of X in C.

Proof: The proof follows by theorem 5 and lemma 4.7.2.

Example 11. Consider the constraint graph $\mathcal{G}_C$ shown in Figure 4.1. Suppose that the range of the real variable X_1 in C is to be computed. There are exactly two paths from X_1 to 0 in $\mathcal{G}_C$. Their compound labels are $(<, -2.5)$ and $(<, -6.5)$. There is no path from 0 to X_1 in $\mathcal{G}_C$. Therefore, according to Theorem 8, the range of X_1 in C is $(-\infty, -6.5)$.

Suppose also that the range of the integer variable Y_1 in C is to be computed. There is exactly one path from Y_1 to 0 and from 0 to Y_1. Their compound labels are $(\leq, -1)$ and $(\leq, 4)$, respectively. Therefore, according to Theorem 8, the range of Y_1 in C is $[-4, -1]$.

The following algorithm summarizes the previous analysis for the range of a variable in a constraint that does not involve $\neq$. This constraint is not required to be in standard form.

The next theorem characterizes the complexity of Algorithm 4. Let N_C denote the number of atomic constraints in C.

Theorem 9. *Computing the range of a variable in a constraint that does not involve $\neq$ can be done in $\mathcal{O}(N_C log N_N)$.*

Proof: Steps 1 and 2 of Algorithm 4 can be done in $\mathcal{O}(N_C)$ time as in Algorithm 2. The number of conjuncts in the resulting constraint C_s is also bounded by $\mathcal{O}(N_C log N_N)$. The minimal compound labels in step 3 can be computed using Dijkstra algorithm twice

Algorithm 4 Computing the range of a variable in a constraint

```
Input: A constraint $C$ that does not involve $\neq$, and a variable $X$ in $C$
Output: The range $R_X^C$ of $X$ in $C$
    {* Step 1: Remove trivial atomic constraints. *}
 1: Remove from $C$ all the valid atomic constraints of the form $Y\,\theta Y + c$
 2: If $C$ contains an unsatisfiable atomic constraint of the form $Y\,\theta Y + c$, return $R_X^C$ is empty
 3: If all the atomic constraints of $C$ are valid constraints, return $R_X^C$ is $(-\infty,\ +\infty)$
    {* Step 2: Construct $\mathcal{G}_{C_s}$. *}
 4: Put $C$ in standard form as in step 2 of Algorithm 2, Let $C_s$ be the resulting constraint. Construct the constraint graph $\mathcal{G}_{C_s}$ representing $C_s$.
    {* Compute minimal compound labels. *}
 5: Compute the minimal compound labels $(\theta, c)$ and $(\theta', c')$ of the paths from 0 to $X$ and from $X$ to 0, respectively.
    {* Compute the range of $X$ in $C$. *}
 6: if there is no path between 0 and $X$ then
 7:   return $R_X^C$ is $(-\infty,\ +\infty)$
 8: else if there is a path from 0 to $X$ and there is no path from $X$ to 0 then
 9:   return $R_X^C$ is $(-c,\ +\infty)$ if $\theta$ is $<$, return $R_X^C$ is $[-c,\ +\infty)$ if $\theta$ is $\leq$
10: else if there is a path from $X$ to 0 and there is no path from 0 to $X$ then
11:   return $R_X^C$ is $(-\infty,\ c')$ if $\theta'$ is $<$, return $R_X^C$ is $(-\infty,\ c']$ if $\theta'$ is $\leq$
12: else if there is a path from 0 to $X$ and a path from $X$ to 0 then
13:   return $R_X^C$ is $(-c,\ c')$ if $\theta$ and $\theta'$ are $<$, return $R_X^C$ is $[-c,\ c')$ if $\leq$ and $\theta'$ is $<$, return $R_X^C$ is $(-c,\ c']$ if $\theta$ is $<$ and $\theta'$ is $\leq$, return $R_X^C$ is $[-c,\ c']$ if $\theta$ and $\theta'$ are $\leq$
14: end if
```

from node 0 on $\mathcal{G}_C$ and on the graph obtained by reversing the edges of $\mathcal{G}_C$. Each one takes $\mathcal{O}(N_C log N_N)$ time. Thus, step 3 can be done in $\mathcal{O}(N_C log N_N)$. Step 4 takes constant time. Therefore, the complexity of Algorithm 4 is $\mathcal{O}(N_C log N_N)$.

The results of this section generalize to the mixed arithmetic constraint case the results of [63] on the computation of the range of a variable in an arithmetic constraint that is a conjunction of atomic arithmetic constraints of the form $X \theta Y$ (without offset) or $X \theta\, c$, and the variables range exclusively over the integers or exclusively over the reals.

4.8 Conclusion

Arithmetic constraints with inequalities are very important in different areas of databases. Problems related to arithmetic constraints have been studied in the past when all the variables (attributes) range exclusively over the integers or exclusively over the reals. In order to deal with constraints involving variables of different type (real or integer), very common in databases, and with arithmetic constraints involving aggregate functions, the mixed arithmetic constraints is introduced where variables are allowed to range simultaneously over the integers or over the reals. The satisfiability and implication problems for conjunctive mixed arithmetic constraints and the problem of computing ranges of variables in a mixed arithmetic constraint are studied. It is shown in this chapter, as expected, these problems are NP-hard when unequality ($\neq$) is allowed to occur in the constraints. Using a graph representation for fixed arithmetic constraints that accounts for both types of variables, necessary and sufficient conditions for all three problems when $\neq$ is not allowed to occur in the constraints are provided. Finally, polynomial algorithms for the three problems are provided and their corresponding complexity .

A motivation (and extension direction) of this work is the exploitation of these results in studying the satisfiability and implication problems for arithmetic constrains involving aggregate functions. These problems are hard or undecidable and the present work can be used to derive specific subclasses of aggregation arithmetic constraints where they can be solved efficiently.

CHAPTER 5

CONSTRUCTING A SEARCH SPACE FOR THE VIEW SELECTION PROBLEM

Deciding which views to materialize is an important problem in the design of a Data Warehouse. Solving this problem requires generating a space of candidate view sets from which an optimal or near-optimal one is chosen for materialization. This chapter addresses the problem of constructing this search space. This is an intricate issue because it requires detecting and exploiting common subexpressions among queries and views. The approach described in this chapter suggests adding to the alternative evaluation plans of multiple queries views called closest common derivators (CCDs) and rewriting the queries using CCDs. A CCD of two queries is a view that is as close to the queries as possible and that allows both queries to be (partially or completely) rewritten using itself. CCDs generalize previous definitions of common subexpressions. Using a declarative query graph representation for queries, both necessary and sufficient conditions for a view to be a CCD of two queries are provided. The results are exploited to describe a procedure for generating all the CCDs of two queries and for rewriting the queries using each of their CCDs.

5.1 Introduction

One of the most important issues in Data Warehousing is the design of a Data Warehouse. This issue can be abstractly modeled as a problem (called materialized view selection problem) that takes as input a set of queries and a number of cost-determining parameters (e.g., query frequencies or source relation change propagation frequencies). The output is a set of views to materialize in the data warehouse that minimizes a cost function (e.g., query evaluation cost, view maintenance cost, or their combination) and satisfies a number of constraints (e.g., storage space restrictions or view maintenance cost constraints) [130]. Solving the materialized view selection problem involves addressing two main tasks: (1) generating a search space of alternative view sets for materialization, and (2) designing optimization algorithms that select an optimal or near-optimal view set from the search space. There is a substantial body of work dealing with the second task, often suggesting elaborate greedy, heuristic or randomized optimization algorithms [75, 13, 147, 70, 125, 71, 90, 134, 93, 84]. However, these approaches assume that the search space is available, usually in the form of multiple common evaluation plans for all the input queries represented as an AND/OR graph (for the case of general queries) or in the form of multi-dimensional lattices of views (for the case of star grouping/aggregation queries). Even though the construction of multidimensional lattices is straightforward [75, 13, 125], the construction of alternative common plans for multiple general queries is an intricate issue. The reason is that common sub-expressions [80] among the input queries need to be detected and exploited, and the queries need to be rewritten using these common subexpressions. In this chapter, the problem of constructing a search space for materialized view selection is addressed. Clearly, this is an important problem since, if the search space does not include a view selection that is close to the optimal, the optimization algorithms will fail to produce satisfactory results.

5.1.1 Previous Work

The need to construct evaluation plans for multiple queries by exploiting common subexpressions first appeared in the area of multi-query optimization [121]. The goal there is to determine a global evaluation plan (a plan for all the queries together) that is more efficient to evaluate than evaluating separately the optimal evaluation plan of each query. Initially "common subexpression" referred to identical or equivalent expressions [49]. Subsequently the term included subexpression subsumption [80, 21, 121]. Later, commonalities between the queries included the possibility for overlapping selection conditions [30]. A typical approach for constructing evaluation plans for multiple queries is to proceed in a bottom-up way (from the base relations to the roots of the plans) and to combine nodes from different plans that are defined over the same set of base relations. Equivalent nodes are merged, nodes related through subsumption are linked with selection operations, and overlapping nodes are linked through new nodes that are able to compute the overlapping nodes. The disadvantage of this approach is that all the alternative plans of the input queries need to be considered in order to guarantee that the optimal view selection is included in the search space. Clearly, it is not feasible to generate and combine all the alternative plans of the input queries. This approach is followed in [10] where merge rules are suggested for plans of queries involving selection, projection, join and grouping aggregation operations. In this chapter, however, node combination is restricted to view subsumption (node overlapping is ignored) and there are no results as to the completeness of the suggested rules. [56, 55] propose an algorithm to determine a set of candidate views for materialization for queries and views that are nested expressions involving selection, projection, join and grouping/aggregation operations. New views are generated in the search space by computing the ancestor of two expressions, i.e. the coarsest view using which both expressions can be completely rewritten [96, 36]. This approach is restricted to nested queries and views involving star joins over multidimensional star schemas. [133, 131] provide transformation rules for generating candidate views for materialization from a set of input select-project-join queries. The transformation rules are used by heuristic optimization algorithms that prune the space of views generated in a cost-based fashion. A restriction of this work is that self-joins are not allowed in queries and views.

5.1.2 Contributions

The approach presented in this chapter for creating a search space for view selection consists in adding to the alternative evaluation plans of multiple queries views called closest common derivators and in rewriting the queries using these closest common derivators. A closest common derivator of two queries is a view that is as close to the queries as possible and that allows both queries to be (partially or completely) rewritten using itself. Intuitively, a closest common derivator does not involve join operations that are not necessary for computing both queries. A traditional query optimizer can be used to generate alternative evaluation plans for a closest common derivator and to choose the cheapest one among them. It can also be used to access the cost of computing each of the queries using a closest common derivator by viewing this last one as a base relation. A view selection optimization algorithm can choose to materialize a closest common derivator and/or any of the nodes (views) in its alternative evaluation plans in a cost-based manner. It is worth noting that a closest common derivator is defined using the definitions of the queries, independently of any of their evaluation plans. Further, a closest common derivator can be determined for queries that involve several occurrences of the same relation (self-joins) and are not defined over the same set of base relations.

The main contributions of the chapter are the following:

- A common derivator of two queries as a view that can be used to rewrite both queries is defined. Based on a closeness relationship on common derivators, the concept of closest common derivator is formally introduced. This concept generalizes previous definitions of common subexpressions since the queries can be not only completely but also partially rewritten using a closest common derivator.
- This chapter shows under what conditions selection and join predicates can be merged and how their merge can be computed. The merge of two predicates is the least restrictive predicate that is implied by both predicates, and it is used in the computation of closest common derivators of queries.
- A declarative graph-based representation of queries and views (query graphs) is introduced and both necessary and sufficient conditions for a query graph to be a closest common derivator of two queries are provided in terms of query graphs.
- The previous results are used to outline a procedure for computing all the closest common derivators of two queries. This procedure can be used within a view selection algorithm that can decide in a cost-based manner which closest common derivators to generate and which ones to exclude from generation, thus pruning the search space of alternative view sets for materialization.

5.2 Class of Queries Considered and Initial Definitions

5.2.1 Query Language

Queries and views considered in this chapter involves selections, projections, join and relation renaming operations under set semantics. Base relations can have multiple occurrences in a query. It is assumed that if a relation has more than one occurrence in a query, every one of its occurrences is renamed with a distinct name. The notation $R_i[R]$ denotes an occurrence of a relation R renamed R_i. The *name* of a relation occurrence in a query is its new name, in case it is renamed, or its original name otherwise. Since the names of relation occurrences in a query are distinct, relation occurrences in a query are referred to by their name. Attributes range over dense totally ordered domains (e.g., real or rational numbers or string spaces) [1]. Relation restrictions in queries are conditions formed as conjunctions of selection and join atomic conditions. A *join atomic condition* is an expression of the form $A \ \theta \ B + c$ where A and B are relation attributes from different relation occurrences, c is an integer or real value, and $\theta \in \{<, \leq, =, \geq, >\}$. A *selection atomic condition* is an expression of the form $A \ \theta \ B + c$ or of the from $A \ \theta \ c$ where A and B are relation attributes of the same relation occurrence, and c and θ are as before.

It is well known that this type of queries can be represented by relational algebra expressions of the form $\Pi_X(\sigma_C(\mathbf{R}))$ where $\mathbf{R}$ is the Cartesian product of a number of relation occurrences; σ_C denotes the selection operator with C being a condition involving attributes from the relations in $\mathbf{R}$; Π_X denotes a projection operator with X being a nonempty set of attributes from the relations in $\mathbf{R}$ (*projected attributes* of the query.)

5.2.2 Query Containment and Equivalence

Query Q_1 *contains* query Q_2 (denoted $(Q_1 \sqsubseteq Q_2)$ if there is a renaming of the projected attributes of Q_1 (or Q_2) that makes the answers of the two queries have the same schema and, for every instance of the base relations, the instance of the answer of Q_1 under the

[1]However, the results in this chapter equally apply to attributes ranging over discreet totally ordered domains (e.g., integers)

common schema be a subset of the instance of the answer of Q_2. Two queries Q_1 and Q_2 are *equivalent* (denoted $Q_1 \equiv Q_2$) if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$.

5.2.3 Class of Queries and Views

It is assumed that queries and views are not equivalent to a query of the form $Q_1 \times Q_2$, where Q_1 and Q_2 are queries, and $\times$ is the Cartesian product operator. "Pure" Cartesian products are rarely used in practice and can generate huge relations. If necessary, a query $Q_1 \times Q_2$ can be represented by the two subqueries Q_1 and Q_2.

5.2.4 Query Rewritings

A *rewriting* of a query Q is a query equivalent to Q. A *rewriting Q' of a query Q using a view V* is a query that references V and possibly base relations such that replacing V in Q' by its definition results in a query equivalent to Q. Query rewritings using views can also involve attribute renaming operations. If there is a rewriting of Q using V, then Q can be rewritten using V and this is denoted using $Q \vdash V$. A *complete rewriting of a query Q using a view V* is a rewriting of Q that references only V (and no base relations.) If there is a complete rewriting of Q using V then Q can be completely rewritten using V and this is denoted as $Q \Vdash V$. A rewriting of Q using V that is not a complete rewriting of Q using V is called *partial rewriting of Q using V*. Clearly, a partial rewriting references also base relations.

5.2.5 Simple Rewritings

A simple rewriting is introduced to describe a rewriting of a query using views where each view has a single occurrence in the rewriting.

Definition 8. A rewriting Q' of a query Q using a view V is *simple* if view V has a single occurrence in Q'.

If there is a rewriting of a query Q using a view V, then there is also a simple rewriting of Q using V. This statement is not necessarily true for complete rewritings. It is possible that there is no complete simple rewriting of a query Q using a view V even if there is a complete rewriting of Q using V.

Example 12. Consider the query $Q = R_1[R] \bowtie_{R_1.A<R_2.B} R_2[R]$ and the view $V = R_3[R]$. The query $Q' = V_1[V] \bowtie_{V_1.A<V_2.B} V_2[V]$ is a complete rewriting of Q using V. Clearly, there is no simple complete rewriting of Q using V.

In the following of this chapter *only simple rewritings are considered*, and 'rewriting' means 'simple rewriting'. This assumption concerns both complete and partial rewritings. Based on the previous definitions the following proposition can be shown.

Proposition 5.2.1. Let Q_1 and Q_2 be two queries. If $Q_1 \Vdash Q_2$ and $Q_2 \Vdash Q_1$, $Q_1 \equiv Q_2$.

5.2.6 Minimal Rewritings

It is also interesting to consider rewritings where the computation done in the view is not repeated when evaluating the query using the view materialization. To this end minimal rewritings are introduced.

Definition 9. A rewriting Q' of a query Q using a view V is *minimal* if for every relation R that has $n,\ n > 0$, occurrences in Q, R has $k,\ 0 \leq k \leq n$, occurrences in V and $n - k$ occurrences in Q'. If there is a minimal rewriting of Q using V then Q can be minimally rewritten using V, and this can be denoted as $Q \vdash_m V$.

Example 13. Consider the query $Q = \Pi_{R_1.A}(R_1[R] \bowtie_{R_1.A<R_2.B} R_2[R] \bowtie_{R_2.C<R_3.D} R_3[R])$ and the view $V = \Pi_{R_4.A,\, R_5.B,\, R_5.C}(R_4[R] \bowtie_{R_4.A\leq R_5.B} R_5[R])$.
$Q' = \Pi_{V.A}(V \bowtie_{V.A<R_2.B} R_2[R] \bowtie_{R_2.C<R_3.D} R_3[R])$ is a rewriting of Q using V that is not minimal: relation R has three occurrences in Q, two in V and two in Q'. In contrast, $Q'' = \Pi_{V.A}(\sigma_{V.A<V.B}(V) \bowtie_{V.C<R_3.D} R_3[R])$ is a minimal rewriting of Q using V.

Roughly speaking, under the assumptions about redundant relation occurrences stated below, a minimal rewriting of a query Q using a view V has the minimum number of base relation occurrences among all the rewritings of Q using V. Notice that there might not exist a minimal rewriting of a query Q using a view V even if a rewriting of Q using V exists. This can happen if the attributes required for a minimal rewriting are not projected out in the view.

Example 14. Consider the query $Q = \Pi_{R_1.A}(R_1[R] \bowtie_{R_1.A<R_2.B} R_2[R] \bowtie_{R_2.C<R_3.D} R_3[R])$ of Example 13 above and the view $V' = \Pi_{R_4.A,R_5.B}(R_4[R] \bowtie_{R_4.A\leq R_5.B} R_5[R])$. View V' is similar to view V of Example 13 with the exception of attribute $R_5.C$ which is not projected out in V'. Query $Q' = \Pi_{V'.A}(V' \bowtie_{V'.A<R_2.B} R_2[R] \bowtie_{R_2.C<R_3.D} R_3[R])$ is a rewriting of Q using V' that is not minimal. One can see that there is no minimal rewriting of Q using V' since attribute $R_5.C$ required to join V' with R_3 is not projected out in V'.

A minimal rewriting of a query Q using a view V may not exist even if a complete rewriting of Q using V exists. In this case, the attributes required for a minimal rewriting are projected out in the view V. Therefore, what prevents the existence of a minimal rewriting is the number of occurrences of a relation in view V which may be greater than that of the same relation in query Q.

Example 15. Consider the query $Q = \Pi_{R_1.A}(R_1[R])$, which has a single occurrence of R, and the view $V = \Pi_{R_2.A,R_3.C}(R_2[R] \bowtie_{R_2.B\leq R_3.B} R_3[R])$, which has two occurrences of R. Query $Q' = \Pi_{V.A}(V)$ is a complete rewriting of Q using V. Clearly, Q cannot be minimally rewritten using V.

The next proposition shows that queries that can be minimally rewritten using each other are equivalent.

Proposition 5.2.2. Let Q_1 and Q_2 be two queries. If $Q_1 \vdash_m Q_2$ and $Q_2 \vdash_m Q_1$, $Q_1 \equiv Q_2$.

5.2.7 Redundant Relation Occurrences

The definition of minimal rewritings above introduces some dependency on the syntax of queries and views: a minimal rewriting of a query Q using a view V might not exist even if a minimal rewriting of an equivalent query Q' using V exists. Similarly, a minimal rewriting of a query Q using a view V might not exist even if a minimal rewriting of an Q using a view V' equivalent to V exists. The reason is that query Q and/or view V may contain redundant relation occurrences. This dependency is refuted below after formally defining redundant relation occurrences.

Definition 10. A relation R has a *redundant* occurrence in a query Q, if Q can be rewritten with fewer occurrences of R.

Example 16. Consider the query $Q = \Pi_{R_1.A}(R_1[R] \bowtie_{R_1.B \leq R_2.B} R_2[R] \bowtie_{R_2.C \leq R_3.C} R_3[R])$. Query Q has two redundant occurrences of relation R: the query $Q' = \Pi_{R_4.A}(R_4[R])$, which has a single occurrence of R, is equivalent to Q. Consider also the view $V = \Pi_{R_5.A, R_6.C}(R_5[R] \bowtie_{R_5.B \leq R_6.B} R_6[R])$. It is not difficult to see that view V does not have any redundant relation occurrences. Query $Q'' = \Pi_{V.A}(V \bowtie_{V.C = R_3.C} R_3[R])$ is a minimal rewriting of Q using V. Clearly, Q' cannot be minimally rewritten using V.

Detecting redundant relation occurrences and minimizing queries, that is, removing redundant relation occurrences from queries, has been studied in the past for the class of queries considered here [89]. In the following it is assumed that queries and views do not contain redundant relation occurrences. This assumption guarantees that minimal rewritings are independent of the syntax of queries and views.

5.3 Closest Common Derivators of Queries

In this section the concept of closest common derivator of queries is introduced.

Definition 11. Let Q_1 and Q_2 be two queries and $\mathbf{R}_1, \mathbf{R}_2$ be two sets of relation occurrences from Q_1 and Q_2, respectively, that have the same number of relation occurrences of each relation. A *common derivator* (CD) of Q_1 and Q_2 over the respective sets $\mathbf{R}_1$ and $\mathbf{R}_2$ is a view V such that there is a minimal rewriting of Q_1 (resp. Q_2) using V that involves V and only those relation occurrences of Q_1 (resp. Q_2) that do not appear in $\mathbf{R}_1$ (resp. $\mathbf{R}_2$.)

Notice that queries Q_1 and Q_2 need not be defined over the same set of base relations, and their rewriting using V need not be complete. Clearly, V has as many relation occurrences of each base relation as they appear in $\mathbf{R}_1$ and $\mathbf{R}_2$.

Example 17. Consider the relation schemas $U(F, A)$, $R(A, B, C)$, $S(C, D)$, and $T(D, E)$, and the queries
$Q_1 = \Pi_{R_1.B, R_2.A, R_3.C}\ (U \bowtie_{U.A \leq R_1.A} R_1[R] \bowtie_{R_1.B \leq R_2.A} \sigma_{B<3}\ (R_2[R]) \bowtie_{R_2.C \leq R_3.B} \sigma_{A \geq 4 \wedge A \leq 7}$ $(R_3[R]) \bowtie_{R_3.C = S_1.C} \sigma_{D>3}\ (S_1[S]))$, and $Q_2 = \Pi_{R_4.C, R_5.A, S_3.C}\ (S_2[S] \bowtie_{S_2.C \leq R_4.C} \sigma_{B=3}\ (R_4[R])$ $\bowtie_{R_4.C = R_5.B} \sigma_{A \geq 5 \wedge A \leq 9}\ (R_5[R]) \bowtie_{R_5.C \leq S_3.C} \sigma_{D \geq 3}\ (S_3[S]) \bowtie_{S_3.D.C = T.D} T)$, Query Q_1 has three occurrences of relation R and one occurrence of relation S, while query Q_2 has two occurrences of relation R and one occurrence of relation S. These schemas and queries are used as a running example in this section.

The view
$V' = R_6[R] \bowtie_{R_6.C \leq R_7.B} R_7[R]$ is a CD of Q_1 and Q_2 over $\{R_2, R_3\}$ and $\{R_4, R_5\}$. The views
$V'' = R_6[R] \bowtie_{R_6.C \leq R_7.B} \sigma_{A \geq 3 \wedge A \leq 9}\ (R_7[R]) \bowtie_{R_7.C \leq S_4.C} S_4[S]$, $V''' = \sigma_{B \leq 3}\ (R_6[R]) \bowtie_{R_6.C \leq R_7.B}$ $\sigma_{A \geq 3 \wedge A \leq 9}\ (R_7[R]) \bowtie_{R_7.C \leq S_4.C} \sigma_{D \geq 3}\ (S_4[S])$,
$V = \Pi_{R_6.A, R_6.B, R_6.C, R_7.A, R_7.B, R_7.C, S_4.D}\ (\sigma_{B \leq 3}\ (R_6[R]) \bowtie_{R_6.C \leq R_7.B} \sigma_{A \geq 3 \wedge A \leq 9}\ (R_7[R]) \bowtie_{R_7.C \leq S_4.C}$ $\sigma_{D \geq 3}\ (S_4[S]))$ are CDs of Q_1 and Q_2 over $\{R_2, R_3, S_1\}$ and $\{R_4, R_5, S_3\}$.

Both queries have minimal rewritings using each of the views V', V'', V''' and V. For instance, Q_1 and Q_2 can be minimally rewritten using V as follows:
$Q_1' = \Pi_{R_1.B, R_6.A, R_7.C}\ (U \bowtie R_1[R] \bowtie_{R_1.B \leq V_4.R_6.A} \sigma_{R_6.B<3 \wedge R_7.C = S_4.C \wedge R_7.A \geq 4 \wedge R_7.A \leq 7 \wedge S_4.D>3}$ $(V))$, and
$Q_2' = \Pi_{R_4.C, R_7.A, S_4.C}\ (S_2[S] \bowtie_{S_2.C \leq V_4.R_6.C} \sigma_{R_6.C = R_7.B \wedge R_6.B = 3 \wedge R_6.C = R_7.B \wedge R_7.A \geq 5 \wedge R_7.A \leq 9}\ (V)$ $\bowtie T)$.

View V has two occurrences of R and one occurrence of S. Rewriting Q'_1 has one occurrence of R, while rewriting Q'_2 has one occurrence of S.

A CD of two queries can be "closer" to the queries than some other CD of the same queries. This concept of closeness is explained below with an example before providing a formal definition. Then it is used to define a closest common derivator of two queries. Roughly speaking, a closest common derivator of two queries is a CD that has as many relation occurrences as possible, and as few attributes and as few tuples in its answer as possible.

Example 18. Consider the queries and the CDs of example 17.

- The CD $V'' = \sigma_{R_6.C \le R_7.B \wedge R_7.A \ge 3 \wedge R_7.A \le 9 \wedge R_7.C \le S_4.C}\ (R_6[R] \times R_7[R] \times S_4[S])$ is closer to Q_1 and Q_2 than the CD $V' = \sigma_{R_6.C \le R_7.B}(R_6[R] \times R_7[R])$ because:

  (a) V' is defined over the relation occurrence sets $\mathbf{R}'_1 = \{R_2, R_3\}$ of Q_1 and $\mathbf{R}'_2 = \{R_4, R_5\}$ of Q_2 while V'' is defined over the relation occurrence sets $\mathbf{R}''_1 = \{R_2, R_3, S_1\}$ of Q_1 and $\mathbf{R}''_2 = \{R_4, R_5, S_3\}$ of Q_2 and clearly $\mathbf{R}'_1 \subset \mathbf{R}''_1$ and $\mathbf{R}'_2 \subset \mathbf{R}''_2$.

  (b) V' is not more restrictive than V'' on their common relation occurrences (in fact, V' is less restrictive than V''.)

  Notice that the following conditions are satisfied:

  (a) $V'' \vdash V'$.

  (b) $\sigma_{R_6.C \le R_7.B \wedge R_7.C \le S_4.C}\ (R_6[R] \times R_7[R] \times S_4[S]) \nVdash$
  $\sigma_{R_6.C \le R_7.B \wedge R_7.A \ge 3 \wedge R_7.A \le 9 \wedge R_7.C \le S_4.C}\ (R_6[R] \times R_7[R] \times S_4[S])$.

- The CD $V''' = \sigma_{R_6.B \le 3 \wedge R_6.C \le R_7.B \wedge R_7.A \ge 3 \wedge R_7.A \le 9 \wedge R_7.C \le S_4.C \wedge S_4.D \ge 3}\ (R_6[R] \times R_7[R] \times S_4[S])$ is closer to Q_1 and Q_2 than the CD V'' because:

  (a) V'' and V''' are defined over the same relation occurrence sets of Q_1 and Q_2.

