

ABSTRACT

KORMILITSIN, MAXIM. Algorithms for Selecting Views and Indexes to Answer Queries. (Under the direction of Dr. Rada Chirkova and Dr. Matthias Stallmann).

In many contexts it is beneficial to answer database queries using derived data called views. Using views in query answering is relevant in applications in information integration, data warehousing, web-site design, and query optimization. The problem of answering queries using views can be divided into a number of subproblems. The first step in the process of view selection is to identify which view can be used to answer queries from the given set. The second step is to determine possible reformulations of the workload queries. The last step is choosing views that can be maintained appropriately and that minimize the processing time of the input query workload.

In our work we address the problem of selecting and precomputing indexes and materialized views in a database system, with the goal of improving the processing performance for frequent and important queries. The focus of our work is to develop a unified quality-centered view- and index-selection approach, for a range of query, view, and index classes that are typical in practical database systems. To the best of our knowledge, we are the first to adopt the solution-quality focus for this generic practical problem setting.

Algorithms for Selecting Views and Indexes to Answer Queries

by
Maxim Kormilitsin

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2009

APPROVED BY:

Dr. Rada Chirkova
Chair of Advisory Committee

Dr. Yahya Fathi

Dr. Matthias Stallmann
Co-Chair of Advisory Committee

Dr. Christopher Healey

TABLE OF CONTENTS

LIST OF TABLES	iv
LIST OF FIGURES	v
LIST OF ALGORITHMS	viii
1 Introduction.....	1
1.1 Usable Views and Query Rewritings	4
2 View- and Index-Selection Problem.....	12
2.1 Well-Known Taxonomy [Gup97] for a View-Selection Problem	12
2.2 Defining the View-Selection Problem	14
2.3 Past Work in the Framework of OR View Graphs	15
2.3.1 Near-Optimal Solutions	18
2.4 Past Work in the Framework of AND-OR View Graphs	24
2.4.1 Automated Selection of Materialized Views and Indexes for SQL Databases [ACN00a]	24
2.4.2 Automatic Physical Database Tuning: A Relaxation-based Ap- proach [BC05]	26
2.5 Selecting Views Under a Maintenance-Cost Constraint	27
3 Our Approach to the View- and Index-Selection Problem.....	29
3.1 The Architecture	33
3.2 Efficient Evaluation Plans for CQAC Queries	37
3.2.1 Finding Efficient Plans for a Single CQAC Query	37
3.2.2 Finding Efficient Plans for Multiple CQAC Queries	45
3.2.3 More aggressive pruning.	53
3.3 Experimental Results	57
3.3.1 Instance Generation	58
3.3.2 Comparisons with RBA	60
3.3.3 Scalability Results	64
3.3.4 Comparison on TPC-H workload	64
3.3.5 Summary	69
3.4 Discussion and Extensions	69

4	Second Stage Plug-in	73
4.1	Complexity of the Problem	75
4.1.1	Formal Problem Statement	75
4.1.2	NP-completeness of PSP	76
4.2	Integer Linear Programming	77
4.2.1	An ILP model for PSP	78
4.2.2	Branch and bound	79
4.3	Finding Upper Bounds	80
4.4	Finding Lower Bounds	84
4.4.1	Lagrangian Heuristics	84
4.4.2	Greedy Algorithm	84
4.5	Experimental Results	85
4.5.1	Random Generation of Problem Instances	86
4.5.2	B&B behavior	88
4.5.3	OLAP-alike Problem Instances	95
4.5.4	Comparison with CPLEX	96
4.5.5	Comparison with <i>Greedy</i> (k, m)	101
4.5.6	Other Observations	104
5	Extensions and Future Work	105
5.1	View-Selection Problem for the More General Case of Queries	105
5.2	Applicability of Our Approach to the Multiple Query Optimization	108
5.3	ADR Under a Maintenance-Cost Constraint	112
6	Conclusion	116
	Bibliography	118

LIST OF TABLES

Table 4.1 Mean and standard deviation of runtime distribution (in seconds) for different values of maximum allowed error.....	90
Table 4.2 Example output of our algorithm in the interactive mode.....	94
Table 4.3 Mean and standard deviation of runtime distribution (in seconds) for a fixed value of OL	97

LIST OF FIGURES

Figure 2.1	Examples of AND and AND-OR view graphs.	13
Figure 2.2	Representation of data as a data cube.	16
Figure 2.3	OR view graph.	18
Figure 2.4	Example of a lattice with space costs.	19
Figure 3.1	Our two-stage architecture.	34
Figure 3.2	Constructing (possibly view-based) evaluation plans for a single CQAC query.	39
Figure 3.3	Removing dominated (in the sense of Definition 2) plans from the list of plans of a given subquery.	41
Figure 3.4	Sample database.	43
Figure 3.5	Plans for the 2 subqueries of length 2.	43
Figure 3.6	All available plans for query Q with unrestricted amount of the available disk space.	44
Figure 3.7	Ranking of view-based plans for query Q	44
Figure 3.8	Example of DP lattice for multiple chained queries. (D_{ij} , with $i < j$, denotes a chain of subgoals $i, i + 1, \dots, j - 1, j$).	47
Figure 3.9	Constructing (view-based) evaluation plans for multiple CQAC queries.	50
Figure 3.10	Removing globally dominated (in the sense of Definition 6) plans from the list of plans of a given subquery.	51
Figure 3.11	Constructing merged views for a given subquery.	56
Figure 3.12	Comparing solution quality of our AR (Section 3.2) versus the RBA of [BC05].	62

Figure 3.13 Scalability of aggressive pruning rules.....	63
Figure 3.14 AR20 versus RBA for different space bounds.....	64
Figure 3.15 AR20 versus RBA as a function of number of queries.....	65
Figure 3.16 Scalability of CPLEX when used to implement stage two.....	66
Figure 3.17 Relative difference between the solution values returned by AR and RBA, $(S_{RBA} - S_{AR})/S_{RBA}$	67
Figure 3.18 Relative difference between the solution values returned by AR44 and RBA, $(S_{RBA} - S_{AR44})/\max(S_{RBA}, S_{AR44})$, for randomly generated instances	68
Figure 3.19 Distribution of the solution relative difference between AR44 and RBA.....	68
Figure 4.1 Scalability of $B\&B$ for various input errors.....	89
Figure 4.2 Analysis of the runtime of $B\&B$	91
Figure 4.3 Runtime of $B\&B$ on a fixed instance with various input errors.	91
Figure 4.4 Gap between lower and upper bounds.	93
Figure 4.5 Complexity of the problem for different values of OL	96
Figure 4.6 Scalability of the algorithms for $OL = 1.4$ (problem instances in which a view appears in 1.4 plans in average).....	98
Figure 4.7 Scalability of the algorithms for database-oriented instances.....	99
Figure 4.8 Scalability of $B\&B$ and $Greedy(k, m)$	101
Figure 4.9 Quality of the $Greedy(k, m)$ solution.....	103
Figure 4.10 Error distribution for $Greedy(k, m)$	103
Figure 5.1 Graph representation of a comb query.....	106
Figure 5.2 Constructing (view-based) evaluation plans for multiple queries.	107
Figure 5.3 Constructing plans for MQO.....	110

Figure 5.4	Update lattice.	114
------------	----------------------	-----

LIST OF ALGORITHMS

Algorithm 1	SINGLEQUERYPLANGEN	39
Procedure 2	CONSTRUCTJOINPLANS(q_1, q_2, q, B)	39
Procedure 3	CONSTRUCTVIEWPLAN(q, B)	39
Procedure 4	SINGLEQUERYPRUNEPLANS()	41
Algorithm 5	MULTIQUERYPLANGEN	50
Procedure 6	CONSTRUCTJOINPLANS(q_1, q_2, q, B)	50
Procedure 7	CONSTRUCTVIEWPLANS(q, Q, B)	50
Procedure 8	MULTIPLEQUERYPRUNEPLANS()	51
Algorithm 9	Heuristic view-selection.	56
Algorithm 10	Lagrangian Heuristics	85
Algorithm 11	Greedy Algorithm	86
Algorithm 12	GENERALMULTIQUERYPLANGEN	107
Algorithm 13	MQOPLANGEN	110
Procedure 14	INITIALIZELEVELONE(Q, H)	110
Procedure 15	CREATEJOINPLANS(q_1, q_2, q)	110

Chapter 1

Introduction

In many contexts it is beneficial to answer database queries using derived data called materialized views and indexes. A *materialized view* is a named query whose answer is stored in a database system. A user query can be answered using views via a new definition that is called a *rewriting* and is built in terms of the views. An index is a supplementary structure that allows an optimizer to access tuples by their keys instead of scanning entire tables.

Using materialized views in query answering [LMSS95] is relevant in applications in information integration, data warehousing, web-site design, and query optimization [ZCL⁺00, TS97]. Two main directions in answering queries using views and indexes are (1) feasibility: to obtain some answer to a given query using given views and indexes, as in the information-integration scenario [CM77, Ull89, PKL02], and (2) efficiency: to reduce query-execution time by using the views and indexes, as in the query-optimization scenario [ALU01, Gup97, HRU96a, CKPS95a]. Within the efficiency direction, the objective is typically to use views and indexes to obtain *equivalent query rewritings* — that is, definitions that give an exact answer to the query on all databases. Answering queries using views and indexes has been explored in depth for relational database systems and for conjunctive queries, which can be defined via positive existential conjunctive formulas of first order logic [End72].

In the past few years, significant research efforts have been concentrated on view and index selection, that is, on developing methods for defining and precomputing

materialized views and indexes on these views to answer predefined queries. Existing approaches differ in their main objective (feasibility or efficiency) and in how they explore the search space of views and rewritings for the given queries. Formally, starting with a set of database relations and a set of queries, the problem is to design a set of views and indexes of the database relations that (1) can be materialized (precomputed and stored on disk) under a given restriction (such as a *storage limit*, i.e., the amount of disk space available for storing the view relations) and, once materialized, (2) can be used by a given evaluation algorithm in answering the queries equivalently and more efficiently than the original relations.

Standard database management systems (DBMS) can keep track of the queries asked during a certain period of time. Based on this information we can find the set of the most popular queries. As users of databases tend to ask similar queries or even repeat them, the set of most popular queries can sometimes serve as a prediction for the future query workload. In order to reduce the processing time of this set of queries we could simply calculate the answers to the queries and save them on disk. However, this solution is not acceptable and, in general, not feasible. Here are some reasons why the “materialize all” technique is not acceptable.

- First of all, in many real applications we have a constraint on the available disk space. Thus, we cannot materialize everything we may need in the future.
- The second reason is less intuitive, but is more important — the maintenance-time constraint. After creating a set of materialized views, we must keep updated the data in the answers to these views. This means, if the data in base tables were changed during some database transactions, we must propagate the changes to all views based on these tables. During this period of time, the database, or at least its part, remains inaccessible, because it might pass through inconsistent states during the updates and users can get incorrect results to their queries. Therefore, we cannot allow the update operation to take too much time, and must consider only those sets of views that will satisfy a given maintenance-time constraint.

- The third reason why the “materialize all” technique is not the most beneficial can be illustrated using the following example.

Example 1. *Suppose a user asks for the total sales of Honda Civic in Raleigh for last year. Following the “materialize all” technique we save the answer on disk, but next day the user asks the same query about Toyota Corolla, and never asks again about Honda Civic. The point is that these queries are so specific it might be useful to save the answer to a more general query. For example, list all car models together with the total sales for Raleigh for last year.*

Therefore, the problem of finding a viewset that maximizes the DBMS performance is a very important aspect in database optimization, and, at the same time, it is a very hard problem. This problem is called a *view-selection problem*.

Picking a right set of views is a difficult task, because there can be dependencies between views: choosing one view for materialization can help answer other views more quickly. The view that we choose for materialization may not be very important by itself, but materializing it can help to process other queries faster.

The rest of this dissertation is organized as follow. Chapter 1.1 discusses the feasibility aspect of the view-selection problem. Here, we discuss the conditions under which a view or a set of views can be used to answer a query. We also present our work [GKC06] on the local conditions under which two views can be combined in a query rewriting. In Chapter 2 we overview the state of the art for the view-selection problem. Here, we present existing algorithms and show that though the view-selection problem has received a lot of attention, it still has much potential for the further research. In Chapter 3 and 4 we talk about our proposed approach to the view- and index-selection problem and demonstrate its competitiveness. In Chapter 5.1 we discuss how to extend our proposed techniques to more general shapes of queries. In Chapter 5.2 we demonstrate applicability of our approach to the Multiple-Query Optimization problem.

1.1 Usable Views and Query Rewritings

In this section we present several examples and discuss the conditions under which a view or a set of views can be used to answer a query. We begin by presenting an example that demonstrates how views can be used to improve the query-evaluation process.

Example 2. *Suppose we have a database with two base tables (key attributes are underlined):*

`Students (studentID, name, GPA)`

`Courses (studentID, courseID)`

Suppose we want to know what courses are taken by students whose GPA is at least 3.8. The query for such a question can be formulated as follows in SQL [SQL92]

Q1: `SELECT DISTINCT courseID
FROM Students s, Courses c
WHERE s.studentID = c.studentID
AND s.GPA >= 3.8;`

Assume that we also have a materialized view that contains records about students with GPA of at least 3.8.

`CREATE VIEW GoodStudents as
SELECT studentID, name, GPA
FROM Students s
WHERE s.GPA >= 3.8;`

It is easy to see that we can use the view `GoodStudents` to find the answer to the query Q1:

`SELECT DISTINCT courseID
FROM GoodStudents g, Courses c
WHERE g.studentID = c.studentID;`

This reformulation of Q1 is cheaper to execute than the original definition of Q1 as we have already filtered from `Students` all records with $GPA < 3.8$. Notice that this view could be useful even if in the query we were interested in courses of students with $GPA \geq 3.9$. Furthermore, this view is still beneficial for Q1 as it prunes a part of irrelevant records from the table `Students`.

Thus, a very important question in the view-selection problem is how to determine whether a view can be used to answer a query or not. In other words, we want to know whether a view contains enough information to contribute to a query. Many contributions have been made to the problem of how to determine whether a view can be used to answer a query [CM77, CG00, AGK99, GL01, PH01]. Most of them try to find conditions on query and view definitions that must be satisfied in order for the view to be usable in answering the query. The following three intuitive conditions defined in [Hal01] can help the reader to understand the nature of these research results:

- There must exist a mapping of the tables in the view definition into the tables of the query definition. The intuition here is that all tables that appear in a view must appear in a query.
- Selection predicates of a view must be weaker than those of the query. If it's not so, the view can miss some of the records needed in the query answer.
- If the attributes of the tables used by the view appear in the output of the query, then these attributes must be in the output of the view, unless these attributes can be retrieved from some other view.

Even having such a set of rules we still have a problem what views to choose, because the number of views we can create using these rules is exponential in the number of base tables, attributes, and selection conditions. Thus, considering all such views becomes prohibitive even for small sets of queries. The state-of-the-art paper by Agrawal et al. [ACN00a] presents a tool for automated selection of materialized views and indexes for a wide variety of query, view, and index classes in relational database systems. The approach of [ACN00a], implemented in Microsoft SQL Server, is based partly on the authors' previous work by Chaudhuri and Narasayya [CN97] on index selection. The main idea of their approach is to find sets of *interesting* table subsets, that is, table subsets that appear in the most important queries or in many less important queries, and construct views for individual queries based on these table subsets applying all applicable selection conditions. After that, in order

to create views that are usable by several queries, they propose a merging technique that takes pairs of views based on the same table subsets and creates a view that contains both parent views in it. The main contribution of their work is that they reduce the search space of applicable views in such a way that searching over the reduced space of candidate views preserves most of the gains of searching the entire space.

After determining a set of views that can be used to answer a query, a very important step is to find a set of possible ways of reformulating the query using this set of views. Such query reformulations are called rewritings.

In [CKPS95a], Chaudhuri et al. proposed an optimization algorithm that produces valid view-based query rewritings for SQL select-project-join (SPJ) queries and views with inequality comparisons and without grouping or aggregation. This is a large group of queries, and, in general, the majority of queries belongs to this class.

The algorithm proposed in [CKPS95a] is based on a recursion. On input it takes a query definition and a set of view definitions. For each subproblem, which is a valid rewriting, it finds a subexpression of the rewriting that matches the definition of a view from the viewset. For each such view, the algorithm replaces its definition in the rewriting by its name, creates a subproblem for the newly obtained rewriting, and then runs the above procedure on the next subproblem. If for some rewriting there are no more possible substitutions, the algorithm returns it as a valid rewriting of the query.

If we are to find the best rewriting using a set of views, then for each problem we keep only the best rewriting returned from its subproblems according to a chosen cost model.

Example 3. *Suppose we have a database with four base tables $P(A,B)$, $R(B,C)$, $S(C,D)$, and $T(D,E)$ and a query $Q1$ in SQL*

```
Q1: SELECT P.A, T.D
      FROM P, R, S, T
      WHERE  P.B = R.B
      AND    R.C = S.C
      AND    S.D = T.D;
```

This query can be rewritten in Datalog [Ull89] as follows

$$Q(A,D) \leftarrow P(A,B), R(B,C), S(C,D), T(D,E)$$

*Here Q is the name of the query, A and D are the output variables from the **SELECT** clause of the SQL definition of Q_1 , and equivalence of two attributes is expressed by using the same variable name. For example, variable B appears in both P and R ; this means that $P.B = R.B$. We use the Datalog notation, because it simplifies understanding of the algorithms we present later.*

Suppose we also have a set of views $\{U, V, W\}$ defined in Datalog:

$$U(A,C) \leftarrow P(A,B), R(B,C)$$

$$V(B,D) \leftarrow R(B,C), S(C,D)$$

$$W(C,D,E) \leftarrow S(C,D), T(D,E)$$

Let us follow the steps of the algorithm of [CKPS95a].

Subproblem 1 *This subproblem is based on the initial query definition $Q_1 \leftarrow P(A,B), R(B,C), S(C,D), T(D,E)$. For each view we try to find its occurrence in Q_1 . We see that U can be used in Q_1 , thus, we can replace the definition of U by its name. The resulting rewriting is $Q_2 \leftarrow U(A,C), S(C,D), T(D,E)$. We initialize new Subproblem 2 with this query rewriting on input.*

The same can be done for views V and W . We get rewritings $Q_3 \leftarrow P(A,B), V(B,D), T(D,E)$ and $Q_4 \leftarrow P(A,B), R(B,C), W(C,D,E)$, respectively, and initialize two new Subproblems 3 and 4.

Keep only the best of rewritings returned from the Subproblems 2, 3, and 4.

Subproblem 2 *Input: $Q_2 \leftarrow U(A,C), S(C,D), T(D,E)$. We have an occurrence of W in Q_2 , thus we get rewriting $Q_5 \leftarrow U(A,C), W(C,D,E)$. No other substitutions of views are possible, send this rewriting back to Subproblem 1.*

Subproblem 3 *Input: $Q_3 \leftarrow P(A,B), V(B,D), T(D,E)$. No other substitutions of views are possible, send this rewriting back to Subproblem 1. Subproblem 1 already has rewriting Q_5 . We compare the costs of Q_3 and Q_5 . Suppose Q_5 is cheaper, because it has fewer tables in its definition. So, we keep Q_5 and reject Q_3 .*

Subproblem 4 *Input: $Q_4 \leftarrow P(A,B), R(B,C), W(C,D,E)$. We have an occurrence of U in Q_4 , we get rewriting $Q_6 \leftarrow U(A,C), W(C,D,E)$. No more substitutions of views are possible, send this rewriting back to Subproblem 1. Subproblem 1 already contains*

rewriting Q_5 , which is actually the same as Q_6 . So, we keep one of them, say, Q_5 .

As we saw in Example 3, when we try to find a rewriting using a set of views, we try to find a subexpression of the query definition that can be replaced by a view. In other words, each view “covers” a subset of tables of the query, and no two views can cover the same table — views in the query rewriting do not overlap. In [ALU01], Afrati et al. show that for the case when tables in a database do not contain duplicates (such databases are called *set-valued*) there exist rewritings that use *overlapping views* — views that have in their definitions some common tables. We show this in Example 4.

Example 4. *Suppose we have a database of a restaurant. This database contains three base tables (key attributes are underlined).*

```
Clients (clientID, clientName)
Orders (orderID, menuID, clientID, date, price)
Menu (menuID, mainDish)
```

Suppose that the Orders table contains many more records than the two other tables. Suppose we want to know for each client who visited the restaurant since January 2006 his name and the maximum price he paid together with the main dish he ordered that time. Such a query can be expressed in Datalog as follows:

```
Q(cName, mDish, max(price)) ← Clients(cID, cName),
                                Orders(oID, mID, cID, date, price),
                                Menu(mID, mDish),
                                date >= '2006-01'
```

Suppose the database also maintains two views

```

V(cName, mID, max(price)) ← Clients(cID, cName),
                             Orders(oID, mID, cID, date, price),
                             date >= '2006-01'
W(cID, mID, mDish, max(price)) ← Orders(oID, mID, cID, date, price),
                                 Menu(mID, mDish),
                                 date >= '2006-01'

```

The view V contains for each client who visited the restaurant since January 2006 his name and maximum price he paid together with the ID of the menu he ordered that time.

The view W contains for each client who visited the restaurant since January 2006 his ID and maximum price he paid together with the ID of the menu and main dish he ordered that time.

Notice that the view V is at most as big as the table **Client**, because it contains one record for each client who visited the restaurant since January 2006; and the table **Clients** is much smaller than the table **Orders**. The view W is smaller than the table **Orders**, because it contains only the orders since January 2006, and also for each pair of client and menu W can have at most one record, while the **Orders** table can contain several records with the same pair of values (**menuID**, **clientID**).

If we use the algorithm we used in Example 3, we find two rewritings that use views

```

QV(cName, mDish, max(X)) ← V(cName, mID, X), Menu(mID, mDish)
QW(cName, mDish, max(X)) ← Clients(cID, cName), W(cID, mID, mDish, X)

```

We simply combine each view with the missing table and obtain an equivalent rewriting. This is a standard technique used in most commercial query optimizers [CKPS95a, LMSS95, ZCL⁺00]. Both rewritings Q_V and Q_W are worth considering, because either of them may reduce significantly the evaluation cost of Q .

In fact, there is one more rewriting that uses views:

$$Q_{VW}(cName, mDish, \max(X)) \leftarrow V(cName, mID, X), W(cID, mID, mDish, Y)$$

It can be shown that this rewriting is also equivalent to Q , and, as we explained earlier, this rewriting can be more beneficial than Q_V and Q_W .

If we look at the definitions of V and W , we can see that the table `Orders` appears in both of them, thus, it is “covered” by both views. Hence, under certain circumstances it is possible to combine in a rewriting several overlapping views. This possibility increases the space of possible rewritings, and this additional group of rewritings can give much better plans of query evaluation.

The problem this example brings up is how to determine whether two overlapping views can be combined in an equivalent rewriting of a query. We develop efficient local conditions under which two views can be combined in a query rewriting, for SPJ queries and views with or without aggregation [GKC06]. Here is one of the results: Given a set-valued database, an SPJ query Q , and two applicable views V and W , such that these views have in common a nonempty set G of tables. Then V and W can be combined in an equivalent rewriting of Q on this database if and only if all the attributes in G are the output attributes of both views.

Let’s demonstrate how this rule works on the following example.

Example 5. *Suppose we have a query and three views:*

$$\begin{aligned} Q(A) &\leftarrow P(A,B), R(B,C), S(C,D) \\ V_1(A,B,C) &\leftarrow P(A,B), R(B,C) \\ V_2(B,C) &\leftarrow R(B,C), S(C,D) \\ V_3(C) &\leftarrow S(C,D) \end{aligned}$$

All three views are applicable, as we can substitute corresponding parts of the query definition by them. Views V_1 and V_2 overlap on the table `R`. The set of attributes of this table $\{B,C\}$ is a subset of the sets of output attributes of both V_1 and V_2 . Thus, according to our rule we can combine views V_1 and V_2 in an equivalent rewriting

$$Q(A) \leftarrow V_1(A, B, C), V_2(B, C)$$

Views V_2 and V_3 overlap on the table S . The set of attributes of this table is $\{C, D\}$. This set is not a subset of the set of output attributes of V_3 , thus, these two views cannot be combined in an equivalent rewriting.

Notice that view V_3 can still be used in a rewriting together with V_1 :

$$Q(A) \leftarrow V_1(A, B, C), V_3(C)$$

In [GKC06] we proposed a dynamic-programming algorithm based on our local conditions, for finding efficient execution plans for user queries using materialized views; queries and views may have grouping and aggregation. Our approach is a relatively simple generalization of the query optimization algorithm of [CKPS95a]; similarly to [CKPS95a], our approach can be used to exploit cached results. While exploring strictly more rewritings than the approach of [CKPS95a], in general our algorithm is incomplete; we presented a theoretical study of the completeness-efficiency tradeoff and described enhancements that would make the algorithm complete. Finally, we provided experimental results that show good efficiency and scalability of our algorithms.

Chapter 2

View- and Index-Selection Problem

In this chapter we give an overview of existing methods for the view-selection problem. This chapter is organized as follows. In (2.1) we describe a well-known taxonomy for the view-selection problem. In (2.2) we formally state the view-selection problem. In (2.3) we examine the problem for the data warehouses environment, i.e., for the case when each query can be answered using only one view. In (2.4) we consider the problem for a more general case when a query plan can contain a set of views. In 2.5 we present existing results for view-selection problem with the maintenance-time constraint.

2.1 Well-Known Taxonomy [Gup97] for a View-Selection Problem

In this section we present an *AND-OR* view graph notion defined in [Gup97]. We use this taxonomy because it allows us to classify cases of the view-selection problem, thus we are going to use it later to classify the research results we discuss in this section.

Suppose we are given a set of queries and a set of views together with possible rewritings of the queries using the views. Then, for each query Q , we can construct a directed acyclic graph (DAG) that represents possible rewritings of this query. In

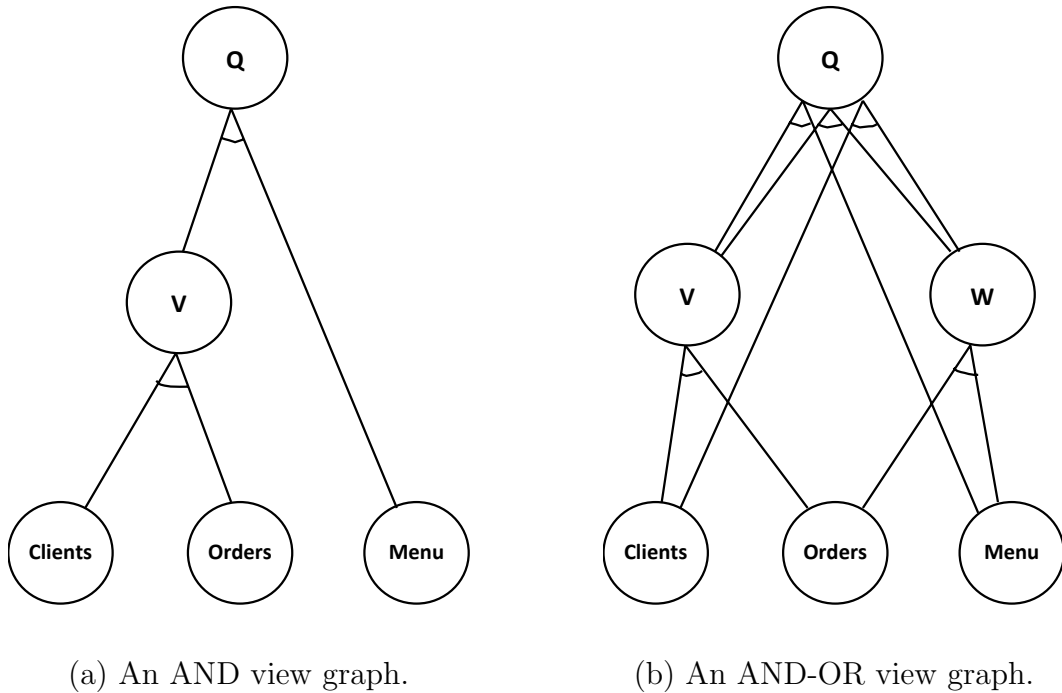


Figure 2.1: Examples of AND and AND-OR view graphs.

this graph, Q is the “source” node and the base relations are “sinks”. If a node u in this graph has outgoing edges to nodes v_1, v_2, \dots, v_k , then u can be computed if *all* of the views v_1, v_2, \dots, v_k are present and this dependence is indicated by drawing a semicircle, called an *AND arc*.

Figure 2.1a shows an example of such a graph, or *AND view graph*, for view **V** and the plan that uses this view to answer query **Q** in Example 4. In Figure 2.1a, query **Q** can be computed using a rewriting that involves views **V** and base relation **Menu**. Notice that the relation “can be answered using” is transitive, i.e., if the answer to query q_1 can be found using the result of query q_2 , and q_2 can be answered using the result of query q_3 , then q_1 can be answered using the result of q_3 . Thus, another possible rewriting for the query **Q** involves base relations **Client**, **Orders**, and **Menu**.

A more general case, where a query can be answered using various subsets of views, can be expressed using an *AND-OR view graph*. The only difference between *AND* and *AND-OR* view graphs is that a non-sink node in an *AND-OR* graph can

have more than one *AND* arc associated with it.

In Figure 2.1b, query *Q* can be answered using either $\{\mathbf{V}, \mathbf{Menu}\}$, or $\{\mathbf{Clients}, \mathbf{W}\}$, or $\{\mathbf{V}, \mathbf{W}\}$. By transitivity, query *Q* can also be answered using $\{\mathbf{Clients}, \mathbf{Orders}, \mathbf{Menu}\}$.

An *AND-OR* view graph for a set of queries Q_1, Q_2, \dots, Q_n is a join of *AND-OR* graphs for each query. That is, for each query Q_i we can find a subgraph of the *AND-OR* graph which is an *AND-OR* graph of Q_i .

2.2 Defining the View-Selection Problem

In this section we formally define the view-selection problem.

- Given:
1. A set of queries.
 2. A function that, for any view or index, returns the amount of space it occupies on disk, or for a given subset of views and indexes, returns its maintenance cost.
 3. A bound on available disk space or maintenance cost.
 4. A function that, for each query and each set of views and indexes, returns the cost of evaluating the query using these views and indexes.

Find: a subset of views and indexes that

- (i) fits into available memory or whose maintenance cost does not exceed the bound, and
- (ii) minimizes the response time of the given query set.

Item 4 of the problem formulation is a very important aspect of the view-selection problem. It is called a *cost model*. The choice of a cost model plays an important role in any work on the problem of view selection. Not only do we want to know which view accelerates a query best, but we would also like to have some quantitative measure that allows us to compare different subsets of views with each other. We are interested in how many elementary operations we need to perform to answer a

query using each view. This is the purpose of a cost model. A cost model must reflect relations between sets of operations. It means the following: if one set of operations is cheaper than another in real database, then it must be cheaper in the cost model. The choice of a cost model is a very important decision fundamental to database optimization.

2.3 Past Work in the Framework of OR View Graphs

The special case of an *AND-OR* view graph in which each AND arc binds only one edge is called *OR view graph*. This type of view graphs is often present in data warehouse environments [TS97]. Much work on the view-selection problem has been done for the data warehouses environment, because applications based on these works form a crucial part of decision support systems (DSS), which are used for data analysis by many companies. In this section we survey work for this subclass of the problem.

Data warehouses are collections of data designed to support management decision making [JS96]. For users of data warehouses, data are usually represented as a multidimensional cube, where the value of interest, for example *quantity of details*, is organized by several dimensions which are the attributes of interest. For instance, in Example 6, the quantity of ordered parts is organized by customer, part number, and time stamp of order.

Example 6. *Suppose we have four relations in our database:*

`Customer(ckey, cname, caddress), Part(pkey, pname), Time(dateID, day, month, year), Orders(ckey, pkey, dateID, quantity).`

Here, Orders is the fact table, and Customer, Part and Time are dimension tables [Ull89]. Each cell in this cube contains a value of interest. In our example, for each part p bought by customer c at time t we store the quantity purchased.

Users of data warehouses are usually interested in identifying trends rather than looking at individual records [GBLP96]. That is why queries in data warehouses

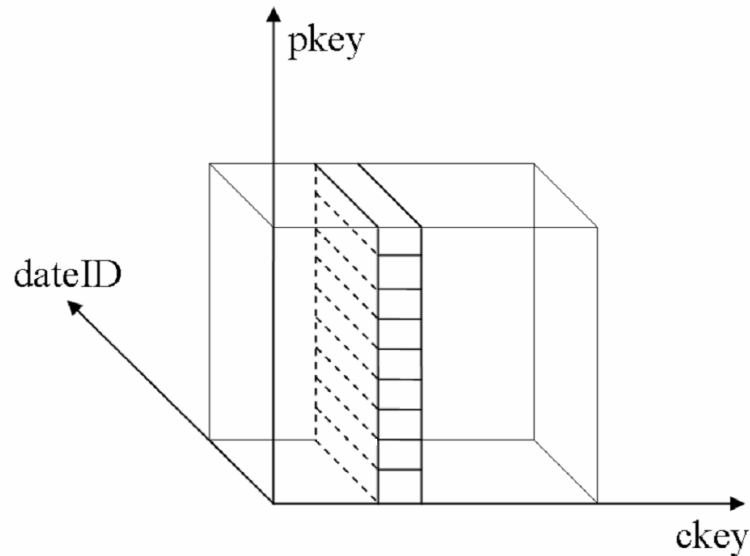


Figure 2.2: Representation of data as a data cube.

typically make heavy use of aggregation.

A typical query for such a database would be “for each customer find the total quantity of each part he purchased”:

```
Q1: SELECT ckey, pkey, SUM(quantity)
      FROM Orders
      GROUP BY ckey, pkey;
```

or, “for each part find the total quantity sold”:

```
Q2: SELECT pkey, SUM(quantity)
      FROM Orders
      GROUP BY pkey;
```

Figure 2.2 gives an intuitive representation of query Q1: for each pair (ckey, pkey) we sum up quantities along the bar parallel to the dateID axis.

Such queries do “dimensionality reduction”, by aggregating data along some dimensions. Notice that for an n -dimensional “data cube” there are 2^n different projections. Each projection is defined by a subset of the n attributes in the GROUP-BY clause. In Example 6, we have eight possible subsets:

- pkey, ckey, dateID
- pkey, ckey
- pkey, dateID
- ckey, dateID
- pkey
- ckey
- dateID
- none of the attributes

For example, the third subset corresponds to the query

```
V: SELECT pkey, dateID, SUM(quantity)
    FROM Orders
    GROUP BY pkey, dateID;
```

If, for instance, we choose to save the result of this query to disk as a materialized view, then we can use it to answer query Q2:

```
Q2(V): SELECT pkey, SUM(quantity)
        FROM V
        GROUP BY pkey;
```

Notice that we cannot answer query Q1 using this view, because the GROUP-BY clause of query Q1 contains attribute ckey, which is not present in the result of view V. Thus, view V does not contain enough information to answer query Q1. In general, we can answer query q using view v only if the set of attributes in the GROUP-BY and WHERE clauses of the query is a subset of attributes of the GROUP-BY clause of the view.

Therefore, each subset of attributes defines a view and for each two view from this group we can easily determine whether one view can be answered using the answer of another view. We can represent this dependency between queries from the set above as shown in Figure 2.3. It is also very often called *a view lattice*.

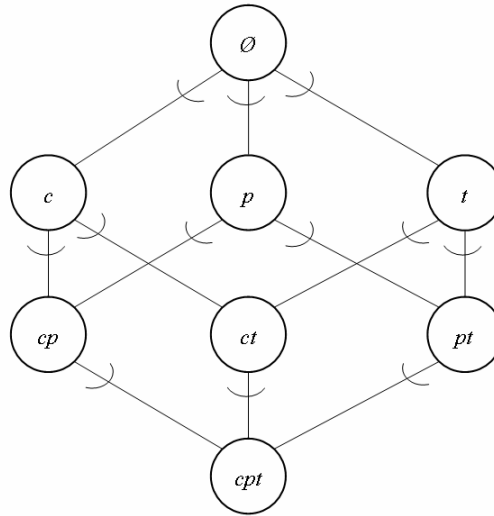


Figure 2.3: *OR* view graph.

All queries in data warehouses can be answered using at least one of the views from the view lattice. Thus, it is very important to materialize all or some of these views, because this can drastically reduce query response time. But in general, it is not possible to materialize the whole set because of a space or maintenance-time constraint. Hence, we need an algorithm that would choose a subset of views, which satisfy the constraints and would maximally accelerate query answering.

2.3.1 Near-Optimal Solutions

In this subsection we describe approaches for finding near-optimal solutions of the view-selection problem in data warehouses. Such methods [Gup97, HRU96a, GHRU97] use greedy heuristics in search of a set of views for materialization. The advantage of these approaches is that they usually have lower complexity, in comparison with exact methods, and scale better with problem size growth. One disadvantage of such algorithms is that the solution is not guaranteed to be optimal. Such works prove bounds on the error of the solution with respect to the optimal solution and try to improve these bounds.

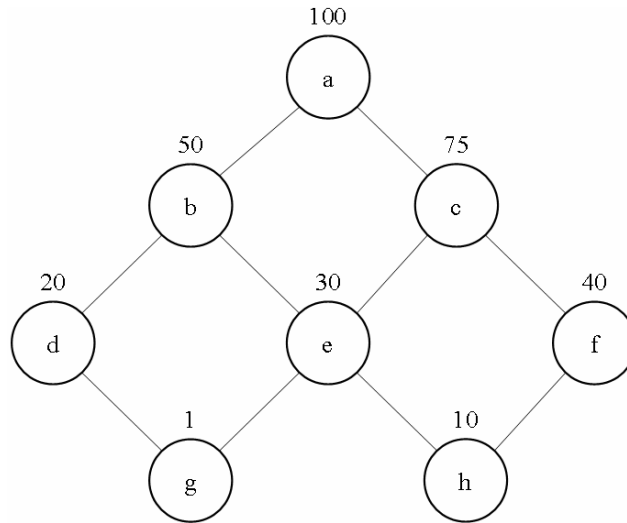


Figure 2.4: Example of a lattice with space costs.