  (b) V''' is more restrictive than V'' on their common relation occurrences.

In this case the following conditions are satisfied:

(a) $V''' \vdash V''$.

(b) $\sigma_{R_6.B\leq3 \wedge R_6.C\leq R_7.B \wedge R_7.A\geq3 \wedge R_7.A\leq9 \wedge R_7.C\leq S_4.C \wedge S_4.D\geq3}(R_6[R] \times R_7[R] \times S_4[S]) \Vdash$
$\sigma_{R_6.C\leq R_7.B \wedge R_7.A\geq3 \wedge R_7.A\leq9 \wedge R_7.C\leq S_4.C}(R_6[R] \times R_7[R] \times S_4[S])$.

(c) $V'' \nVdash V'''$.

- The CD $V = \Pi_{R_6.A,R_6.B,R_6.C,R_7.A,R_7.B,R_7.C,S_4.D}$
$(\sigma_{R_6.B\leq3 \wedge R_6.C\leq R_7.B \wedge R_7.A\geq3 \wedge R_7.A\leq9 \wedge R_7.C\leq S_4.C \wedge S_4.D\geq3}(R_6[R] \times R_7[R] \times S_4[S]))$ is closer to Q_1 and Q_2 than the CD V''' because:

  (a) V''' and V are defined over the same relation occurrence sets of Q_1 and Q_2.

  (b) V and V''' are equally restrictive on their common relation occurrences.

  (c) The set of projected attributes of V is a proper subset of that of V'''.

  The following conditions are satisfied:

  (a) $V \vdash V'''$.

  (b) $\sigma_{R_6.B\leq3 \wedge R_6.C\leq R_7.B \wedge R_7.A\geq3 \wedge R_7.A\leq9 \wedge R_7.C\leq S_4.C \wedge S_4.D\geq3}(R_6[R] \times R_7[R] \times S_4[S]) \Vdash$
  $\sigma_{R_6.B\leq3 \wedge R_6.C\leq R_7.B \wedge R_7.A\geq3 \wedge R_7.A\leq9 \wedge R_7.C\leq S_4.C \wedge S_4.D\geq3}(R_6[R] \times R_7[R] \times S_4[S])$.

  (c) $V''' \nVdash V$.

In general, the names of relation occurrences in two CDs that are related with a closeness relationship may be distinct even if these CDs are defined over the same relation occurrence sets.

Definition 12. Let Q_1 and Q_2 be two queries, $V = \Pi_X(\sigma_C(\mathbf{R}))$ be a CD of Q_1 and Q_2 over $\mathbf{R}_1$ and $\mathbf{R}_2$, $V' = \Pi_{X'}(\sigma_{C'}(\mathbf{R}'))$ be a CD of Q_1 and Q_2 over $\mathbf{R}'_1$ and $\mathbf{R}'_2$, and let $\mathbf{R}_1 \subseteq \mathbf{R}'_1$ and $\mathbf{R}_2 \subseteq \mathbf{R}'_2$. The CD V' is *closer* to Q_1 and Q_2 than the CD V (denoted $V' \prec_{Q_1,Q_2} V$) if the following conditions are satisfied:

(a) $V' \vdash V$.

(b) If $\sigma_{C'}(\mathbf{R}') \Vdash \sigma_C(\mathbf{R})$ then $V \not\Vdash V'$.

Clearly, two CDs of the same queries may not be related with a closeness relationship, even if they are defined over the same relation occurrence sets. Now a closest common derivator of two queries is formally defined as follows.

Definition 13. Let Q_1 and Q_2 be two queries. A *closest common derivator* (CCD) of Q_1 and Q_2 over $\mathbf{R}_1$ and $\mathbf{R}_2$ is a CD V of Q_1 and Q_2 over $\mathbf{R}_1$ and $\mathbf{R}_2$ such that there exists no CD of Q_1 and Q_2 that is closer to Q_1 and Q_2 than V.

Often, when referring to CDs and CCDs, the relation occurrence sets over which they are defined is not explicitly mentioned, if this is not necessary for accuracy.

Example 19. Consider the queries Q_1 and Q_2 and the CDs V', V'', V''' and V of Example 17. The CDs V', V'' and V''' are not CCDs of Q_1 and Q_2 since, as shown in Example 18, for each of them there is a CD of Q_1 and Q_2 which is closer to Q_1 and Q_2. However, it can be shown that V is a CCD of Q_1 and Q_2.

For queries related through a minimal rewriting the following proposition holds.

Proposition 5.3.1. Let Q_1 and Q_2 be two queries. If $Q_1 \vdash_m Q_2$ then Q_2 is a CCD of Q_1 and Q_2.

Proposition 5.3.1 does not hold for a simple rewriting, even if this is a complete rewriting. As shown in Example 15, it is possible that $Q_1 \nvdash_m Q_2$ (in which case Q_2 cannot be a CCD of Q_1 and Q_2) even if $Q_1 \Vdash Q_2$.

Two queries may have several CCDs. The relation occurrence sets of these CCDs for each query may be disjoint, overlapping or contained.

Example 20. Consider the queries Q_1 and Q_2 of Example 17. In Example 19 a CDD V of Q_1 and Q_2 over $\{R_2, R_3, S_1\}$ and $\{R_4, R_5, S_3\}$ is shown. It can be shown that view $U = \Pi_{S_5.C, S_5.D, R_8.A, R_8.B, R_8.C}(S_5[S] \bowtie_{S_5.C \leq R_8.C} R_8[R])$ is a CCD of Q_1 and Q_2 over

$\{S_1, R_3\}$ and $\{S_2, R_4\}$, respectively. Set $\{S_1, R_3\}$ is contained into $\{R_2, R_3, S_1\}$, while set $\{S_2, R_4\}$ overlaps with $\{R_4, R_5, S_3\}$.

In the next sections, a method for computing all the CCDs of two queries along with minimal rewritings of these queries using each of the CCDs is presented.

5.4 Condition Merging

This section shows how conditions can be merged. Merged conditions are used later to construct CCDs of queries.

A condition C is *consistent* if there is an assignment of values to its attributes that makes C true. A condition that is not consistent is called *inconsistent*. A condition C is *valid* if every assignment of values to its attributes makes C true. For instance, $A = A$ is a valid condition. Clearly, a condition is valid if it is a conjunction of valid atomic conditions. A condition C_1 *implies* a condition C_2 (notation $C_1 \models C_2$) if every assignment of values to the attributes of C_1 that makes C_1 true, also makes C_2 true. Two conditions C_1 and C_2 are *equivalent* (notation $C_1 \equiv C_2$) if $C_1 \models C_2$ and $C_2 \models C_1$. The complexity of the consistency and implication problems have been studied previously [116, 89, 138, 64, 63] for attributes ranging over dense totally ordered domains and for attributes ranging over the integers. Both problems have been shown to be polynomial for the class of conditions considered here.

A selection operation with an inconsistent condition returns an empty relation, while a selection operation with a valid condition returns all the tuples of the input relation. In the following, it is assumed that query conditions are consistent and not valid. Further, it is also assumed that a query condition C does not include valid atomic conditions (otherwise, the condition resulting by removing from C all the valid atomic conditions which is a condition equivalent to C can be considered.)

Definition 14. Two conditions C_1 and C_2 are *mergeable* if there is a non-valid condition C such that $C_1 \models C$ and $C_2 \models C$ and there exists no condition C', $C' \not\equiv C$, such that $C_1 \models C'$, $C_2 \models C'$ and $C' \models C$. Condition C is called a *merge* of C_1 and C_2.

-5.1
0
6
7
9
A
$A > -5.1 \wedge A \leq 6$
$A \geq 7 \wedge A < 9$
(a)
-5.1
0
9
A
$A > -5.1 \wedge A < 9$
(b)

Figure 5.1 (a) Conditions C_3 and C_4 (b) $merge(C_3, C_4)$.

A merge of two conditions C_1 and C_2 is unique (otherwise, the conjunction of two non-equivalent merges C' and C'' of C_1 and C_2 is a merge of C_1 and C_2 that implies both C' and C''). The merge of C_1 and C_2 as can be denoted as $merge(C_1, C_2)$.

Example 21. Let C_1 be the condition $B < D + 1.2 \wedge B \geq D$ and C_2 be the condition $B \leq D \wedge E < H$. The condition $C_1 \wedge C_2$ is consistent. It can be shown that C_1 and C_2 are mergeable and $merge(C_1, C_2)$ is $B < D + 1.2$.

Let C_3 be the condition $A > -5.1 \wedge A \leq 6$ and C_4 be the condition $A \geq 7 \wedge A < 9$. The condition $C_3 \wedge C_4$ is inconsistent. It can be shown that C_3 and C_4 are mergeable and $merge(C_3, C_4)$ is $A > -5.1 \wedge A < 9$. Conditions C_3 and C_4 and their merge are depicted on Figures 5.1(a) and (b).

Let C_5 be the condition $A > 5$ and C_6 be the condition $D = E + 3 \wedge B < 3$. The condition $C_5 \wedge C_6$ is consistent. It can be shown that no non-valid constraint can be implied by both C_5 and C_6, and therefore, C_5 and C_6 are not mergeable.

Let C_7 be the condition $A < 5$ and C_8 be the condition $A > 7$. The condition $C_7 \wedge C_8$ is inconsistent. It can be shown that C_7 and C_8 are not mergeable.

The previous example shows that the mergeability of two conditions and the consistency of their conjunction are orthogonal properties. The rest of this section shows how the merge of two conditions can be computed, starting with atomic conditions.

Example 22. Let the atomic conditions A_1 and A_2 be $D > E$ and $D \geq E+3$, respectively. Conditions A_1 and A_2 involve the same attributes D and E. They are depicted on Figure 5.2(a). Clearly, $A_2 \models A_1$. Conditions A_1 and A_2 are mergeable and $merge(A_1, A_2) = A_1$.

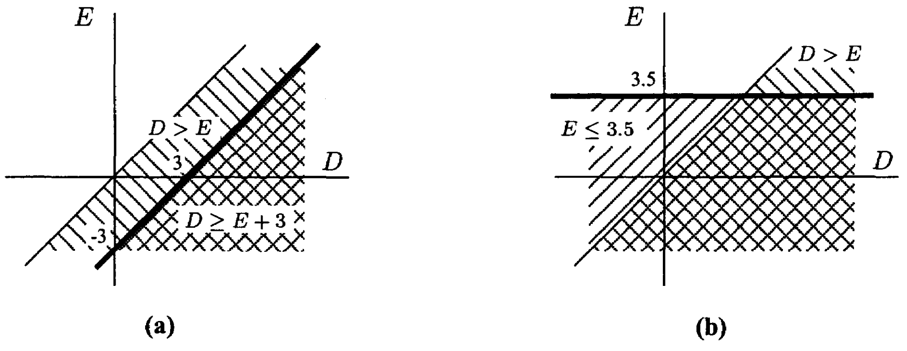


Figure 5.2 (a) Mergeable atomic conditions (b) Non-mergeable atomic conditions.

Let the atomic conditions A_3 and A_4 be $D > E$ and $E \leq 3.5$, respectively. Conditions A_3 and A_4 do not involve the same attributes: condition A_3 involves attributes D and E while condition A_4 involves only attribute E. Clearly, $A_3 \not\models A_4$ and $A_4 \not\models A_3$. One can see that A_3 and A_4 are not mergeable.

More generally, a proposition is shown below. For this proposition it is assumed that atomic conditions do not involve equality (=).

Proposition 5.4.1. Let A_1 and A_2 be two atomic conditions that do not involve equality. Conditions A_1 and A_2 are mergeable if and only if $A_1 \models A_2$ or $A_2 \models A_1$. If $A_1 \models A_2$, $merge(A_1, A_2) = A_2$. If $A_2 \models A_1$, $merge(A_1, A_2) = A_1$.

As shown in Example 22 an atomic condition can imply another atomic condition that involves the same attributes. This is not the case for atomic conditions that do not involve the same attributes:

Proposition 5.4.2. Let A_1 and A_2 be two atomic conditions that do not involve the same attributes. Then, $A_1 \not\models A_2$ and $A_2 \not\models A_1$.

Thus, two atomic conditions can be mergeable only if they involve the same attributes.

Let us now consider general conditions. An atomic condition that involves equality can be equivalently replaced in a condition by the conjunction of two atomic conditions that involve $\leq$ and $\geq$. For instance, $D = E+2$ can be replaced by $D \leq E+2 \wedge D \geq E+2$. An atomic condition A_1 in a condition C is *strongly redundant* in C, if there is another atomic condition A_2 in C such that $A_2 \models A_1$. For instance, $D < E + 2$ is strongly redundant in a condition C if the atomic condition $D < E$ is also present in C. Removing from a condition C an atomic condition that is strongly redundant in C results in a condition equivalent to C. Thus, without loss of generality, it can be considered that conditions do not involve equality and do not include strongly redundant atomic conditions.

In order to determine the merge of two conditions, a form for conditions that directly represents the atomic formulas that can be implied by the conditions is needed.

Definition 15. A condition C is in *full* form if:

1. For every atomic condition A_i such that $C \models A_i$, there is an atomic condition A_j in C such that $A_j \models A_i$.
2. Condition C does not include strongly redundant atomic conditions.

A query $\Pi_X(\sigma_C(\mathbf{R}))$ is in *full* form if its condition C is in full from.

Every condition can be equivalently put in full form in polynomial time by computing the closure of its atomic conditions [89, 138, 64] and removing strongly redundant atomic conditions. Note that a condition C in full form may include redundant atomic conditions, that is, atomic conditions that can be implied by the conjunction of the rest of the atomic conditions in C.

The following therorem determines whether two conditions are mergeable and shows how their merge can be computed. By convention, it is assumed that if two conditions C_1 and C_2 are not mergeable, $merge(C_1, C_2) = T$, where T denotes the truth value TRUE.

Theorem 10. Let C_1 and C_2 be two conditions in full form that do not involve equality. Let

$$\mathcal{M} = \bigwedge_{A_i \text{ in } C_1,\ A_j \text{ in } C_2} merge(A_i, A_j).$$

where A_i and A_i denote atomic conditions. If $\mathcal{M} = T$, conditions C_1 and C_2 are not mergeable. Otherwise, C_1 and C_2 are mergeable and $\mathcal{M} = merge(C_1, C_2)$.

Example 23. Applying Proposition 10 to the conditions of Example 21 validates the claims made there.

The merge of two conditions can be computed in polynomial time. Usually the number of atomic conditions in a query is not large. Therefore, the merging of conditions can be performed efficiently even for a large number of queries.

5.5 Computing CCDs using query graphs

This section shows how queries can be represented using graphs called query graphs. Then, both necessary and sufficient conditions for a view to be a CCD of two queries in terms of query graphs are provided. These conditions are used to compute all the CCDs of two queries.

Definition 16. Given a query $Q = \Pi_X(\sigma_C(\mathbf{R}))$, a *query graph* for Q is a node and edge labeled graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ where:

1. $\mathcal{N}$ is the set of nodes of $\mathcal{G}$. Set $\mathcal{N}$ comprises exactly one node for every relation occurrence in $\mathbf{R}$. Every node is labeled by:
   (a) the name R of the represented relation occurrence and the name of the corresponding relation if this last one is renamed, and
   (b) the set P_R of the projected attributes of the represented relation occurrence (those attributes of this relation occurrence that appear in X.)

   Nodes in $\mathcal{G}$ can be uniquely identified by the name of the represented relation occurrence. In the following a node in $\mathcal{G}$ is identified with the represented relation occurrence.

2. $\mathcal{E}$ is the set of edges of $\mathcal{G}$. For every node R in $\mathcal{N}$, if R has an attribute involved in a selection atomic condition in C, there is a loop edge $\langle R, R\rangle$ in $\mathcal{E}$ labeled by the conjunction C_R of all the selection atomic conditions in C involving attributes of R. For every two nodes R, S in $\mathcal{N}$ that have a join atomic condition in C involving an attribute of R and an attribute of S, there is an edge $\langle R, S\rangle$ in E labeled by the conjunction C_{RS} of all the join atomic conditions in C involving one attribute of R and one attribute of S. The conditions C_R and C_{RS} are called *edge conditions* of the edges $\langle R, R\rangle$ and $\langle R, S\rangle$, respectively.

Example 24. Figure 5.3 shows the query graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ for the queries Q_1 and Q_2 of Example 17. The names of the nodes are shown by the nodes. The projected attributes of a node follow the name of the node separated by colon. Edge conditions are shown by the corresponding edges.

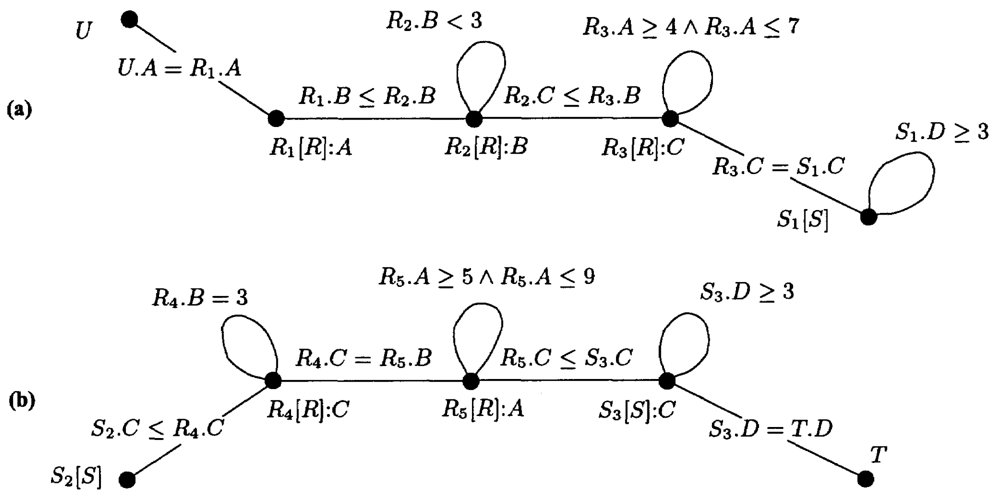


Figure 5.3 (a) Query graph $\mathcal{G}_1$ (b) Query graph $\mathcal{G}_2$.

In the following, a query is identified with its query graph. The concept of a candidate CCD of two query graphs can be defined in the following. This definition uses node mapping functions between two query graphs. If C is a condition, and f is a node mapping function that maps all the relation occurrences (nodes) of C, $f(C)$ is used to denote the condition obtained by renaming the relation occurrences in C according to the relation renaming induced by f. Further, if $R_i.A$ is an attribute of the relation occurrence R_i, and

f is a node mapping function that maps R_i to a relation occurrence R_j, $f(R_i.A)$ is used to denote the attribute $R_j.A$.

Definition 17. Let $\mathcal{G}_1 = (\mathcal{N}_1, \mathcal{E}_1)$ and $\mathcal{G}_2 = (\mathcal{N}_2, \mathcal{E}_2)$ be the query graphs of two queries in full form, and $\mathbf{R}_1$ and $\mathbf{R}_2$ be two sets of nodes in $\mathcal{G}_1$ and $\mathcal{G}_2$, respectively ($\mathbf{R}_1 \subseteq \mathcal{N}_1$ and $\mathbf{R}_2 \subseteq \mathcal{N}_2$). A query graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ is a *candidate CCD* of $\mathcal{G}_1$ and $\mathcal{G}_2$ over $\mathbf{R}_1$ and $\mathbf{R}_2$ if and only if there are two one-to-one onto functions f_1 and f_2 from $\mathcal{N}$ to $\mathbf{R}_1$ and from $\mathcal{N}$ to $\mathbf{R}_2$, respectively, such that:

1. For every node $R \in \mathcal{N}$, the nodes R, $f_1(R)$ and $f_2(R)$ are relation occurrences of the same relation.
2. For every $\langle R, S\rangle \in \mathcal{E}$, $\langle f_i(R), f_i(S)\rangle \in \mathcal{E}_i$, $i = 1, 2$. That is, edges in $\mathcal{G}$ are mapped through f_i, $i = 1, 2$, to edges in $\mathcal{G}_i$.
3. For every condition C of an edge $\langle R, S\rangle \in \mathcal{E}$, the condition $f_1^{-1}(C_1)$, where C_1 is the condition of the edge $\langle f_1(R), f_1(S)\rangle$, and the condition $f_2^{-1}(C_2)$, where C_2 is the condition of the edge $\langle f_2(R), f_2(S)\rangle$, are mergeable and $C = merge(f_1^{-1}(C_1), f_2^{-1}(C_2))$. That is, each condition of an edge in $\mathcal{G}$ is the merge of the conditions of the images of this edge in $\mathcal{G}_1$ and $\mathcal{G}_2$ under the relation occurrence renamings induced by the inverse of the node mapping functions.
4. For every node $R \in \mathcal{N}$, its set of projected attributes is $\bigcup_{i=1,2}(Y_i \cup Z_i)$, where:
   (a) Y_i is the set of attributes $R.A$ such that $f_i(R.A)$ is a projected attribute of $f_i(R)$ in $\mathcal{G}_i$, and
   (b) Z_i is the set of attributes $R.A$ such that attribute $f_i(R.A)$ is involved in an atomic condition A_i of the condition of an edge $\langle f_i(R), f_i(S)\rangle$ of $\mathcal{G}_i$ and $C \not\models f_i^{-1}(A_i)$, where C is the condition of the edge $\langle R, S\rangle$ of $\mathcal{G}$. Edge $\langle f_i(R), f_i(S)\rangle$ can be a loop edge, i.e. $f_i^{-1}(S)$ can be identical to $f_i^{-1}(R)$.
5. There is no query graph $\mathcal{G}' = (\mathcal{N}', \mathcal{E}')$, and extensions f_i', $i = 1, 2$, of f_i from $\mathcal{N}'$ to $\mathbf{R}'_i$, where $\mathbf{R}_i \subset \mathbf{R}'_i \subseteq \mathcal{N}_i$, that satisfy the properties 1 - 4 above.

Example 25. Figure 5.4 shows two candidate CCDs $\mathcal{G}$ and $\mathcal{G}'$ of the query graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ of Example 24. Query graph $\mathcal{G}$ on Figure 5.4(a) is a CCD of $\mathcal{G}_1$ and $\mathcal{G}_2$ over the set of nodes $\{R_2, R_3, S1\}$ and $\{R_4, R_5, S_1\}$. Query graph $\mathcal{G}'$ on Figure 5.4(b) is a CCD of $\mathcal{G}_1$ and $\mathcal{G}_2$ over the set of nodes $\{S_1, R_3\}$ and $\{S_2, R_4\}$.

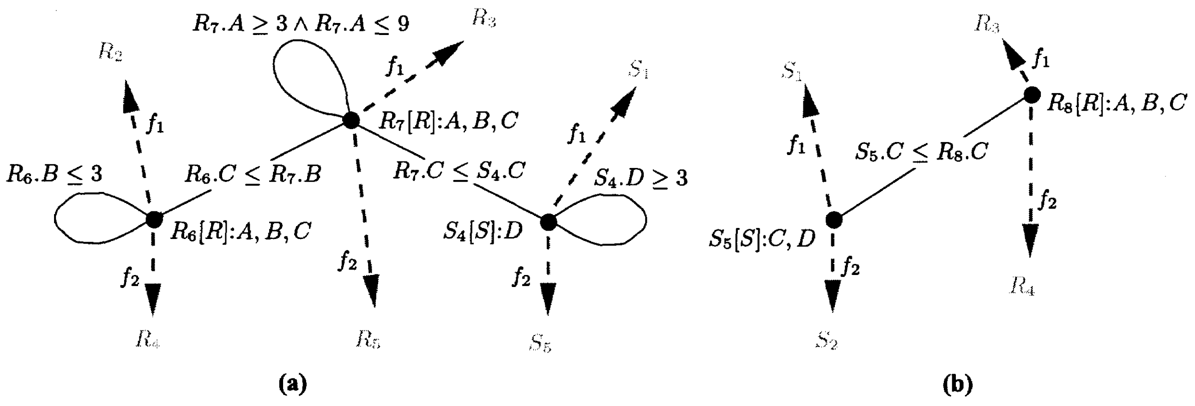


Figure 5.4 Two CCDs of query graphs (a) $\mathcal{G}_1$ and (b) $\mathcal{G}_2$.

Clearly, a candidate CCD of two query graphs is a CD of these query graphs. Next how a closeness relationship between two candidate CCDs can be expressed in terms of query graphs is shown below.

Proposition 5.5.1. Let $\mathcal{G}_1 = (\mathcal{N}_1, \mathcal{E}_1)$ and $\mathcal{G}_2 = (\mathcal{N}_2, \mathcal{E}_2)$ be the query graphs of two queries in full form, and $\mathbf{R}_1$, $\mathbf{R}_2$, $\mathbf{R'}_1$ and $\mathbf{R'}_2$ be sets of nodes such that $\mathbf{R}_1 \subseteq \mathbf{R'}_1 \subseteq \mathcal{N}_1$ and $\mathbf{R}_2 \subseteq \mathbf{R'}_2 \subseteq \mathcal{N}_2$. Let also $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a candidate CCD of $\mathcal{G}_1$ and $\mathcal{G}_2$ over $\mathbf{R}_1$, and $\mathbf{R}_2$ and $\mathcal{G}' = (\mathcal{N}', \mathcal{E}')$ be a candidate CCD of $\mathcal{G}_1$ and $\mathcal{G}_2$ over $\mathbf{R'}_1$ and $\mathbf{R'}_2$. $\mathcal{G}' \prec_{\mathcal{G}_1, \mathcal{G}_2} \mathcal{G}$ if and only if $\mathcal{G}$ and $\mathcal{G}'$ are not equivalent and there is a one-to-one node mapping function f from $\mathcal{G}$ to $\mathcal{G}'$ such that:

1. For every node $R \in \mathcal{N}$, the nodes R and $f(R)$ are relation occurrences of the same relation.
2. For every $\langle R, S \rangle \in \mathcal{E}$, $\langle f(R), f(S) \rangle \in \mathcal{E}'$.
3. For every condition C of an edge $\langle R, S \rangle \in \mathcal{E}$, $f(C) \models C$, where $f(C)$ is the condition of the edge $\langle f(R), f(S) \rangle \in \mathcal{E}'$.
4. If f is an onto function (that is, if $\mathcal{G}$ and $\mathcal{G}'$ have the same number of nodes), for every node $R \in \mathcal{N}$, $P_R \supseteq P_{R'}$, where P_R is the set of projected attributes of node R and $P_{R'}$ is the set of projected attributes of node $R' = f(R)$.

The following theorem provides necessary and sufficient conditions for a query graph to be a CCD of two query graphs.

Theorem 11. A query graph $\mathcal{G}$ is a CCD of two query graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ over $\mathbf{R}_1$ and $\mathbf{R}_2$ if and only if $\mathcal{G}$ is a candidate CCD of $\mathcal{G}_1$ and $\mathcal{G}_2$ over $\mathbf{R}_1$ and $\mathbf{R}_2$, and there exists no candidate CCD $\mathcal{G}'$ of $\mathcal{G}_1$ and $\mathcal{G}_2$ over $\mathbf{R}'_1$ and $\mathbf{R}'_2$, where $\mathbf{R}_1 \subseteq \mathbf{R}'_1$ and $\mathbf{R}_2 \subseteq \mathbf{R}'_2$, such that $\mathcal{G}' \prec_{\mathcal{G}_1,\mathcal{G}_2} \mathcal{G}$.

The previous results suggest a process for computing all the CCDs of two queries which is outlined below.

Input: two query graphs $\mathcal{G}_1 = (\mathcal{N}_1, \mathcal{E}_1)$ and $\mathcal{G}_2 = (\mathcal{N}_2, \mathcal{E}_2)$.

Output: the set $\mathcal{C}$ of all the CCDs of $\mathcal{G}_1$ and $\mathcal{G}_2$

1. Compute the set $\mathcal{C}$ of all the candidate CCDs of $\mathcal{G}_1$ and $\mathcal{G}_2$ by identifying node mapping functions f from $\mathcal{N}_1$ to $\mathcal{N}_2$ that map an edge $\langle R_1, S_1 \rangle \in \mathcal{E}_1$ to an edge $\langle R_2, S_2 \rangle \in \mathcal{E}_2$ such that the condition $f(C_1)$, where C_1 is the condition of $\langle R_1, S_1 \rangle$, and the condition C_2 of $\langle R_2, S_2 \rangle$ are mergeable.
2. Remove from $\mathcal{C}$ a candidate CCD $\mathcal{G}$ of $\mathcal{G}_1$ and $\mathcal{G}_2$ if there is a candidate CCD $\mathcal{G}'$ of $\mathcal{G}_1$ and $\mathcal{G}_2$ such that $\mathcal{G}' \prec_{\mathcal{G}_1,\mathcal{G}_2} \mathcal{G}$.

The previous process can be extended in a straightforward way to generate also minimal rewritings of the queries using the computed CCDs based on the identified node mapping functions.

5.6 Experimental Results

This sections shows the complexity of the algorithm of finding all candidate CCDs between two given queries. The performance of this algorithm depends on several factors such as the number of occurrences, the number of overlapping relations and the number of self-join occurrences of the pair of queries. Note that in these experiments, when the candidate CCDs between queries are computed, the rewritings of each query using the candidate CCDs are generated at the same time.

The number of overlapping occurrences of relations between the two queries has a significant effect on the time of computing all candidate CCDs between two queries. For two queries Q_1 and Q_2, an overlapping factor $OF = 2n_c/(n_{r_1} + n_{r_2})$ is introduced to measure the overlapping between them. n_{r_1} and n_{r_2} represent the number of relations in Q_1 and Q_2, respectively. n_c is the number of common relations of Q_1 and Q_2. Figure 5.6 shows the time needed to find all candidate CCDs between two queries when the number of relation occurrences in each query increases from 4 to 30. Three curves are shown in Figure 5.6, each corresponding to a specific overlapping factor, 0, 0.5 and 1.0, respectively. No self-join is allowed in either query of Q_1 and Q_2. When $OF = 0$, there is no overlapping at all between Q_1 and Q_2. The algorithm can detect that there is no candidate CCD between Q_1 and Q_2 very fast. When the overlapping factor OF increases from 0 to 0.5 and 1.0, the running time of finding the candidate CCDs between Q_1 and Q_2 also increases. When $OF = 1$, the relation occurrences of Q_1 are exactly identical to that of Q_1. In this case, the algorithm finds all candidate CCDs between Q_1 and Q_2 in 1 millisecond and 10 milliseconds when there are 10 and 30 relation occurrences in each query of Q_1 and Q_2, respectively. The result in Figure 5.6 shows that the running time finding all candidate CCDs between two queries scales well to the number of relation occurrences in each query.

The number of self-joins in each query also has a significant effect on the running time of computing all candidate CCDs between two queries. This is because self-joins introduce different mappings of relation occurrences of the two queries. A self-join factor $SF = n_o/n_r$ is introduces to measure the number of self-joins in a query. n_r is the number of relations in the query. n_o is the number of relation occurrences in the query. The bigger a self-join factor is, the more self-joins exists in a query when the number of relation occurrences is fixed. When $SF = 1.0$ for a query Q, there is no self-join at all in Q. When $SF = 2.0$, each relations occurs twice averagely in the query, which represents a heavily used self-join in the query. Figure 5.6 shows the running time of computing all candidate CCDs between two queries when the number of relation occurrences in each query increases from 4 to 32. The overlapping factor is fixed to be 0.5. Three curves are

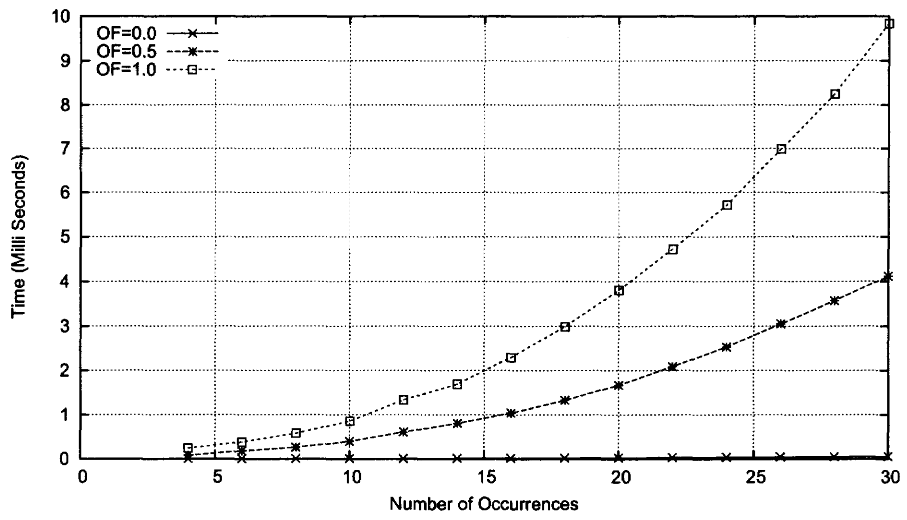


Figure 5.5 Time of finding candidate CCDs between queries for different overlapping factors.

shown in Figure 5.6, each corresponding a different self-join factor, 1.0, 1.5 and 2.0. When there are 30 relation occurrences in each query, the running time of finding all candidate CCDs between the queries increases from 4 milliseconds to 10 milliseconds when the self-join factors in each query increase from 1.0 to 2.0 The running time of computing all candidate CCDs of two queries scales well to the number of self-joins in both queries. This can be explained as follows: when the two queries are quite similar, there is less possibility to build many CCDs. This indicates less mappings need to be considered. When the two queries are not similar, there is less possibility to build big CCDs.

5.7 Conclusion

The problem of constructing a search space for materialized view selection is addressed in this chapter. This is an intricate problem since it requires the detection and exploitation of common subexpressions between queries and views. A novel approach which is based on adding to the alternative plans of the queries CCDs, and on rewriting the queries using these CCDs is suggested. CCDs are defined using the query definitions and do not depend

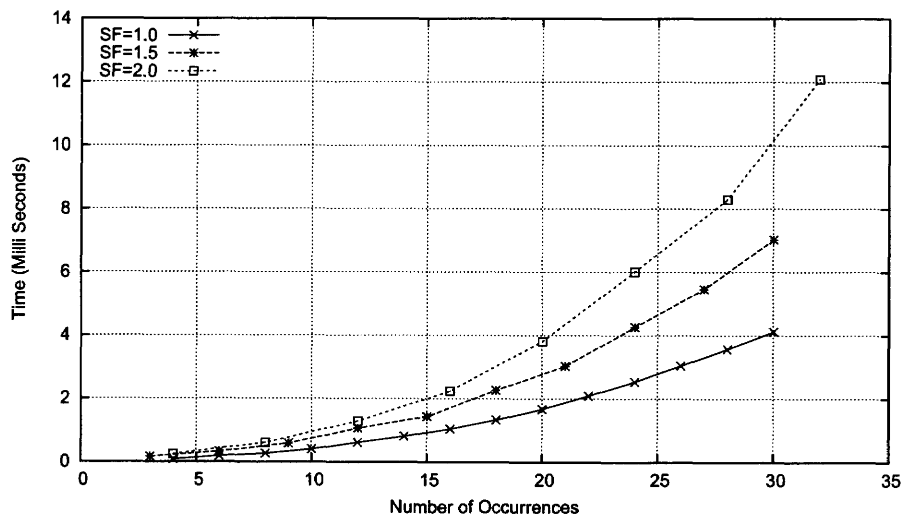


Figure 5.6 Time of finding candidate CCDs between queries for different self-join factors.

on a specific query evaluation plan. CCDs generalize previous definitions of common subexpressions since two queries can be partially rewritten using their CCD. Adding CCDs to the alternative evaluation plans of the queries generates a search space which is expected to comprise all the interesting views for materialization since the CCD of two queries is as close to these queries as possible. A traditional query optimizer can be used to generate alternative evaluation plans for a CCD. The existence of a CCD in the search space does not force its materialization by the view selection algorithms, nor does it exclude the materialization of other nodes (views) in any of its alternative evaluation plans.

CCDs are formally defined based on a closeness relationship on CDs. Using a declarative query graph representation for queries, necessary and sufficient conditions for a view to be a CCD of two queries are provided. Based on these results, a process for computing all the CCDs of two queries and for generating minimal rewritings of the queries using their CCDs is outlined.

CHAPTER 6

AN EXTENDED APPROACH FOR SEARCH SPACE CONSTRUCTION

The previous chapter proposes approaches of finding candidate CCDs of two queries. In most cases, a candidate CCD of two queries represents all the common computations of these two queries. The concept of CCD can be further extended to more than two queries by finding the CCDs of CCDs and other queries or CCDs. For a general view selection problem, the more common computation included in a view, the better this view be materialized. CCDs of queries can be used to construct the search space of a view selection problem. In this chapter, the concept of CCD between two queries is further extended to include some computation of one of the queries. This extension is implemented and the experimental results shows it brings both some benefit and some overhead to a view selection problem. Whether this extension is used depends on the specific instance of the view selection problem.

6.1 Introduction

Current databases and warehouses are getting bigger in size. Also, queries are getting more complex (e.g., queries involving roll-up and drill down operations and top-k queries) [129] while the user's requirements on query performance are getting stricter. Traditional query optimizers based on relations and indices can not satisfy these new needs. Materialized views have been found to be a very efficient way to speed up query evaluation. They are increasingly supported by commercial database management systems [14, 151, 8]. A lot of research work has focused on the problem of view selection, which can be abstractly modeled as follows: given a set of queries and a number of cost-determining parameters (e.g., query frequency/importance and source relation update propagation frequencies), output a set of view definitions that minimizes a cost function and satisfies a number of constraints. The cost function can be, for instance, the query evaluation cost, the view maintenance cost, or a combination of them. A constraint can be a storage space constraint, or an upper bound on the materialized view maintenance cost [130]. Depending on the type of problem considered, the selected views can be permanently materialized (e.g., in the case of the data warehouse design problem) or transiently materialized (e.g., in the case of intermediate results during the concurrent evaluation of multiple queries) or both (e.g., in the case of the maintenance of multiple materialized views where some views are computed because they are permanently stored while other views are stored transiently as auxiliary views during the maintenance process, in order to assist the maintenance of multiple other views [106, 144]).

Most the work on view selection problem focuses on data cube operations on multi-dimensional datasets [75, 13, 125, 85, 101, 31]. The reason is not only that this kind of operations is particularly important in On Line Analytical Processing(OLAP) queries, but also because the search space for the problem with this simplified class of queries can be easily modeled and constructed as a multi-dimensional lattice. There are also some works on view selection problem on general database systems. However these approaches assume, explicitly or implicitly, the existence of a search space in a form of an AND/OR graph [118, 69, 71]. However, these AND/OR graphs require the construction of alternative common plans for multiple general queries and the common subexpressions among input queries need to be detected and exploited. This is an intricate problem. Further, the original queries need to be rewritten using the common subexpressions.

In order to determine a search space for a view selection problem, common sub-expressions among workload queries need to be identified. These common subexpressions represent part of the work needed to compute a query. When identified they can be computed only once and the result be used by all queries that share it. This is expected to importantly save computation time. Depending on the problem, a common subexpression might also be a good candidate view for storage (e.g., in the case of the data warehouse design problem).

This chapter provides an extension to the approach of finding common subexpressions of queries introduced in the previous chapter.

The concept of common subexpression initially referred to identical or equivalent expressions [49]. Later on, the term included subsumption [81]. Then the term included overlapping selection conditions [30]. More generally. the term common subexpression between two queries refers to a view that can be used in the rewritings of both queries, either completely or partially [94]. One approach to exploit common subexpressions is on the query evaluation plan level [121, 120, 106]. This method can give the global evaluation plan in addition to selecting a set of views to materialize. It requires the enumeration of all possible evaluation plans for all queries in the workload. Although some heuristics can be applied to reduce the number of evaluation plans considered, this method is still too expensive when the number of queries is big or when the queries in the workload are complex. The resulting search space is also too big for most view selection problems. Another approach to exploit common subexpressions is on the query definition level[30, 94, 129]. In [30], all queries in a workload are represented as a global multi-graph. Using heuristic transformation rules, this multi-graph is transformed to a one where no more transformations can be applied. This approach can give the common subexpressions as well as the corresponding rewritings. All the queries in the workload can be considered together. However, this chapter assumes there are no self-joins in the queries and this assumption is too restrictive for the queries of current applications. In [94], a general concept of common subexpression is introduced and some constraints (e.g., key/foreign key constraints) are considered. An artificial common subexpression (common subsumer) between a pair of queries is constructed if there is no subsumption relationship between them. Then rewritings (called compensations) of the original queries using the common subexpressions are given. This approach considers query pairs that follow some specific patterns. In [129], common subexpressions between different parts of one single query are exploited and used to apply multi query optimization on a single query. The algorithm to exploit commonalities in this chapter is more on a topology similarity level than on a predicate level. This is because it is not known in advance which parts of the query need to considered for comparison. In general, the problem of answering queries using views is a NP-hard problem [2]. Implementation aspects of this problem, for restricted classes of queries and views have been addressed in [112, 151].

The previous chapter defines CCDs by matching a relation occurrence in one query to at most one relation occurrence in the other query. In this chapter, this restriction is relaxed.It is shown that this relaxation brings more computation in the CCDs. Additionally, it reduces the size of the search space (by reducing the number of CCDs). Both improvements are expected to be beneficial to the subsequent application of view selection algorithms.

6.2 Maximum Commonality Between two Queries

6.2.1 Queries and Rewritings

Just as in the previous chapter, the queries considered in this chapter are select-project-join (SPJ) queries, possibly with self-joins. This class of queries constitutes the basis for more complex queries that involve grouping aggregation operations. For simplicity of description, the relational algebra expression $\pi_P(\sigma_C(O))$ is used to represent a query. O denotes a set of occurrences representing their Cartesian product. C denotes a set of atomic conditions (conjunctive queries are assumed). P denotes the set of projected attributes. Set-theoretic semantics is assumed. Since self-joins are allowed in queries considered, the same relation may occur more than once in O. Therefore, relation occurrence renaming may be needed. For simplicity, each occurrence is given a unique name in addition to the relation name. The expression $N[R]$ is used to represent an occurrence N of a relation R. For example, $O = \{R_1[R], S[S], R_2[R]\}$ is a possible occurrence set. Note that the occurrence name may be the same as the relation name, but must be unique among the names in the occurrence set. C represents the predicates to be applied to O. It consists of a set of atomic conditions of the form $A\ \theta\ B{+}c$ or $A\ \theta\ c$ where θ is an operator from $\{<, \leq, =$

$, \geq, >\}$, A and B are attributes and c is a constant. The latter atomic condition is a selection condition. The former one is a selection condition when A and B are from the same occurrence, and it is a join condition when A and B are from different occurrences. For example, $C = \{R_1.A < S.B + 3,\ R_1.C = 5,\ S.C = R_2.C,\ R_2.D \leq R_2.E\}$ are possible atomic conditions for the previous occurrence set. Note that "pure" Cartesian product is disallowed in queries. This means that every occurrence in the occurrence set is involved in at least one atomic join condition. P represents all projected attributes. Attribute renaming in the projected attribute set is allowed. The expression $R_1.A \rightarrow A$ renames attribute $R_1.A$ as A. For example, set P can be $\{R_2.A \rightarrow A, R_1.B \rightarrow B_1, R_2.B \rightarrow B_2\}$. Note that the renaming of one attribute to more than one attributes in the projected attribute set is allowed. For example, one possible projected attribute set may be $\{R_1.A \rightarrow A_1, R_1.A \rightarrow A_2\}$. Similarly to relation occurrence names, the new names of the attributes must be unique among all attribute names in the projected attribute set. An example query follows:

Example 26. Query Q_1:

- SQL

```
select R1.E as E1, S.A as A, S.B as B1, T.B as B2,
R2.E as E2
from R as R1, S as S, T as T, R as R2
where R1.E<3 and R1.A=S.A and S.B=T.B and R2.C<T.D+3
```

- Relational Algebra expression: $\pi_P(\sigma_C(O))$

- $P = \{R_1.E \rightarrow E_1, S.A \rightarrow A, S.B \rightarrow B_1, T.B \rightarrow B_2, R_2.E \rightarrow E\}$

- $O = \{R_1[R], S[S], T[T], R_2[R]\}$

- $C = \{R_1.E < 3, R_1.A \leq S.A - 3, S.B = T.B, R_2.C < T.D + 3\}$

The order of the projected attributes, atomic conditions and relation occurrences are of no importance. Query *rewritings* are queries of the same form that involve also at least one view.

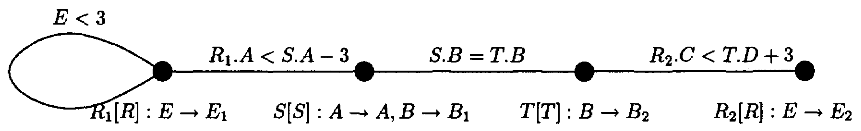


Figure 6.1 Query graph for Q1.

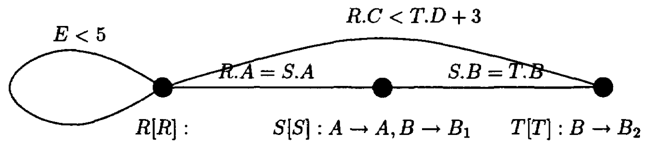


Figure 6.2 Query graph for Q2.

query graphs introduced in the previous chapter is used to graphically represent queries. Relation occurrences are represented by nodes. Relation occurrence and relation names along with new and old attribute names are shown by the corresponding nodes. The query graph of query Q_1 of the previous example is shown in Figure 6.1. Figure 6.2 shows the query graph of another query Q_2.

6.2.2 Query Commonalities

In the previous chapter, CCDs of two queries are defined by requiring minimal rewritings of both queries using each CCD. A rewriting is *minimal* if, for each relation, the sum of the number of its occurrences in the rewriting and the views of the rewriting equals the number of its occurrences in the query. Although this requirement simplifies the definition and computation of a CCD, it also prevents the detection of some commonalities between queries. Consider the queries Q_1 and Q_2 shown in Figures 6.1 and 6.2. Based on the old definition of a CCD, these queries have two CCDs V_1 and V_2 shown in Figure 6.3. In general, an attribute A is called *redundant* in a query if a condition of the form $A = B + c$, where B is another attribute and c is a constant, can be implied from the condition of the query.

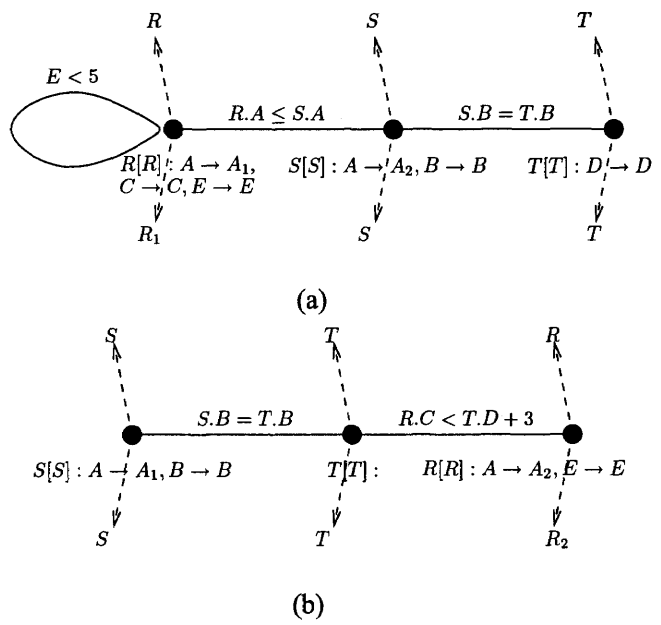


Figure 6.3 Query graphs of CCDs (a) V_1 and (b) V_2.

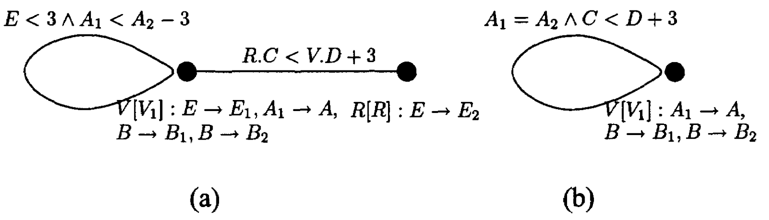

$E < 3 \wedge A_1 < A_2 - 3$
$R.C < V.D + 3$
$V[V_1]: E \to E_1, A_1 \to A,$ $R[R]: E \to E_2$
$B \to B_1, B \to B_2$
$A_1 = A_2 \wedge C < D + 3$
$V[V_1]: A_1 \to A,$
$B \to B_1, B \to B_2$
(a)
(b)

Figure 6.4 Rewritings (a) Q_1' and Q_2' of queries Q_1 and Q_2 using V_1.

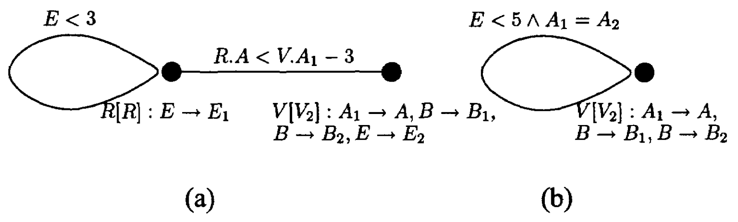


Figure 6.5 Rewritings (a) Q_1' and (b) Q_2' of queries Q_1 and Q_2 using V_2.

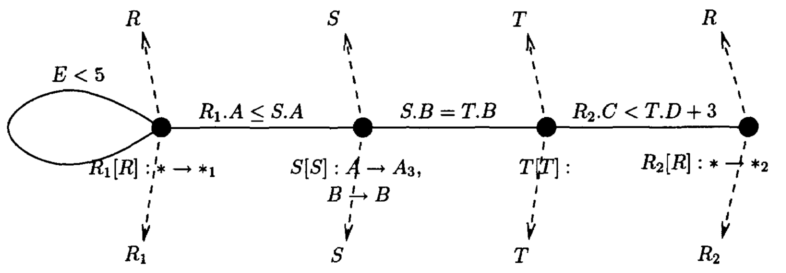


Figure 6.6 Common subexpression V' of Q_1 and Q_2.

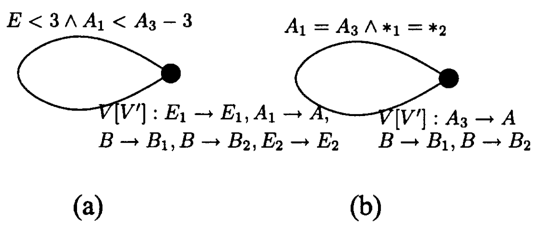

$E < 3 \wedge A_1 < A_3 - 3$
$V[V'] : E_1 \rightarrow E_1, A_1 \rightarrow A,$
$B \rightarrow B_1, B \rightarrow B_2, E_2 \rightarrow E_2$
(a)
$A_1 = A_3 \wedge *_1 = *_2$
$V[V'] : A_3 \rightarrow A$
$B \rightarrow B_1, B \rightarrow B_2$
(b)

Figure 6.7 Rewritings (a) Q'_1 and Q'_2 of queries Q_1 and Q_2 using V'.

Note that in both V_1 and V_2, attribute B is not projected from T although it is projected in both Q_1 and Q_2. This is because the condition in each of V_1 and V_2 imply that $S.B = T.B$. Therefore, only one attribute among $S.B$ and $T.B$ (e.g., in V_1 and V_2, $S.B$ is projected) is projected. Then attribute renaming is used twice in the rewriting of the queries using the CCDs. This is shown in both Figure 6.4 and Figure 6.5. In Figure 6.4, the two queries Q_1 and Q_2 are rewritten using V_1. In Figure 6.5, they are rewritten using V_2.

However, for the previous two queries Q_1 and Q_2, a common subexpression V' shown in Figure 6.6 can be found. Symbol * is an abbreviation for all the attributes of a relation. Observe that V' has 4 relation occurrences in stead of 3, V' can be used for rewrite Q_1 minimally, but cannot be used for rewriting Q_2 minimally. This is because V' contains two occurrences of R while Q_2 contains only one occurrence of R. Notice though that V' can be used for rewriting both queries. Figure 6.7 shows rewritings of Q_1 and Q_2 using V'.

It is possible that V' is smaller than CCD_1 and CCD_2. In this case, if the goal is to reduce the materialization space, it is better to use V' as a materialized view for rewriting Q_1 and Q_2. Materializing V' will bring more benefit per space unit. Even if V' is larger than CCD_1 and CCD_2, it might be beneficial to consider V' instead of CCD_1 and CCD_2 because V' comprises one more join operation. This increases the possibility to find more useful common subexpressions between V' and other queries in the workload. In the next section, the definition of CCD introduced in the previous chapter is relaxed so that a minimal rewriting is not a necessary requirement for a CCD. The goal of the relaxation is to consider V' as a CCD of the two queries Q_1 and Q_2.

6.3 Definition and Computation of CCDs

In this section, a new definition is provided a CCD based on the relaxation of removing the minimal rewriting. Then an algorithm to compute CCDs with the new definition is given. As in the previous chapter, the rewriting of queries using their CCDs can be computed when the CCDs are computed. Both the CCDs and the corresponding rewritings are used in a search space construction for a view selection algorithm.

6.3.1 CCD Definition

If a query Q can be rewritten using a view V, the view V is called a *subexpression* of Q.

Definition 18. A view V is a CCD of queries Q_1 and Q_2 if and only if:

1. V contains no redundant attribute
2. V is a subexpression of Q_1 and Q_2
3. There exists not another subexpression of Q_1 and Q_2, V' such that V is a subexpression of V while V' is not a subexpression of V.

6.3.2 CCD Computation

To compute the CCDs of two queries, nodes in one query are mapped to nodes of the other query so that the induced edge mapping associates join edges of the two queries whose conditions are mergeable. Two join conditions are mergeable if there is another join condition that is implied by each of them. Clearly this is possible only if a node of one query is mapped to a node of the other query labeled by the same relation. The mapped nodes and edges in each one of the two queries should form a connected component. CCDs correspond to mappings that cannot associate more edges of the two queries.To compute CCDs according to the definition in the previous chapter, the computation of one-to-one mappings between the nodes of the two queries is sufficient.

With the definition of CCD in this chapter, one node in a query can be mapped to more than one node in the other query. This is based on the observation that a query can be

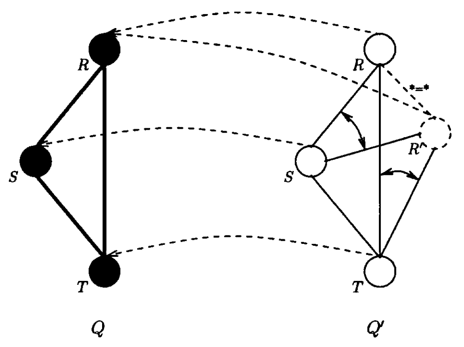


Figure 6.8 Node splitting.

rewritten equivalently by splitting a relation occurrence. For example, R can be rewritten as $R \bowtie_{*=*} R$. An example of occurrence splitting is shown in Figure 6.8 with query graph. Note that in this example, a new edge exists between the original occurrence node and the splitting node with edge condition $* = *$. This condition denotes a conjunction of equalities between all attributes with the same names from the two relation occurrences. This edge is referred as an all-attribute-equal edge. In the following examples of this section, it is depicted by a dashed line. If there is an edge (S, R) between a node S and the split node R, then a new edge (S, R') may be added between S and the new node R' with the same condition as that of edge (S, R).

Before introducing the process of finding CCDs of two queries, the process of how a common subexpression of them can be extended to contain one more edge is shown. If this common subexpression can not be extended any more, then it is a CCD.

Assume there is a common subexpression V of queries Q_1 and Q_2. Initial V contains only one node. For a node N in V, find its corresponding nodes N_1 in Q_1 and N_2 in Q_2 according to the mapping. For an edge $e_1 = (N_1, N_1')$ in Q_1, find an edge $e_2 = (N_2, N_2')$ from Q_2 that is mergable to e_1. Let C_m be the merge condition. Based on whether e_1 and e_2 have been mapped and whether N_1' and N_2' has been mapped, the extension rules are presented as follows:

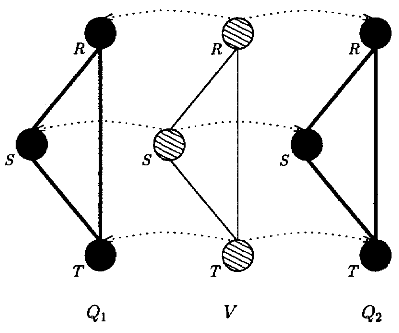


Figure 6.9 One to one node and edge mapping.