Near-Optimal Solution of View-Selection Problem

Work by Harinarayan et al. [HRU96a] is one of the fundamental works for the problem of view selection for the case of data warehouses. In this work, a view lattice similar to an *OR* view graph defined earlier is used to represent dependencies between views.

In [HRU96a] the authors assume that the cost of answering a query is proportional to the number of rows scanned. Though this cost model cannot be very precise, it reflects a relationship between views very well. The idea of this model is that if a query can be answered using two different views, then a view with smaller size is preferable, as its scan takes a smaller amount of time.

To solve the problem of the optimal view selection the authors propose a greedy technique for near-optimal selection of materialized views. At each step, the algorithm first calculates the benefit of materializing each view in the lattice assuming that all previously chosen views are materialized and can be used to answer queries. After that, a view with the greatest benefit is chosen for materialization. Then the algorithm repeats these two steps until there is no more free space.

Let us take a look at an example from [HRU96a]. Suppose we have a view lattice

as shown in Figure 2.4, with space costs shown. The query workload is all the nodes of the lattice. The top view a must be chosen in order to be able to answer all the queries. Suppose we can choose three more views. The cost of processing query u using view v is equal to the size of view v .

Let S be the set of views we choose for materialization. Initially $S = \{a\}$. The total cost of evaluating all queries of the lattice using view a is $8 \times 100 = 800$. We now calculate the profit of each view. For instance, view c has profit $5 \times 25 = 125$ as it allows answering queries c, e, f, g, h with the cost 75, instead of 100. At this stage, view b has the best profit $5 \times 50 = 250$. Thus, $S = \{a, b\}$.

Now we recalculate the profits of all the views that are not yet chosen, assuming that S is materialized. For example, the profit of c is $2 \times 25 = 50$, as it improves the cost of c and f by 25 each. View f has the profit $60 + 10 = 70$ because it improves the cost of f by $100 - 40 = 60$ and the cost of h by $50 - 40 = 10$. View f has the best profit at this stage. Thus, after this step $S = \{a, b, f\}$. If we repeat this procedure one more time, we find that view d gives the highest profit, assuming that the views already in S are materialized. Therefore, the solution for this problem is $S = \{a, b, d, f\}$.

The authors present a proof that for the view-selection problem with restriction on the number of views that can be materialized, the solution given by this greedy algorithm has a benefit no less than $(e - 1)/e \approx 63\%$ of the optimal benefit. Unfortunately, in 1999 the paper by Karloff and Mihail [KM99] disproved the strong performance bounds of these algorithms, by showing that the underlying approach of [HRU96a] cannot provide the stated worst-case performance ratios unless $P = NP$. In fact, [KM99] proved that no polynomial time algorithm can achieve worst case performance ratio better than $n^{1-\epsilon}$, if $P \neq NP$.

The main drawback of the algorithm proposed in [HRU96a] is that the algorithm is greedy, thus, it is not exact. In addition, the algorithm shows good results when the restriction is the number of views to materialize, but in more realistic situation, when we have a bound on available disk space, the resulting viewset can be much worse than an optimal viewset.

The work [HRU96a] reveals another important issue in the problem of view selec-

tion. Most works in this field consider view selection either by using indexes or by using views, whereas, as we discussed earlier, these two aspects must be considered as one problem, because, first, these two types of structures share the same resource — disk space; second, indexes are useless without views. We consider the problem of selection of a set of views and indexes in the following subsection.

Near-Optimal Selection of a Set of Views and Indexes

In [GHRU97], Gupta et al. examine the problem of selecting materialized views and indexes for data warehouses. In data warehouses, data are represented as a cube — in the way we discussed in Section 2.3.

A straightforward extension of the algorithm presented in [HRU96a] (see previous subsection) would be to add to the set of views a set of indexes, and then to apply the same algorithm. That is, at each step the goal is to find a data structure — a view or an index — that maximizes profit under the assumption that all previously chosen structures are materialized, and add the chosen structure to the set of structures that we have chosen for materialization. Repeat the step until there is no more free space.

Most works in this area had considered a modification of this algorithm — a two-step algorithm. In a two-step algorithm, available space is divided into two parts. The first part is intended for views, and the second for indexes. After that, two separate problems are solved. The first problem is exactly the problem considered in [HRU96a], when for a given amount of available space we find a near-optimal subset of views. The second problem consists of choosing a near-optimal set of indexes. This step exploits the same idea. For a given amount of free disk space and a given set of materialized views, we iteratively and greedily pick indexes that maximize profit.

This two-step algorithm has several problems. First of all, there is no general algorithm for finding a partition of free space into two parts. Such an algorithm must consider the structure of views and available indexes, which appears to be a very difficult problem. The second disadvantage is that we choose views without knowing anything about indexes for these views. A view may not be very useful by itself without an appropriate index; thus, it will not be materialized in the first stage,

while if that view were materialized, a good index could be created giving us a big advantage in using that view.

For instance, let us have two views V_1 and V_2 with benefits 50 and 45, respectively. Suppose we have an index I_1 for the view V_1 and an index I_2 for the view V_2 with benefits 5 and 100, respectively, if the corresponding views are materialized. Suppose we are allowed to materialize one view and one index. In the first step we choose to materialize the view V_1 , because it gives a better benefit. In the second step we must choose one index. Index I_1 has profit 5 and index I_2 has profit 0, because the view V_2 is not materialized. Thus, we choose index I_1 . The total benefit is 55, while it is easy to see that if we choose V_2 and I_2 , then the total benefit is 145.

In their work [GHRU97], the authors proposed a slightly different approach to this problem that bounds the error of the algorithm. The main idea of this method is that views and indexes must be considered at the same stage. They presented a family of r -greedy algorithms. In each step, instead of picking only one structure at a time, as we discussed in the beginning of this subsection, the algorithm chooses a subset of at most r structures that maximize the benefit.

The set of structures that we choose at each step can consist either of

- A view and some of its indexes, or
- A single index whose view has already been selected at one of the previous steps.

The selection step is repeated until

- the maximum number of structures is selected, if the bound is on the number of structures; or
- no more structures can fit into the allocated space, if the bound is on the available space.

Notice that the complexity of this algorithm is polynomial in the total number of given data structures. Its complexity is $O(km^r)$, where m is the total number of structures and k is the number of structures selected by the algorithm. Thus, this method is efficient even for big instances of the problem.

An advantage of the proposed method is that we can improve the average behavior of the algorithm by increasing the complexity, i.e., we can use the trade-off between complexity and performance to regulate the error. Experiments in [GHRU97] showed that an algorithm of moderate complexity can perform fairly close to the optimal.

The main disadvantage of this algorithm is that it uses greedy choice, thus it is not exact. Again, the proposed algorithm does not give any guarantee of the solution quality. Also, the algorithm does not always respect the bound on the available space. It guarantees only that its solution will not occupy more than $2S$ units of space, where S is the initial bound on available space, which is not always acceptable. Another drawback of this approach is its scalability. The algorithm works with the set of all applicable views and indexes, while for the n -dimensional data cube, we can create 2^n different views and up to $n!$ indexes for each view.

Optimal Solution of the View-Selection Problem

In this subsection we examine methods for finding an optimal solution for the view-selection problem [LACF05]. This problem is NP -hard [KM99], therefore, is intractable assuming that $NP \neq P$. The good point of exact algorithms is that they guarantee an optimal solution. The disadvantage is that they are typically not very scalable, thus finding a solution for a big problem can take very long. The main direction of research in this area is to find methods that reduce the search space of possible solutions.

One of the approaches is to use an integer programming (IP) model [Sch86] for optimal viewset problem [LACF05]. In this approach, a lattice similar to the one in Figure 2.4 is considered. The set of queries is a subset of nodes of the lattice. It is assumed that the view corresponding to the top node is always materialized [HRU96a], thus we can always answer any query of the lattice.

In [LACF05] this problem is solved using the CPLEX solver [CPL00] with AMPL 9.0 [FGK94] interface. An instance of the problem with 32768 (2^{15}) nodes in the lattice was solved in less than 19 seconds.

The experimental results of [LACF05] show that the computational requirements

of solving this problem become prohibitive once the size of the problem exceeds certain limits. However, some heuristics that use the special structure of the problem could be found to solve larger instances of this problem. Knowing that such heuristics have been found for the facility location problem [KP83] and for the k -median problem [MC79], we can expect that similar heuristics can be found for the view-selection problem.

2.4 Past Work in the Framework of AND-OR View Graphs

In this section we consider a general case of the view-selection problem. In this case, a query can use several views. Furthermore, several queries can use the same view, which gives this view a better chance to be chosen for materialization.

Not so many results have been obtained in this area. Most of the papers in this domain are results of implementations of database management systems, such as Microsoft SQL Server [ACN00a, GL01, BC05] and Oracle [BDD⁺98]. These works mainly use greedy-based heuristics for the view set selection.

2.4.1 Automated Selection of Materialized Views and Indexes for SQL Databases [ACN00a]

In [ACN00a], Agrawal et al. present a tool for automated selection of materialized views and indexes. This paper considers the problem of view selection for query workloads consisting not only of aggregate queries, but also of other types of queries based on various subsets of tables with various join and selection conditions. The authors present an algorithm that for a given query workload and space constraint defines a set of “good” views and indexes, and then finds a subset of views and indexes that fits into available space and minimizes the processing cost of this query set.

Here is an outline of the algorithm proposed in [ACN00a].

- In the first step, the tool proposed in [ACN00a] identifies interesting sets of

tables which can be materialized as views. Many previous works were skipping this step, assuming the set of candidate views is equal to the set of possible table subsets that appear in the query workload. This approach is not scalable, because the size of a viewset grows exponentially in the number of base tables and join conditions. To choose interesting sets of tables [ACN00a] proposed a cost formula that uses a simple heuristic — it chooses a subset of tables whose weight is greater than some threshold.

- In the second step, the algorithm tries to combine views that are based on the same subsets of tables, but have different sets of GROUP BY and selection attributes. For each new view the tool tests that this view is “good”.
- After selecting all “interesting” views in this way, the algorithm adds to this set a set of possible indexes on the base tables and on all interesting views.
- Finally, the algorithm chooses a subset of structures that fits into the allocated memory and maximizes performance. At this step, the tool uses a **Greedy(m,k)** algorithm. The algorithm returns a total of **k** materialized views and indexes. It first finds an optimal configuration of size up to **m** by exhaustive search, and then picks the remaining structures greedily.

This tool was implemented in Microsoft SQL Server 2000 [ACN00a].

The main weakness of the proposed algorithm is that it uses heuristics without any theoretical guarantee on quality. The problem of finding parameter values for a selection of interesting views is an open problem (the authors mention that for different workloads these values are different). The proposed algorithm pays attention to the choice of “good” views, but does not do the same for indexes. Thus, the search space of data structures for materialization may become quite big because of the big number of indexes.

2.4.2 Automatic Physical Database Tuning: A Relaxation-based Approach [BC05]

The paper by Bruno Chaudhuri [BC05] is another notable work in this area. This paper presents the Relaxation-based Approach (RBA). The authors address the same problem as in [ACN00a]. Although the algorithm that they propose uses similar ideas, the overall solution is quite different.

As many other approaches, the algorithm starts by selecting a set of “good” structures. But unlike [ACN00a], it: (1) in addition to views, selects indexes; (2) uses a standard optimizer to identify interesting views and indexes. To identify the set of beneficial structures, the algorithm launches optimization of the input queries one-by-one, intercepts all calls of the optimizer for the views and indexes that could be useful for the plan, and simulates these structures in the candidate pool.

On the second stage, the algorithm starts with the configuration of views and indexes that was created on the first stage. This configuration serves as the initial solution, but it is usually infeasible, because it violates the space constraint. After that, the algorithm “shrinks” the configuration using transformations, such as view and index merging, index splitting and prefixing, or by simply removing some of the structures. Using this technique, the algorithm applies an heuristic search for feasible configurations, and stops when reaches a predefined time limit.

One advantage of this algorithm is that it finds a feasible solution relatively fast, especially for problems with a large space bound. Another advantage is that it can return to the DBA a set of solutions for various space bounds, so that the administrator may decide to accept a recommendation that requires slightly more space, but is “good enough” for the database. But one of the main advantages of the proposed algorithm is that it bases its choices of “good” views and indexes on actual approximations of the optimizer. This gives a more precise estimations of costs and sizes, and guarantees that the optimizer will use these structures if they are chosen for the materialization.

The disadvantage of this algorithm is that it does not give any performance guarantees, and in fact, gets much worse for lower space bounds, because in this case, it requires more time to get from the initial configuration to a feasible solution.

2.5 Selecting Views Under a Maintenance-Cost Constraint

Most of the work on selecting views to materialize have been done for the case where the main constraint is a disk-space bound. In practice, this approach is not very useful as disk space is very cheap and in some cases we really can materialize most needed views. In fact, the real constraining factor is maintenance costs — the time we need to spend to maintain data in view answers up to date. Note that this variant of the view-selection problem is actually harder than the one with the disk space constraint. The added hardness comes from the fact that the maintenance cost of a view depends on the set of views which are already selected for materialization. That is, to update a view we can use another view which is already updated, and the time needed to update two views separately is, in general, greater than the time needed to update these views together. Thus, the disk-space constraint version of the problem is a special case of the maintenance-time constraint problem, when all views are independent and can be updated from base relations only.

The paper by Gupta and Mumick [GM99] is the first work that considers the problem of view selection under a maintenance-cost constraint. As we already explained, this problem is harder than the problem with the disk space constraint, and the solutions proposed to the latter (such as [Gup97, HRU96a]) are not applicable for this case.

For a special case of the view-selection problem with *OR* view graph, [GM99] define the notion of *inverted tree set*. Intuitively, an inverted tree set is a subtree of an inverted *OR* view graph. The authors proved a result that states that for any subset of views there is a partition of this set into a set of inverted trees which satisfy a monotonicity condition of the maintenance-cost function. This means that if

we operate at the level of inverted tree sets, then the maintenance-cost function is monotonic.

Experimental results in the paper show that this technique is quite effective and, in most cases, finds an optimal solution, but the complexity of the proposed algorithm is exponential in the number of views and base tables.

For the general case of the view-selection problem with *AND-OR* view graph, [GM99] adopt the A* algorithm. The proposed algorithm is guaranteed to find an optimal solution, but its complexity in the worst case is exponential in the number of views and base tables.

As we have seen, much of work on the view-selection problem has been done for the data-warehouse case, i.e., for sets of aggregate queries. At the same time, the general case of queries that use combinations of views is still relatively unexplored, and existing algorithms for this case are limited to the use of greedy heuristics without rigorous proofs of quality and error bounds. In the next chapter, we present our approach to the view-selection problem with the general case of *AND-OR* view graph.

Chapter 3

Our Approach to the View- and Index-Selection Problem

In this chapter we address the View- and Index-Selection problem with the constraint on the available disk space: Given a set of frequent and important queries on a relational database, generate a set of evaluation plans that provides the lowest evaluation costs for the input queries on the given database. Each plan requires the materialization of a set of views and indexes, and cannot be executed unless all of the required views and indexes are materialized. The total size of materialized views and indexes must not exceed a given space (disk) bound.

To solve the problem, we propose a novel end-to-end approach — Automated Database Restructuring (ADR) that focuses on *systematic* exploration of view- and index-based *plans* for evaluating the input queries. Specifically, we propose a framework (architecture) and algorithms to select views and indexes that contribute to *the most efficient* plans for the input queries, subject to the space bound. We present strong optimality guarantees on the proposed architecture. The algorithms we propose search for sets of competitive plans for queries expressed in the language of conjunctive queries with arithmetic comparisons. This language captures the full expressive power of SQL select-project-join queries, which are common in practical database systems. Our experimental results on synthetic and benchmark instances demonstrate the competitiveness and scalability of our approach.

Formally, an instance of the problem is a tuple $(\mathcal{Q}, B, \mathcal{S}, \mathcal{L}_1, \mathcal{L}_2)$ defined with respect to a database \mathcal{D} . Here, \mathcal{Q} is a workload of $n \in \mathbb{N}$ input (frequent and important) queries, the natural number B represents the input storage limit in bytes, and \mathcal{S} represents statistical information about \mathcal{D} . \mathcal{L}_1 is the language of views that can be considered in solving the instance, and \mathcal{L}_2 is the language of rewritings represented by the plans in the solution for this instance. More precisely, the problem output is a set $P = \{p_1, \dots, p_n\}$ of n evaluation plans, one plan p_i for each query q_i in \mathcal{Q} , such that each plan p_i (a) is associated with an equivalent rewriting of q_i in query language \mathcal{L}_2 , and (b) can reference only stored relations of D and views defined on D in query language \mathcal{L}_1 . Finally, (1) for the sum s of the sizes (in bytes) of the tables for all the views mentioned in the set of plans P , it holds that $s \leq B$, and (2) for the costs $c(p_i)$ of evaluating the plans $p_i \in P$, the sum $\sum_{i=1}^n c(p_i)$ is minimal among all sets of plans whose views satisfy condition (1).

We now provide the details on the database statistics \mathcal{S} in the problem input. Access to the database statistics is not handled directly by the algorithms in our architecture. Rather, the algorithms assume availability (and use the standard optimizer APIs) of a module for viewset simulation and evaluation-cost estimation for view-based query plans. That is, we assume the availability of a “what-if” optimizer similar to those used in the work (e.g., [ACN00b, BC05]) on view/index selection for Microsoft SQL Server. Observe that the use of such a *black-box* module in our architecture guarantees that the plans in the ADR problem outputs are going to be considered by the actual (“target”) optimizer of the database system once the views mentioned in the plans are materialized. We assume that the target optimizer in question can perform query rewriting using views (see, e.g., [GKC06, CKPS95b]) and that the what-if optimizer module used in our architecture uses the same algorithms as the target optimizer in the database system.

The CQAC problem: For the algorithms that we introduce in Section 3.2, in the above problem inputs we restrict the language of input queries, as well as each of languages \mathcal{L}_1 and \mathcal{L}_2 , to express SQL queries that are single-block select-project-join expressions whose **WHERE** clause consists of a conjunction of simple predicates. (This language corresponds to conjunctive queries with arithmetic comparisons, *CQACs*,

see, e.g., [Klu88].) Further, we make the common assumption (see, e.g., [OL90]) of no cross products in query- or view-evaluation plans. In our cost model, we assume nested-loop joins only. The cost formula for this type of joins, as well as the cost formulas for the remaining operators that occur in the plans we consider, are as described in [GMUW02]. Finally, in the language of input queries and in the language \mathcal{L}_1 we restrict all queries to be chain queries.

Definition 1. *Chain queries are the queries whose tables can be arranged in a sequence, such that, the join conditions (e.g., conditions of the form $Table1.attr1 = Table2.attr2$) occur only between neighboring in the chain tables.*

Thus, we consider queries of the form:

SELECT $attr_1, \dots, attr_k$ FROM T_1, \dots, T_n WHERE	$T_1.jattr_1 = T_2.jattr_1$ AND \dots $T_{n-1}.jattr_{n-1} = T_n.jattr_{n-1}$ AND $sattr_1 \leq C_1$ AND $sattr_2 \geq C_2$ AND $sattr_3 = C_3$ AND \dots	$\left. \begin{array}{l} \dots \\ \dots \end{array} \right\}$	join conditions selection conditions (or, constraints)
---	---	---	---

In our approach, we consider only chain views — views corresponding to answers to chain queries. Although, there are works (e.g., [OL90]) that show that for some special cases it might be beneficial to materialize a cross-product of tables, we do not consider such views, because in general they are less efficient. This is a standard assumption made by most modern commercial optimizers.

Example 7 clarifies the use of CQAC queries, views, and rewritings. Please see Section 3.4 for more general classes of the problem inputs that are covered by generalizations of the approach of Section 3.2.

Example 7. Suppose we are given a database with tables $A(a, b)$, $B(b, c)$, $C(c, d)$ and chain query

```
SELECT  A.a, B.c
FROM    A, B, C
WHERE
        A.b = B.b AND
        B.c = C.c AND
        C.d ≤ 100
```

Possible chain views for this query would be

```
CREATE TABLE V1 AS
SELECT  A.a, B.c
FROM    A, B
WHERE
        A.b = B.b
```

or

```
CREATE TABLE V2 AS
SELECT  B.b, B.c
FROM    B, C
WHERE
        B.c = C.c AND
        C.d ≤ 100
```

Note how the views have attributes that either match outputs of the query or are used in the **WHERE** clause of the query. Also note that we do not consider a view based on tables A and C , because the query does not have a join condition between these two tables.

We now provide plans that use the above views:

```

SELECT  V1.a, V1.c
FROM    V1, C
WHERE
        V1.c = C.c AND
        C.d ≤ 100

```

and

```

SELECT  A.a, V2.c
FROM    A, V2
WHERE
        A.b = V2.b

```

In this work we use the term “automated database restructuring” to point out the possibility of “inventing” new views (see [Chi02]) in specific algorithms in stage one of our proposed architecture. This way, the algorithms could ensure *completeness* of the exploration of the search space of view-based plans. For our query-language restrictions on the algorithms of Section 3.2, our proposed algorithms (presented in that section) *are* complete in that sense.

3.1 The Architecture

As discussed in Chapter 2, the problem of view selection is hard for a number of reasons. Thus, most *view*-selection approaches in the literature rely on heuristics with no guarantees of optimality or approximate optimality in the sense of [ACG⁺99].

Our problem statement ADR, see Chapter 3, emphasizes *plans* at the expense of *views*, and thus necessitates a different architecture from those proposed in the literature for the *view*-selection problem. Our architecture is natural for our problem statement, in that the architecture first forms a search space of plans, and then does selection of the best combination of plans in that space. We propose a branch-and-bound approach to the selection of an optimal configuration of the plans in Chapter 4. Alternatively, an optimal combination of the given plans can be selected by a general

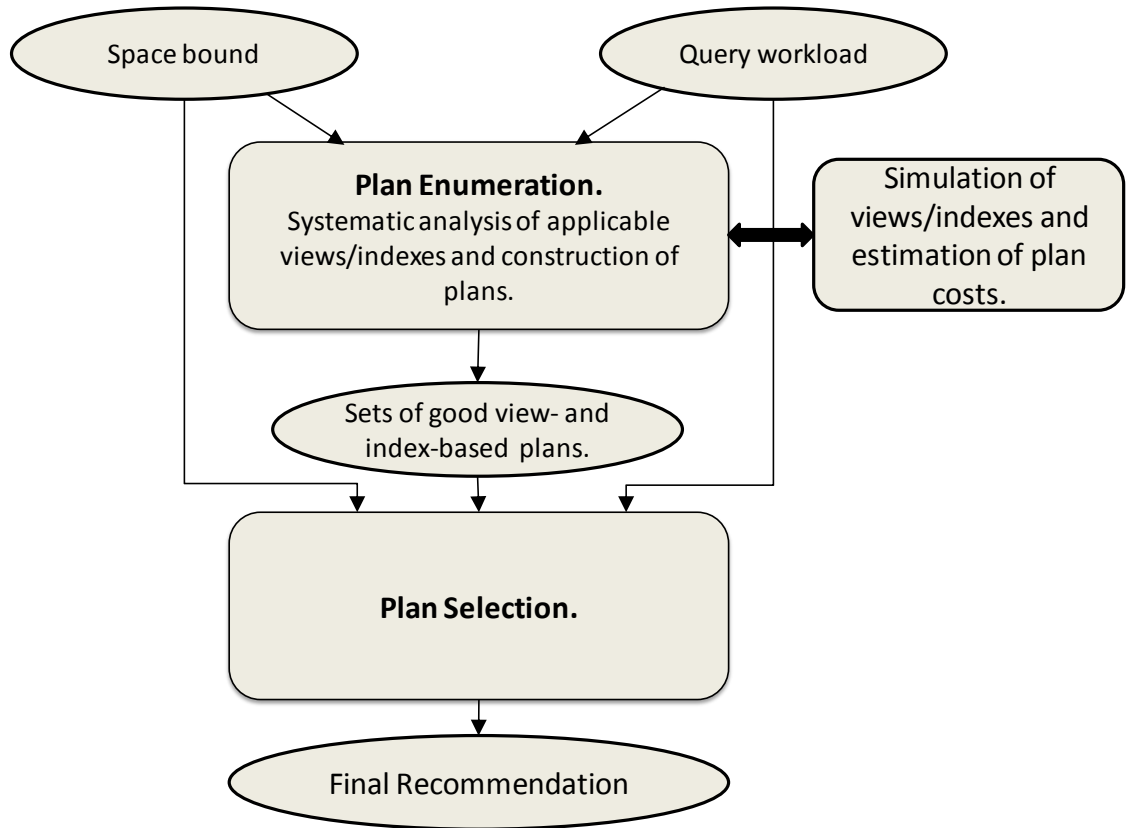


Figure 3.1: Our two-stage architecture.

Integer Linear Program (ILP) problem solver such as CPLEX [ILO04], as will be discussed below.

As shown in Figure 3.1, our architecture has two stages: (1) A search for sets of competitive plans for the input queries (“plan enumeration”), and (2) Selection of one efficient plan for each input query (“plan selection”). The plans in the output of the second stage are formulated as sets of views and indexes whose materialization would permit evaluation of the plans in the database systems. The first stage begins with a query workload and, optionally, a space bound. It produces a set of plans and corresponding views so that there is at least one plan for each query. The output of stage one, along with the original query workload and space bound, becomes the input

to stage two. (The space bound is optional as input to stage one because different bounds can be introduced in stage two.)

All problem-specific details are encapsulated in stage one of our architecture. For example, we can restrict the nature of the queries allowed, the types of views, the operators, etc. The only output from stage one is a set of *IDs* of plans and of the views used in the plans. In fact, nothing about the details of the plans or views needs to be conveyed to stage two other than (a) the set of plans for each query; (b) the set of IDs of views required by each plan; (c) the cost of each plan; (d) the space required by each view; and (e) the space bound.

Stage two is then free to solve a generalized knapsack problem: find the lowest-cost plan for each query such that the total space used by the required views is no greater than the given bound. An ILP formulation of this problem is given in Section 4.2.1.

While a large variety of exact algorithms and heuristics can be employed in stage two, the last 10-15 years have seen the emergence of sophisticated commercial solvers directed at general problems in mathematical programming, such as integer programming, quadratic programming, and constraint programming. These include Ilog CPLEX [ILO04], COIN-OR [COI], and SAS/OR [SAS]. Their superiority over problem-specific heuristics and algorithms has been demonstrated in various domains, including design automation [LSB05]. Furthermore, these solvers are enhanced regularly with significant improvements that directly impact their use in application areas where they may not have been competitive a few years ago.¹

Another major advantage of the general-purpose solvers is their ability to incorporate new constraints and problem-specific heuristics at runtime via *callbacks*. The solver can be configured so that, at any point during the search, the current state can be used to decide whether to (a) add a constraint, (b) invoke a heuristic, or (c) to bias the search in a particular direction.

¹Our collective experience bears this out. The [LSB05] paper, based on CPLEX 7.5, reports instances where an approach that combines general-purpose and domain-specific techniques outperforms CPLEX; with CPLEX 9 these results no longer hold. More recently, our experiments showed that a Lagrangian relaxation for our ILP model in Section 4.2.1 outperformed CPLEX 9 but not CPLEX 11, as the latter added a powerful presolver and randomized heuristics for finding feasible solutions.

Our work strives to exploit the performance of CPLEX and other ILP solvers by transforming ADR instances into ILP instances. The biggest challenge is that, for a given query workload, the number of potential views, indexes, and (by extension) plans grows exponentially or worse in the number of queries. This combinatorial explosion plagues *all heuristics and algorithms* for view (or index) selection, either directly, or in most cases indirectly (for example, when a heuristic is only able to explore a small portion of the solution space).

Our architecture demonstrates that (a) the difficulties can be isolated (in stage one) and, for special cases of practical interest, overcome; and (b) even large instances of the ILP formulation in stage two can be solved efficiently.

Given the fact that the input to stage two abstracts the details of the original ADR instance and reduces the problem to an ILP model, the following propositions hold with respect to any exact stage-two Algorithm *A*. Their correctness derives directly from the design of our architecture.

Proposition 1. *If the set of plans P has a subset P' such that P' includes at least one optimal plan for each query and the views and indexes required by P' satisfy the space limit B , then any solution produced by Algorithm *A* using the subset P' will be optimal with respect to the original query workload and B .*

Proposition 2. *If the set of plans P has a subset P' such that the total cost of plans in P' is within relative error ϵ of the optimum cost for the original query workload and the views and indexes required by P' satisfy the space limit B , then any solution produced by Algorithm *A* will be within relative error ϵ of optimal with respect to the original query workload and space B .*

In other words, the quality of the output of stage one directly determines the quality of the solution produced by our architecture.

In the following sections we will talk about implementations of both stages of our architecture for variants of the CQAC problem (Sections 3.2 and Chapter 4), and present results of the experiments confirming the competitiveness of our approach (Section 3.3). Finally (Section 3.4), we discuss extensions of the CQAC problem that can be handled by straightforward generalizations of our algorithms of Section 3.2.

3.2 Efficient Evaluation Plans for CQAC Queries

We present two variations of a specific algorithm implementing stage one of our architecture of Section 3.1. The algorithm is applicable to conjunctive queries with arithmetic comparisons (CQAC queries) in the problem input, and considers views and rewritings in the language of CQACs. One variation that we propose is an optimal algorithm in the sense of Proposition 1. The other generates fewer plans, trading optimality for efficiency. As our experiments in Section 3.3 show, the solution quality of the second approach is still quite good, optimal or almost so for small instances, superior to those of [BC05] for larger ones (for which the optimum is not known). Please see Section 3.4 for a discussion of more general classes of problem inputs that are covered by generalizations of these algorithms.

The remainder of this section proceeds as follows. In Section 3.2.1 we illustrate the main idea via an algorithm that finds an optimal plan for a single CQAC query. This algorithm, while of no interest in this context to our architecture, introduces the framework for our proposed stage-one algorithms for multiple input queries. In Section 3.2.2 we deal with the complications arising with multiple queries, and show that our algorithms still maintain optimality. At the end of Section 3.2.2 we present two *pruning rules* that significantly reduce the overall number of plans under consideration. One of these maintains optimality in the sense of Proposition 1, the other drastically reduces the number of plans at the expense of the optimality guarantee, but still yields high-quality solutions, see Section 3.3.

3.2.1 Finding Efficient Plans for a Single CQAC Query

The standard System-R-style optimizer [SAC⁺79] uses dynamic programming (*DP*) to find best plans for all subqueries of a given query, in order of increasing sizes. For each subquery, it creates new plans that join plans for the component subqueries. After that, it chooses and saves the cheapest plan for each *interesting order* of tuples. We say that the result returned by a (partial) plan is in an interesting order, if the ordering of tuples of the result is useful (i.g., permits the use of efficient join

algorithms) in the subsequent joins. Example of an interesting order is the ordering of tuple by value of an attribute that has a join condition on it.

We adapt this algorithm to ADR, with two important modifications:

1. In addition to the plans created by joins of subplans, we consider one more plan: a simulated *covering view* — a view that matches the subquery exactly (if not already in the database). This allows us to consider plans with a variety of combinations of simulated views.
2. We keep *all* relevant² plans for each subquery. This is important for feasibility: The views required by the plan for each of the subqueries may satisfy the space bound, but the total space bound of those views when the plans are joined may fail to do so.

We introduce our approach by way of Algorithm SINGLEQUERYPLANGEN in Figure 3.2. The algorithm (as well as the algorithm of Section 3.2.2) finds all *bushy* plans of a query, thus ensuring optimality. Most modern optimizers limit themselves to the smaller (incomplete) search space of linear plans. We could do the same and obtain much more efficient algorithms.

In Algorithm 1, each subquery is represented with a *node* that contains a list of plans. The nodes are organized into the *DP lattice*, with single-table subqueries at the bottom, followed by the subqueries based on two tables, etc., and with a single node corresponding to the whole query at the top.

The algorithm investigates the nodes of the lattice in the bottom-top manner and builds the plans using two techniques: (1) joining of plans for smaller subqueries (Procedure CONSTRUCTJOINPLANS); (2) creating plans that use only one view containing the answer to the subquery (Procedure CONSTRUCTVIEWPLAN).

Note that for a chain query on k tables, there are 2^{k-1} different view-based rewritings. It is the number of different ways to partition the chain of size k into at most k non-empty pieces. Each such partition can be translated into a rewriting by creating a view for each piece.

²Even when optimal plans are sought, it is not necessary to keep all plans. The number of plans can be pruned significantly, as described in Section 3.2.2.

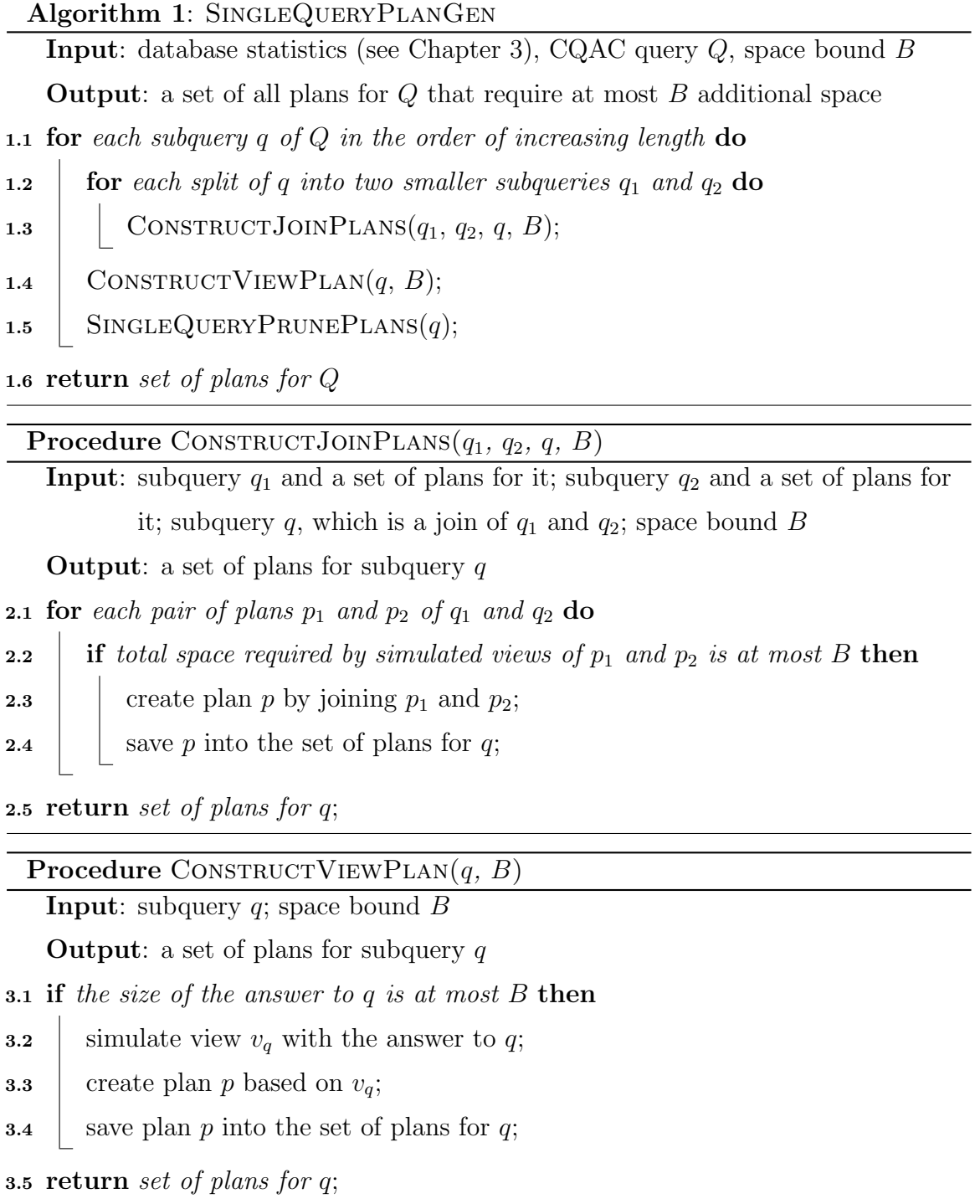


Figure 3.2: Constructing (possibly view-based) evaluation plans for a single CQAC query.

We assume that the operations in lines 2.2-2.4 of `CONSTRUCTJOINPLANS` can be executed in constant time. Suppose, subquery q_1 has length k_1 and q_2 has length k_2 . Then, in the worst case, in the for-loop in line 2.1, we consider joins of 2^{k_1-1} plans for q_1 with 2^{k_2-1} plans for q_2 , for a total of 2^{k-2} joins, where $k = k_1 + k_2$ is the length of q . The for-loop in line 1.2 considers $k - 1$ splits of subquery q of length k in two smaller subqueries. Thus, the total number of join-based plans that we build for a subquery of size k is $(k - 1)2^{k-2}$. After that, in lines 3.1-3.4, we create one more plan that is based on a view corresponding to the answer to subquery q . We assume that creating such plan takes approximately the same amount of time as creating a join based plan. Note that we keep only 2^{k-1} plans for q , because some of the plans correspond to the same rewriting, and we keep only the best plan for each rewriting.

Thus, the overall complexity of the algorithm is proportional to

$$\sum_{k=1}^n (n - k + 1)((k - 1)2^{k-2} + 1) \leq n^3 2^n,$$

where n is the number of tables in the query.

Observe that for a given subquery, the total number of possible plans may be exponential in the number of tables. Thus, considering all of them may become prohibitive even for small-size queries. One of the contributions of this work is a set of pruning rules that allow us to avoid considering large parts of the space of the view-based plans for the input queries.

We now define formally the notion of plan domination, and formulate our pruning rule for the single-query case. Let $cost(p)$ be the cost of executing plan p and $weight(p)$ be the total size of all views used by p .

Definition 2. *Let p_1 and p_2 be two plans for the same subquery. If p_1 and p_2 return the same tuples in exactly the same order, such that $cost(p_2) \leq cost(p_1)$ and $weight(p_2) \leq weight(p_1)$ each hold, with at least one strict inequality, then plan p_2 dominates plan p_1 .*

Procedure `SINGLEQUERYPRUNEPLANS` in Algorithm 1 removes plan p from the list of plans for a subquery whenever p is dominated by another plan p' . The implementation of this procedure can be found in Figure 3.3.