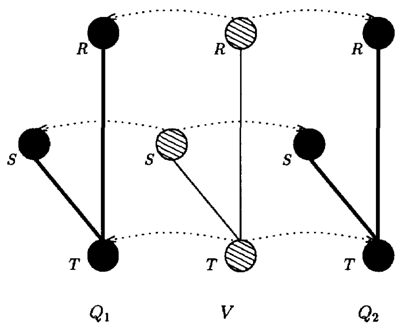


Figure 6.10 One to one node and edge mapping.

1. Edges e_1 and e_2 have not been mapped in V:

   (a) Rule 1: If N_1' and N_2' have been mapped to the same node N' in V, add the merging edge (N, N') with condition C_m to V. An example of an application of this extension rule is shown in Figure 6.9. Initially edges (S, T) and (R, T) have been mapped and Rule 1 adds edge (S, R) to V.

   (b) Rule 2: If neither N_1' nor N_2' has been mapped in V, add a new node N' and map N_1' in Q_1 and N_2' in Q_2 to N'. Add a new edge (N, N') with condition C_m to V. An example of an application of this extension rule for edge (T, R) is shown in Figure 6.10.

   (c) Rule 3: If N_1' has been mapped to N' in V while N_2' has not been mapped, rewrite Q_1 as follows: split node N_1' to get a new nodeN_1'', add a virtual edge (N_1, N_1'') and an all-attribute-equal edge (N_1', N_1''), remove edge (N_1, N_1'). After the rewriting, map edge (N_1, N_1'') in the rewriting Q_1' of Q_1 to (N_2, N_2') in Q_2 with extension Rule 2. An example of an application of this extension rule is shown in Figure 6.11.

   (d) Rule 4: If N_1' in Q_1 maps to N_1' in V and N_2' in Q_2 maps to N_2' in V, rewrite Q_2 as follows: split node N_2' to get anew node N_2'', add the virtual edge (N_2, N_2'') and the all-attribute-equal edge (N_2', N_2''), remove edge (N_2, N_2'). After the rewriting, map edge (N_1, N_1') in Q_1 to (N_2, N_2'') in the rewriting Q_2' of Q_2 using

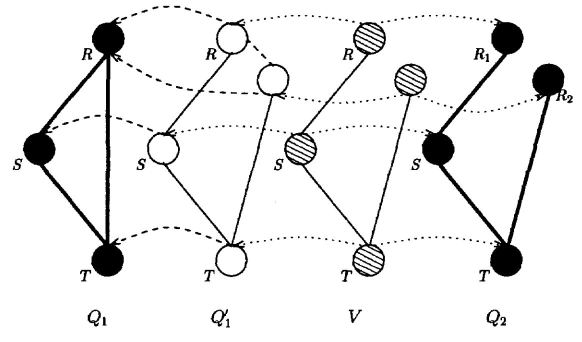


Figure 6.11 One node splitting.

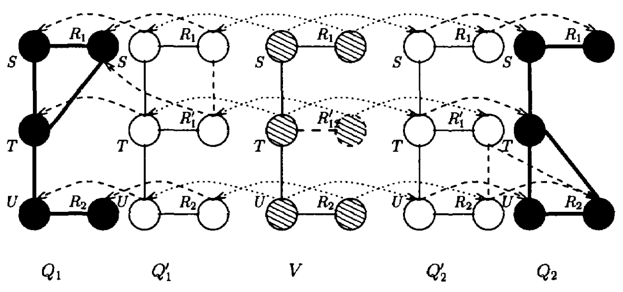


Figure 6.12 Two nodes splitting.

extension Rule 3. This will generate a node splitting in Q_1 similar to that in Q_2. An example of an application of this extension rule is show in Figure 6.12.

2. Edge e_1 in Q_1 has been mapped to $e = (N, N')$ in V while e_2 in Q_2 has not been mapped:

   (a) Rule 5: If N_2' in Q_2 has not been mapped, rewrite Q_1 to Q_1' as follows: split node N_1' to get a new node N_1'', and add to Q_1 an all-attribute-equal edge (N_1', N_1'') and a virtual edge (N_1, N_1''). After this rewriting, map edge (N_1, N_1'') Q_1' to (N_2, N_2') in Q_2 with extension Rule 2. An example of an application of this extension rule is show in Figure 6.13.

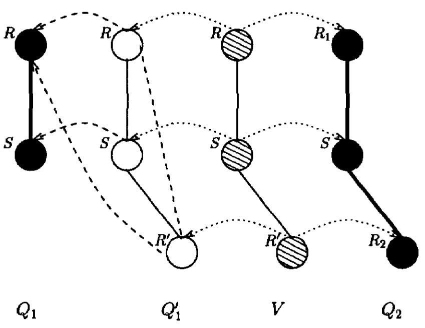


Figure 6.13 One node splitting.

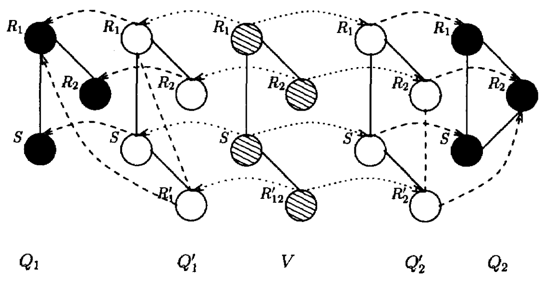


Figure 6.14 Two nodes splitting.

(b) Rule 6: If N_1' in Q_1 has been mapped to N_1' in V, while N_2' in Q_2 has been mapped to N_2' in V, rewrite Q_2 to Q_2' as follows: split node N_2' to get a new node N_2'', add to Q_2 an all-attribute-equal edge (N_2', N_2'') and a virtual edge (N_2, N_2''), and remove edge (N_2, N_2') from Q_2. After the rewriting, map edge (N_1, N_1') in Q_1 to (N_2, N_2'') in Q_2' using extension Rule 3. An example of an application of this extending rule is show in Figure 6.14.

All the above six extension rules map regular edges and not virtual or all-attribute-equal edge. There may be cases where mapping those edges with regular edges generates some useful CCDs. For simplicity here, this kind of mapping is ignored.

Note that extension Rules 1 and 2 apply one to one node mapping; extension Rules 3 and 4 apply one to many node mapping, but only one to one edge mapping; extension Rule 5 and 6 apply one to many edge mapping (and consequently one to many node mapping). Using the previous extension rules, one can specify the new CCD computation process as follows:

Using extension rules 1 and 2, one can get CCDs exactly the same as they are defined in the previous chapter. Using additionally Rules 3-6 makes the process more expensive since more possible extensions are considered. To reduce the execution time, one possible heuristic disallows extending rules 3-6 if one common subexpression can be extended using Rule 1 or 2.

6.3.3 Rewriting the Queries Using the CCDs

The general problem of rewriting queries using views is NP-hard. This is even true for simple queries and views such as SPJ queries with self-joins, the case considered in the previous chapter and in this chapter. However, here in both chapters, only simple rewritings are considered, which means that queries are rewritten using only one occurrence of a view. Further, since a mapping function from all occurrence nodes of the view to occurrence nodes of the query have already been constructed, (generated during the computation of the CCDs), the rewriting is straightforward. The process is outlined below:

1. *Construct the relation occurrence set:* Put an occurrence of the CCD to the occurrence set of the rewritten query. Put also to the occurrence set of the rewritten query all the relation occurrences in the query which have not been mapped to a node in the CCD. Note that all occurrence names must be unique in the constructed occurrence set.
2. *Construct the projected attribute list:* For each relation occurrence in the query which has been mapped to a node in the CCD, list its projected attributes after the CCD occurrence. List all other projected attributes in the query after their original relation occurrences. All projected attributes on occurrences which has a mapped node in CCD, list the attribute under the occurrence of CCD. Note that in the former case, one attribute may be derived from another attribute listed in the CCD. In this case, attribute renaming is needed to recover the projected attribute of the query in the rewriting.

Algorithm 5 Computation of the extended CCDs

```
Input: Two queries $Q_1$ and $Q_2$
Output: A set of views that are CCDs of $Q_1$ and $Q_2$
1: Put both $Q_1$ and $Q_2$ into full form {the same as in the previous chapter}
2: Find all views that are CCDs defined in the previous chapter and put them in a set $\mathcal{V}$
3: for each $V \in \mathcal{V}$ do
4:     extend $V$ in all possible ways using the above extension rules until it can not be
       extended anymore. Put $V$ into a set $\mathcal{NV}$
5: end for
6: return $\mathcal{NV}$
```

3. *Construct the predicate set:* For edges between occurrences (join edges) or on one occurrence (loop edges) in the query that have not been mapped to occurrences in the CCD, add edges with the same condition between the corresponding occurrences in the rewriting of the query. For edges between an unmapped occurrence and a mapped occurrence, add an edge with the same condition between the corresponding occurrence and the CCD occurrence in the query rewriting. For edges between mapped occurrences, add a loop edge to the CCD occurrence in the query rewriting with the same condition. If the edge already exists, just add the condition to the edge.

Figures 6.4, 6.5 and 6.7, show some examples of rewriting queries using views.

6.4 Experimental Results

The extensions of CCDs has been implemented and some experiments are run to show the effectiveness of the new definition of CCDs. In this section, some experimental results are shown to compare the some characteristics of CCDs of the previous chapter and of this chapter. The definition of CCD in the previous chapter is referred as one-to-one CCD because it is constructed by allowing each occurrence node from one query to be mapped to at most one occurrence node from the other query. The new definition of CCD in this chapter is referred as many-to-many because it allows many occurrence nodes from one query to be mapped to many occurrence nodes of the other query. In the experiments, an initial schema is generated which includes a set of base relations. Based on this schema, a series of workload queries (20 queries in each workload) is generated. For a given workload, all possible CCDs between each pair of queries and the corresponding rewritings are computed. Then, for each one of the two CCD definitions, the average time for computing the CCDs of a pair of queries, the average number of CCDs generated for a pair of queries, and the average number of occurrences in each CCD is recorded and compared for the two CCD definitions. Each workload follows the following rules:

- Each query contains m base relations.

- Each query contains n relation occurrences, $n \geq m$. In general, the larger the number of occurrences for a fixed number of relations, the higher the number of node mappings between the two queries.
- Approximately, $n/3$ of the queries have selection conditions in every workload.
- In order to avoid having query graphs that are too dense or too sparse, each one of them has in it slightly over 20% of all possible edges in the query graph.

In the first experiment, n is fixed to be 8 and m varies from 7 down to 5. The result is shown in Figure 6.4. In the second experiment, the difference of n and m is fixed to 3 n varies from 6 to 10. The result is shown in Figure 6.4. One can see that for the same values of m and n, the number of many-to-many CCDs is less than the number of one-to-one CCDs, and the number of occurrences per many-to-many CCD is larger than the number of occurrences per one-to-one CCD. These remarks show that the new CCD incorporates more operations. This is expected to improve the overall cost of evaluating all the input queries together. Further, the many-to-many CCD gives less options to view selection algorithms since less CCDs are generated. Therefore, the many-to-many CCD is expected to increase the speed and accuracy of view selection algorithms in determining the optimal solution.

When the ratio of relations to occurrences of relation is high (75% or higher), the computation time of the CCDs for the two approaches is similar (e.g., column sets 1 and 2 in Figure 6.4(a), and column set 3 in Figure 6.4(a)). When this ratio drops bellow 75% the computation time of the many-to-many CCD is much higher (e.g., column set 3 in Figure 6.4(a), and column sets 1 and 2 in Figure 6.4(a)). However, this loss in CCD computation time might be compensated by the important gains in the time of the view selection algorithm (due to the reduction in the number of many-to-many CCDs - Figure 6.4(b) column set 3 and Figure 6.4(b) column sets 1 and 2).Therefore, when the ratio drops bellow 75%, the type (e.g., greedy, heuristic) and the speed of the employed view selection algorithm will determine whether the one-to-one or the many-to-many CCD definition is preferable.

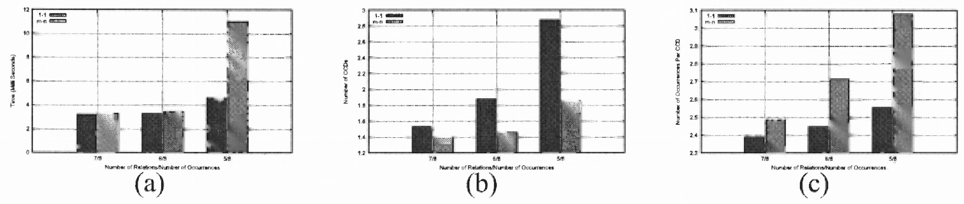


Figure 6.15 Effectiveness of CCD definitions.

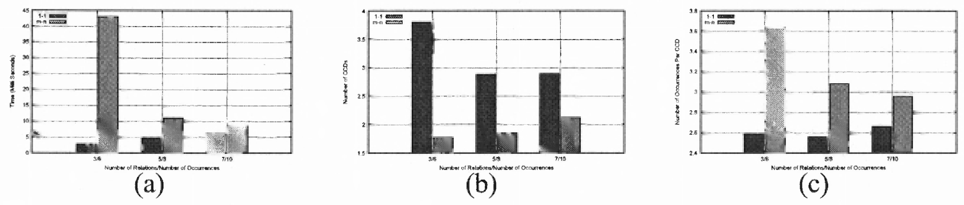


Figure 6.16 Effectiveness of CCD definitions.

6.5 Conclusion

In this chapter, the definition of CCD is extended. CCDs are uesd for constructing search spaces for view selection problems that involve multiple queries. The new definition extends the previous one to allow many to many mappings of relation occurrences in queries. It is particularly useful for query workloads that involve also self-joins. An experimental evaluation shows that the new approach produces CCDs that involve more relational occurrences. Therefore, it increases the chances for these CCDs to be exploited in computing other input queries. It also reduces the size of the search space to be exploited by cost based view selection algorithms.

CHAPTER 7

APPLYING CCDS IN VIEW MAINTENANCE PROBLEMS

The previous two chapters discusses how to define and compute CCDs (common sub-expressions of some specific characteristics) of two queries. Because CCDs of queries include most common computations of queries, they can be used to construct search spaces for general view selection problems. In this chapter, CCDs are applied to a specific instance of the view selection problem: maintaining multiple materialized views while a set of views can be transiently materialized. View maintenance problem is a complementary problem to view selection problem. All materialized views must be updated automatically or manually by database administrator once there are changes to the underlying base relations. These updates can be incremental update whenever base tables upon which the materialized views change, or complete re-computation of the whole materialized views from base tables when there are big changes to the underlying base tables. Because all materialized view are updated as a batch, multi-query optimization techniques can be utilized in both kinds of updates to materialized views.

7.1 Introduction

Selecting materialized views has been proposed as an effective strategy for improving the performance of database application. It is especially important when the database is very big and the queries of the workload are complex [23]. Two critical issues need to be solved here: The optimizer must know how to rewrite queries using materialized views. The materialized views must be updated to keep consistency with the underlying base tables upon which the materialized views are defined.

Maintenance of views has a significant effect on the view selection problem. Consider a view selection problem with maintenance time constraint [71], the more efficient the maintenance algorithm is, the more views can be materialized. This means more benefit is got for the workload of queries. If the update statements are combined into the workload [154], then the maintenance cost is combined into the workload execution time as a minimization goal. A more efficient method of maintaining materialized views directly reduces the total execution time of the whole workload. Although an incremental view maintenance approach may performs better than a complete re-computation approach, the strict conditions (e.g., the percentage of updated records in the base relations must be relatively small comparing to the original data, both the old data and the new data in each updated base relations must be available, etc.) it requires prevent it from being applied in general view materialization problem. In this chapter, a complete re-computation approach is applied in the view maintenance problem.

The process of a complete re-computation approach for a view maintenance problem can be described in two steps: in the first step, the queries on which the materialized views are defined are computed from base tables; in the second step, the results of the previous step are inserted into into the view tables. To improve the performance of maintaining multiple views using the complete re-computation approach, multi-query optimization techniques can be applied [94, 106]. To speed up the maintenance of a set of input materialized views, some additional views can be materialized transiently during the view maintenance process. This approach of view maintenance itself forms an instance of the view selection problem.

The most important difference between Multi-query optimization and single query optimization is whether to find and utilize common subexpressions among multiple queries. By materializing some of the common subexpressions, some expensive common operations execute only once in the common subexpressions and then more than one queries can be answered from them instead of from base tables [121, 120, 102, 85, 31]. Q. Zhu et al. consider the exploitation of common subexpressions inside a single query and utilize them in query optimization for very complex queries [129]. Although it is applied to single query, we still consider it as an application of multi-query optimization technique.

7.2 View Maintenance Using CCDs

The view maintenance problem can be defined as follows: given a set $\mathcal{V}$ of materialized views defined over a database instance, find a maintenance plan for these views that minimizes the overall maintenance cost. We assume that views are maintained by complete recomputation.

As mentioned in the above section, multi-query optimization techniques are employed to maintain multiple views in which common subexpressions are exploited to derive a global evaluation plan (that is, a plan that computes all the views together). The goal is to minimize the cost of this plan. A global evaluation plan usually needs to materialize a set of views $\mathcal{V}'$. These views are materialized transiently during the execution of the plan and are dropped once the "permanently" materialized views in $\mathcal{V}$ are updated. Different evaluation plans materialize different sets of views. Therefore, the maintenance problem of multiple views using multi-query optimization techniques can be formulated as a view selection problem: given a set $\mathcal{V}$ of permanently materialized views, find a set $\mathcal{V}'$ of additional views so that the cost of the global evaluation plan that transiently materializes the views in $\mathcal{V}'$ is minimized. The optimal global evaluation plan can also be computed as a by-product of the computation of the optimal view set $\mathcal{V}'$.

Because there are a lot of views for a set of materialized views to be maintained, the search space for such a view selection problem can be huge if every possible view is considered for materialization. Instead of considering all possible views, it is assumed that only CCDs of views can be considered as the candidate views to be selected in the view maintenance problem. An advantage of the use of CCDs is that the views in the search space are restricted in a non cost-based manner to views that are expected to be useful. Further, as discussed in the previous sections, this approach also generates rewritings of the queries using materialized views when CCDs are computed. These rewritings can be exploited directly in constructing the global evaluation plan that uses the selected materialized views (CCDs).

Assume that there is a set $\mathcal{V}$ of materialized views to maintain through complete re-computation. The *maintenance cost* $mc(V)$ of a view V in $\mathcal{V}$ is the cost of evaluating the view V from base relations and writing the answer to the disk. The view V can be rewritten using the views in a set $\mathcal{V}'$ and base relations (*partial rewriting of* V using the views in $\mathcal{V}'$), or using exclusively the views in a set $\mathcal{V}'$ (*complete rewriting of* V *using the views in* $\mathcal{V}'$). In this case, the maintenance cost of V, denoted $mc(V, \mathcal{V}')$, is the cost of evaluating V using the views in $\mathcal{V}'$ and the cost of writing the answer to the disk. In computing this cost, assume that the views in $\mathcal{V}'$ are materialized and they are viewed as base relations. The maintenance cost of a view (using or without using other views) can be computed by a DBMS optimizer. Current DBMS optimizers are able to assess the cost of evaluating queries using materialized views [7, 154]. The *total cost* $MC(V)$ of maintaining a view V equals to $mc(V)$ if V if evaluated from base relations, or equals to $mc(V, \mathcal{V}') + \Sigma_{V_i \in \mathcal{V}'} MC(V_i)$ if V is evaluated using views in set $\mathcal{V}'$. The views in $\mathcal{V}'$ may also be rewritten using other views in $\mathcal{V}'$. In this case, $MC(V_i)$ recursively reflects this fact. Note that trivial rewritings are not allowed for the views in $\mathcal{V}'$. Therefore, the rewritings define a hierarchy for the views in $\mathcal{V}'$. This hierarchy reflects the order of materialization of views in $\mathcal{V}'$.

This technique can be extended to multiple views. The total cost of maintaining a set $\mathcal{V}$ of materialized views using another set $\mathcal{V}'$ of views is computed as follows: $MC(\mathcal{V}) = \Sigma_{V_i \in \mathcal{V}} MV(V_i) + \Sigma_{V_i \in \mathcal{V}'} MC(V_i)$.

Adding a new view V into set $\mathcal{V}'$ will increase the second component of $MC(\mathcal{V})$ since this view needs to be maintained. However, this addition might reduce the first component of $MC(\mathcal{V})$ since it might allow cheaper rewritings of the views in $\mathcal{V}$ using $\mathcal{V}'$. If this gain in the evaluation time for the views in $\mathcal{V}$ is larger than the maintenance cost of V, it is beneficial to add V to the set of transiently materialized views $\mathcal{V}'$. In our approach, the views in $\mathcal{V}'$ are CCDs of other views.

Example 27. *Assume that there is a set $\mathcal{V}$ of two views $V_1 = R \bowtie_{A=B} S \bowtie_{C>D+3 \wedge E \leq F} T$ and $V_2 = U \bowtie_{G=H} S \bowtie_{C>D \wedge E<F} T$ to be maintained. Let's also assume that the optimizer has found the optimal plans for materializing V_1 and V_2 which are shown in Figures 7.1 and 7.2. The total cost of maintaining views in $\mathcal{V}$ using base relations is $MC(\mathcal{V}) = mc(V_1) + mc(V_2)$. The discussion in the previous sections suggests that there is only one CCD of V_1 and V_2, $V = S \bowtie_{C>D \wedge E \leq F} T$. Our approach computes also the rewritings of each one of V_1 and V_2 using V. These rewritings are provided as input to the optimizer. The optimizer can find the optimal evaluation plans for materializing V and for materializing V_1 and V_2 using V. Let's assume that these are the evaluation plans shown in Figure 7.3. Note that the optimal plan for a view evaluated over the base relations and the optimal plan of the same view evaluated using a view do not necessarily employ the same join order. The total cost of maintaining the views in $\mathcal{V}$ using the view set $\mathcal{V}' = \{V\}$ is $MC(\mathcal{V}, \{V\}) = mc(V_2, \{V\}) + mc(V) + mc(V_1, \{V\})$. The* benefit *of materializing V is $MC(\mathcal{V}) - MC(\mathcal{V}, \{V\})$. If this benefit is positive, it is worthy using V for maintaining V_1 and V_2. Our approach also outputs an optimal global evaluation plan for V_1 and V_2. This is the plan resulting by merging the three nodes labeled by V of the optimal evaluation plans of Figure 7.3.*

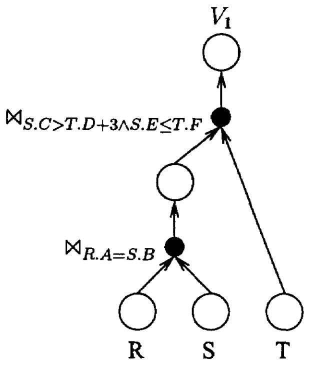


Figure 7.1 Optimal maintenance plan for V_1.

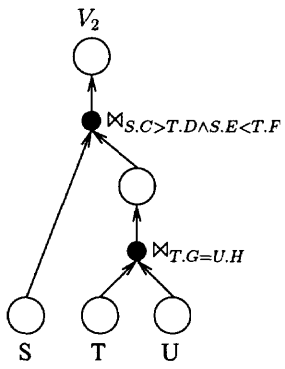


Figure 7.2 Optimal maintenance plan for V_2.

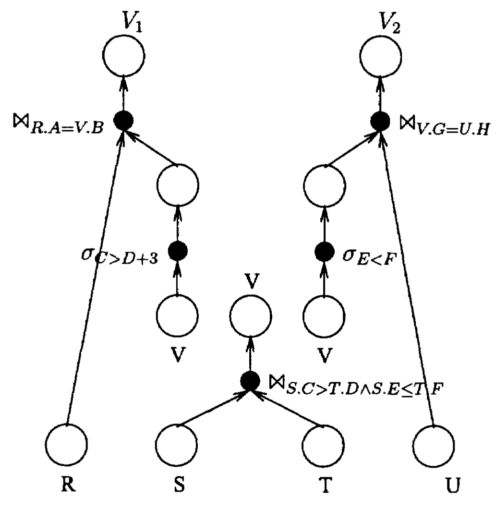


Figure 7.3 Optimal maintenance plans for V_1' and V_2' using V.

When many materialized views need to be maintained, the problem becomes complex because even if the candidate views are restricted to CCDs of views, there might still be too many candidate views to consider for transient materialization. An additional problem is that the benefit of a view depends on other views selected for materialization. Therefore, it might need to be recomputed for multiple candidate views whenever a new view is selected for materialization. In the next section, a general approach of constructing a search space for the view maintenance problem using CCDs is introduced.

7.3 Construct Search Spaces for View Maintenance Problems Using CCDs

As mentioned in the previous chapters, CCDs can be used to construct the search space for a view selection problem. Although the strategy of considering only CCDs reduces the search space of the view maintenance problem significantly, it is still not practical to consider all CCDs of views to maintain a big number of materialized views. Further pruning techniques can be applied to generate a search space of CCDs for the view selection problem with reasonable size. Using the cost model introduced in the above section, the benefit of a CCD can be changed when other CCDs are selected. In the following, a level-based algorithm is introduced to construct the search space for the view maintenance problem using CCDs. In this algorithm, new CCDs of views are introduced level by level. Only CCDs that can bring positive benefit are kept. The algorithm is described as follows:

Algorithm 6 Construct search space for the view maintenance problem using CCDs

Input: A set of views $\mathcal{V}$ to be maintained

Output: A set of views $\mathcal{V}'$ which are CCDs of views in $\mathcal{V}$

```
1: Let set $\mathcal{V}' = \emptyset$
2: Let set $\mathcal{G} = \mathcal{V}$
3: Let set $\mathcal{N}$ contains all CCDs of views in $\mathcal{G}V$
4: while $\mathcal{N} \neq \emptyset$ do
5:    Compute the benefit of each CCD in $\mathcal{N}$ and remove those without positive benefit
6:    while There are more CCDs in $\mathcal{N}$ do
7:       Remove from $\mathcal{N}$ the CCD with the highest benefit and add it into $\mathcal{V}'$
8:       Recompute the benefit of each CCD in $\mathcal{N}$ with respect to the CCDs in $\mathcal{V}'$ being
         materialized and remove those CCDs without positive benefit
9:    end while
10:   Let $\mathcal{N}$ contains all CCDs between each CCD $C \in \mathcal{V}'$ and each view $V \in \mathcal{G}$ where
      $C$ is not a CD of $V$
11:   Let $\mathcal{G} = \mathcal{G} \cup \mathcal{V}'$
12:   Let $\mathcal{V}' = \emptyset$
13: end while
```

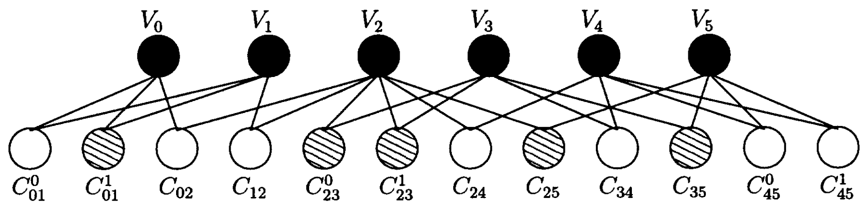
This algorithm starts by considering CCDs of each pair of views in $\mathcal{V}$ in the first level. The benefit of each CCD is computed as in the previous section. CCDs are selected one by one based on their benefit. The benefits of the CCDs not selected are recomputed with respect to the selected CCDs if necessary. Selected CCDs and CCDs without positive benefits are remove. This process terminates when there is no CCDs left in this level. Then in the next level, the CCDs considered are those of the selected CCDs of the previous levels and view in the input set of views $\mathcal{V}$. This process continues level by level until no more

CCDs can be found in the next level. The set of views selected in this process comprise a search space for maintaining views in $\mathcal{V}$

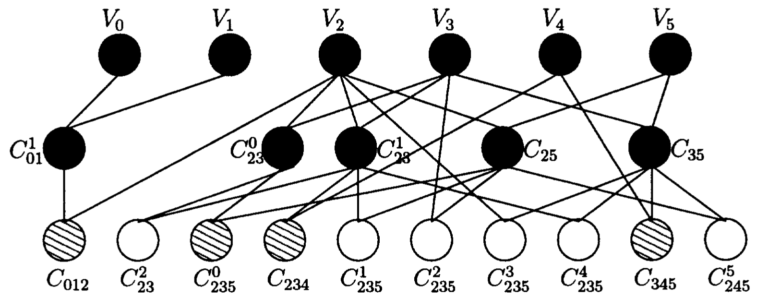
Figure 7.4 demonstrates how this algorithm works to generate a search space of maintaining a set $\mathcal{V} = \{V_0, V_1, V_2, V_3, V_4, V_5\}$ of six views

1. In Step 1, all CCDs of each pair of views in $\mathcal{V}$ are constructed. The optimizer can be used to compute the benefit of each CCD. The algorithm selects CCDs C^1_{23}, C^1_{01}, C^0_{23}, C_{35}, C_{25} in a specific order such that each time the CCD with the highest benefit is selected. Note that after each selection, the benefit of the left CCDs are recomputed with respect to the selected CCDs if necessary. For example, after selecting the CCD $C^1 23$, the benefit of the CCDs C^0_{23}, C_{35}, C_{25} may be recomputed because all of them are CCDs of the view V_2 which can be answered by the selected CCD C^1_{23}.
2. In Step 2, all CCDs of each CCD selected CCDs in Step 1 and the views in $\mathcal{V}$ are constructed. With the assumption that all CCDs selected in Step 1 also need to be maintained as well as views in $\mathcal{V}$, the benefit of the new CCDs can be computed in the same way as in Step 1. A similar process as in Step 1 selects CCDs C_{012}, C^0_{235}, C_{234} and C_{345}.
3. In Step 3, all CCDs of each CCD select in Step 2 and Step 1 and the views in $\mathcal{V}$ are constructed. With the assumption that all CCDs selected in Step 1 and Step 2 also need to be maintained, the benefit of the new CCDs can be computed. A similar process select CCDs $C^0 2345$ and C^3_{2345}.
4. Because no more CCDs can be found of CCDs selected in Step 3 and those in previous steps and views in $\mathcal{V}$, the process terminates in Step 4 with the search space for maintaining the views in $\mathcal{V}$ being CCDs selected in Step 1, Step 2 and Step 3.

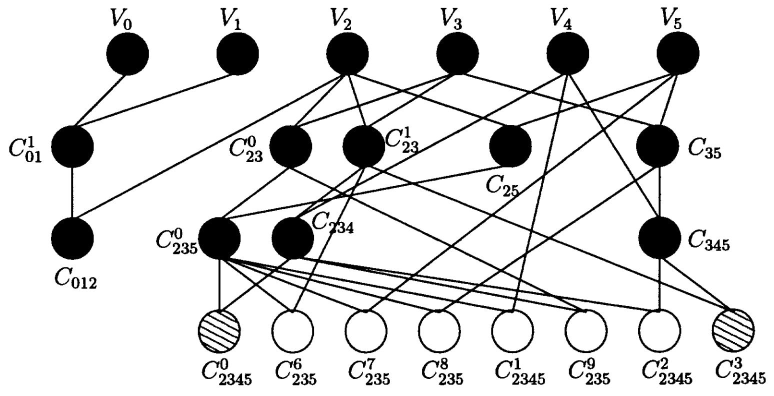
Note that the algorithm not only output a set of views, but also gives the hierarchy of the views that tells which view can be used to answer other views. This is similar to an AND/OR view graphs introduced [69]. The algorithms introduced in [69] can be used directly for the view maintenance problem. Specifically, if it is restricted that each view should be maintained using at most one transiently materialized view, the output hierarchy is an OR DAG which simulates the lattice framework introduced in [75]. The algorithm introduced in [75] can be applied to this view maintenance problem with a light modification.


V_0 V_1 V_2 V_3 V_4 V_5
C^0_{01} C^1_{01} C_{02} C_{12} C^0_{23} C^1_{23} C_{24} C_{25} C_{34} C_{35} C^0_{45} C^1_{45}

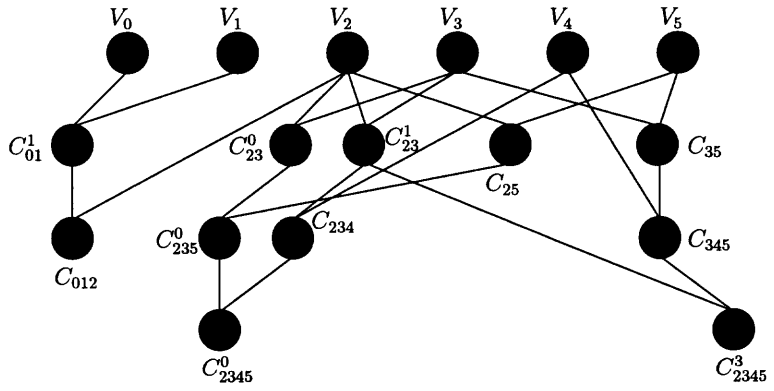
(a) Step 1


V_0 V_1 V_2 V_3 V_4 V_5
C^1_{01} C^0_{23} C^1_{23} C_{25} C_{35}
C_{012} C^2_{23} C^0_{235} C_{234} C^1_{235} C^2_{235} C^3_{235} C^4_{235} C_{345} C^5_{245}

(b) Step 2


V_0 V_1 V_2 V_3 V_4 V_5
C^1_{01} C^0_{23} C^1_{23} C_{25} C_{35}
C_{012} C^0_{235} C_{234} C_{345}
C^0_{2345} C^6_{235} C^7_{235} C^8_{235} C^1_{2345} C^9_{235} C^2_{2345} C^3_{2345}

(c) Step 3


V_0 V_1 V_2 V_3 V_4 V_5
C^1_{01} C^0_{23} C^1_{23} C_{25} C_{35}
C_{012} C^0_{235} C_{234} C_{345}
C^0_{2345} C^3_{2345}

(d) Step 4

Figure 7.4 Constructing search space for maintaining six views.

Although the above approach of generate a sound search space for the view selection problem such that some previously introduced algorithms can be applied easily, it is not practical in most cases when the number of view to be maintained is big. This is because that the algorithm 6 is very expensive. Whenever a CCD is selected in a specific level, the benefit of all other CCDs in that level need to be recomputed, which involves a cost estimation of each selected CCDs or input views.

Instead of applying some heuristics on the algorithm of constructing search space for the view maintenance problem, heuristics are introduced in the next section which combines the process of constructing a search space and selecting materialized views for the view maintenance problem.

We present now heuristic approaches for selecting views (CCDs) for the multiple view maintenance problem. Our first heuristic approach, called *single-level* CCD selection greedy approach, proceeds as follows: first we compute the CCD C_{ij} of every pair of input views V_i and V_j. The rewritings of the views V_i and V_j using C_{ij} are also computed at this time. Then, we compute the benefits of the CCDs, and we rank the CCDs in descending order of their benefits. Note that some (or all) CCDs might have a negative benefit. If all the views have a negative benefit, we we do not select any view for materialization as no candidate view can contribute to the reduction of the view maintenance cost of the input views. We select for materialization the CCD, say C_{ij}, with the highest benefit. We exclude from consideration the input views V_i and V_j and every other candidate view that is a CCD of input view V_i or V_j and some other view. We iteratively repeat this process until no more CCDs with positive benefit is left. This approach considers only candidate views that are CCDs of the input queries. It is in line with previous approaches for multiple query optimization techniques [94, 8]. If we consider a logical plan for multiple input views that shows their dependence from candidate views and base relations for their computation, then the candidate views considered by the this approach all lie at the same single level of the logical plan. This feature explains the name of this approach.

Even though the single-level greedy approach is fast because it considers a restricted number of candidate views, it has a limitation: it considers for materialization only CCDs among the input views. To overcome this limitation, we present below a *multi-level* CCD selection greedy approach. A key feature of this approach is that it considers for materialization also CCDs between CCDs and input views and CCDs between CCDs selected for materialization. By doing so this approach can identify and exploit common subexpressions among multiple input views. Algorithm 7 implements the multi-level CCD selection greedy approach. We assume we have an optimizer that can find optimal plans of views from the base relations and optimal plans of views using possibly other views.

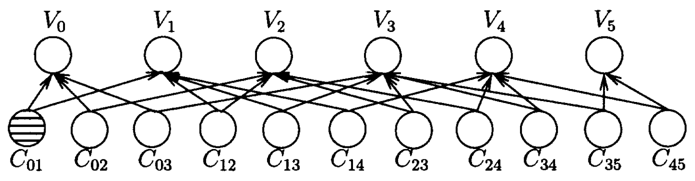
The algorithm is described in Algorithm 7:

We give below an example that shows how the multi-level greedy CCD selection approach proceeds. We assume we have a set $\mathcal{V} = \{V_0, V_1, V_2, V_3, V_4, V_5\}$ of six views to maintain. For simplicity, we assume that there is at most one CCD for each pair of views. The views and their CCDs are shown in Figure 7.5(a). Only CCDs with positive benefit are shown. Some view pairs do not have a CCD either because they do not have an overlapping or because a CCD between them exists but has negative benefit.

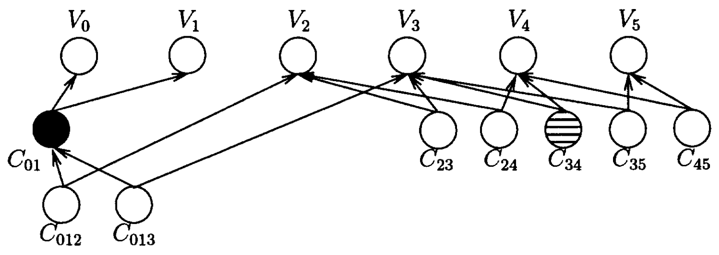
1. Figure 7.5(a) represents the state after the CCDs of each pair of views are constructed. Note that not all pairs of queries have a CCD. For example, there is no CCD between V_0 and V_3. We assume that the CCD C_{01} has the highest benefit among the CCDs and is chosen for materialization. A CCD selected for materialization is shown shadowed in the figures.

2. We select C_{01} and form the logical plan for V_0 and V_1 using V_{01}. We then remove all other CCDs that involve V_0 or V_1 and add new CCDs between C_{01} and views other than V_0 and V_1. The new CCDs are C_{012} and C_{013}. This state is shown in Figure 7.5(b). A black node denotes a materialized CCD. Note that because there is no CCD between V_0 and V_4, there is no CCD between C_{01} and V_4 either. We assume that CCD C_{34} has the highest benefit and it is shown shadowed in Figure 7.5(b).

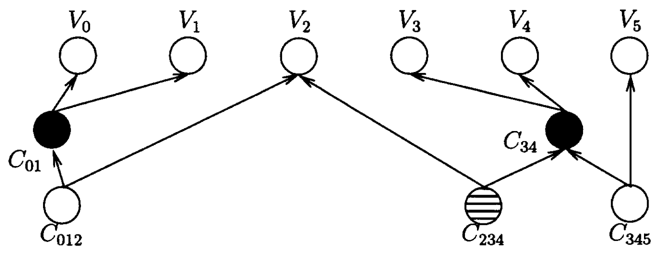
3. We materialize CCD C_{34} and form the logical plan for V_3 and V_4 using C_{34}. We then remove all other CCDs that involve V_3 and V_4 and add new CCDs between C_{34} and all the materialized CCDs and the input views other than V_0, V_1, V_3 and V_4. This state is shown in Figure 7.5(c). We assume that CCD C_{234} has the highest benefit and it is shown shadowed in Figure 7.5(c).


V_0 V_1 V_2 V_3 V_4 V_5
C_{01} C_{02} C_{03} C_{12} C_{13} C_{14} C_{23} C_{24} C_{34} C_{35} C_{45}

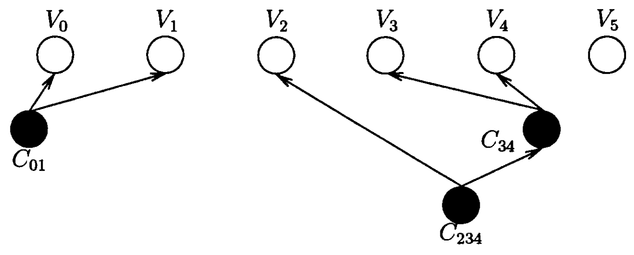
(a) Step 1


V_0 V_1 V_2 V_3 V_4 V_5
C_{01}
C_{23} C_{24} C_{34} C_{35} C_{45}
C_{012} C_{013}

(b) Step 2


V_0 V_1 V_2 V_3 V_4 V_5
C_{01}
C_{34}
C_{012} C_{234} C_{345}

(c) Step 3



(d) Step 4

Figure 7.5 Logical maintenance plan list for six queries

4. We materialize CCD C_{234} and form the logical plan for V_2 and C_{34} using C_{234}. We then remove all other CCDs that involve V_2 and C_{34} and add new CCDs between C_{234} and all materialized CCDs other than C_{34} and between C_{234} and all input views other than V_0, V_1, V_2, V_3 and V_4. We assume no CCD can be added. The resulting state is shown in Figure 7.5(d). Since no CCD is left for consideration, the process terminates. We are left with a set of materialized CCDs $\mathcal{V}' = \{C_{01}, C_{34}, C_{234}\}$ and a global logical plan for maintaining the six input views.

5. We use a query optimizer on the logical maintenance plan to produce a global maintenance plan for all the six queries. The total maintenance cost for $\mathcal{V}$ is: $MC(\mathcal{V}) = mc(C_{01}) + mc(V_0, \{C_{01}\}) + mc(V_1, \{C_{01}\}) + mc(C_{234}) + mc(C_{34}, \{C_{234}\}) + mc(V_2, \{C_{234}\}) + mc(V_3, \{C_{34}\}) + mc(V_4, \{C_{34}\}) + mc(V_5)$

Algorithm 7 Multi-level greedy algorithm for Multiple View Maintenance

```
Input: A set $\mathcal{V}$ of input views to be maintained
Output: A set $\mathcal{V}'$ of views to be transiently materialized and a global view maintenance
    plan that uses views in $\mathcal{V}'$
1: Construct the global logical plan for all the views in $\mathcal{V}$ where every view is computed
   from base relations
2: Initially let $\mathcal{V}' = \emptyset$
3: Let set $\mathcal{VS}$ contain all CCDs of all pairs of views in $\mathcal{V}$
4: while $\mathcal{VS} \neq \emptyset$ do
5:     Remove the CCD $C$ from $\mathcal{VS}$ with the highest benefit and add it into $\mathcal{V}'$. Assume
       that $C$ is a CCD of views $V_1$ and $V_2$.
6:     Modify the global logical maintenance plan so that $V_1$ and $V_2$ are maintained from
       $C$
7:     Remove from $\mathcal{VS}$ all views which are CCDs of $V_1$ (or $V_2$) and some other view
8:     Add to $\mathcal{VS}$ new CCDs between $C$ and each view in $\mathcal{V} \cup \mathcal{V}'$; CCDs between $C$ and
       views in $\mathcal{V}'$ that are ancestors of $C$ in the logical plan do not need to be considered
9: end while
10: Use the query optimizer to compute a global global view maintenance plan using a
    single query optimizer
```

7.4 Experimental Evaluation

We have implemented our approach for using CCDs in the maintenance of multiple materialized views. We have run experiments to measure the performance of our approach in maintaining multiple materialized views, and the quality of the solutions returned.

The experiments were conducted on a machine with a 2.4 GHz Intel Pentium 4 processor running a 2.6 Linux kernel. The machine has 512MB main memory and 80 GB hard disk. We used java as the programming language for the whole experiment implementation.

We created a relational database schema and we recorded meta data about it. This setting allowed us to control easily a number of features useful for the experiments including condition selectivity, condition implication, relation instance cardinalities etc. For the experiments, we generated different workloads of queries and view defined on this schema.

Selecting only CCDs (and not other types of views) to be transiently materialized when maintaining multiple materialized views reduces the search space of the view selection problem significantly and makes possible solving the problem for a large number of views. For measuring the performance of our approaches and the quality of the solutions returned, we run experiments on sets of materialized views which contain from 10 to 120 views. We evaluated both approaches introduced in the previous section, the single-level view selection approach and the multi-level view selection approach.

As discussed in the previous section, in order to evaluate our approaches we need a query optimizer to find a global maintenance plan of a set of materialized views based on a logical maintenance plan. The logical maintenance plan indicates for every view V what other views need to be used for its maintenance (if any). It also provides a rewriting of V using the other views (and possibly base relations). The optimizer should have the following properties:

1. The optimizer should be able to find the optimal maintenance plan of a view from the base relations and return its cost.

2. If the logical plan shows that a view V should be maintained using other views, the optimizer should be able to find the optimal maintenance plan of V using the other views.

Modern database management systems have these properties[1]. Note however that our approaches are independent of the cost model and the optimizer used. For simplicity and flexibility, in our experiments we implemented a simple optimizer. Our cost model measures I/O time which is usually a dominant factor of the query evaluation cost. Our single query optimizer pushes all selection operations down as much as possible and enumerates all possible join sequences to select one with the least cost. Choosing a more sophisticated optimizer is not of great importance for the experiments since our goal is to provide a comparative presentation of different approaches.

In our first experiment we compare the quality of the solutions returned by the two approaches. The quality of the solution returned is represented by the maintenance cost of the solution. This in turn is measured as a percentage of the maintenance cost of the solution returned by the naive maintenance plan where no views are transiently materialized. Clearly, the lower this percentage is, the better the quality of the solution is. We ran both algorithms on a workload W_2. Workload W_2 contains 120 views. Each view contains six occurrences of four relations, two selection conditions and seven join conditions. The selection and join conditions are not necessarily atomic and the views are cyclic. From each relation occurrence a random number of 0 to 4 attributes are projected. The average number of mergeable (join and selection) conditions of a pair of views in the workload W_2 is 4. We ran the two view maintenance algorithms, the single-level view selection greedy algorithm and the multi-level view selection greedy algorithm on a set of views which contains the first n views of W_2 where n varies from 10 to 120.

The result is shown in Figure 7.6. Both algorithms return better solutions than the naive approach. For any number of views to be maintained, the multi-level view

[1]IBM DB2 has the required ability through the EXPLAIN utility, while Microsoft SQL Server provides a similar feature with the WHAT-IF utility.

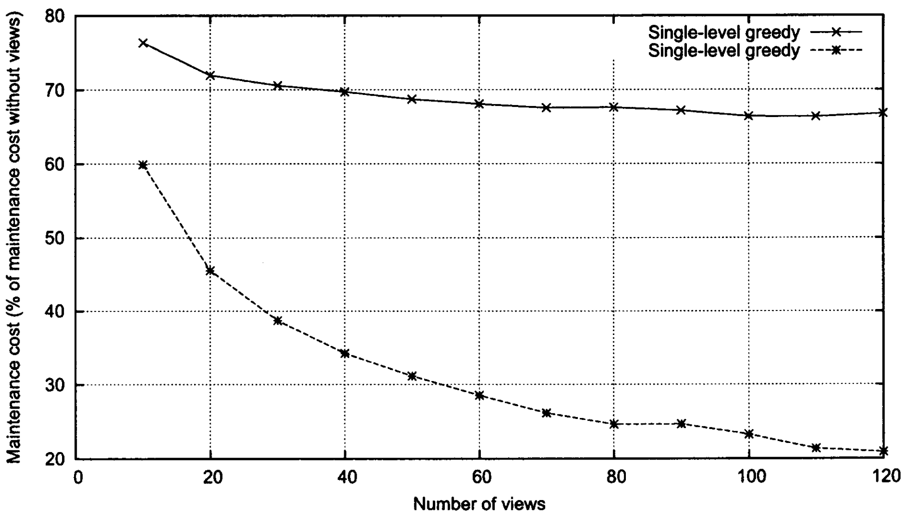


Figure 7.6 Quality vs. number of views for both algorithms

selection greedy algorithm returns better solutions than the single-level view selection greedy algorithm. When the number of views to be maintained increases, the quality of the solutions returned by both algorithms increases too. For example, when the number of views to be maintained increases from 10 to 120, the maintenance cost of the plan generated using the single-level view selection greedy algorithm ranges from 77% to 67% of the cost of the naive plan. The maintenance cost of the plan generated by the multi-level view selection algorithm ranges from 60% to 21% of the cost of the naive plan. This is expected because when the number of views increases, the number of common subexpressions between them increases too, and so does the possibility to find common subexpressions (CCDs in our case) that can bring substantial benefit to the view maintenance cost. Note also that when the number of views to be maintained increases, the improvement of the multi-level view selection greedy algorithm is more important than that of the single-level view selection greedy algorithm. This demonstrates that the multi-level view selection greedy algorithm can better identify and exploit the overlapping of the views.

To further investigate the effect of query overlapping on the quality of the solution returned, we ran the two algorithms on four workloads that differ on the overlapping

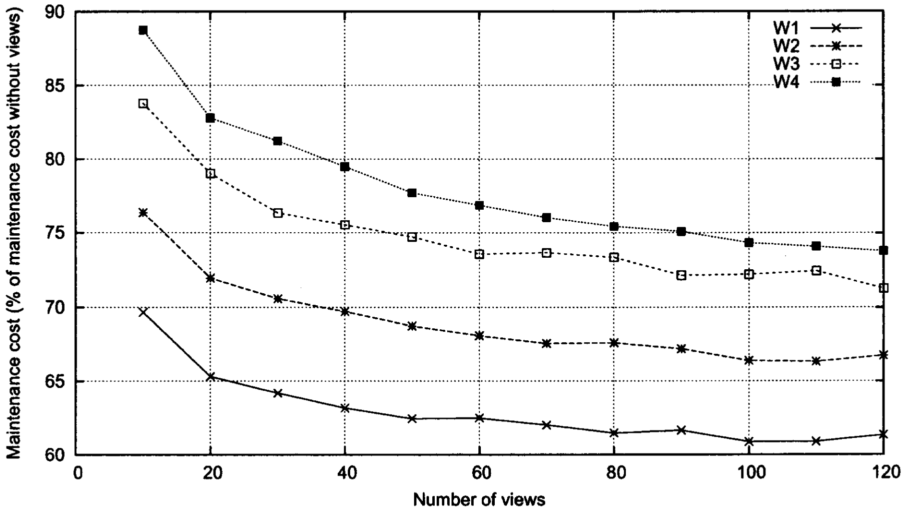


Figure 7.7 Quality vs. number of views for the single-level greedy algorithm

between the views. The overlapping is measured by the number of mergeable (join and selection condition) edges between views in the workload. We generated three workloads W_1, W_3 and W_4 in addition to workload W_2. Workloads W_1, W_3 and W_4 are similar to W_2 in terms of the characteristics of the queries. Only the overlapping of views is different in these workloads: workload W_1 has $5 - 6$ mergeable conditions for each pair of views, W_2 has $3 - 5$ mergeable conditions, W_3 has $1 - 4$ mergeable conditions and W_4 has $0 - 3$ mergeable conditions. The results are shown in figures 7.7 and 7.8. One can see that for both algorithms the higher the overlapping is, the better the quality of the solution is. This is due to the fact that when we increase the overlapping, we also increase the probability for the algorithms to find common subexpressions that bring substantial benefit to the view maintenance cost.

In our third experiment, we compared the execution time of the two view selection algorithms when the number of views to be maintained increases. We considered the workload W_2 and we varied the number of views to be maintained from 10 to 120. The result is shown in Figure 7.9. Note that the measured time is higher than what we can obtain in practice. This is due to the simplicity of our query optimizer. Nevertheless, this does

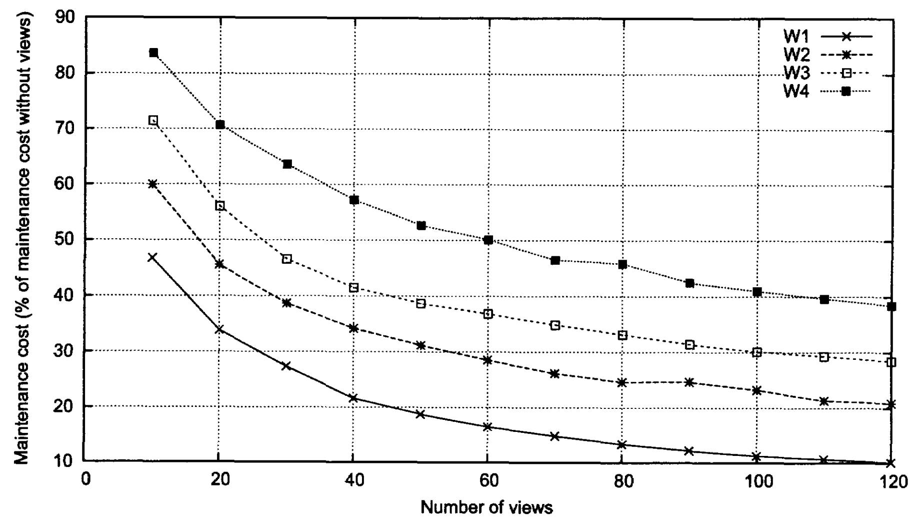


Figure 7.8 Quality vs. number of views for the multi-level greedy algorithm

not affect the comparison between the two approaches. One can see that the single-level greedy performs better than the multi-level greedy algorithm. Both approaches scale well (they are almost linear) on the number of views.

The choice between the two algorithms is a trade off between the quality of the solution sought and the available computation time.

7.5 Conclusion

In this chapter, we propose an application for the concept of CCD: view maintenance using multi-query optimization techniques. We propose two algorithms of multi-query optimization for the view maintenance problem using CCDs. The first one considers materializing only CCDs between queries while the second one also considers CCDs between queries and CCDs and even CCDs between CCDs. Experimental results show that both algorithms scale well to the number of queries we consider in a workload. While the first algorithm performs better on optimization time, the second one generate maintenance plans of better quality. This quality difference gets larger when we have more similarity in the workload. When the database is not big in size and there is not too much similarity in

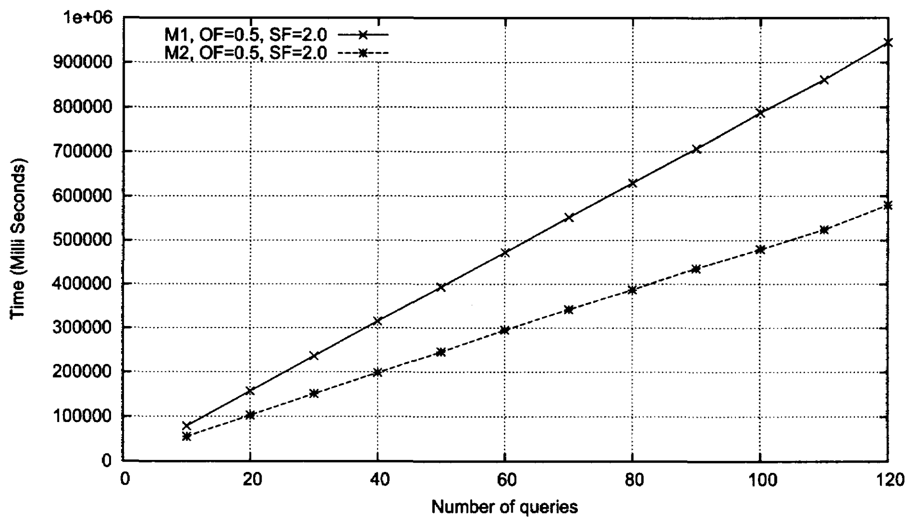


Figure 7.9 Running time vs. number of views for the two algorithms

the workload, the first algorithm may perform better. Otherwise, the second algorithm is preferred.

CHAPTER 8

HEURISTIC VIEW SELECTION FOR MULTIPLE VIEW MAINTENANCE IN IBM DB2

Materialized views (MVs) are used in databases and data warehouses to improve query performance. In this context, a great challenge is to exploit commonalities among the views and to employ multi-query optimization techniques in order to derive an efficient global evaluation plan for refreshing the MVs concurrently. *IBM® DB2® Universal Database*™ product (DB2 UDB and DB2 9) provides two query matching techniques, query stacking and query sharing, to exploit commonalities among the MVs, and to construct an efficient global evaluation plan. When the number of MVs is large, memory and time restrictions prevent one from using both query matching techniques in constructing efficient global plans. An approach that applies the query stacking and query sharing techniques in different steps are suggested in this chapter. The query stacking technique is applied first, and the outcome is exploited to define groups of MVs. The number of MVs in each group is restricted. This allows the query sharing technique to be applied only within groups in a second step. Finally, the query stacking technique is used again to determine an efficient global evaluation plan. An experimental evaluation shows that the execution time of the plan generated by this approach is very close to that of the plan generated using both query matching techniques without restriction. This result is valid no matter how big the database is.