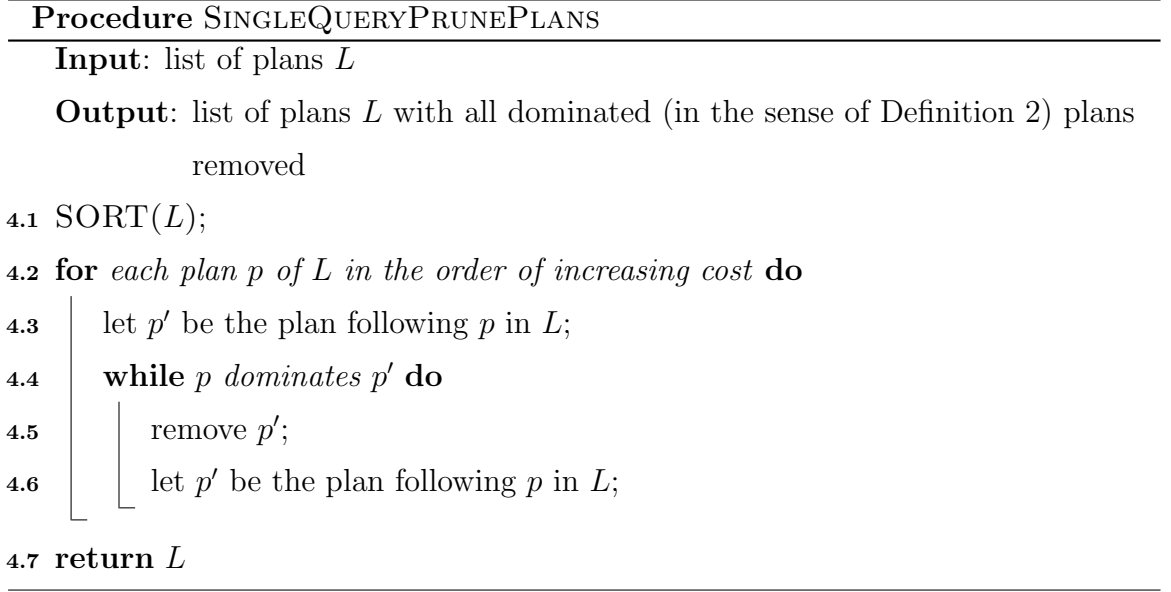


Figure 3.3: Removing dominated (in the sense of Definition 2) plans from the list of plans of a given subquery.

Proposition 3. *Suppose we have three plans for the same sub-query: p_1 , p_2 , and p_3 . Also, suppose that $\text{cost}(p_1) < \text{cost}(p_2) < \text{cost}(p_3)$ and p_1 does not dominate p_2 . Then p_1 dominates p_3 only if p_2 dominates p_3 .*

Proof. We prove proposition by contradiction. Suppose, p_2 does not dominate p_3 , while p_1 dominates p_3 . This means that $\text{weight}(p_2) > \text{weight}(p_3)$ and $\text{weight}(p_1) \leq \text{weight}(p_3)$. From this, it follows that $\text{weight}(p_1) \leq \text{weight}(p_2)$, thus, p_1 dominates p_2 . This contradicts to the fact that p_1 does not dominate p_2 .

■

In this algorithm, we use the property described by Proposition 3. Mainly, we sort the plans of the query in the order on increasing costs (alternatively, we can sort them by weights) and compare only neighboring in the list plans. For a given plan, if it does not dominate the immediately following it plan, then we do not need to compare it with the rest of the plans.

We now analyze the complexity of SINGLEQUERYPRUNEPLANS. Suppose, sub-query q has s plans. First, we sort these plans in the order of increasing costs,

$O(s \log s)$. After that, for each plan starting with the one with the lowest cost, we need to check the domination condition of this plan against the neighboring plan with the higher cost. If the domination is confirmed, we remove the latter and check the next plan with higher cost. We stop when we find a plan that is not dominated. Thus, for each plan, we perform as many domination checks as the number of removed plans plus one, and the total number of domination checks is

$$\sum_{i=1}^s (t_s + 1) = \sum_{i=1}^s t_s + s,$$

where t_s is the number of plans removed by domination on s^{th} iteration.

Note that we can remove at most s plans from the list. Thus, the total number of the domination condition checks for a given subquery is proportional to $O(s)$. This means the worst case complexity of SINGLEQUERYPRUNEPLANS is $O(s \log s) + O(s)$.

For a subchain of size k , the total number of view-based rewritings is 2^k , thus replacing s with 2^k , we get $O(k2^k)$. We need to perform this set of operations for each subquery, thus the overall added complexity of SINGLEQUERYPRUNEPLANS is $O(n^3 2^n)$, where n is the number of tables in the query. We can see that the worst case complexity remains the same.

Example 8 illustrates the work of Algorithm SINGLEQUERYPLANGEN.

Example 8. *Consider the query*

```

SELECT  A.a, B.c
FROM    A, B, C
WHERE
        A.b = B.b AND
        B.c = C.c AND
        C.d ≤ 100

```

from Example 7 and suppose the database is populated as in Figure 3.4.

Algorithm SINGLEQUERYOPT starts by creating plans for the subqueries A, B, and C based on sequential scans of the corresponding tables: call these p_A , p_B , and p_C , respectively. A second alternative for C is to create a view for $C(c, d) d \leq 100$:

A		B		C	
a	b	b	c	c	d
1	2	2	2	2	50
2	4	2	5	3	100
3	4	3	4	4	150
3	5	4	5	5	200
4	6	4	3		

Figure 3.4: Sample database.

subquery	plan	cost	space
AB	$p_{AB,1} : -p_A \bowtie p_B$	16	0
	$p_{AB,2} : -V_{AB}$ where $V_{AB} : -A \bowtie B$	6	6
BC	$p_{BC,1} : -p_B \bowtie p_C$	11	0
	$p_{BC,2} : -p_B \bowtie p_{C,2}$	9	2
	$p_{BC,3} : -V_{BC}$, where $V_{BC} : -BC(d \leq 100)$	2	2

Figure 3.5: Plans for the 2 subqueries of length 2.

call this plan $p_{C,2}$. The plan p_C requires no additional space and has cost 4 while $p_{C,2}$ requires space 2 and has cost 2.

Figure 3.5 shows the plans for subqueries of length 2. In each case, the first is derived by joining scans of two base tables and the last from a view created specifically for the subquery. In the case of BC , the middle alternative arises because there was a second plan for subquery $C(d \leq 100)$. Because of the pruning rule the plan $p_{BC,2}$ is eliminated: it is more expensive than $p_{BC,3}$ and uses the same amount of space.

Fig. 3.6 shows the plans for the whole query, those that do not use the already pruned plan. Pruning can eliminate all but 3 of these 7 plans. Plan 1 dominates plan 3, while plan 2 dominates plans 3, 4, and 5. With a space bound of 0, the best cost we can achieve is 19; in this case only the two plans with space 0 would have survived the creation of length 2 queries. Given a space bound of 2, the cost can be reduced to 10; with a bound of 3 it can be reduced to 3. Fig. 3.7 shows the actual tables and views for these three plans, ranked by increasing space bound, with brackets

plan id	plan	cost	space
1	$p_A \bowtie p_{BC,1}$	19	0
2	$p_A \bowtie p_{BC,3}$	10	2
3	$p_{AB,1} \bowtie p_C$	23	0
4	$p_{AB,1} \bowtie p_{C,2}$	21	2
5	$p_{AB,2} \bowtie p_C$	13	6
6	$p_{AB,2} \bowtie p_{C,2}$	11	8
7	$V_{AC}(a, c)$ where $V_{AC}(a, c) : -A(a, b)B(b, c)C(c, d)d \leq 100$	3	3

Figure 3.6: All available plans for query Q with unrestricted amount of the available disk space.

plan id	plan	cost	space
1	$A(a, b) \bowtie [B(b, c) \bowtie [C(c, d)d \leq 100]]$	19	0
2	$A(a, b) \bowtie V_{BC}(b, c)$	10	2
7	$V_{AC}(a, c)$	3	3

Figure 3.7: Ranking of view-based plans for query Q .

indicating the join order.

Theorem 1. *Algorithm SINGLEQUERYPLANGEN returns all view-based plans that do not violate the input space bound and are not dominated by other plans.*

We prove, by induction on the length of the (sub)query q , that the list of plans for q in Algorithm SINGLEQUERYOPT includes all plans that are within the space bound and are not dominated. This is clearly true for queries on the original base tables.

Let p be a plan for (sub)query q and suppose p obeys the space bound and is not dominated. If p is simply the view representing q , then the algorithm adds it to the list in lines 3.2-3.4. Otherwise p is $p_1 \bowtie p_2$ possibly followed by some selections, where p_1 and p_2 are plans for smaller subqueries q_1 and q_2 , respectively. Since p satisfies the space bound, so do p_1 and p_2 . And neither p_1 nor p_2 is dominated. If, for instance, p_1 was dominated by p'_1 , then plan $p'_1 \bowtie p_2$ would dominate p , contradicting the fact that p is not dominated.

By induction we know that p_i is on the list for q_i for $i = 1, 2$ and p is considered in the loop in line 2.1 of the algorithm.

■

3.2.2 Finding Efficient Plans for Multiple CQAC Queries

In this subsection we discuss how to adjust the algorithm of Section 3.2.1 to work for multiple CQAC queries. Our proposed algorithm is applicable to *chained queries*.

Definition 3. *A set of queries Q is a set of chained queries, if there exists a sequence of base tables L (possibly, with several occurrences of the same tables), such that, for each query q of Q , its set of tables matches with a subsequence of L , and the joins in q occur only between neighboring tables in L .*

We refer to L as the *global chain*. In this work, we do not address the question of constructing L . We assume that the global chain can be easily deduced from the database schema. Note that the queries in the TPC-H benchmark [TH] are chained queries on a chain of length 9, with possibly some additional queries that turn the chain into a cycle or follow a single branch away from the chain. The case of the branch can be incorporated into our approach (see Section 3.4), while the cycle is currently under investigation.

A naive approach to processing problem inputs with multiple chained queries would be to find plans for each input query separately using the algorithm of Section 3.2.1. This approach has several obvious problems:

1. One problem is with efficiency, especially when queries have common parts, as in this case we may end up doing some of the work repeatedly.
2. If we consider queries in isolation, we can miss some structures that are suboptimal for single queries, but are beneficial for groups of queries. For instance, if one query has an arithmetic comparison $(AC) 0 \leq B \leq 3$, where B is an attribute name, and another has $1 \leq B \leq 4$, then materializing a view with AC “ $0 \leq B \leq 4$ ” may be a competitive option compared to materializing one view for each of the original comparisons.

3. Finally, the pruning rule for the single-query algorithm might remove plans that are needed for an optimal solution. Example 9 describes such a situation.

Example 9. Consider two queries $Q_1:-ABC$ and $Q_2:-BCD$. We omit attributes, because they are not relevant here. Suppose, in the node corresponding to subgoals ABC , we have two plans $p_1:-V_1 \bowtie C$ and $p_2:-A \bowtie V_2$, where $V_1:-AB$ and $V_2:-BC$; and in node BCD we have $p_3:-V_2 \bowtie D$ and $p_4:-B \bowtie V_3$, where $V_3:-CD$. Let

$$\begin{aligned} cost(p_1) &< cost(p_2) \\ cost(p_4) &< cost(p_3). \end{aligned}$$

and

$$\begin{aligned} weight(V_1) &< weight(V_2) \\ weight(V_3) &< weight(V_2). \end{aligned}$$

Following the single-query pruning rule, we would eliminate both p_2 and p_3 , because they are less efficient than p_1 and p_4 , respectively. If we consider the combination of plans (p_2, p_3) versus plans (p_1, p_4) , then (p_1, p_4) has lower cost, but the relative weights of the plan pairs are unknown:

$$\begin{aligned} cost(p_1) + cost(p_4) &< cost(p_2) + cost(p_3) \\ weight(V_1) + weight(V_3) &? weight(V_2) \end{aligned}$$

If $weight(V_1) + weight(V_3) > weight(V_2)$, then the combination (p_2, p_3) becomes a viable option.

Thus, the single-query pruning rule is not valid for multiple queries.

In what follows we discuss the basic framework of a multi-query algorithm and two pruning rules for it, one that guarantees the presence of an optimal solution for each query, the other trading off optimality for efficiency.

Chained Queries without ACs. For chained queries without arithmetic comparisons (ACs), our algorithm uses the same DP structure as the single-query case, with one important difference: some subchains of the combined chain are not subqueries of any query and do not require plans to be created for them.

Consider an illustration. Suppose we have two queries $Q_1 : -ABCD$ and $Q_2 : -CDE$. (This format of defining the queries enumerates just the predicate names

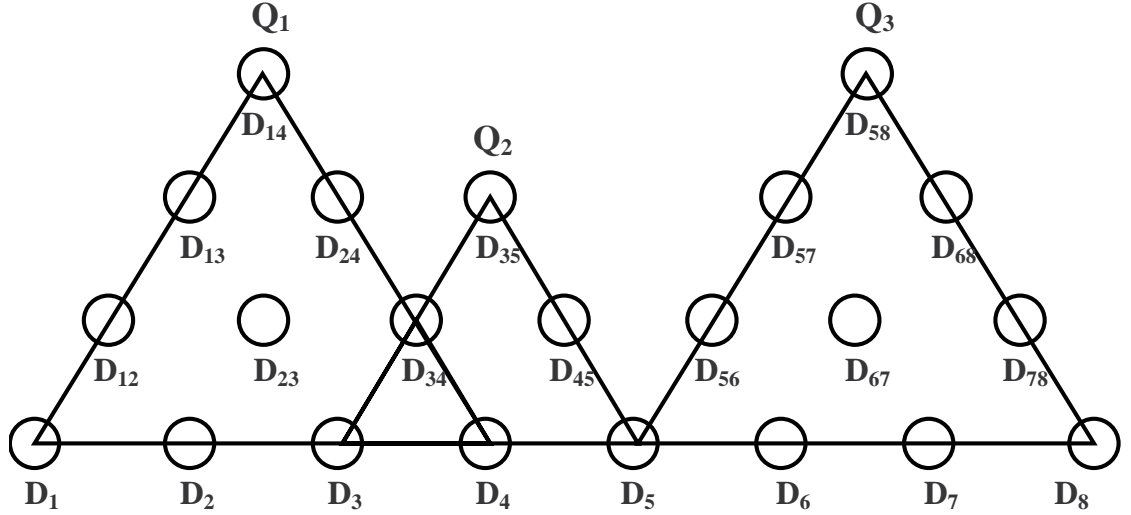


Figure 3.8: Example of DP lattice for multiple chained queries. (D_{ij} , with $i < j$, denotes a chain of subgoals $i, i + 1, \dots, j - 1, j$).

of all the relational subgoals of the queries.) Then the combined chain is $ABCDE$. Subchain $BCDE$ is not a part of either query, nor is $ABCDE$ – we do not need plans for either of them. The DP lattice for the case of multiple queries resembles a collection of mountain peaks as illustrated in Figure 3.8, which shows the lattice for three queries on a combined chain of length 8. In this diagram, circles represent subchains for which we need to construct plans. Examples of subchains for which we do not need to construct plans are D_{46} , D_{25} , and D_{36} . (Notation D_{ij} , with $i < j$, refers to a chain of subgoals $i, i + 1, \dots, j - 1, j$.)

Our pruning rule for the case of a single query may not be correct for the case of multiple queries. Suppose that in the multi-query case, we prune p_1 because its cost is higher than that of p_2 , and p_1 uses at least as much space as p_2 . In the multi-query situation, if we replace all occurrences of p_1 with p_2 then the overall cost of the solution will decrease but the total size of views used by the plan may actually increase, as the views used by p_1 might be useful for evaluating other queries. Thus, removal of p_1 might actually eliminate feasible plans that take advantage of the space

savings when views are used by plans for more than one query.

To avoid this problem, we propose the following definitions.

Definition 4. *For a given set of queries Q , we say that a view v is **exclusive** if it can be used by exactly one query in Q .*

Definition 5. *The **exclusive weight** of plan p , $ew(p)$, is the total size of the exclusive views of p .*

Definition 6. *Let p_1 and p_2 be two plans for the same subquery. If p_1 and p_2 return the same tuples in exactly the same order, such that $cost(p_2) \leq cost(p_1)$ and $weight(p_2) \leq ew(p_1)$ each hold, with at least one strict inequality, then plan p_2 globally dominates plan p_1 .*

In our algorithm **MULTIQUERYPLANGEN** (Algorithm 5), procedure **PRUNEPLANS** removes from the list of plans for a subquery all plans p such that p is globally dominated by another plan p' for the same subquery. In order to prune plans that are globally dominated, we keep track of both the exclusive weight and the total weight — sum of sizes of all used views — of each plan. Then we can execute **PRUNEPLANS** either by comparing each pair of plans or, more efficiently, by first sorting the plans by cost or by maintaining a search tree.

Adding arithmetic comparisons. We now discuss what happens when we allow selection conditions in the **WHERE** clause of the input queries. For ease of exposition, we assume that all the selection conditions are range (i.e., inequality) arithmetic comparisons (ACs), although, as we explain later, most of the techniques that we discuss here apply to other types of selection conditions.

In presence of ACs, our algorithm needs several adjustments. First, each node of the lattice implied by our algorithm **SINGLEQUERYPLANGEN** (Algorithm 1) corresponds to a subset of tables (relational subgoals of a query) and contains plans for the subqueries based on these tables. But when the problem input has multiple queries with different selection conditions, two plans built on the same set of tables might differ with respect to their selection conditions and not be usable for the same set

of queries. As a result, the same node might contain plans for different subqueries. Therefore, for each plan, we need to keep a list of queries that can use this plan.

Second, we must take care of so-called *merged* views – views that are usable by more than one query. If we have two queries that use the same subset of tables but different sets of ACs, then it may benefit both to create a merged view whose set of ACs is the disjunction (i.e., OR) of the ACs of the queries.

In other words, we merge the tuples of the individual views with the hope that the size of the result is smaller than the sum of the sizes of the individual views. Using such view is less efficient for each of the participating queries in isolation, but can prove to be beneficial in the situation with the limited available disk space.

It is easy to see that for the case where three queries overlap on the same set of subgoals, we may need to create one merged view for each pair of the queries, and one merged view for all three queries. Although in theory this means that the number of merged views that we need to create is exponential in the number of queries, in practice this number is much lower. Examples that follow illustrate that the actual number of merged views may not be that large in practice.

Suppose we have $n > 2$ queries that overlap on the same larger chain (set of tables). Suppose query Q_1 has ACs on attributes B_1 and B_2 , query Q_2 has ACs on B_2 and B_3 , etc. We can assume that the attributes that do not have ACs on them have ACs that match their whole domains. Then, when we merge views for Q_i and Q_j , such that $|i - j| > 1$, for any attribute a_k , the disjunction of constraints from Q_i and Q_j is a constraint that covers the whole domain of a_k , which means there is no AC on a_k . The same is true for any subset of the queries containing more than two queries. Therefore, in this case, we have only $n - 1$ possible merged views.

For another possible scenario, suppose that one query has AC ($B > 100$) on attribute B , and another query has AC ($B < 200$). In this case, the merged view for these two queries will have attribute B unbounded.