8.1 Introduction

The advent of data warehouses and of large databases for decision support has triggered interesting research in the database community. With decision support data warehouses getting larger and decision support queries getting more complex, the traditional query optimization techniques, which compute answers from the base tables, cannot meet the stringent response time requirements. The most frequent solution used for this problem is to store a number of materialized views (MVs). Query answers are computed using these materialized views instead of using the base tables exclusively. Materialized views are manually or automatically selected based on the underlying schema and database statistics so that the frequent and long-running queries can benefit from them. These queries are rewritten using the materialized views prior to their execution. Experience with the TPC-D benchmark and several customer applications has shown that MVs can often improve the response time of decision support queries by orders of magnitude [151]. This performance advantage is so big that TPC-D [1] had ceased to be an effective performance discriminator after the introduction of the systematic use of MVs [151].

Although this technique brings a great performance improvement, it also brings some new problems. The first one is the selection of a set of views to materialize in order to minimize the execution time of the frequent queries while satisfying a number of constraints. This is a typical data warehouse design problem. In fact, different versions of this problem have been addressed up to now. One can consider different optimization goals (e.g., minimizing the combination of the query evaluation and view maintenance cost) and different constraints (e.g., MV maintenance cost restrictions, MV space restrictions, etc.). A general framework for addressing those problems is suggested in [130]. Nevertheless, polynomial time solutions are not expected for this type of problem. A heuristic algorithm to select both MVs and indexes in a unified way has been suggested in [154]. This algorithm has been implemented in the IBM DB2 design advisor [154].

The second problem is how to rewrite a query using a set of views. A good review of this issue is provided in [73]. Deciding whether a query can be answered using MVs is an NP-hard problem even for simple classes of queries. However, exact algorithms for special cases and heuristic approaches allow one to cope with this problem. A novel algorithm that rewrites a user query using one or more of the available MVs is presented in [151]. This algorithm exploits the graph representation for queries and views (*Query Graph Model - QGM*) used internally in DB2. It can deal with complex queries and views (e.g., queries involving grouping and aggregation and nesting) and has been implemented in the IBM DB2 design advisor.

The third problem is related to the maintenance of the MVs [95]. The MVs often have to be refreshed immediately after a bulk update of the underlying base tables, or periodically by the administrator, to synchronize the data. Depending on the requirements of the applications, it may not be necessary to have the data absolutely synchronized. The MVs can be refreshed incrementally or recomputed from scratch. This chapter focuses on the latter approach for simplicity. When one or more base tables are modified, several MVs may be affected. The technique of multi-query optimization [121] can be used to detect common subexpressions [135] among the definitions of the MVs and to rewrite the views using these common subexpressions. Using this technique one can avoid computing complex expressions more than once.

An algorithm for refreshing multiple MVs in IBM DB2 UDB is suggested in [94]. This algorithm exploits a graph representation for multiple queries (called *global QGM*) constructed using two query matching techniques: query stacking and query sharing. Query stacking detects subsumption relationships between query or view definitions, while query sharing artificially creates common subexpressions which can be exploited by two or more queries or MVs. Oracle 10g also provides an algorithm for refreshing a set of MVs based on the dependencies among MVs [51]. This algorithm considers refreshing one MV using another one which has already been refreshed. This method is similar to the query stacking technique used in DB2 UDB. However, it does not consider using common subsumers for optimizing the refresh process (a technique that corresponds to query sharing used in DB2 UDB). This means they may miss the optimal evaluation plan.

When there are only few MVs to be refreshed, one can apply the method proposed in [94] to refresh all MVs together. This method considers both query stacking and query sharing techniques, and a globally optimized refresh plan is generated. However when the number of MVs gets larger, a number of problems prevent this method from being applied. The first problem relates to the generation of a global plan. When there are many MVs to be refreshed, too much memory is required for constructing a global QGM using both query sharing and query stacking techniques. Further, it may take a lot of time to find an optimal global plan from the global QGM. The second problem relates to the execution of the refresh plan. There are several system issues here. The process of refreshing MVs usually takes a long time, since during this period, MVs are locked. User queries which use some MVs either have to wait for all MVs to be refreshed, or routed to the base tables. Either solution will increase the execution time. Another system issue relates to the limited size of the statement heap which is used to compile a given database statement. When a statement is too complex and involves a very large number of referenced base tables or MVs considered for matching, there may not be enough memory to compile and optimize the statement. One more system issue relates to transaction control. When many MVs are refreshed at the same time (with a single statement), usually a large transaction log is required. This is not always feasible. Further if something goes wrong during the refreshing, the whole process has to start over.

To deal with the problems above, the following approach is proposed. When too many MVs need to be refreshed and the construction of a global QGM based on query stacking and query sharing together is not feasible, the MV set is partitioned into smaller groups based on query stacking alone. Then, query sharing technique is applied to each group independently. Consequently, the execution plan is separated into smaller ones, each involving fewer MVs. Intuitively, by partitioning MVs into smaller groups, query sharing are restricted to views within groups while query stacking are restricted to views of different groups. In this way MVs from the lower groups are potentially exploited by the groups above. An implementation and experimental evaluation of this approach shows that this method has a comparable performance to the approach that uses a globally optimized evaluation plan while at the same time avoiding the aforementioned problems.

The next section presents the QGM model and the two query matching techniques. Section 3 introduces the MV partition strategy. Section 4 presents the experimental setting and results. Section 5 concludes and suggest future work.

8.2 Query Graph Model and Query Matching

In this section, we introduce the concept of QGM model which is used in DB2 UDB to graphically represent queries. First the QGM model for a single query is introduced. Then it is extended to a global QGM model for multiple queries. This extension requires the concept of query matching using both query stacking and query sharing techniques.

8.2.1 Query Graph Model

The QGM model is the internal graph representation for queries in the DB2 UDB database management system. It is used in all steps of query optimization in DB2 UDB such as parsing and semantic checking, query rewriting transformation and plan optimization. An example is shown next to demonstrate how queries are represented in the QGM model. A query in the QGM model is represented by a set of boxes (called *Query Table Boxes – QTBs*) and arcs between them. A QTB represents a view or a base table. Typical QTBs are *select* QTBs and *group-by* QTBs. Other kinds of QTBs include the *union* and the *outer-join* QTBs.

Below, a query with *select* and *group-by* operations is given in SQL.

```
select c.c3, d.d3, sum(f.f3) as sum
from c, d, fact f
where c.c1 =f.f1 and d.d1 = f.f2 and c.c2 = 'Mon' and d.d2 > 10
group by c.c3,d.d3 having sum > 100
```

Figure 8.1 shows a simplified QGM representation for this query.

8.2.2 Query Matching

To refresh multiple MVs concurrently, a global QGM for all of the MVs is generated using the definitions tied together loosely at the top. All QTBs are grouped into different levels with the base tables belonging to the bottom level. Then, from bottom to top, each QTB is compared with another QTB to examine whether one can be rewritten using the other. If this is the case, it is called that the latter QTB *subsumes* the former QTB. The latter QTB is called the *subsumer* QTB while the former is called the *subsumee* QTB. A rewriting of the subsumee QTB using the subsumer QTB may also be generated at the time of matching, and this aditional work is called *compensation*. The comparison continues with the parent QTBs of both the sumsumer and subsumee QTBs. This process continues until no more matches can be made.

If the top QTB of one MV subsumes some QTB of another MV, then the former MV subsumes the latter MV. This kind of matching is called *query stacking* because it

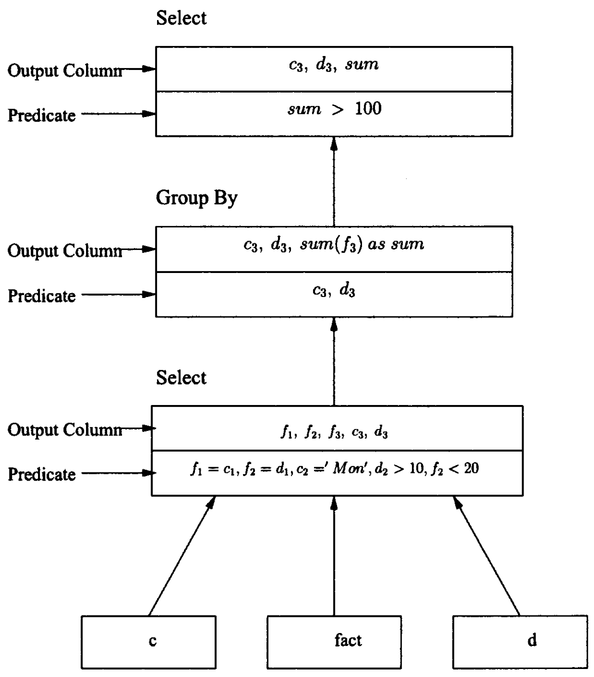
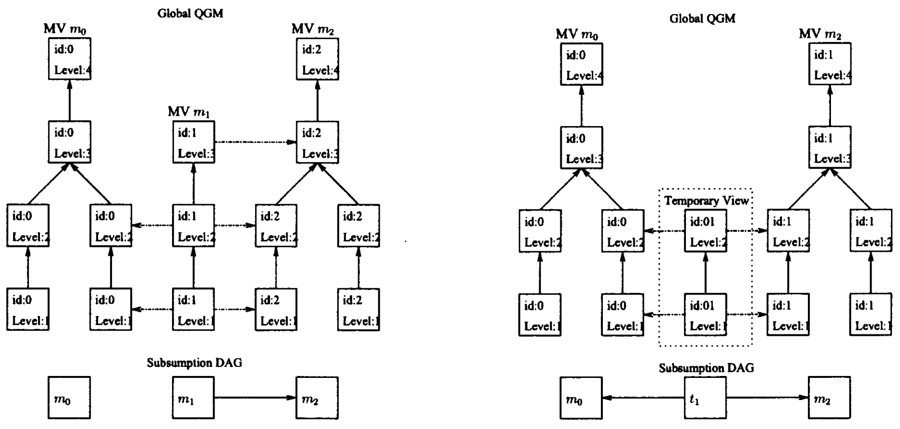


Figure 8.1 QGM graph for query Q_1.

ultimately determines that one MV can be rewritten using the other and the subsumee MV can be “stacked” on the subsumer MV.

In some cases, it is possible that one may not find a strict subsumption relationship between two MVs even if they are quite close to having one. For instance, a difference in the projected attributes of two otherwise equivalent MVs will make the matching fail. The matching technique of DB2 UDB is extended in [94] to deal with this case. In some cases when there is no subsumption relationship between two MVs, an artificially built common subexpression (called *common subsumer*) can be constructed such that both MVs can be rewritten using this common subsumer. Because this common subsumer is “shared” by both MVs, this matching technique is called *query sharing*. With query sharing, matching techniques can be applied to a wider class of MVs.

In Figure 8.2, examples of query matching techniques are shown. In Figure 8.2(a), the matching using query stacking only is shown. In this example, there are three queries m_0, m_1, m_2. For each query pair, their QTBs are matched from bottom to top until the top QTB of the subsumer query is reached. Since there is a successful matching of the top QTB of query m_1 with some QTB of query m_2, there is a subsumption relationship from



(a) Query stacking (b) Query sharing

Figure 8.2 Query matching.

m_1 to m_2 (m_1 subsumes m_2). This is not the case with queries m_0 and m_1. The matching process defines a query *subsumption DAG* shown in Figure 8.2(a). In this DAG, each node is a query. Since query m_1 subsumes query m_2, a directed line can be drawn from m_1 to m_2. In Figure 8.2(a), the subsumption DAG for queries m_0, m_1 and m_2 is shown. There is one subsumption edge from m_1 to m_2 while query m_0 is a disconnected component. This subsumption DAG can be used to optimize the concurrent execution of the three queries. For example, one can compute the results of m_0 and m_1 from base tables. Then, query m_2 can be computed using m_1 based on the rewriting of m_2 using m_1, instead of computing it using exclusively base tables. Query m_2 is "stacked" on m_1 since it has to be computed after m_1.

In this example, it is also observed that although m_2 can be rewritten using m_1, m_0 can not be rewritten using m_2 or vise versa. one cannot even find a successful match of the bottom QTBs of m_0 and m_2 based on query stacking. This is quite common in practice. When trying to answer m_0 and m_2 together, and a subsumption relationship between them cannot be found, one can try to create a new query, say t_1, which can be used to answer both queries m_0 and m_2. This newly constructed query is called *common subsumer* of the two queries m_0 and m_2 because it is constructed in a way so that both queries m_0 and m_2 can be rewritten using it. Although the common subsumer is not a user query to be answered, its answer can be computed and then used to compute the answers of both queries m_0 and m_2. As a "common part" between m_0 and m_2, t_1 is computed only once, and therefore, its computation might bring some benefit in the concurrent execution of m_0 and m_2. In the example of Figure 8.2(b), there is no subsumption edge between m_0 and m_2. However, after adding a common subsumer t_1 of m_0 and m_2, there are two subsumption edges: one from t_1 to m_0 and one from t_1 to m_2.

The subsumption relationship graph is a DAG because there is no cycle in it. In most cases, if one MV subsumes another one, the latter one cannot subsume the former one. Nevertheless in some cases, two or more MVs may subsume each other, thus generating a subsumption cycle. The DB2 UDB matching techniques will ignore one subsumption relationship randomly, when this happens, to break any cycles. This will guarantee the result subsumption graph to be a real DAG. In drawing a subsumption DAG, if $m_1 \rightarrow m_2$, and $m_2 \rightarrow m_3$, no transitive subsumption edge is shown in the DAG such as $m_1 \rightarrow m_3$. However, this subsumption relationship can be directly derived from the DAG, and it is of the same importance as the other subsumption relationships in optimizing the computation of the queries.

8.3 Group Partition Strategy

If there are too many MVs to be refreshed then, as we described above, one may not be able to construct the global QGM using both query stacking and query sharing techniques. The goal is to partition the given MV set into groups that are small enough so that both query matching techniques can be applied, and the problems mentioned in the introduction is not an issue. The approach introduced in this section first creates a subsumption DAG using query stacking only which is a much less memory and time consuming process. This subsumption DAG is used for generating an optimal global evaluation plan. The different levels of this plan determine groups of materialized views on which query sharing is applied.

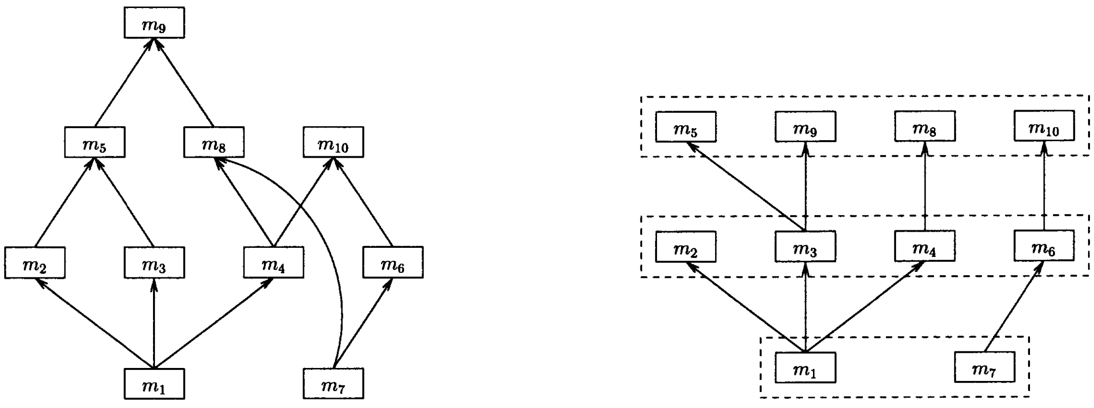
Building an Optimal Plan Using Query Stacking Given a set of MVs to be refreshed, a global QGM using only query stacking is constructed. Then, a subsumption DAG as described in the previous section is created . The query optimizer is applied to choose an optimal plan for computing each MV using either exclusively base relations or using other MVs in addition to base tables as appropriate. The compensations stored in the global QGM of a MV using other MVs are used to support this task. The optimizer decides whether using a MV to compute another MV is beneficial when compared to computing it from the base relations. These optimal "local" plans define an optimal global plan for refreshing all the queries. Figure 8.3(a) shows an example of a subsumption DAG for ten

MVs. Transitive edges are ignored for clarity of presentation. Figure 8.3(b) shows an optimal global evaluation plan.

Groups are defined by the views in the optimal plan. If one group is still too big for the query sharing technique to be applied, it can be divided into suitable subgroups heuristically based on some common objects and operations within the group or possibly randomly.

Adding also Query Sharing By considering the query stacking technique only, some commonalities between queries may be missed which can be beneficial to the refreshing process. Therefore, both query matching techniques, query stacking and query sharing, are enabled within each group to capture most of those commonalities. This process is outlined below.

1. Apply query stacking and query sharing techniques to the MVs of each group. Even though no subsumption edges will be added between MVs in the group, some common subsumers may be identified and new subsumption edges will be added from those common subsumers to MVs in the group.
2. Apply the query stacking technique to the common subsumers of one group and the MVs of lower groups. Lower groups are those that comprise MVs from lower levels



(a) Subsumption DAG (b) Optimal global logical plan

Figure 8.3 Query stacking based refreshing.

of the optimal global plan. This step might add some new subsumption edges from MVs to common subsumers in the subsumption DAG.

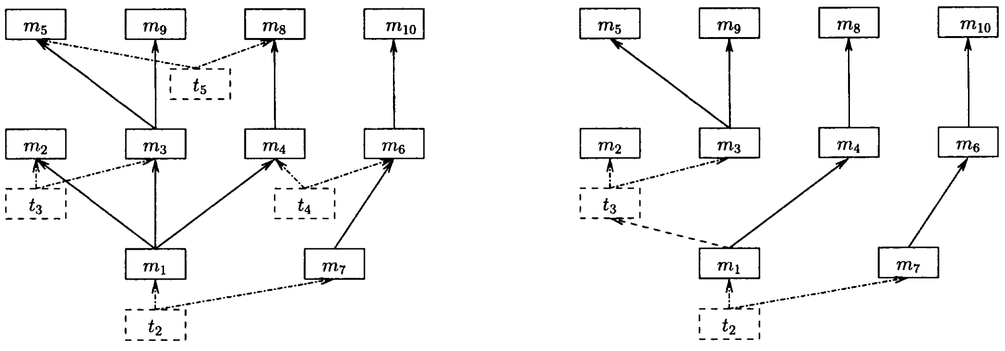
3. Using a common subsumer induces additional view materialization cost. However, if this cost is lower than the gain obtained in computing the MVs that use this common subsumer, it is beneficial to materialize this common subsumer. Such a common subsumer is called a candidate common subsumer. The use of a candidate common subsumer may prevent the use of other candidate common subsumers. Heuristics are applied to retain those candidate common subsumers such that no one of them prevents the use of the others and together yield the highest benefit. This process is applied from the bottom level to the top level in the subsumption DAG.
4. Have the optimizer to create a new optimal global plan using the retained candidate common subsumers. Compared to the optimal global plan constructed using only query stacking, this optimal global plan contains also some new MVs, in the form of the retained candidate common subsumers.

During the refreshing of the MVs, a common subsumer is first materialized when it is used for refreshing another MV and it is discarded when the last MV that uses it has been refreshed.

Figure 8.4 shows the construction of an optimal global plan taking also query sharing into account. Figure 8.4(a) shows the subsumption DAG of Figure 8.4(b) along with some common subsumers. Dotted directed edges indicate subsumption edges involving common subsumers. Among the candidate common subsumers, some of them are retained in the optimal global plan. Such an optimal global plan is shown in Figure 8.4(b). This optimal global plan will have a better performance than the one of Figure 8.3(b).

8.4 Performance Testing

For the experimental evaluation a database with a star schema is considered. Also a number of MVs to be refreshed (16 in the test) are considered. The number of MVs are kept small small enough so that, in finding an optimal global evaluation plan, both the query stacking and query sharing techniques can be applied without restrictions. The performance comparison test is not feasible when there are too many MVs. The goal is to compare the



(a) Subsumption DAG (b) Optimal global logical plan

Figure 8.4 Query sharing based refreshing.

performance of different approaches. The performance of four kinds of MV refreshing approaches are compared for different sizes of databases. These approaches are as follows:

1. *Naive Refreshing(NR):* Refresh each MV one by one by computing its new state using the base tables referenced in the MV definition exclusively. This approach disallows any multi-query optimization technique or other already refreshed MV exploitation.

2. *Stacking-Based Refreshing(STR):* Refresh each MV one by one in the topological order induced by the optimal global plan constructed using the query stacking technique only (for example, the optimal global plan of Figure 8.3(b)) in the previous section. This approach disallows query sharing. With this approach some MVs are refreshed using the base relations exclusively. Some other MVs are refreshed using other MVs if they have a rewriting using those MVs that are in lower groups in the optimal global plan.

3. *Group-Sharing-Based Refreshing(SHR):* Refresh each MV in the topological order induced by the optimal global evaluation plan constructed using query stacking first and then query sharing only within groups (for example, the optimal global plan of Figure 8.4(b)).

4. *Unrestricted-Sharing-based Refreshing(USR):* Refresh all MVs together based on an optimal global plan constructed using, without restrictions, both query matching techniques.

The test schema consists of one fact table and three dimension tables. Each dimension table has 10,000 tuples, while the number of tuples of the fact table varies from 100,000 to 10,000,000. The set of MVs are refreshed with each one of the four refreshing approaches mentioned above, and the overall refreshing time is measured. The following table shows the platform on which this experiment is run configuration.

Model	*OS*	*Memory*	*CPUs*	*rPerf*	*Database*
P640-B80	AIX 5.2 ML06	8 GB	4	3.59	DB2 V91

Figure 8.5 shows the experimental results to refresh a set of views using the above schemes. For any size of the database, the unrestricted-sharing-based approach always has the best performance since it allows unrestricted application of both query stacking and query sharing techniques. The group-sharing-based approach has the second best performance because, even though it exploits both query matching techniques, they are considered separately in different steps and query sharing is restricted only within groups. The stacking-based approach is the next in performance because it cannot take advantage of the query sharing technique. Finally, far behind in performance is the naive approach which does not profit of any query matching technique. As can be seen in Figure 8.5 the group-sharing based approach is very close to the unrestricted sharing approach. This remark is valid for all database sizes and the difference in those two approaches remains insignificant. In contrast, the difference between the naive and the pure stacked approach compared to other two grows significantly as the size of the database increases. In a real data warehouse, it is often the case that MVs have indexes defined on them. The group-sharing-based refresh may outdo the unrestricted approach if there is occasion to exploit the indexes of MVs when used by the refreshing of the higher group MVs.

8.5 Conclusion

The problem addressed in this chapter is to refresh concurrently multiple MVs. In this context, two query matching techniques, query stacking and query sharing, are used in DB2 to exploit commonalities among the MVs, and to construct an efficient global evaluation plan. When the number of MVs is large, memory and time restrictions prevent from being used both query matching techniques in constructing efficient global plans. An heuristic

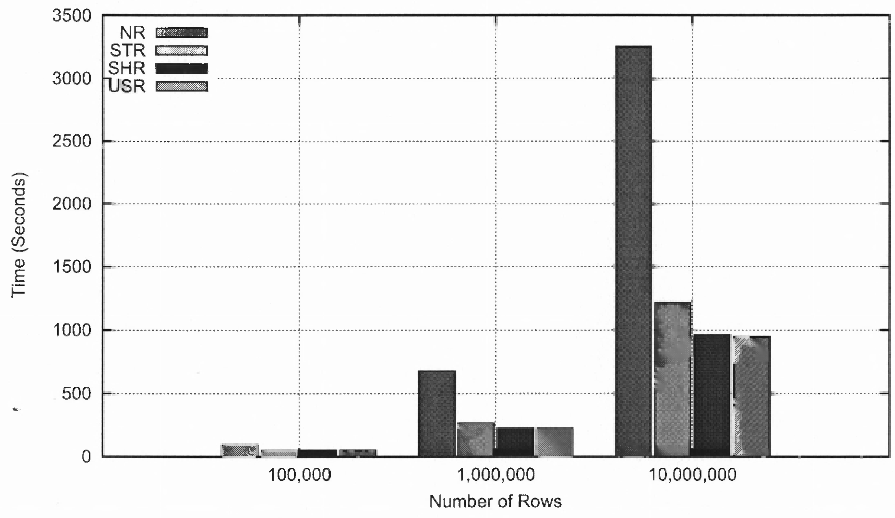


Figure 8.5 Performance test result for different refreshing method.

approach that applies the two techniques in different steps is suggested. The query stacking technique is applied first, and the generated subsumption DAG is used to define groups of MVs. The number of MVs in each group is smaller than the total number of MVs. This will allow the query sharing technique to be applied only within groups in a second step. Finally, the query stacking technique is used again to determine an efficient global evaluation plan. An experimental evaluation shows that the execution time of the optimal global plan generated by our approach is very close to that of the optimal global plan generated using, without restriction, both query matching techniques. This result is valid no matter how big the database is.

The approach can be further fine-tuned to deal with the case where the groups of MVs turn out to be too small. In this case, merging smaller groups into bigger ones may further enhance the potential for applying the query sharing technique. Although it is assumed that a complete re-population of all MVs in our approach for simplicity is applied, the approach can actually be applied to incremental refreshing of MVs. In a typical data warehouse application, there usually exist indexes on MVs. The approach can be extended to adapt to this scenario. Actually, because the existence of indexes increases the complexity of the global QGM, the approach may achieve better performance.

CHAPTER 9

A DYNAMIC VIEW MATERIALIZATION SCHEME FOR SEQUENCES OF QUERY AND UPDATE STATEMENTS

9.1 Introduction

In a data warehouse design context, a set of views is selected for materialization in order to improve the overall performance of a given workload. Typically, the workload is a set of queries and updates. In many applications, the workload statements come in a fixed order. This scenario provides additional opportunities for optimization. Further, it modifies the view selection problem to one where views are materialized dynamically during the workload statement execution and dropped later to free space and prevent unnecessary maintenance overhead. The problem of dynamically selecting and dropping views when the input is a sequence of statements in order to minimize their overall execution cost under a space constraint is addressed in this chapter. It is modeled as a shortest path problem in directed acyclic graphs. Then a heuristic algorithm is provided that combines the process of finding the candidate set of views and the process of deciding when to create and drop materialized views during the execution of the statements in the workload. The experimental results show that this dynamic approach performs better than previous static and dynamic approaches.

Data warehousing applications materialize views to improve the performance of workloads of queries and updates. The queries are rewritten and answered using the materialized views [73]. A central issue in this context is the selection of views to materialize in order to optimize a cost function while satisfying a number of constraints [132, 148]. The cost function usually reflects the execution cost of the workload statements that is, the cost of evaluating the workload queries using possibly the materialized views and the cost of applying the workload updates to the affected base relations and materialized views. The constraints usually express a restriction on the space available for view materialization [75], or a restriction on the maintenance cost of the materialized views [71], or both [106]. Usually the views are materialized before the execution of the first statement and remain materialized until the last statement is executed. This is the static view selection problem which has been studied extensively during the last decade [75, 133, 71]. Currently most of the commercial DBMSs (e.g., IBM DB2, MS SQL Server, Oracle) provide tools that recommend a set of views to materialize for a given workload of statements based on a static view materialization scheme [153, 7, 40].

If a materialized view can be created and dropped later during the execution of the workload, we face a dynamic version of the view selection problem. A dynamic view selection problem is more complex than its static counterpart. However, it is also more flexible and can bring more benefit since a materialized view can be dropped to free useful space and to prevent maintenance overhead. When there is no space constraint and there are only queries in the workload, the two view materialization schemes are the same: any view that can bring benefit to a query in the workload is materialized before the execution of the queries and never dropped.

Although in most view selection problems the workload is considered to be unknown or a set of statements without order, there are many applications where the workload forms a sequence of statements. This means that the statements in the workload are executed in a specific order. For example, in a typical data warehousing application, some routine queries are given during the day time of every weekday for daily reports; some analytical queries are given during the weekend for weekly reports; during the night the data warehouse is updated in response to update statements collected during the day. This is, for instance, a case where the workload is a sequence of queries and updates. Such a scenario is shown in Figure 9.1. The information on the order of the statements in the workload is important in selecting materialized views.

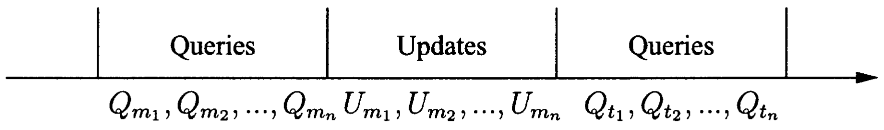

Queries
Updates
Queries
$Q_{m1}, Q_{m2}, \dots, Q_{mn}$ $U_{m1}, U_{m2}, \dots, U_{mn}$ $Q_{t1}, Q_{t2}, \dots, Q_{tn}$

Figure 9.1 A workload as a sequence.

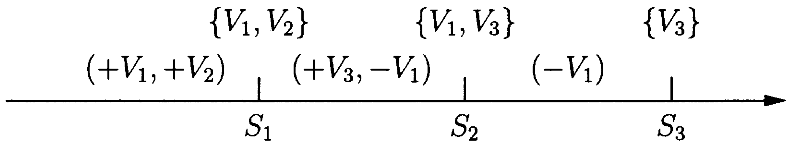


Figure 9.2 Output of the problem.

The solution to the view selection problem when the input is a sequence of statements can be described using a sequence of create view and drop view commands before the execution of every statement. An alternative representation determines the views that are materialized during the execution of every statement. Figure 9.2 shows these two representations. $+V_i$ $(-V_i)$ denotes the materialization (de-materialization) of view V_i.

This chapter addresses the dynamic view selection problem when the workload is a sequence of query and update statements. This problem is more complex than the static one because not only which views to materialize, but also when to materialize and drop them with respect to the workload statements need to be decided. The main contributions of this chapter are as follows:

1. The problem is exploited as a shortest path problem in a directed acyclic graph (DAG) [9]. Unlike that approach, the approach presented in this chapter generates the DAG in a dynamic way. Therefore, it is autonomous in the sense that it does not rely on external modules for constructing the DAG.

2. In order to construct the nodes in the DAG, The "maximal" common subexpressions of queries and/or views are extracted and utilized. The rewritings of the queries using these common subexpressions are produced at the same time thus avoiding the application of expensive processes that match queries to views.

3. A heuristic approach that controls the generation of nodes for the DAGs is based on nodes generated in previous steps suggested.

4. The new approach is implemented and an extensive experimental evaluation conducted. The results show that the new approach performs better than previous static and dynamic approaches.

9.2 Related Work

In order to solve a view selection problem, one has to determine a search space of candidate views from which a solution view set is selected [135]. The most useful candidate views are the common subexpressions on queries since they can be used to answer more than one query. Common subexpression for pure group-by queries can be determined in a straightforward way [75]. In [56] this class of queries is extended to comprise, in addition, selection and join operations and nesting. In [30] the authors elaborate on how to common subexpressions of select-project-join queries without self-join can be found. This results are extended in [135] to consider also self-joins. Currently, most major commercial DBMSs provide utilities for constructing common subexpressions for queries [8, 151]. More importantly, they provide utilities to estimate the cost of evaluating queries using materialized views. The What-If utility in Microsoft SQL Server [7] and the EXPLAIN utility in IBM DB2 [154] are two such examples.

One version of the dynamic view selection problem has been addressed in the past in [41] and [90]. Kotidis et al. [90] show that using a dynamic view management scheme, the solution outperforms the optimal solution of the static scheme. Both approaches focus on decomposing and storing the results of previous queries in order to increase the opportunity of answering subsequent queries partially or completely using these stored results. However, both approaches are different to ours since they assume that the workload of statements is unknown. Therefore, they focus on predicting what views to store in and what views to delete from the cache.

Agrawal et al. [9] consider a problem similar to ours. They model the problem as a shortest path problem for a DAG. However, their approach assumes that the candidate view set from which views are selected for materialization is given. This assumption is problematic in practice because there are too many views to consider. In contrast, our approach assumes that the input to the problem is a sequence of queries and update statements from which it constructs candidate views by identifying common subexpressions among the statements of the sequence.

9.3 Problem Specification

It is assumed that a sequence $\mathcal{S} = (S_1, S_2, ..., S_n)$ of query and update statements is provided as input. The statements in the sequence are to be executed in the order they appear in the sequence. If no views are materialized, the total execution cost of $\mathcal{S}$ includes (a) the cost of evaluating all the queries in $\mathcal{S}$ over the base relations, and (b) the cost of updating the base relations in response to the updates in $\mathcal{S}$. If the materialized views are created just before the execution of statement S_i and dropped before the execution of statement S_j, then the total execution cost includes (a) the cost of evaluating the queries in $(S_1,\ ...,\ S_{i-1})$ and in $(S_j,\ ...,\ S_n)$ over the base relations, (b) the cost of materializing view V (i.e., the cost of computing and storing V), (c) the cost of evaluating all queries in $(S_i, ..., S_{j-1})$ *using the materialized view* V (V is used only if it provides some benefit), (d) the cost of updating view V in response to the changes of the base relations resulting by the update statements in $(S_i, ..., S_{j-1})$, and (e) the cost of updating the base relations in response to the updates in $\mathcal{S}$. For simplicity the cost of dropping the materialized view V is ignored. The total execution cost of $\mathcal{S}$ when multiple materialized views are created and dropped on different positions of sequence $\mathcal{S}$ is defined in a similar way. Since the cost of updating the base relations in response to updates in $\mathcal{S}$ is fixed, it is considered as an overhead cost, and is ignored in the following.

The problem addressed in this chapter can be now formulated as follows. Given a sequence of n queries and updates $(S_1, S_2, ..., S_n)$ and a space constraint B, find a sequence $(O_1, O_2, ..., O_n)$ of sets of "create view" and "drop view" statements such that (a) the total execution cost of $\mathcal{S}$ is minimized and, (b) the space used to materialize views during the execution of $\mathcal{S}$ does not exceed B. Each create view statement contains also the definition of the view to be created. The set O_i of create view and drop view statements is executed before the execution of the statement S_i

The output of this dynamic view selection problem can also be described by a sequence of sets of views $l = (C_1, C_2, ..., C_n)$. Each set C_i contains the views that have been created and not dropped before the evaluation of the statement S_i. l is called a solution to the problem. Further, if the views to be materialized can only be selected from a set of candidate views $\mathcal{V}$, l is called a solution to the problem for the view set $\mathcal{V}$.

9.4 Modeling the Dynamic View Selection Problem

This section shows how the dynamic view selection problem for a sequence of query and update statements can be modeled as a shortest path problem on a directed acyclic graph following the approach introduced by [9]. That approach assumes that the set of candidate views (i.e., the pool from which views can be chosen to be materialized) is given. The approach presented in this chapter is different. As shown later in this section, the candidate views are dynamically constructed from the workload statement by considering common subexpressions among them.

S. Agrawal et al. [9] show how to model the problem assuming that a candidate set of views $\mathcal{V}$ is given and contains only one view V. There are only two options for each statement S_i in the workload: either the view V is materialized or not. Those two options are represented as $C_i^0 = \{\}$ and $C_i^1 = \{V\}$, respectively, in a solution where C_i is the set of views that are materialized before the execution of S_i. Now the process of constructing a directed acyclic graph (DAG) can be described as follows:

1. For each statement S_i in the workload, create two nodes N_i^0 and N_i^1 which represent the execution of statement S_i without and with the materialized view V, respectively. Create also two virtual nodes N_0 and N_{n+1} which represent the state before and after the execution of the workload, respectively. Label the node N_i^0 by the empty set $\{\}$ and the node N_i^1 by the set $\{V\}$.

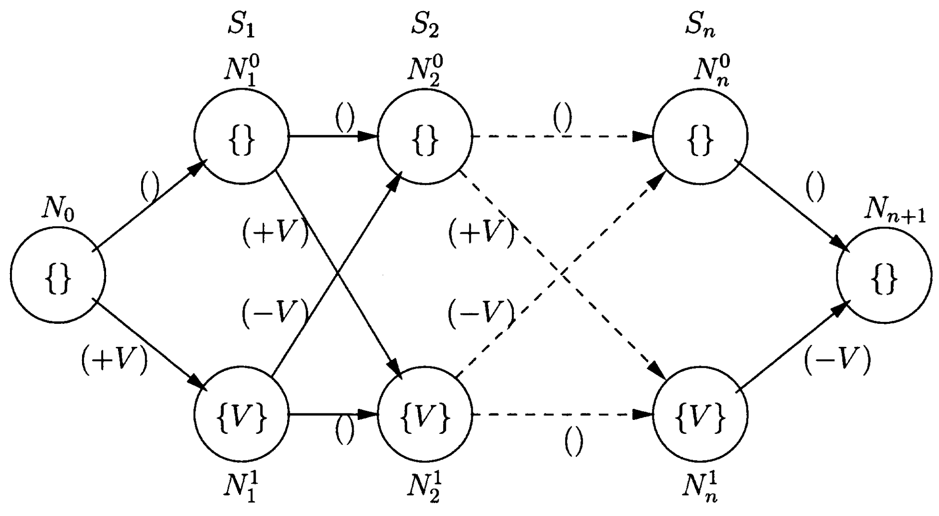


Figure 9.3 Directed acyclic graph for candidate view set $\{V\}$.

2. Add an edge from each node of S_i to each node of S_{i+1} to represent the change in the set of materialized views. If both the source node and the target node are labeled by the same set, then label the edge by a empty sequence "()". If the source node is labeled by $\{\}$ and the target node is labeled by $\{V\}$, then label the edge by a sequence $(+V)$ to represent the operation "create materialized view V". If the source node is labeled by $\{V\}$ and the target node is labeled by $\{\}$, then label the edge by $(-V)$ to represent the operation "drop materialized view V".

3. Compute the cost of each edge from a node of S_i to a node of S_{i+1} as the sum of: (a) the cost of materializing V, if a $(+V)$ labels the edge, (b) the cost of dropping V, if $(-V)$ labels the edge, and (c) the cost of executing the statement S_{i+1} using the set of views that label the target node of the edge.

Figure 9.3 shows the DAG constructed the way described above. Each path from the node N_0 to the node N_{n+1} represents a possible execution option for the workload. The shortest path among them represents the optimal solution for the dynamic view selection problem. The labels of the edges in the path denote the solution represented as a sequence of "create view" and "drop view operations". The labels of nodes in the path denote the solution represented as a sequence of sets of materialized views.

For a candidate set $\mathcal{V}$ of m views, one can construct a DAG in a similar way. Instead of having two nodes for each statement in the workload, 2^m nodes are created, one for each subset of $\mathcal{V}$. Again, the shortest path represents the optimal solution for the dynamic view selection problem.

The shortest path problem for a DAG can be solved by well known algorithms in linear time on the number of edges of the DAG. Therefore the complexity of the process is $O(n \cdot 2^{2m})$ where m is the number of candidate views and n is the number of statements in the workload. To compute the cost of each edge, an optimizer can be used to assess the cost of executing the target statement using the corresponding set of materialized views. This is too expensive even for a small number of candidate views. In practice, the set of candidate views is expected to be of substantial size. The dynamic view selection problem is shown to be NP-hard [9]. Therefore, a heuristic approach must be employed to efficiently solve this problem.

In practice, it cannot be assumed that the set of candidate views is given. It has to be constructed. Tools that suggest candidate views for a static view selection problem [9] are not appropriate for determining candidate views for a dynamic view selection problem. Our approach constructs candidate views which are common subexpressions of queries in the workload and of views. More specifically, subexpressions are considered of two queries which represent the maximum commonalities of these two queries as these are defined in [135]. An additional advantage of this approach is that the rewritings of the queries using the common subexpressions are computed along with the common subexpressions. These rewritings can be fed to the query optimizer to compute the cost of executing the queries using the materialization of the common subexpressions. If a view V is defined as a common subexpression of a set of queries $\mathcal{Q}$, each query $Q \in \mathcal{Q}$ is called a parent of the V. Every $Q \in \mathcal{Q}$ can be answered using V. Only rewritings of a query Q using its common subexpression with other queries and views are considered. A major advantage of this approach is that one do not have to check whether a query matches a view (i.e., check if there is a rewriting of the query using the view) which is in general an expensive process. In the following, rewriting of a query using a common subexpression is ignored if the cost of evaluating this rewriting is not more than the cost of evaluating the query over base relations.

The process of finding a solution for the dynamic view selection problem cab be further simplified as follows. Assume a view V is the only view in the candidate view set. Let $\{Q_i, ..., Q_j\}$ be the parent view set of V where Q_i and Q_j are the first and last parent query of V appearing in the sequence of the workload. If there are no update statements in the workload, we can get the solution directly by creating V before the execution of Q_i and drop it after the execution of Q_j. If the cost of this solution is less than the cost of the solution that does not materialize V, then it is optimal. If there are update statements in the workload, the shortest path algorithm has to be applied to determine the optimal solution. During the construction of the DAG, one does not need to consider materializing V before Q_i and after Q_j. If there are multiple candidate views, this simplification still applies. This process reduces the total number of nodes in the DAG.

To solve the dynamic view selection problem, one can generate different views that are common subexpressions of subsets of the queries in the workload. Finding the solution for this problem using the shortest path algorithm is expensive because of the large number of candidate views and the large number of nodes in the DAG. The next section introduces a heuristic approach that combines the process of generating candidate views with the process of selecting views for a solution of the view selection problem.

9.5 A Heuristic Approach

The heuristic approach starts by considering that there are only queries in the workload. The next section discusses how this approach can be extended to the case where there are also update statements in the workload.

The heuristic approach described in 9 uses two solution merging functions $Merge1$ and $Merge2$. Each function takes as input two solutions to the dynamic view selection problem, each one for a specific set of candidate views, and outputs a new solution.

Consider two solutions $l_1 = (C_1^1, C_2^1, ..., C_n^1)$ and $l_2 = (C_1^2, C_2^2, ..., C_n^2)$, for the candidate view sets $\mathcal{V}_1$ and $\mathcal{V}_2$, respectively. Function $Merge1$ is analogous to function $UnionPair$ [9]. It first creates a DAG as follows: for each statement S_i in the workload, it creates two nodes, one labeled by C_i^1, the other labeled by C_i^2. If the new view set $C_i = C_i^1 \cup C_i^2$ satisfies the space constraint B, it also creates another node labeled by C_i. In addition it creates two virtual nodes representing the starting and ending states. Finally it creates edges as explained earlier from nodes of query S_i to nodes of query S_{i+1}. Once the DAG is constructed, it returns the solution corresponding to the shortest path in the DAG.

Function $Merge1$ does not add new views to the set of candidate views. Instead, for each statement, it might add one more alternative which is the union of two view sets. In

Algorithm 8 Function $Merge2$

```
Input: solution $l_1$, solution $l_2$, space constraint $B$
Output: solution $l$
1: Create a list of solutions $\mathcal{L}$ which initially contains $l_1$ and $l_2$
2: for each view $V_1$ in $l_1$ and each view $V_2$ in $l_2$ do
3:    Find the set of common subexpressions $\mathcal{V}_{12}$ of $V_1$ and $V_2$
4:    for each view $V \in \mathcal{V}_{12}$ do
5:       Find the solution for the candidate view set $\{V\}$ and add it into $\mathcal{L}$
6:    end for
7: end for
8: Find the solution $l$ from $\mathcal{L}$ with the lowest execution cost and remove it from $\mathcal{L}$
9: for all solutions in $\mathcal{L}$ do
10:   Find the solution $l'$ with the lowest execution cost and remove it from $\mathcal{L}$
11:   $l = Merge1(l, l')$
12: end for
13: Return $l$
```

Algorithm 9 A heuristic algorithm for the dynamic view selection problem

Input: *list of queries $\mathcal{S}$, space constraint B*

Output: *solution l*

```
1: Create a set of solutions $\mathcal{L}$ which is initially empty
2: for each pair of queries $S_i$ and $S_j \in \mathcal{S}$ do
3:     Find the set of common subexpressions $\mathcal{V}_{ij}$ of $S_i$ and $S_j$
4:     Find the solution for each view set $\mathcal{V} = \{V\}$ where $V \in \mathcal{V}_{ij}$ and add it into $\mathcal{L}$
5: end for
6: Find the solution $l$ from $\mathcal{L}$ with the lowest execution cost and remove it
7: for all solutions in $\mathcal{L}$ do
8:     Find the solution $l'$ lowest execution cost and remove it from $\mathcal{L}$
9:     $l = Merge2(l, l')$
10: end for
11: Return the solution $l$
```

contrast, function $Merge2$ shown in the previous page introduces new views into the set of candidate views.

The heuristic approach for the dynamic view selection problem is implemented by Algorithm 9 shown above using function $Merge2$. Algorithm 9 first generates an initial set of candidate views which are common subexpressions of each pair of queries. For each view, it finds a solution. Then, it uses Function $Merge2$ to introduce new views into the set of candidate views. At the same time it further refines the solution using the views.

9.6 Considering Update Statements in the Sequence

In general, an update statement that involves updating more than one base relations can be modeled by a sequence of update statements each of which updates one base relation. The update statements are evaluated using the incremental view maintenance strategy. Thus one can abstract an update statement as (R, P) where R is a relation and P is the percentage of tuples in R to be updated. The evaluation cost of an update statement U for a materialized view V can be computed as the materialization cost of V multiplies P if V contains occurrence(s) of R. Otherwise, it is 0. For example, for a materialized view $V = R \bowtie S \bowtie T$ with its materialization cost $mc(V)$, the evaluation cost of an update statement $(R, 10\%)$ is $mc(V) * 10\%$.

Assume that a view V contains an occurrence of the base relation R, and an update statement U in the workload updates R. Then, if V is materialized when U is executed, V has to be updated. This incurs a maintenance cost. An optimizer can assess the cost for maintaining U. Roughly speaking, the unaffected part of a view V with respect to an update U can be defined to be the set of subexpressions of V resulting by removing R from V. If an expression of the unaffected part of a view is materialized or if it can be rewritten using another materialized view, the maintenance cost of V can be greatly reduced. For example, the unaffected part of a view $V = R \bowtie \sigma_{A>10}(S) \bowtie \sigma_{C=0}(T)$ with respect to R is the view $V' = \sigma_{A>10}(S) \bowtie \sigma_{C=0}(T)$. View V can be maintained in response to changes in R either incrementally or through re-computation much more efficiently if V' is materialized.

If there are update statements in the sequence, the heuristic approach still starts with a set of candidate views which are common subexpressions of pairs of queries in the workload sequence. Then, for each view and each update statement, add to the candidate set of views the unaffected parts of the view with respect to the updates. If two update statements update the same base relation, only one is used to generate the unaffected parts of a view. Finally the heuristic algorithm is applied using the new set of views as a candidate view set.

9.7 Experimental Evaluation

The heuristic algorithm for the dynamic view selection problem for a sequence of query and update statements is implemented. In order to examine the effectiveness of our algorithm, a static view selection algorithm similar to the one presented in [153] is also implemented.

Further, the greedy heuristic algorithm $GREEDY-SEQ$ presented in [9] is also implemented. Provided as input to $GREEDY - SEQ$ is a set of candidate views recommended by the static view selection algorithm. Select-Project-Join queries are considered. The "maximal" common subexpressions of two queries are computed using the concept of closest common derivator as is defined in [135]. In order to deal with update statements, the unaffected parts of the views with respect to update statements are considered. A cost model is utilized that assesses the cost of a query (or an update statement) when this is rewritten completely or partially using one or more materialized views.

We measure the performance of each approach as a percentage using the percentage of the total execution cost of a workload using the set of materialized view suggest by the approach to the execution cost of the same workload without using any view. For each experiment, we consider three schemes, *Static* (the static view selection approach), *Dynamic* (the view selection approach presented in this paper), $GREEDY$-SEQ (the algorithm in [9] fed with the result of Static).

First, the effect of the space constraint on the three algorithms is studied. Two workloads W_1 and W_2 are are considered, each of which consists of 25 queries (no updates). The queries in W_1 have more overlapping than that of W_2. The space constraint varies from 1 to 10 times the total size of the base relations. The results are shown in Figures 9.4 and 9.5 for W_1 and W_2, respectively. When the space constraint is restrictive, the dynamic view selection schemes have better performance than the static one. This superiority is the result of the capacity of these approaches for creating and dropping materialized views dynamically. As expected, when the space constraint relaxes, all view selection schemes generate similar results. Among the two dynamic view selection approaches, *Dynamic* performs better than the $GREEDY-SEQ$ algorithm. This shows that a statically selected view set is not appropriate for a dynamic view selection scheme. The new approach does not suffer from this shortcoming since its set of candidate views is constructed dynamically.

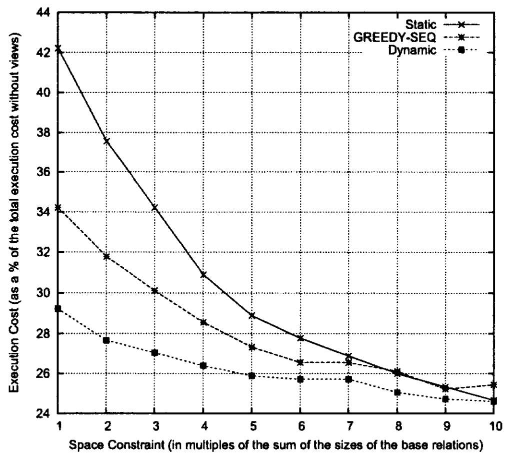


Figure 9.4 Performance vs. space constraint, workload W_1.

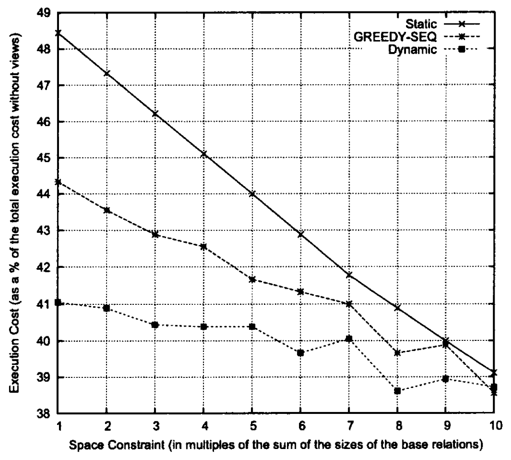


Figure 9.5 Performance vs. space constraint, workload W_2.

Then, the effect of update statements on the three approaches is studied. Two series of workloads WS_1 and WS_2 are considered, each workload contains the same 25 queries. However the number of update statements in each workload varies from 1 to 10. Each update statement updates 30% tuples of base relation chosen randomly. In the workloads of WS_1, all the update statements follow all the queries. In the workloads of WS_2, the

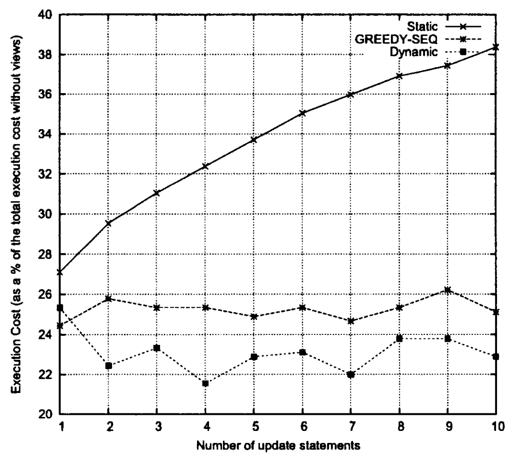


Figure 9.6 Performance vs. number of update statements (workload W_{S1}).

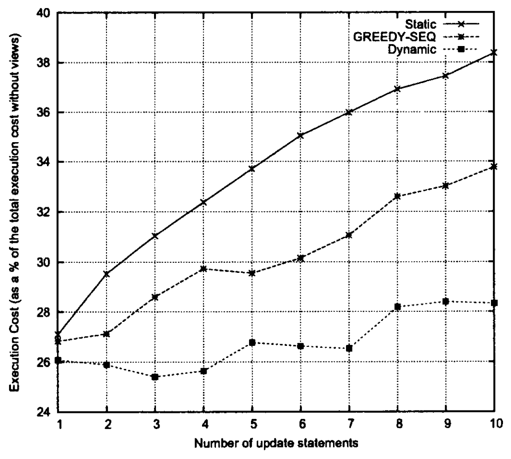


Figure 9.7 Performance vs. space constraint (workload W_{S1p}).

update statements are interleaved randomly with the queries. The space constraint is fixed to 10 times the total size of the base relations. The results are shown in Figures 9.6 and 9.7, respectively. In all cases, when the number of update statements in the workload increases, the dynamic view selection approaches perform better compared to the static one. This is expected since the dynamic algorithms can drop materialized views before the evaluation of update statements and save the maintenance time of these views. The static view selection scheme does not depend on the order of query and update statements in the workload. Thus, for both workload series, the static view selection scheme performs the same. The dynamic view selection scheme depends on the order of query and update statements in the workload. When the update statements follow the queries, the dynamic view selection schemes perform better. The reason is that materialized views that are needed for queries are dropped after the queries are executed and therefore do not contribute to the maintenance cost. In any case, the $Dynamic$ outperforms the $GREEDY - SEQ$.

9.8 Conclusion

This chapter addresses the problem of dynamically creating and dropping materialized views when the workload is a sequence of query and update statements. The problem is modeled as a shortest path problem in DAGs where the nodes of the DAG are dynamically constructed by exploiting common subexpressions among the query and update statements in the workload. A heuristic algorithm is proposed that combines the process of finding the candidate set of views and the process of deciding when to create and drop materialized views during the execution of the statements in the workload. An experimental evaluation of our approach showed that it performs better than previous static and dynamic ones.

CHAPTER 10

CONCLUSION

The problem of selecting views for materialization in current database applications is an important one mainly for reasons of performance. In this dissertation, we provide a comprehensive study of this problem focusing on performance.

We initially provided a framework for the view selection problem by studying the related issues of maintaining materialized views, computing common sub-expressions of queries, and answering queries using other materialized views.

We proposed a new methodology for generating a search space for the view selection problem. A search space is essentially a structure which represents alternative candidate view sets for materialization. Most previous approaches generate sets of candidate views by making the queries in the workload less specific. That technique increases the possibility of using views to answer queries by sacrificing some common operations in the candidate views. Instead, our approach generates candidate views by finding common subexpressions among queries that involve all their common operations. Thus all common operations are kept in the candidate view set. Our approach is general enough to allow the generation of search spaces for different view selection problems. These search spaces can be used by specific cost-based view selection algorithms that take into account the specificities of a given view selection problem to find the solution. Our approach can generate a search space dynamically. Therefore, it can be applied in cases where a statically generated search space is not appropriate. Our experimental results showed that the new approach creates smaller search spaces and exploits more commonalities among queries. In this dissertation, we have applied our method to different instances of the view selection problem.

We studied the problems of satisfiability and implication for mixed arithmetic constraints. We characterized the complexity of the problems and suggested polynomial algorithms for particular cases. We used these theoretical results for computing common subexpressions of queries and search spaces for view selection problems.

As an application of the new method for constructing search spaces for view selection problems, we studied the problem of maintaining multiple view using multi-query optimization techniques. Our approach for this multiple view maintenance problem is based on exploiting maximum commonalities among queries and on rewriting queries using views. Our experimental results showed that our approach significantly reduces the overall view maintenance cost.

We also addressed the same view maintenance problem in the context of a commercial database management system (IBM DB2), in the presence of memory and time restrictions. We proposed a heuristic approach for this problem that guarantees the satisfaction of the restrictions. Our experimental results showed that our approach under memory and time restrictions has a performance which is comparable to that of approaches operating without restrictions.

Finally we exploited our approach for constructing candidate views to study a dynamic version of the view selection problem where the workload is a sequence of query and update statements. We proposed a method that dynamically creates and drops materialized views during the evaluation of the workload. An experimental evaluation showed that our approach outperforms previous static and dynamic view selection algorithms.

REFERENCES

[1] *TPC (Transaction Processing Performance Council) Web Site: http://www.tpc.org.*

[2] ABITEBOUL, S., AND DUSCHKA, O. M. Complexity of Answering Queries Using Materialized Views. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (1998), pp. 254–263.