Our approach to generating efficient query plans for the CQAC version of problem ADR is encoded in algorithm `MULTIQUERYPLANGEN` (Algorithm 5). The algorithm uses two auxiliary structures: $plans(q)$ is the list of plans for subquery q ; $queries(p)$ is the list of IDs of the queries that can use (partial) plan p .

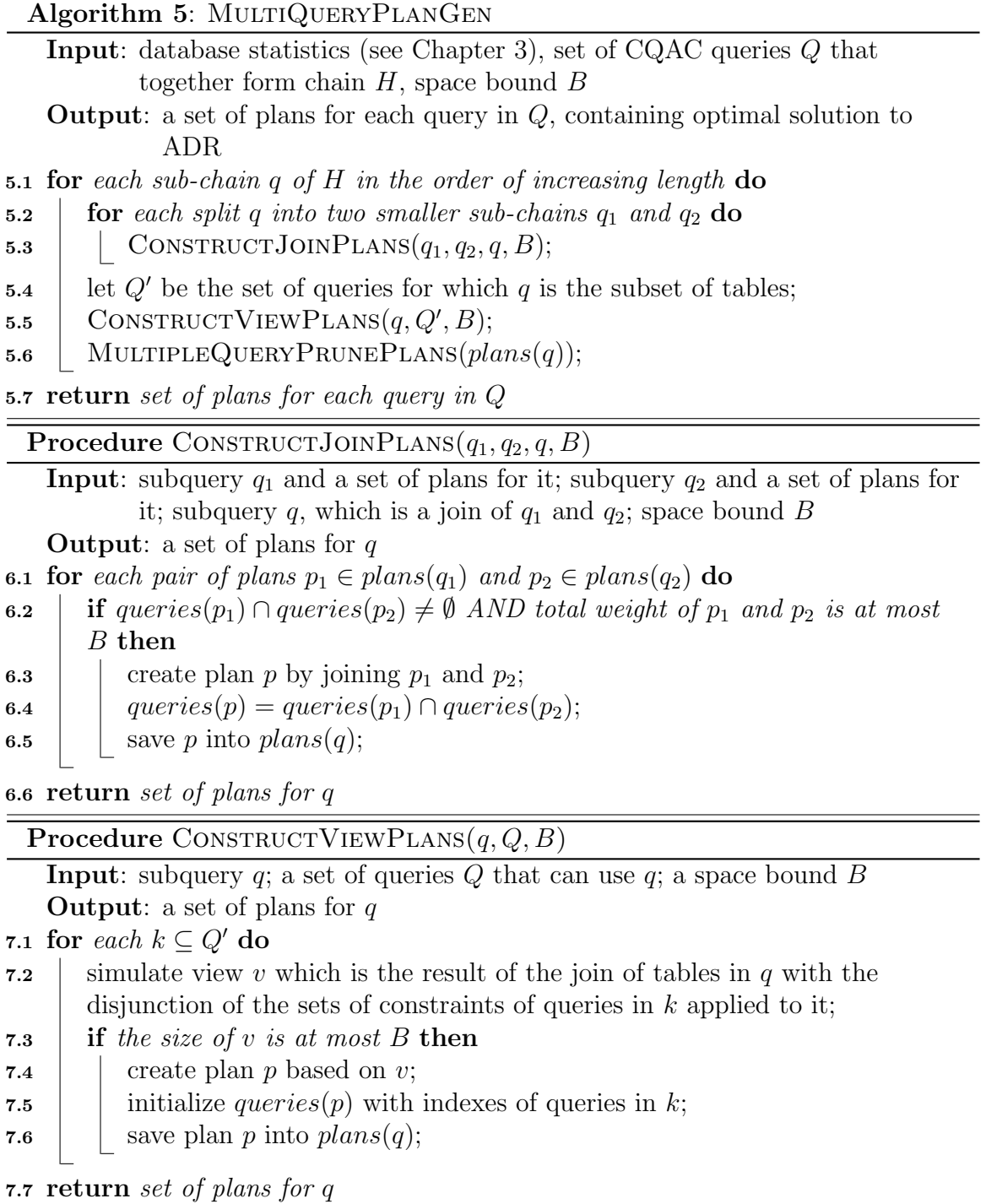


Figure 3.9: Constructing (view-based) evaluation plans for multiple CQAC queries.

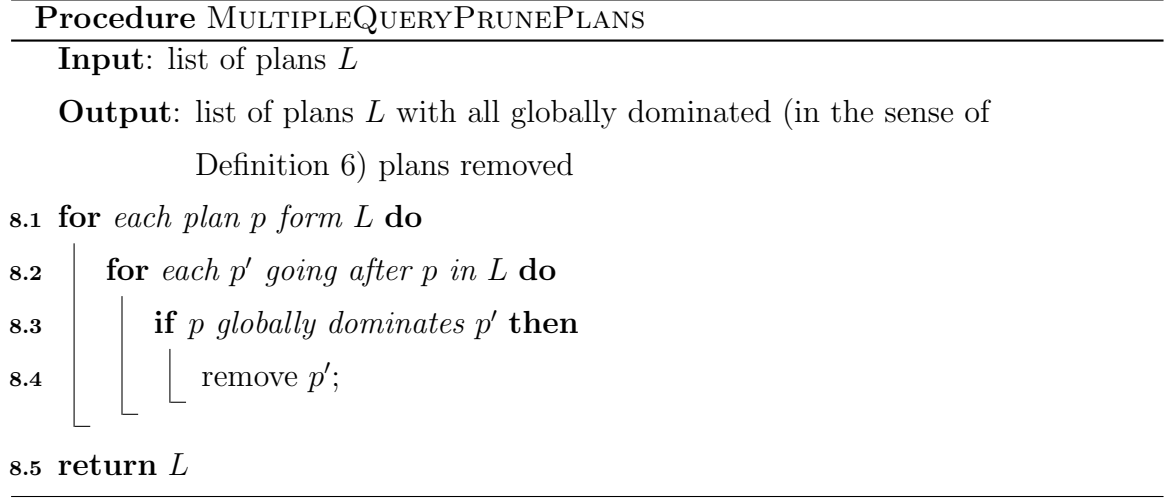


Figure 3.10: Removing globally dominated (in the sense of Definition 6) plans from the list of plans of a given subquery.

If m queries overlap on a subchain of length n , the total number of view-based rewritings for this subchain is proportional to $O(2^{nm})$. The intuition behind this is the same as for the single-query case, but this time, for a subchain common to m queries, in the worst case, we can create 2^m different views. One more difference, if compared to SINGLEQUERYPLANGEN, is that now, in line 7.1-7.6, we create 2^m plans that are based on a single view, not just one as we did in the single-query case. Thus, the overall complexity of Algorithm 5 is $O(n^3 2^{nm})$, where n is the length of the global chain and m is the number of input queries.

The implementation of MULTIPLEQUERYPRUNEPLANS is shown in Figure 3.10. Unlike the single-query case, in the multi-query case, if a plan does not dominate its neighbor with higher cost, it still may dominate a plan with even higher cost, due to the undefined relationship between weights and exclusive weights. Thus, in the worst case, for each new plan, we need to check the global domination rule against all other plans, and there is no need to sort plans of the subquery. Therefore, the worst case complexity of the pruning procedure for a given subchain is $O(s^2)$, or $O(2^{2nm})$, where n is the length of the subchain and m is the number of queries.

The added complexity of MULTIPLEQUERYPRUNEPLANS is $O(n^2 2^{2nm})$. For real databases instances, the number of input queries (parameter m) is likely to be very

big, while the number of base tables (n) is relatively small and is not likely to grow at all. Thus, we can see that MULTIPLEQUERYPRUNEPLANS actually increases the overall worst case complexity of MULTIQUERYPLANGEN, but as our experiments showed, the based on the definition of global domination pruning rule reduces the runtime of MULTIQUERYPLANGEN.

Theorem 2. *Algorithm MULTIQUERYPLANGEN returns a set of view-based plans P such that there exists $S \subseteq P$ where S is an optimal set of plans.*

Using the same technique as in the proof (by induction) of Theorem 1, we can prove that without function PRUNEPLANS the algorithm constructs all possible plans that use exclusive or merged views. The only thing that remains to be proved is that our pruning function retains an optimal solution.

Suppose an optimal solution S is not among the plans returned by Algorithm MULTIQUERYPLANGEN. This means there exists plan p for query q , where p is in some optimal solution and not in the output of the algorithm. The absence of p means that, for some subplan p_s of p , there is another subplan p'_s that globally dominates p_s . Note that, by the definition of global domination, we can replace p_s by p'_s in p without increase in the cost of p . At the same time, when we replace p_s by p'_s , we reduce the total weight of the solution by at least the exclusive weight of p_s and increase it by at most the total weight of p'_s . Thus, by the definition the global domination, the total weight of the solution does not increase. Therefore, replacing p_s by p'_s in p , we obtain a solution which is at least as good as S ; and, if we find in solution S all such dominated subplans and replace them with their dominating alternatives, we get a solution which is at least as good as S and is among the plans return by Algorithm MULTIQUERYPLANGEN.

■

Theorem 2 is a very important result. It means that Algorithm MULTIQUERY-PLANGEN performs a systematic investigation of the search space of view-based plans and returns a (reduced-size) list of plans that contains an optimal solution. Thus, the solution quality of the two-stage architecture that we presented in Section 3.1

depends only on the quality guarantees of the algorithm used in stage two of the architecture. Combined with Propositions 1 and 2, this result provides strong optimality guarantees for our overall architecture when applied to the CQAC class of problem inputs considered in this section.

3.2.3 More aggressive pruning.

The strong point of Algorithm MULTIQUERYPLANGEN is that it preserves optimality. Unfortunately, its runtime grows exponentially in the number of queries, as one might expect from an algorithm that solves an NP-hard problem. We now describe a few more aggressive pruning rules that remove many more plans at the expense of losing the optimality guarantee. These rules suggest a *family* of algorithms for processing CQAC queries at stage one of our architecture, representing points along the time/quality continuum.

Using single-query domination rule for the multiple-query case.

In Example 9, we demonstrate that the pruning rule that we used in our single-query algorithm (see Section 3.2.1) does not guarantee optimality if used for the multiple-query case, as it does not account for the views that are shared by multiple queries. At the same time, our experiments (Section 3.3) suggest that the single-query rule, even if applied to the case of *multiple queries*, does not significantly reduce the quality of the solution. We ran an experiment on small instances, for which we could find an optimal solution, to check the solution quality of Algorithm 5 with the single-query pruning rule. The results showed that only in 10% cases our algorithm found a suboptimal solution, and the maximum error was 5.5%.

Limiting the number of plans.

Although the “single-query” aggressive rule of the previous paragraph prunes many more plans than the conservative one, it is still instance dependent and does not guarantee convergence within the allocated time. In other words, we do not know a-priori how many plans the aggressive rule will prune for a given instance, and thus how fast the algorithm will find a solution. We might get really unlucky, in that the aggressive pruning might not prune any plans at all. In the worst case, we can

still get exponential runtime complexity in the number of queries, tables, selection conditions, and attributes for each subproblem. To counter this problem, the idea of our second aggressive pruning rule is to limit the number of plans we keep for each subproblem: we keep only k plans with the largest $\text{profit} * \text{queries} / \text{size}$, where profit is the decrease in cost offered by the plan (over use of base tables), queries is the number of queries that can use the plan, and size is the total size of the views used by the plan.

From the point of view of the worst case complexity, if we limit the number of plans that we store for each subquery to k , then, for a given subquery of length n , the number of join-based plans is proportional to $O(nk^2)$ (see complexity analysis for SINGLEQUERYPLANGEN and MULTIQUERYPLANGEN), while the number of single-view-based plans remains the same $O(2^m)$. This gives us a total of $O(nk^2 + 2^m)$ plans per subquery of length n . After we create all plans for the given subquery, we need to pick k best, according to our heuristic, plans. This can be done in linear in the number of plans time, using the Median of Medians algorithm proposed by Blum et al. in [BFP⁺72].

We need to perform this set of operations for each subquery. Thus, the overall worst case complexity is $O(n^2(nk^2 + 2^m))$.

Heuristic view-selection.

Algorithm MULTIQUERYPLANGEN, for each subquery, generates plans by two means: (1) joining plans for smaller subqueries; (2) creating plans that contain exactly one view that covers the whole subquery. The total number of views that can be beneficial for a subquery, and thus, the total number of plans created through (2), is exponential in the number of queries. Therefore, the number of plans created by (1) grows exponentially as well.

We already proposed a family of algorithms that use plan pruning techniques and limit the number of plans for each subquery. These methods limit the number of plans created through joining to be polynomial in the length of the subquery.

Although we limit the number of plans for each subproblem, we still investigate the whole space of views. How can we limit the number of views we consider? Suppose, we have m queries overlapping on the same subchain. Then, in the worst case, we

try to build 2^m views on this subchain. What we want to do is to avoid building some of them. The most time-consuming part is the construction of (i.e., statistics estimation for) the view itself. When it is built, we can think of it as of a plan, and apply the pruning rules that we use for plans. To get improvement here, we must avoid building some views without knowing what these views are. Thus, we need an heuristic solution.

A problem that we are trying to solve is given a subchain L and a set of queries Q that have L as a subquery, create at most r “best” views for these queries. Word “best” does not necessarily mean views with the biggest benefit to the workload. Ideally, we want to choose a set of views that contains views for all optimal solutions. Note, not just one optimal solution, because for two different subchains we might pick views for different optimal solutions, what will lead to a suboptimal solution. For simplicity of the notation, we assume that all queries in Q are based on tables of L , and differ only in their selection conditions.

To guide our search we use the function that measures the benefit of using view v to answer queries in Q

$$Benefit(v, Q) = \sum_{q \in Q} b_v(q), \quad (3.1)$$

where $b_v(q)$ is a benefit of answering query q using v instead of calculating the answer from the base tables. If view v is not applicable to some query $q_i \in Q$, then $b_v(q_i) = 0$.

Let $cost(q, v)$ be the cost of answering (sub)query q using v . Also, let $cost(q) = cost(q, \emptyset)$ be the cost of answering q using only base tables (or, views that are already in the database, if we start with some views already materialized).

Then,

$$b_v(q) = cost(q) - cost(q, v). \quad (3.2)$$

Putting it back into (3.1), we get

$$Benefit(v, Q) = \sum_{q \in Q_v} (cost(q) - cost(q, v)) \quad (3.3)$$

Algorithm 9: Heuristic view-selection.

Input: set of queries Q ; $\forall q \in Q, ans(q)$; constant r

Output: a set of views of size r

```

9.1 initialize empty priority queue  $K$  of views ordered by decreasing  $Benefit$ ;
9.2 initialize empty list of views  $S$ ;
9.3 create an empty view  $v$  and insert it into  $K$ ;
9.4 while total size of  $S$  and  $K$  is less than  $r$  do
9.5     pop next view  $v$  from  $K$ ;
9.6     for each  $q \in Q$  do
9.7         create view  $v'$  by merging  $v$  and exclusive view of  $q$ ;
9.8         insert  $v'$  into  $K$ ;
9.9     insert  $v$  into  $S$ ;
9.10 return views from  $K$  and  $S$ 

```

Figure 3.11: Constructing merged views for a given subquery.

where Q_v is the set of queries that can be answered using v .

Note that $cost(q, v)$ — cost of answering query q using covering view v — is the cost of a scan of v to retrieve all records required by q . In reality, the cost depends on the indexes available for v . For instance, if we have a $B+$ -tree index, then the cost is linear in the number of returned records, while if we do not have any index, then we need to perform a sequential scan of v and the cost is linear in the number of tuples of v . In this work, we assume that we do not have any indexes and we use sequential scans to access base tables and views. Thus, $\forall q \in Q : cost(q, v) = \alpha|v|$, where $|v|$ is the size of v . In our experiments, we assume that $\alpha = 1$, therefore $cost(q, v) = |v|$.

We can rewrite (3.3) as

$$Benefit(v, Q) = \sum_{q \in Q_v} cost(q) - n|v|, \quad (3.4)$$

where n is the number of queries in Q_v .

Note, to maximize the benefit, we need to maximize the number of queries that can use v and minimize the size of v , which is not surprising.

In our algorithm, we use a best-first search strategy. On the first cycle of the **While**

loop, we create exclusive views — views that can be used by individual queries. Then, on each next iteration of the **While** loop, we take the view with the highest *Benefit* value and augment it in all possible ways to be useful for one more query. We stop, when we reach the limit, and return the created views.

With a simple tree implementation of the priority queue, we get $O(\log(r + m))$ cost of insertion, and $O(1)$ cost of extraction of the element with the highest *Benefit*. Algorithm 9 creates at most $r + m$ views, where r is the user defined bound and m is the number of input queries. Thus, the time spent on one run of the heuristic view selection is proportional to $O((r + m) \log(r + m))$. We execute the view-selection heuristic for each subchain, therefore, the overall added complexity of this algorithm is $O(n^2(r + m) \log(r + m))$.

If we use the heuristic view-selection in conjunction with the limited number of plans per subquery, then, for each subchain we spend $O(nk^2) + O((r + m) \log(r + m))$ time building (join- and view-based) plans. Then, we pick k best plans in the linear in the number of plans time, $O(nk^2 + (r + m))$. We need to build plans for each subchains, thus, the overall worst case complexity of the algorithm with the limited number of plans and the view-selection heuristic is $O(n^2(nk^2 + (r + m) \log(r + m)))$.

From our experiments, we found that setting r to be equal to $2m$ results in a good trade-off between scalability of the algorithm and quality of the returned solutions. Thus, assuming that r is $O(m)$, we get $O(n^2(nk^2 + m \log m))$.

3.3 Experimental Results

The experiments reported in this section address two questions: (a) How does our two-stage approach compete with the state-of-the-art *relaxation-based approach (RBA)* of [BC05]? (b) Can we safely assume that stage two is not the bottleneck when using CPLEX [ILO04]? Our extensive and thorough investigation gives a positive response to each question.

We begin in Section 3.3.1 with a discussion of how we generate random problem instances with characteristics typical of those arising in practice. In Section 3.3.2 comparisons with the RBA are reported for a large number of instances. Scalability

of the stage two computation using CPLEX is demonstrated in Section 3.3.3.

All of our experiments were done on an AMD Athlon(tm) 64 X2 Dual Core Processor 5200+, with 2 MB of L2 cache, and 4 GB memory, running Red Hat Enterprise Linux 5. The programs were implemented in C++ and compiled using gcc version 4.1.2. We called CPLEX version 11.0 directly from our software.

3.3.1 Instance Generation

Query instances derived from benchmarks such as TPC-H [TH] tend to be small, making it difficult to test scalability, and that algorithms that work well for them may not perform as well in general. Random instances, on the other hand, have to be generated carefully to reflect what occurs in practice. We now give a detailed description of how the problem instances for stage one were generated. We do experiments on both types of instances. This section discusses the details of random instance generation.

Relationships among queries were determined by three *structural parameters*: M = the length (number of tables) of the global chain, N = the number of queries, and L = the maximum length of the subchain for any query. The choice of the values of the parameters has an impact on the degree to which queries overlap: if L and/or N is large w.r.t. M , there is likely to be much overlap.