[3] AFRATI, F. N., CHIRKOVA, R., GERGATSOULIS, M., AND PAVLAKI, V. Finding Equivalent Rewritings in the Presence of Arithmetic Comparisons. In *Proceedings of the 10th International Conference on Extending Database Technology* (2006), pp. 942–960.

[4] AFRATI, F. N., LI, C., AND MITRA, P. Answering Queries Using Views with Arithmetic Comparisons. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (2002), pp. 209–220.

[5] AFRATI, F. N., LI, C., AND MITRA, P. On Containment of Conjunctive Queries with Arithmetic Comparisons. In *Proceedings of the 9th International Conference on Extending Database Technology* (2004), pp. 459–476.

[6] AFRATI, F. N., LI, C., AND ULLMAN, J. D. Generating Efficient Plans for Queries Using Views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2001), pp. 319–330.

[7] AGRAWAL, S., CHAUDHURI, S., KOLLÁR, L., MARATHE, A. P., NARASAYYA, V. R., AND SYAMALA, M. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proceedings of 30th International Conference on Very Large Data Bases* (2004), pp. 1110–1121.

[8] AGRAWAL, S., CHAUDHURI, S., AND NARASAYYA, V. R. Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proceedings of 26th International Conference on Very Large Data Bases* (2000), pp. 496–505.

[9] AGRAWAL, S., CHU, E., AND NARASAYYA, V. R. Automatic physical design tuning: workload as a sequence. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2006), pp. 683–694.

[10] AKINDE, M. O., AND BÖHLEN, M. H. Constructing GPSJ View Graphs. In *Proceedings of the International Workshop on Disign and Management of Data Warehouses* (1999), p. 8.

[11] ATLURI, V., AND GUO, Q. Star-tree: An index structure for efficient evaluation of spatio temporal authorizations. In *DBSec* (2004), pp. 31–47.

[12] BALLINGER, C. TPC-D: Benchmarking for Decision Support. In *The Benchmark Handbook.* 1993.

[13] BARALIS, E., PARABOSCHI, S., AND TENIENTE, E. Materialized Views Selection in a Multidimensional Database. In *Proceedings of 23rd International Conference on Very Large Data Bases* (1997), Morgan Kaufmann, pp. 156–165.

[14] BELLO, R. G., DIAS, K., DOWNING, A., JR., J. J. F., FINNERTY, J. L., NORCOTT, W. D., SUN, H., WITKOWSKI, A., AND ZIAUDDIN, M. Materialized Views in Oracle. In *Proceedings of 24rd International Conference on Very Large Data Bases* (1998), Morgan Kaufmann, pp. 659–664.

[15] BLAKELEY, J. A., COBURN, N., AND LARSON, P.-Å. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *ACM Trans. Database Syst. 14*, 3 (1989), 369–400.

[16] BLAKELEY, J. A., LARSON, P.-Å., AND TOMPA, F. W. Efficiently Updating Materialized Views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1986), pp. 61–71.

[17] BRUNO, N., AND CHAUDHURI, S. Automatic Physical Database Tuning: A Relaxation-based Approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2005), pp. 227–238.

[18] BRUNO, N., CHAUDHURI, S., AND GRAVANO, L. Top-k Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation. *ACM Trans. Database Syst. 27*, 2 (2002), 153–187.

[19] CERI, S., NEGRI, M., AND PELAGATTI, G. Horizontal Data Partitioning in Database Design. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1982), ACM Press, pp. 128–136.

[20] CERI, S., AND WIDOM, J. Deriving Production Rules for Incremental View Maintenance. In *Proceedings of International Conference on Very Large Data Bases* (1991), pp. 577–589.

[21] CHAKRAVARTHY, U. S., AND MINKER, J. Multiple Query Processing in Deductive Databases using Query Graphs. In *Proceedings of 23rd International Conference on Very Large Data Bases* (1986), pp. 384–391.

[22] CHATZIANTONIOU, D., AND ROSS, K. A. Groupwise Processing of Relational Queries. In *Proc. of the 23rd Intl. Conf. on Very Large Data Bases* (1997), pp. 476–485.

[23] CHAUDHURI, S., AND DAYAL, U. An Overview of Data Warehousing and OLAP Technology. *ACM SIGMOD Record 26*, 1 (1997), 65–74.

[24] CHAUDHURI, S., KRISHNAMURTHY, R., POTAMIANOS, S., AND SHIM, K. Optimizing Queries with Materialized Views. In *Proceedings of the International Conference on Data Engineering* (1995), pp. 190–200.

[25] CHAUDHURI, S., AND NARASAYYA, V. R. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proceedings of 23rd International Conference on Very Large Data Bases* (1997), pp. 146–155.

[26] CHAUDHURI, S., AND NARASAYYA, V. R. Microsoft Index Tuning Wizard for SQL Server 7.0. In *Proceedings ACM SIGMOD International Conference on Management of Data* (1998), pp. 553–554.

[27] CHAUDHURI, S., AND SHIM, K. Including Group-By in Query Optimization. In *Proc. of the 20th Intl. Conf. on Very Large Data Bases* (1994), pp. 354–366.

[28] CHAUDHURI, S., AND SHIM, K. Optimizing Queries with Aggregate Views. In *Proc. of the 5th Intl. Conf. on Extending Database Technology* (1996), pp. 167–182.

[29] CHEKURI, C., AND RAJARAMAN, A. Conjunctive query containment revisited. *Theor. Comput. Sci. 239*, 2 (2000), 211–229.

[30] CHEN, F.-C. F., AND DUNHAM, M. H. Common Subexpression Processing in Multiple-Query Processing. *IEEE Trans. Knowl. Data Eng. 10*, 3 (1998), 493–499.

[31] CHEN, Z., AND NARASAYYA, V. R. Efficient Computation of Multiple Group By Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2005), pp. 263–274.

[32] CHILSON, D. W., AND KUDLAC, M. E. Database Design: A Survey of Logical and Physical Design Techniques. In *Databases for Business and Office Applications* (1983), pp. 70–84.

[33] CHIRKOVA, R., LI, C., AND LI, J. Answering queries using materialized views with minimum size. *The International Journal on Very Large Data Base 15*, 3 (2006), 191–210.

[34] COHEN, S. Containment of aggregate queries. *SIGMOD Record 34*, 1 (2005), 77–85.

[35] COHEN, S. Equivalence of queries combining set and bag-set semantics. In *Proc. of the 25th ACM Symposium on Principles of Database Systems* (2006), pp. 70–79.

[36] COHEN, S., NUTT, W., AND SEREBRENIK, A. Rewriting Aggregate Queries Using Views. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (1999), pp. 155–166.

[37] COHEN, S., SAGIV, Y., AND NUTT, W. Equivalences among aggregate queries with negation. *ACM Trans. Comput. Log. 6*, 2 (2005), 328–360.

[38] CONSENS, M. P., BARBOSA, D., TEISANU, A. M., AND MIGNET, L. Goals and Benchmarks for Autonomic Configuration Recommenders. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (2005), pp. 239–250.

[39] CORNELL, D. W., AND YU, P. S. A vertical partitioning algorithm for relational databases. In *ICDE* (1987), pp. 30–35.

[40] DAGEVILLE, B., DAS, D., DIAS, K., YAGOUB, K., ZAÏT, M., AND ZIAUDDIN, M. Automatic SQL Tuning in Oracle 10g. In *Proceedings of 30th International Conference on Very Large Data Bases* (2004), pp. 1098–1109.

[41] DESHPANDE, P., RAMASAMY, K., SHUKLA, A., AND NAUGHTON, J. F. Caching Multidimensional Queries Using Chunks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1998), pp. 259–270.

[42] DEUTSCH, A., POPA, L., AND TANNEN, V. Physical Data Independence, Constraints, and Optimization with Universal Plans. In *Proceedings of International Conference on Very Large Data Bases* (1999), pp. 459–470.

[43] DONG, G., AND SU, J. Incremental and Decremental Evaluation of Transitive Closure by First-Order Queries. *Inf. Comput. 120*, 1 (1995), 101–106.

[44] DONG, G., AND TOPOR, R. W. Incremental Evaluation of Datalog Queries. In *Proceedings of the International Conference on Database Theory* (1992), pp. 282–296.

[45] DUSCHKA, O. M., AND GENESERETH, M. R. Query Planning in Infomaster. In *Proceedings of the ACM symposium on Applied computing* (1997), pp. 109–111.

[46] DUSCHKA, O. M., GENESERETH, M. R., AND LEVY, A. Y. Recursive Query Plans for Data Integration. *J. Log. Program. 43*, 1 (2000), 49–73.

[47] ELKAN, C. Independence of Logic Database Queries and Updates. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (1990), pp. 154–160.

[48] ELMASRI, R., AND NAVATHE, S. *Fundamentals of Database Systems, 5th Edition.* Addison Wesley, 2006.

[49] FINKELSTEIN, S. J. Common Subexpression Analysis in Database Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1982), pp. 235–245.

[50] FINKELSTEIN, S. J., SCHKOLNICK, M., AND TIBERIO, P. Physical Database Design for Relational Databases. *ACM Trans. Database Syst. 13*, 1 (1988), 91–128.

[51] FOLKERT, N., GUPTA, A., WITKOWSKI, A., SUBRAMANIAN, S., BELLAMKONDA, S., SHANKAR, S., BOZKAYA, T., AND SHENG, L. Optimizing Refresh of a Set of Materialized Views. In *Proceedings of the 31st International Conference on Very Large Data Bases* (2005), pp. 1043–1054.

[52] FRANK, M. R., OMIECINSKI, E., AND NAVATHE, S. B. Adaptive and Automated Index Selection in RDBMS. In *EDBT* (1992), pp. 277–292.

[53] GALINDO-LEGARIA, C. A., PELLENKOFT, A., AND KERSTEN, M. L. Fast, randomized join-order selection - why use transformations? In *VLDB* (1994), pp. 85–95.

[54] GOLDSTEIN, J., AND LARSON, P.-Å. Optimizing Queries Using Materialized Views: A practical, scalable solution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2001), pp. 331–342.

[55] GOLFARELLI, M., AND RIZZI, S. Comparing Nested GPSJ Queries in Multidimensional Databases. In *Proceedings of ACM International Workshop on Data Warehousing and OLAP* (2000), pp. 65–71.

[56] GOLFARELLI, M., AND RIZZI, S. View materialization for nested GPSJ queries. In *Proceedings of the International Workshop on Design and Management of Data Warehouses* (2000), pp. 1–10.

[57] GRAHNE, G., AND MENDELZON, A. O. Tableau Techniques for Querying Information Sources through Global Schemas. In *Proceedings of the International Conference on Database Theory* (1999), pp. 332–347.

[58] GRAY, J., CHAUDHURI, S., BOSWORTH, A., LAYMAN, A., REICHART, D., VENKATRAO, M., PELLOW, F., AND PIRAHESH, H. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. *Data Min. Knowl. Discov. 1*, 1 (1997), 29–53.

[59] GRIFFIN, T., AND LIBKIN, L. Incremental Maintenance of Views with Duplicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1995), pp. 328–339.

[60] GRUMBACH, S., RAFANELLI, M., AND TININI, L. Querying Aggregate Data. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (1999), pp. 174–184.

[61] GRUMBACH, S., AND TININI, L. On the Content of Materialized Aggregate Views. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (2000), pp. 47–57.

[62] GUO, S., SUN, W., AND WEISS, M. A. On Satisfiability, Equivalence, and Implication Problems Involving Conjunctive Queries in Database Systems. *IEEE Trans. Knowl. Data Eng. 8*, 4 (1996), 604–616.

[63] GUO, S., SUN, W., AND WEISS, M. A. Solving Satisfiability and Implication Problems in Database Systems. *ACM Trans. Database Syst. 21*, 2 (1996), 270–293.

[64] GUO, S., SUN, W., AND WEISS, M. A. Addendum to "On Satisfiability, Equivalence, and Implication Problems Involving Conjunctive Queries in Database Systems". *IEEE Trans. Knowl. Data Eng. 10*, 5 (1998), 863.

[65] GUPTA, A., HARINARAYAN, V., AND QUASS, D. Aggregate-Query Processing in Data Warehousing Environments. In *Proceedings of International Conference on Very Large Data Bases* (1995), pp. 358–369.

[66] GUPTA, A., JAGADISH, H. V., AND MUMICK, I. S. Data Integration using Self-Maintainable Views. In *Proceedings of the International Conference on Extending Database Technology* (1996), pp. 140–144.

[67] GUPTA, A., AND MUMICK, I. S. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull. 18*, 2 (1995), 3–18.

[68] GUPTA, A., MUMICK, I. S., AND SUBRAHMANIAN, V. S. Maintaining Views Incrementally. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1993), ACM Press, pp. 157–166.

[69] GUPTA, H. Selection of Views to Materialize in a Data Warehouse. In *Proceedings of the 6th International Conference on Database Theory* (1997), pp. 98–112.

[70] GUPTA, H., HARINARAYAN, V., RAJARAMAN, A., AND ULLMAN, J. D. Index Selection for OLAP. In *Proceedings of the International Conference on Data Engineering* (1997), pp. 208–219.

[71] GUPTA, H., AND MUMICK, I. S. Selection of Views to Materialize Under a Maintenance Cost Constraint. In *Proceedings of the 7th International Conference on Database Theory* (1999), pp. 453–470.

[72] HAAS, L. M., FREYTAG, J. C., LOHMAN, G. M., AND PIRAHESH, H. Extensible Query Processing in Starburst. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1989), pp. 377–388.

[73] HALEVY, A. Y. Answering Queries Using Views: A survey. *The International Journal on Very Large Data Bases 10*, 4 (2001), 270–294.

[74] HAMMER, J., GARCIA-MOLINA, H., WIDOM, J., LABIO, W., AND ZHUGE, Y. The Stanford Data Warehousing Project. *IEEE Data Eng. Bull. 18*, 2 (1995), 41–48.

[75] HARINARAYAN, V., RAJARAMAN, A., AND ULLMAN, J. D. Implementing Data Cubes Efficiently. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (1996), ACM Press, pp. 205–216.

[76] HARRISON, J. V., AND DIETRICH, S. W. Maintenance of Materialized Views in a Deductive Database: An Update Propagation Approach. In *Workshop on Deductive Databases, JICSLP* (1992), pp. 56–65.

[77] IOANNIDIS, Y. E. Query optimization. In *The Computer Science and Engineering Handbook*. 1997, pp. 1038–1057.

[78] IOANNIDIS, Y. E., AND KANG, Y. C. Randomized Algorithms for Optimizing Large Join Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1990), ACM Press, pp. 312–321.

[79] JAGADISH, H. V., MUMICK, I. S., AND SILBERSCHATZ, A. View Maintenance Issues for the Chronicle Data Model. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (1995), pp. 113–124.

[80] JARKE, M. Common Subexpression Isolation in Multiple Query Optimization. In *Query Processing in Database Systems*. Springer, 1985, pp. 191–205.

[81] JARKE, M. Common Subexpression Isolation in Multiple Query Optimization. In *Query Processing in Database Systems*. Springer, 1985, pp. 191–205.

[82] JT, H., YJ, C., BJ, L., AND CY, K. Materialized view selection using genetic algorithms in a data warehouse. In *Proceedings of World Congress on Evolutionary Computation* (1999).

[83] J.WIDOM, AND S.CERI. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.

[84] KALNIS, P., MAMOULIS, N., AND PAPADIAS, D. View selection using randomized search. *Data Knowl. Eng. 42*, 1 (2002), 89–111.

[85] KALNIS, P., AND PAPADIAS, D. Optimization Algorithms for Simultaneous Multidimensional Queries in OLAP Environments. In *Proceedings of 3rd International Conference on Data Warehousing and Knowledge Discovery* (2001), pp. 264–273.

[86] KANELLAKIS, P. C., KUPER, G. M., AND REVESZ, P. Z. Constraint Query Languages. *J. Comput. Syst. Sci. 51*, 1 (1995), 26–52.

[87] KARLOFF, H. J., AND MIHAIL, M. On the Complexity of the View-Selection Problem. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (1999), ACM Press, pp. 167–173.

[88] KIM, W. OLTP Versus DSS/OLAP/Data Mining. *Journal Of Object Oriented Programming 10*, 7 (1997), 68–70, 77.

[89] KLUG, A. C. On conjunctive queries containing inequalities. *J. ACM 35*, 1 (1988), 146–160.

[90] KOTIDIS, Y., AND ROUSSOPOULOS, N. DynaMat: A Dynamic View Management System for Data Warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1999), pp. 371–382.

[91] KÜCHENHOFF, V. On the Efficient Computation of the Difference Between Concecutive Database States. In *International Conference on Deductive and Object-Oriented Databases* (1991), pp. 478–502.

[92] LARSON, P.-Å., AND YANG, H. Z. Computing Queries from Derived Relations. In *Proc. of the 11th Intl. Conf. on Very Large Data Bases* (1985), pp. 259–269.

[93] LEE, M., AND HAMMER, J. Speeding Up Materialized View Selection in Data Warehouses Using a Randomized Algorithm. *Int. J. Cooperative Inf. Syst. 10*, 3 (2001), 327–353.

[94] Lehner, W., Cochrane, R., Pirahesh, H., and Zaharioudakis, M. fAST Refresh Using Mass Query Optimization. In *Proceedings of the 17th International Conference on Data Engineering* (2001), pp. 391–398.

[95] Lehner, W., Sidle, R., Pirahesh, H., and Cochrane, R. Maintenance of Automatic Summary Tables. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (2000), pp. 512–513.

[96] Levy, A. Y., Mendelzon, A. O., Sagiv, Y., and Srivastava, D. Answering Queries Using Views. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (1995), pp. 95–104.

[97] Levy, A. Y., and Mumick, I. S. Reasoning with Aggregation Constraints. In *Proc. of the 5th Intl. Conf. on Extending Database Technology* (1996), pp. 514–534.

[98] Levy, A. Y., Rajaraman, A., and Ordille, J. J. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proceedings of International Conference on Very Large Data Bases* (1996), pp. 251–262.

[99] Levy, A. Y., and Sagiv, Y. Queries Independent of Updates. In *Proceedings of International Conference on Very Large Data Bases* (1993), pp. 171–181.

[100] Li, C. Describing and Utilizing Constraints to Answer Queries in Data-Integration Systems. In *Proc. of IJCAI-03 Workshop on Information Integration on the Web* (2003), pp. 163–168.

[101] Li, J., Talebi, Z. A., Chirkova, R., and Fathi, Y. A Formal Model for the Problem of View Selection for Aggregate Queries. In *Proceedings of 9th East European Conference on Advances in Databases and Information Systems* (2005), pp. 125–138.

[102] Liang, W., Orlowska, M. E., and Yu, J. X. Optimizing Multiple Dimensional Queries Simultaneously in Multidimensional Databases. *The International Journal on Very Large Data Bases 8*, 3-4 (2000), 319–338.

[103] Mendelzon, A. O., and Mihaila, G. A. Querying Partially Sound and Complete Data Sources. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (2001).

[104] Michael Lawrence, A. R.-C. Dynamic View Selection for OLAP. In *Proceedings of the 8th International Conference on Data Warehousing and Knowledge Discovery* (2006).

[105] Millstein, T. D., Levy, A. Y., and Friedman, M. Query Containment for Data Integration Systems. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (2000), pp. 67–75.

[106] Mistry, H., Roy, P., Sudarshan, S., and Ramamritham, K. Materialized View Selection and Maintenance Using Multi-Query Optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2001), pp. 307–318.

[107] OMIECINSKI, E. Parallel relational database systems. In *Modern Database Systems*. 1995, pp. 494–512.

[108] OSZU, M. T., AND VALDURIEZ, P. *Principles of Distributed Database Systems, Second Edition*. Prentice-Hall, 1999.

[109] PAIGE, R. Applications of Finite Differencing to Database Integrity Control and Query/Transaction Optimization. In *Advances in Data Base Theory* (1982), pp. 171–209.

[110] PETER C., G. M. *ORACLE Performance Tuning*. OReilly & Associates, Inc., 1993.

[111] POPA, L., DEUTSCH, A., SAHUGUET, A., AND TANNEN, V. A Chase Too Far? In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2000), pp. 273–284.

[112] POTTINGER, R., AND LEVY, A. Y. A Scalable Algorithm for Answering Queries Using Views. In *Proceedings of 26th International Conference on Very Large Data Bases* (2000), pp. 484–495.

[113] QIAN, X. Query Folding. In *Proceedings of the International Conference on Data Engineering* (1996), pp. 48–55.

[114] QIAN, X., AND WIEDERHOLD, G. Incremental Recomputation of Active Relational Expressions. *IEEE Trans. Knowl. Data Eng. 3*, 3 (1991), 337–341.

[115] RAMAKRISHNAN, R., AND GEHRKE, J. *Database Management Systems*. WCB/McGraw-Hill, 1998.

[116] ROSENKRANTZ, D. J., AND III, H. B. H. Processing Conjunctive Predicates and Queries. In *Proc. of the Intl. Conf. on Very Large Data Bases* (1980), pp. 64–72.

[117] ROSS, K. A., SRIVASTAVA, D., STUCKEY, P. J., AND SUDARSHAN, S. Foundations of Aggregation Constraints. *Theor. Comput. Sci. 193*, 1-2 (1998), 149–179.

[118] ROSS, K. A., SRIVASTAVA, D., AND SUDARSHAN, S. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (1996), pp. 447–458.

[119] ROUSSOPOULOS, N. An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis. *ACM Trans. Database Syst. 16*, 3 (1991), 535–563.

[120] ROY, P., SESHADRI, S., SUDARSHAN, S., AND BHOBE, S. Efficient and Extensible Algorithms for Multi Query Optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2000), pp. 249–260.

[121] SELLIS, T. K. Multiple-Query Optimization. *ACM Trans. Database Syst. 13*, 1 (1988), 23–52.

[122] SHI GUANG QIU, T. W. L. View Selection in OLAP Environment. In *Proceedings of 11th International Conference on Database and Expert System Applications* (2000).

[123] SHIM, K., SELLIS, T. K., AND NAU, D. S. Improvements on a Heuristic Algorithm for Multiple-Query Optimization. *Data Knowl. Eng. 12*, 2 (1994), 197–222.

[124] SHMUELI, O., AND ITAI, A. Maintenance of Views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1984), pp. 240–255.

[125] SHUKLA, A., DESHPANDE, P., AND NAUGHTON, J. F. Materialized View Selection for Multidimensional Datasets. In *Proceedings of 24rd International Conference on Very Large Data Bases* (1998), Morgan Kaufmann, pp. 488–499.

[126] SRIVASTAVA, D., DAR, S., JAGADISH, H. V., AND LEVY, A. Y. Answering queries with aggregation using views. In *Proceedings of International Conference on Very Large Data Bases* (1996), pp. 318–329.

[127] SUN, X.-H., KAMEL, N., AND NI, L. M. Processing Implication on Queries. *IEEE Trans. Software Eng. 15*, 10 (1989), 1168–1175.

[128] SWAMI, A. N., AND GUPTA, A. Optimization of Large Join Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1988), pp. 8–17.

[129] TAO, Y., ZHU, Q., AND ZUZARTE, C. Exploiting common subqueries for complex query optimization. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research* (2002), p. 12.

[130] THEODORATOS, D., AND BOUZEGHOUB, M. A General Framework for the View Selection Problem for Data Warehouse Design and Evolution. In *Proceedings of ACM Seventh International Workshop on Data Warehousing and OLAP* (2000), pp. 1–8.

[131] THEODORATOS, D., LIGOUDISTIANOS, S., AND SELLIS, T. K. View selection for designing the global data warehouse. *Data Knowl. Eng. 39*, 3 (2001), 219–240.

[132] THEODORATOS, D., LIGOUDISTIANOS, S., AND SELLIS, T. K. View selection for designing the global data warehouse. *Data Knowl. Eng. 39*, 3 (2001), 219–240.

[133] THEODORATOS, D., AND SELLIS, T. K. Data Warehouse Configuration. In *Proceedings of 23rd International Conference on Very Large Data Bases* (1997), pp. 126–135.

[134] THEODORATOS, D., AND SELLIS, T. K. Incremental Design of a Data Warehouse. *J. Intell. Inf. Syst. 15*, 1 (2000), 7–27.

[135] THEODORATOS, D., AND XU, W. Constructing Search Spaces for Materialized View Selection. In *Proceedings of ACM Seventh International Workshop on Data Warehousing and OLAP* (2004), pp. 112–121.

[136] TOMPA, F. W., AND BLAKELEY, J. A. Maintaining materialized views without accessing base data. *Inf. Syst. 13*, 4 (1988), 393–406.

[137] TSATALOS, O. G., SOLOMON, M. H., AND IOANNIDIS, Y. E. The GMAP: A Versatile Tool for Physical Data Independence. *The International Journal on Very Large Data Bases 5*, 2 (1996), 101–118.

[138] ULLMAN, J. D. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1989.

[139] URPÍ, T., AND OLIVÉ, A. A method for change computation in deductive databases. In *Proceedings of International Conference on Very Large Data Bases* (1992), pp. 225–237.

[140] VALENTIN, G., ZULIANI, M., ZILIO, D. C., LOHMAN, G. M., AND SKELLEY, A. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *Proceedings of the International Conference on Data Engineering* (2000), pp. 101–110.

[141] W. H. INMON, C. K. *Developing the Data Warehouse*. QED Publishing Group/John Wiley, 1993.

[142] WOLFSON, O., DEWAN, H. M., STOLFO, S. J., AND YEMINI, Y. Incremental Evaluation of Rules and its Relationship to Parallelism. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1991), pp. 78–87.

[143] WU, M.-C., AND BUCHMANN, A. P. Research Issues in Data Warehousing. In *Datenbanksysteme in Bro, Technik und Wissenschaft* (1997), pp. 61–82.

[144] XU, W., ZUZARTE, C., AND THEODORATOS, D. Preprocessing for Fast Refreshing Materialized Views in DB2. In *Proceedings of the 8th International Conference on Data Warehousing and Knowledge Discovery* (2006).

[145] YAN, W. P., AND LARSON, P.-Å. Eager Aggregation and Lazy Aggregation. In *Proc. of the 21st Intl. Conf. on Very Large Data Bases* (1995), pp. 345–357.

[146] YANG, H. Z., AND LARSON, P.-Å. Query Transformation for PSJ-Queries. In *Proc. of the 13th Intl. Conf. on Very Large Data Bases* (1987), pp. 245–254.

[147] YANG, J., KARLAPALEM, K., AND LI, Q. Algorithms for Materialized View Design in Data Warehousing Environment. In *Proceedings of 23rd International Conference on Very Large Data Bases* (1997), Morgan Kaufmann, pp. 136–145.

[148] YU, S., ATLURI, V., AND ADAM, N. R. Selective view materialization in a spatial data warehouse. In *DaWaK* (2005), pp. 157–167.

[149] YU, S., ATLURI, V., AND ADAM, N. R. Cascaded star: A hyper-dimensional model for a data warehouse. In *DEXA* (2006), pp. 439–448.

[150] YU, S., ATLURI, V., AND ADAM, N. R. Preview: Optimizing view materialization cost in spatial data warehouses. In *DaWaK* (2006), pp. 45–54.

[151] Zaharioudakis, M., Cochrane, R., Lapis, G., Pirahesh, H., and Urata, M. Answering Complex SQL Queries Using Automatic Summary Tables. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (2000), pp. 105–116.

[152] Zhang, C., and Yang, J. Genetic Algorithm for Materialized View Selection in Data Warehouse Environments. In *Proceedings of the International Conference on Data Warehousing and Knowledge Discovery* (1999), pp. 116–125.

[153] Zilio, D. C., Rao, J., Lightstone, S., Lohman, G. M., Storm, A., Garcia-Arellano, C., and Fadden, S. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *Proceedings of the International Conference on Very Large Data Bases* (2004), pp. 1087–1097.

[154] Zilio, D. C., Zuzarte, C., Lightstone, S., Ma, W., Lohman, G. M., Cochrane, R., Pirahesh, H., Colby, L. S., Gryz, J., Alton, E., Liang, D., and Valentin, G. Recommending Materialized Views and Indexes with IBM DB2 Design Advisor. In *Proceedings of the International Conference on Autonomic Computing* (2004), pp. 180–188.