For each of the N queries we generate a random integer r in $[1, L]$; this r is the length of the subchain for that query. The position of the query in the global chain is determined by choosing a random integer s in $[1, M - r]$ to be the first table of the query. We used fixed values of M and L for increasing the number of queries. For $N = 22$, the choice $L = 8$ reflects the nature of the TPC-H workload. Our choice of $M = 20$ is somewhat larger than the longest global chain in TPC-H, but is typical of databases with a large number of attributes.

We also used a collection of *numerical parameters* to determine the sizes of base tables and related quantities that affect cost and space usage.

- W_{min} and W_{max} are the minimum and maximum number of tuples, respectively,

in a base table; we call the number of tuples the *size* from here on³; these directly affect both space of a view and cost of a plan; W_{max} is also the upper limit on the number of values an attribute can take, its *value counter* (*v.c.*); We used $W_{min} = 5$, $W_{max} = 100$ to avoid arithmetic overflows when computing sizes/costs with longer chains.

- B is the space bound – limit on the amount of available disk space in addition to that used by the base tables; our experimental results depend on the choice of B : a problem instance is trivial if B is the total size of all query results (just create a view for each query) and becomes harder as B decreases.

The global chain of M tables, numbered 1 to M , contains $M + 1$ attributes, numbered 0 to M . Attributes 1 to $M - 1$ are *join attributes*; a join involving tables i and $i + 1$ uses join attribute i . First the v.c. of each attribute and the size of each table is chosen to be a random integer in $[W_{min}, W_{max}]$.

Further, for each table, with probability $P = 0.8$ we decide whether one of its two attributes is a key attribute, and, if so, pick one. In practice joins are frequently done on key attributes – the value chosen here is based on the TPC-H workload. A potential conflict between table size and key attribute v.c. may arise at this point. When this is the case the table size is set to be the v.c. of the key attribute rather than the previous randomly chosen value.

Since the set of values taken on by an attribute, its *global domain*, is independent of that taken on by other attributes, we can safely assume that all global domains are integer intervals $[0, w]$, where w is the v.c. of the attribute.

When an attribute is not a key attribute, a table does not necessarily contain all of its values. For a given table i and join attribute $j = i - 1$ or i , the *local domain* i, j is the set of values in table i for attribute j . The v.c. of each local domain i, j is chosen to be a random integer that is no larger than either the size of table i or the v.c. of attribute j . Each local domain is a sub-interval of the corresponding global domain, also chosen randomly depending on the v.c.

³Size usually means number of bytes, but, in order to avoid introducing another variable to the instance generation, we assume all tuples have the same number of bytes.

The last set of random choices is that of *constraints* on the attributes. In practice, some attributes, such as invoice date, are more likely to be subject to constraints, i.e., are more important than others, such as customer id. Hence we choose an *importance* for each attribute, a random real number I from $[0, 1]$.

A query q on tables i to j involves attributes $i - 1$ to j . For each of these attributes we must decide whether the attribute has a constraint *for that query*; this is true with probability I , the importance of the attribute. If there is a constraint, it is chosen to be a random sub-interval of the attribute’s global domain. Note that both the decision about whether to include a constraint and the actual interval are independent for each query.

3.3.2 Comparisons with RBA

The RBA algorithm we use for comparison purposes is the one described in [BC05]. It consists of two main stages: (a) choose a set of physical structures (i.e., views or indexes) that is guaranteed to result in an optimal configuration, but may take too much space; and (b) “shrink” it using transformations, such as view and index merging, index prefixing, etc., that reduce the total required space.

In the experiments comparing our two-stage approach with the RBA of [BC05] we examined the quality achieved by each approach within a specified time period. Our own implementation of [BC05] was used since we did not have access to the code for it.⁴ Any comparison using time as a measure may therefore not be representative of the actual relative performance of the two approaches. Thus we use a combinatorial measure, as follows.

In the physical database design literature it is a common practice to use the total number of query-optimization calls as a measure of time spent on optimization. We use a more fine-grained unit of time — size estimation. For each operator in a plan tree, the optimizer estimates the size of the results, the execution cost, and the order of tuples of the result. Thus, any query-optimization call consists of a series of size and

⁴Microsoft Research is currently unable to distribute the research prototype externally due to IP considerations.

statistics approximation calls. Although size estimations are not the only operations performed by the optimizer, they tend to dominate the runtime of query optimization.

We ran the RBA algorithm on each instance until it had done as many size estimations as our MULTIQUERYPLANGEN (Section 3.2) using the single-query pruning rule; we refer to this algorithm as *AR*, aggressive (pruning) rule. We then compared the solution quality of the two algorithms.⁵ Fig. 3.12(a) shows that in almost all cases our algorithm achieves higher quality solutions.

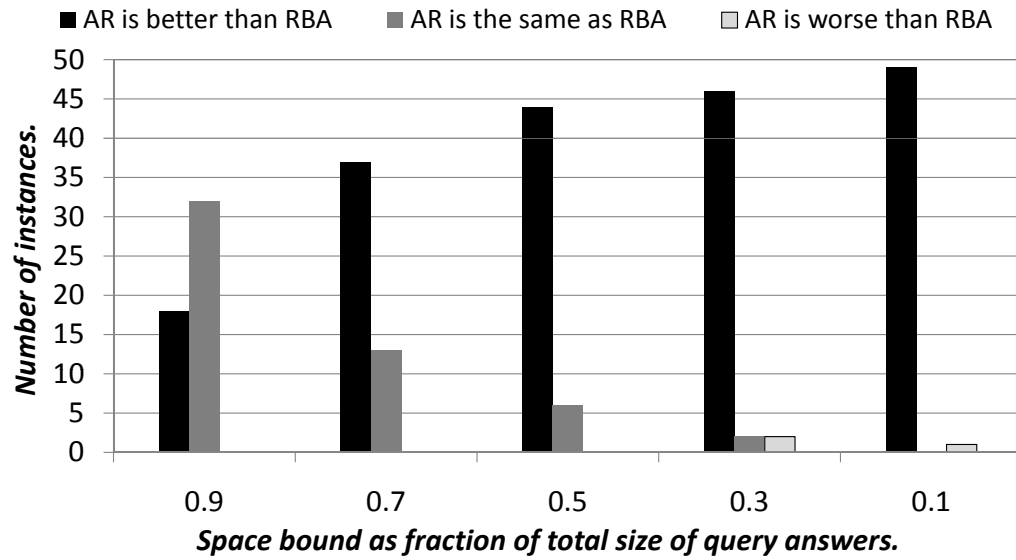
In a few cases RBA failed to find a feasible solution at all. To make sure that the cutoff did not preclude solutions that were *almost* encountered by RBA, we also compared the performance with RBA allowed four times as many size estimations as our algorithm, see Fig. 3.12(b). Note that the RBA's performance did not improve much vs. AR even when allowed four times as many size estimations. In fact in only three (out of 250) cases, RBA was able to improve its solution in comparison to AR.

In both experiments the results differ depending on the relationship between the space bound and the total size of the query results. What both parts of the figure show is that our approach gets better relative to RBA as the problem gets harder, i.e., less space is available. In fact the difference becomes dramatic with only a slight increase in difficulty.

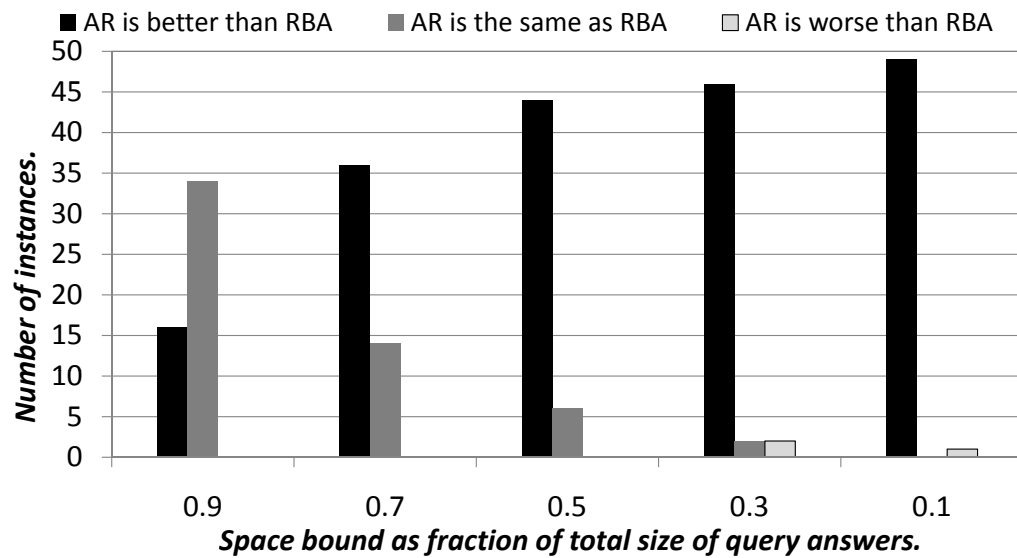
Recall that the runtime of AR is exponential in the worst case. Even in practice its runtime increases dramatically with increasing query length. To mitigate this, we experimented with a version of AR that sets a limit on the maximum number of plans kept for each subquery. The choice of plans is made heuristically using the k plans with largest $profit * queries / size$, where *profit* is the decrease in cost offered by the plan (over use of base tables), *queries* is the number of queries that can use the plan, and *size* is the total size of the views in the plan.

Fig. 3.13 shows how the runtime increases with query workload for $k = 10$ and 20, i.e., AR10 and AR20, versus the original AR. Here, the actual runtime can be used, because we compare three variations of the same algorithm. The much slower growth for AR10 and AR20 is evident. We still need to demonstrate that these algorithms

⁵Stage two using CPLEX is also executed following our MULTIQUERYPLANGEN of Section 3.2. This part took only a small fraction of the total time.



(a) RBA is allowed the same number of size estimations.



(b) RBA is allowed four times as many size estimations.

Figure 3.12: Comparing solution quality of our AR (Section 3.2) versus the RBA of [BC05].

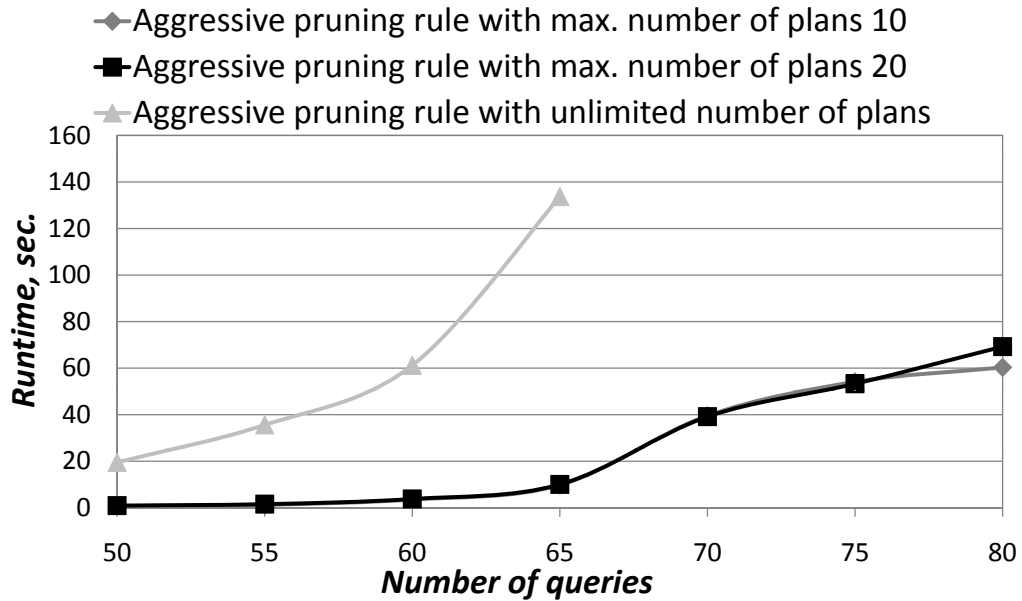


Figure 3.13: Scalability of aggressive pruning rules.

are competitive when it comes to solution quality.

Fig. 3.14 shows the performance of AR20 vs. RBA using the same setting as that of Fig. 3.12. The superiority of AR20 w.r.t. solution quality is not as dramatic as with AR, but still clear. Whereas AR wins practically all the time with the space bound less than 100% of the total size of query answers, AR20 catches up gradually and attains superiority at 50%. After that, the relative results do not change significantly.

Another way to evaluate AR20 is how its performance vis a vis RBA scales with the number of queries. Since AR20 makes significantly fewer size-estimate calls than AR, we end up allowing fewer size estimates for RBA. When the space bound is 50% of the total size of query answers – Fig. 3.15 – the results are mixed; AR20’s advantage in “speed” is offset by poorer relative solution quality. However, when the space bound is 10% of the total size of query answers, the instances are much harder for RBA, while AR20 is still able to come up with significantly better solutions.



Figure 3.14: AR20 versus RBA for different space bounds.

3.3.3 Scalability Results

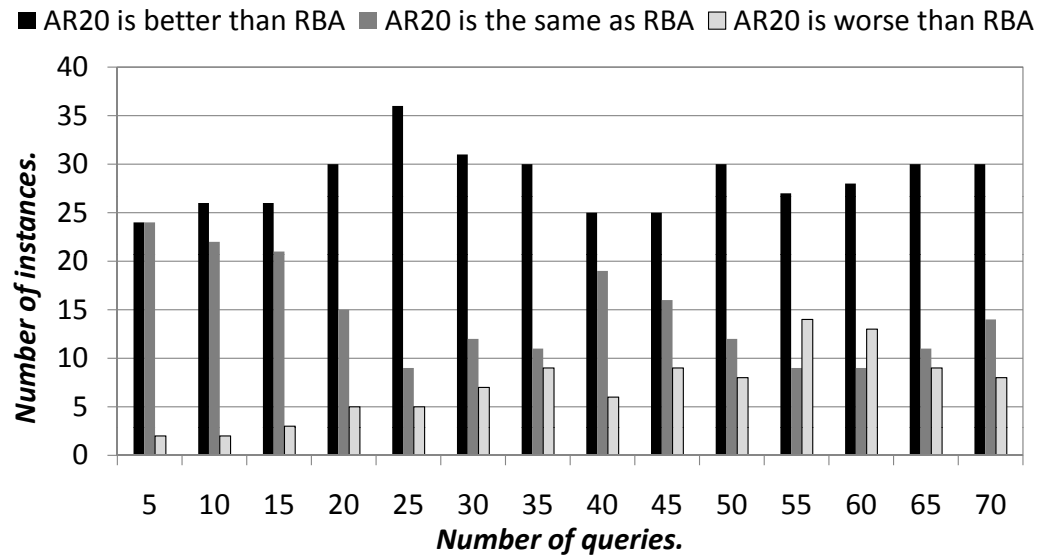
One concern with our two-stage architecture is that the second stage, which solves a generic integer programming problem instead of using problem-specific techniques, might incur prohibitive runtime. Our experiments suggest otherwise.

Fig. 3.16 shows the runtime of CPLEX when it solves problem instances arising as outputs from stage one. As we were unable to generate these outputs directly when the number of queries is large, we generated them randomly using the techniques described in [KCFS07], being careful to set parameters so that the instances had characteristics similar to the stage one outputs of our smaller instances. Instances with a large number of queries can be solved in a few seconds.⁶ The largest instance that we were able to solve within 10 minutes using CPLEX had 800 queries, 1074 plans per query, and 6886 views.

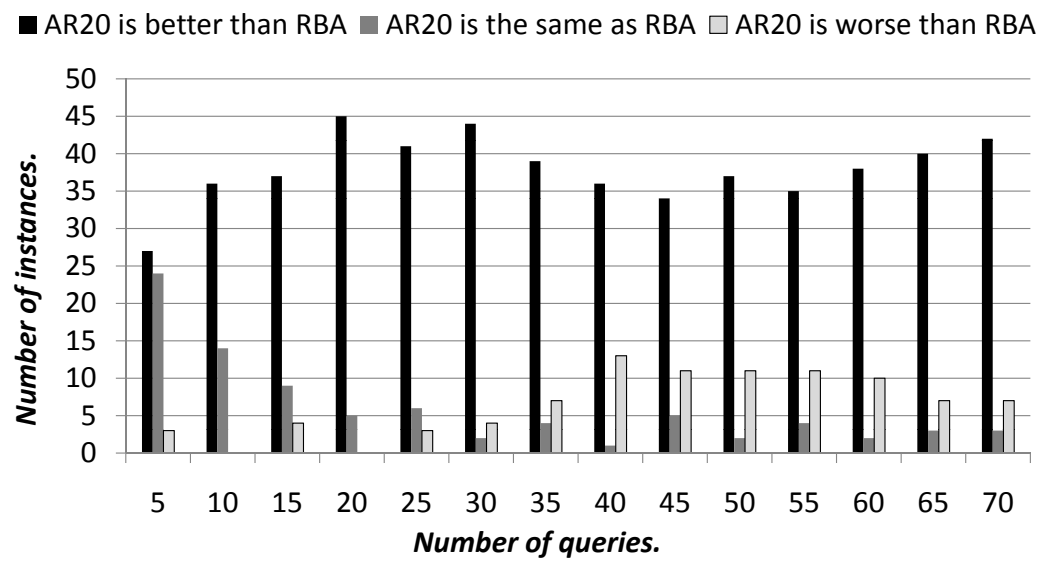
3.3.4 Comparison on TPC-H workload

In this section we present results of the comparison of AR and RBA on TPC-H-shaped instances. As we said earlier, our experimental framework can deal only with

⁶The jag in the curve appears to be related to the fact that CPLEX 11.0 employs a more aggressive preprocessor when the problem size reaches a particular threshold.



(a) Space bound is 0.5 times total size of query answers.



(b) Space bound is 0.1 times total size of query answers.

Figure 3.15: AR20 versus RBA as a function of number of queries.

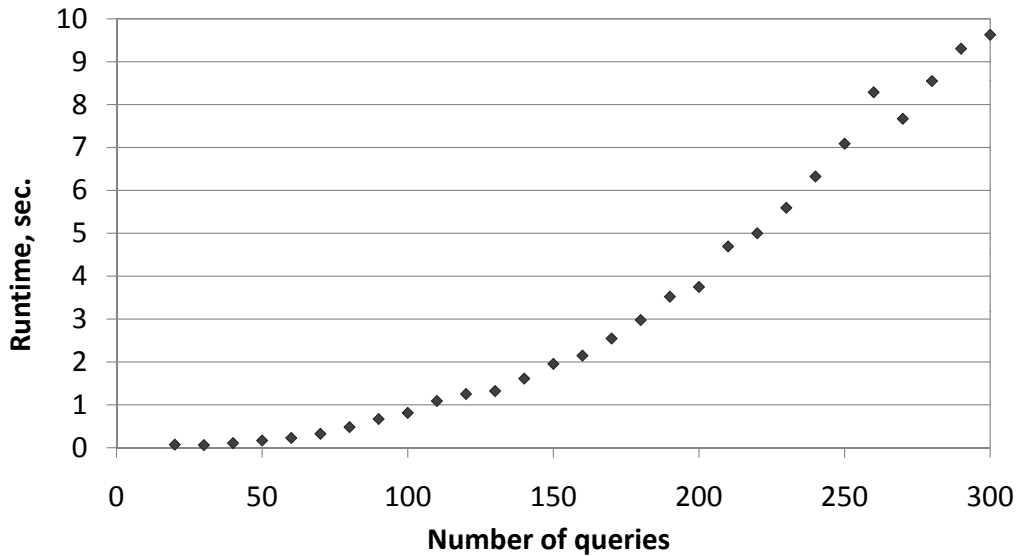


Figure 3.16: Scalability of CPLEX when used to implement stage two.

singe-block chain queries with range constraints. Thus, we preprocessed all queries of the TPC-H workload to match them with our template. One interesting fact is that almost all queries of TPC-H are chain queries, and they can be combined into a global chain consisting of 9 table:

region—*nation*—*supplier*—*partsupp*—*lineitem*—*orders*—*customer*—*nation*—*region*

Note that this chain has two occurrences of tables *region* and *nation*. For our model, it is not important, because we treat them as different tables.

The TPC-H specification gives guidelines for creating queries and allows small variations in the choice of the constraints. Following these instructions, we implemented a query generator with small randomness in the choice of the constraints. Such module can create many similar TPC-H-shaped instances with small variations in constraints.

TPC-H workload consists of 22 queries. As we showed in the sections above, such instances are relatively small for our proposed algorithm. Therefore, we were able to run an experiment on 1000 TPC-H-shaped instances. The average runtime of our algorithm was 3.4 seconds. We allowed RBA to do the same number of size estimations and recorded the solution values returned by the two algorithms. As

before, the results differ depending on the relationship between the space bound and the total size of the query results. Figure 3.17 shows some statistics about the distribution of the relative difference of RBA and AR solution values for various choices of the space bound. Here, we set the space bound as a fraction of the total size of the query answers.

From the results in Figure 3.17, we can see that the cost of the solution returned by AR is on average 20% to 40% lower than that of RBA. Out of the total of 9000 runs, only in 2 cases RBA returned a better solution than that of AR, and in 25 cases RBA and AR returned the same solution.

space bound	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
min	-0.02	0.21	0.21	0.22	0.07	0.00	0.00	0.00	0.00
median	0.29	0.40	0.42	0.41	0.37	0.36	0.24	0.23	0.23
mean	0.31	0.41	0.42	0.42	0.39	0.37	0.26	0.23	0.23
max	0.84	0.75	0.65	0.60	0.61	0.60	0.52	0.55	0.53
standard deviation	0.13	0.10	0.06	0.07	0.08	0.09	0.06	0.03	0.03

Figure 3.17: Relative difference between the solution values returned by AR and RBA, $(S_{RBA} - S_{AR})/S_{RBA}$.

The goal of the second experiment was to compare AR to RBA on randomly generated instances with the parameters chosen to emulate the TPC-H settings. Preliminary results showed that such instances were much harder than those of TPC-H, therefore, instead of AR, we used the version of AR that bounds the number of plans for each subproblem. Obviously, to make the quality of the results consistent over the range of instances of different sizes, it makes sense to bind this limit to the parameters of the problem instance. From our preliminary experiments we found that setting the bound to be twice the number of queries results in a good scalability versus quality trade-off. Therefore, for the experiments on the TPC-H-shaped instances we used AR44.

We set the space bound to be 0.1 of the total size of query answers. As before, we measured the costs of the returned solutions and calculated the relative difference of these costs. The following is the summary of the results.

Out of 1000 instances, AR44 found a better solution in 883 cases and RBA found

space bound	0.1
min	-0.53
median	0.18
mean	0.18
max	0.73
standard deviation	0.16

Figure 3.18: Relative difference between the solution values returned by AR44 and RBA, $(S_{RBA} - S_{AR44})/\max(S_{RBA}, S_{AR44})$, for randomly generated instances

a better solution in 106 cases.

Although Figure 3.18 shows the general relationship between the solutions returned by the two algorithms, the actual distribution is not clear. Figure 3.19 shows a more detailed picture. The first column of the both subtables contains the relative difference between the costs of the returned solutions. The second column shows what fraction of the instances had at least this relative difference. Table 3.19(a) is based on 883 instances for which AR44 found a better solutions, Table 3.19(b) is based on 106 instances on which RBA was better.

relative difference, %	percentrank	relative difference, %	percentrank
0.06	1.0	0.04	1.0
4.33	0.9	0.70	0.9
8.88	0.8	1.51	0.8
12.67	0.7	2.12	0.7
16.20	0.6	4.04	0.6
20.09	0.5	6.39	0.5
23.69	0.4	8.53	0.4
27.99	0.3	10.21	0.3
33.43	0.2	13.98	0.2
40.18	0.1	18.66	0.1
73.36	0.0	52.88	0.0

(a) AR44 is better than RBA. (b) RBA is better than AR44.

Figure 3.19: Distribution of the solution relative difference between AR44 and RBA.

From Figure 3.19, we can see that AR44 outperforms RBA on TPC-H-like randomly generated instances. For instance, in 50% instances where AR44 returned a

better solution, the relative difference between the solutions was more than 20%; while for the instances where RBA returned a better solution, only 10% of the instances had relative difference of 18.7% or more.

3.3.5 Summary

In the experiments reported here we showed that our stage one algorithm with an aggressive pruning rule yielded better solution quality than an RBA approach most of the time. This continued to hold even for a faster version of our algorithm that pruned more plans at every step. We also demonstrated that the computational effort in stage two is relatively insignificant and scales well, allowing very large instances to be solved. We are currently considering even more aggressive pruning rules, to bring down the runtime of stage one while still yielding high-quality solutions. In addition, we are investigating stage one algorithms for other database schemas, for future implementation and experiments.

3.4 Discussion and Extensions

In this section we have considered the problem of selecting views or indexes that minimize the evaluation costs of the frequent and important queries under a given upper bound on the disk space available for storing the views or indexes selected to be materialized. (In the remainder of this section we will refer to views and indexes collectively as *tasks*.) To solve the problem, we proposed a novel end-to-end approach that focuses on *systematic* exploration of possibly task-based *plans* for evaluating the input queries. Specifically, we proposed a framework (architecture, see Section 3.1) and algorithms (Section 3.2) that enable selection of those tasks that contribute to *the most efficient* plans for the input queries, subject to the space bound. We presented strong optimality guarantees on the proposed architecture. Our proposed algorithms search for sets of competitive view-based plans for queries expressed in the language of conjunctive queries with arithmetic comparisons (CQACs). This language captures the full expressive power of SQL select-project-join queries, which are common in

practical database systems. Our experimental results on synthetic and benchmark instances (see Section 3.3) corroborate the competitiveness and scalability of our approach.

We now focus on some classes of problem inputs and of allowed views and rewritings/plans that subsume the CQAC class to which our algorithms of Section 3.2 are applicable. Recall that our architecture has two stages: (1) A search for sets of competitive plans for the input queries, and (2) Selection of one efficient plan for each input query. The plans in the input to and output of the second stage are formulated as sets of IDs of those tasks whose materialization would permit evaluation of the plans in the database. Thus, *all problem-specific details, including the query languages for the input queries and for the views/rewritings of interest to efficient evaluation of the queries, are encapsulated in stage one of our architecture*. It follows that to capture more general query, view, and rewriting languages, as well as the presence of indexes, by our overall approach, it is sufficient to develop algorithms for stage one *only* of our proposed architecture.

In the remainder of this section we outline how processing some of the generalizations of the CQAC class of problem inputs (as well as of the CQAC class of views and rewritings for such problem inputs) in stage one of our architecture can be done by straightforward generalizations of the algorithms that we presented in Section 3.2.

Indexes in addition to views. It is rather straightforward to extend our approach to include indexes. For this, we can use the technique proposed in [BC05]: Each time the optimizer joins two partial plans at least one of which is a base table or a view, it determines the indexes that can make the join operation more efficient. We can intercept such requests and simulate the beneficial indexes in the same way we simulate views in (generalizations of) our approach of Section 3.2. We then create one plan for each candidate index and apply our pruning techniques.

Recall that stage two of our architecture does not distinguish between materialized views and indexes. Both are called tasks and treated the same way in that each has (i) an effect on the cost of the query that uses it, and (ii) a size required to store it. *Our architecture therefore facilitates the incorporation of indexes* and any other features

that can be treated as tasks. Incorporation of any classes of tasks, including indexes, in our stage-one algorithms is orthogonal to the query/view/rewriting languages to which the algorithms are applicable. Note that the addition of indexes enables our stage-one algorithms to consider partial/full query plans with “interesting orders” (of the tuples in the partial query answers) in the sense of [SAC⁺79], and thus to consider sort-merge joins and other types of joins that take advantage of such interesting orders.

Selection conditions other than arithmetic comparisons (ACs). In Section 3.2.2 we discussed how our algorithms deal with ACs, which generalize range selection conditions of SQL queries. The main challenge that we face when adding more general selection conditions to the picture is the problem of how to consider *systematically* the search space of all merged views. As discussed in an example in Section 3.2.2, to build a merged view for a given set of views based on the same tables, we need to take a disjunction of the selection conditions of the “parent” views. Thus, it is straightforward to extend our algorithms of Section 3.2 to any other types of selection conditions for which it is easy to build such disjunctions in the languages of choice for views or rewritings.

Queries, views, and rewritings with grouping and aggregation. Queries, views, and rewritings with grouping and aggregation can be treated in stage one of our architecture using a surprisingly minor extension of our algorithms of Section 3.2. The extension is based on the formal results in the previous work [GKC06, AC05] by some of the authors of this paper; the intuition is as follows. For a given subquery, our algorithm simulates a “covering view,” as discussed in Section 3.2. Such a view may also have grouping and aggregation in it, see the work by Afrati and Chirkova [AC05] for the local conditions for building aggregate views for subqueries of an aggregate query. Aggregations must also be accounted for when building merged views, but this does not increase the search space of plans, because for any two parent views with selection conditions and aggregations, we create only one merged view with the combined selection conditions (i.e., a disjunction of the selection conditions of the parent views) and attributes to which grouping/aggregation apply.

Going beyond chained queries. Our framework can also be applied to the

case of general-shape queries. In this case, instead of a global combined chain, we will have a more general combined graph, with possible cycles in it. Our algorithms systematically build plans for all subqueries of the global chain. Thus, to adapt our algorithms to the general case, we just need to find a systematic way of building plans for each connected subgraph of the combined graph.

The attractiveness of the chained-query case is in that the total number of connected subgraphs of a chain graph is quadratic in the length of the chain. For a graph of general shape, this number can grow exponentially. An appealing lower-complexity generalization of chains is the “comb” graph — a graph that consists of a chain with possible sub-chains (of bounded length) branching off each node of the global chain. This means that each table in the input queries can join with at most two other tables. This limits the total number of linked subgraphs, and thus the complexity of the algorithm. The only modification that we need to do is, instead of taking two-way splits of the chain, we consider three-way splits of the comb. We have developed single-query algorithms for comb graphs and simple cycles. These algorithms are based on dynamic programming using lattices and are therefore adaptable to multiple queries. We talk about this special case in more details in Section 5.1.

Yet other query types can be handled using algorithms from the literature. For instance, “star”-shaped queries and views are generally common only for OLAP databases [CD97], where one expects only aggregate queries to be asked on the stored data. When only aggregate queries are expected, then the framework of [HRU96b] of having only one view in a query plan can be tackled by methods known in the literature, rather than by the methods proposed in this work.

Chapter 4

Second Stage Plug-in

In [KCFS07], we propose an alternative to CPLEX plug-in for the second stage of our architecture. In this work, we apply problem-specific techniques to the branch-and-bound algorithm with the goal of improving its performance on the Plan-Selection Problem (PSP).

Our specific optimization problem PSP is as follows: Given a set of possible plans for each query, choose a subset of plans that provides the greatest reduction in query cost. Each plan requires the materialization of a set of views and/or indexes, and cannot be executed unless all of the required views and indexes are materialized. For practical reasons, the total size of materialized views and indexes must not exceed a given space (disk) bound.

Our problem statement for view and index selection does not require any information about the input plans other than the views or indexes that they require and the cost reduction the plans yield. Thus, our solution is not tied to any particular database model (that is, the queries and even database schemas can take any form), nor do we need to know how the indexes or views affect the query costs. These details are abstracted by the cost function, which in turn can come from whatever cost model most suits the application.

The focus of our work is to develop a *unified quality-centered* view- and index-selection approach, for a range of query, view, and index classes that are typical in practical database systems. To the best of our knowledge, we are the first to adopt

the solution-quality focus for this generic practical problem setting. Our problem inputs include the query workload of interest and the amount of available storage (disk) space for the views and indexes to be materialized.

In this work we assume that each problem instance specifies one or more evaluation plans for each workload query. Each such query plan is viewed by our approach as just a set of candidate views and indexes that provides acceptable — “good enough” in the sense of [Loh07] — time costs of evaluating the query. Thus, the input query plans form the search space of candidate solutions in our view- and index-selection problem.¹

We show that this version of the view- and index-selection problem is NP hard. To mitigate the complexity of the problem, we develop efficient methods that deliver user-specified quality with respect to the input query plans. Here, quality means proximity to the *globally optimal* performance for this query workload.

Our main contributions are as follows:

- a problem statement that is flexible in the sense of being adaptable to the full spectrum of data models and query languages (Section 4.1),
- proof of NP-completeness of the decision version of our problem (Section 4.1),
- an integer linear program formulation that suggests a natural branch and bound solution strategy (Section 4.2),
- effective upper and lower bounding techniques that lead to attractive tradeoffs between time and solution quality and to interactive quality control by the user (Sections 4.3 and 4.4),
- experimental results on benchmark instances as well as on random instances of increasing size (to illustrate scalability) (Section 4.5), and
- specification and discussion of a practically important easy-to-solve special case (Section 4.5.6).

¹Note that defining such “good” query-evaluation plans is not part of our framework; see [ACN00a, GKC06] for possible approaches to this problem.

The *runtime versus solution quality tradeoff is extremely important*. No approach can guarantee optimal solutions in reasonable time with increasing instance size unless $P = NP$. However, we are able to guarantee a relative error with respect to the optimum. The desired precision is given as input to our algorithm instead of being one of its limitations. And, precision being a worst-case guarantee, the output solution often has better quality than requested.

Alternatively, the algorithm can be run in an *interactive setting*, where it behaves as follows. It begins under the assumption that it is seeking an optimal solution. As soon as it finds a feasible solution, it reports the quality of that solution, asking whether to stop or continue to search for a better one. In both of these settings, the computation of branch and bound is sped up by our interaction between upper and lower bounds – see Sections 4.3 and 4.4.

4.1 Complexity of the Problem

In this section we formally state our problem 4.1.1 and prove that it is NP-complete 4.1.2.

4.1.1 Formal Problem Statement

We first formally define our problem PSP, by specifying its inputs and outputs. The problem inputs are as follows:

- Inputs:
- a set of queries Q ;
 - a set of views and indexes V ;
 - a (space) bound B ;
 - each view/index $v_j \in V$ has an associated weight w_j ;
 - a view subset $V' \subset V$ can be materialized, if its total weight is $\leq B$;
 - for each query $q_i \in Q$ we have a set of plans P_i ;
 - each plan $p_{ij} \in P_i$ is a subset of V , $p_{ij} \subset V$;

- each plan $p_{ij} \in P_i$ has an associated benefit b_{ij} ,
- a plan p_{ij} can be chosen, if it is a subset of the set of chosen for materialization views/indexes;
- for each query $q_i \in Q$ at most one plan can be chosen from P_i ;

Find: $V' \subset V$ that can be materialized and that maximizes the total benefit of plans that can be chosen.

4.1.2 NP-completeness of PSP

Theorem 3. *PSP is NP-complete.*

Proof. We prove the NP-completeness of PSP by reduction from the Dense k -Subgraph Problem, which is known to be NP-complete.

The decision version PSP-D of PSP is as follows. Given the problem inputs of PSP and a positive integer C , is there a subset V' of the set V of input views and indexes that can be materialized, such that the total benefit of plans that can be chosen is at least C .

The Dense k -Subgraph Problem (DkSP) is defined as follows. Given a weighted graph $G = (V, E)$, with weight w_e associated with each edge $e \in E$, and given positive integers k and W , is there a subset V' of size k of vertices V of graph G , such that the weight of the graph induced by V' is at least W .

Suppose we have an instance I_D of the DkSP problem. We want to construct an instance I_V of PSP-D in polynomial time, such that if we later find an answer to I_V then we can find an answer to I_D in polynomial time. (All the times are to be polynomial in the size of the instance I_D of DkSP).

For each edge $e_i \in E$ in I_D we create a query $q_i \in Q$ in I_V , and for each vertex $v_j \in V$ in I_D we create a view v_j in I_V . Each query q_i in I_V has exactly one plan p_{i1} that contains views $\{v_{j1}, v_{j2}\}$, such that in the graph G of I_D there is an edge $(v_{j1}, v_{j2}) = e_i$. In problem I_V , the weights w_j are set to 1 for all j . The gain of the plan $p_{1i} = \{v_{j1}, v_{j2}\}$ in I_V is set to the weight of edge $(v_{j1}, v_{j2}) \in E$ in I_D . C and B in I_V are initialized with W and k , respectively, from I_D .

It is easy to see that we can construct I_V from I_D in this manner within polynomial time in the size of I_D . By construction, the YES answer to I_V , that is, we can choose a subset of views with total weight at most B and total gain at least C , means the YES answer to I_D , that is, we can choose a subset of vertices of size at least k , such that that weight of the graph induced by this subset of vertices is at least W . By the same reason, the YES answer to I_D means the YES answer to I_V . Thus, problem PSP-D is at least as hard as problem DkSP, i.e., NP-hard.

Now, we need to prove that PSP-D is in NP. It means, we need to show that, given a solution to PSP-D, we can verify it in the polynomial in the size of PSP-D time. To do this, for a solution, we must check:

- the total weight of the views chosen for materialization is $\leq B$; clearly, this can be done in linear in the number of views time, $O(|V|)$;
- for each query, at most one plan is chosen; for each query, we count the number of chosen plans, this can be done in linear in the number of queries and plans time, $O(|Q||P|)$;
- for each chosen plan, it is a subset of the materialized views, $p_{ij} \subset V'$; this can be done in linear in the number of queries (because, after the previous check we have at most one plan per query) and views time, $O(|Q||V|)$.

Thus, PSP-D is in NP, and, as we already proved that it is NP-hard, it is NP-complete. \square

4.2 Integer Linear Programming

In this section we formulate our plan-selection problem PSP as an integer linear program (ILP), and discuss the standard branch-and-bound (B&B) technique for solving ILP's.² The problem-specific heuristics and algorithms we present later use B&B as their basic framework. Subsection 4.2.1 gives a formal ILP definition

²For a general discussion of B&B, see [LW66].

of PSP; Subsection 4.2.2 outlines the B&B approach used in both general-purpose and problem-specific solvers. We close the section with remarks on how the basic framework presented here lays the groundwork for the rest of the work.

4.2.1 An ILP model for PSP

Our ILP model uses the following 0/1 variables: x_{ij} is 1 when the j -th plan is chosen for query i , 0 otherwise; y_t is 1 if the t -th view or index is materialized, 0 otherwise.

The objective is to maximize $\sum_{i=1}^n \sum_{j=1}^m b_{ij} x_{ij}$, where b_{ij} is the improvement (gain) in query response when the j -th plan is chosen for query i . A query can have at most one plan; this is expressed by the constraint

$$\sum_{j=1}^m x_{ij} \leq 1 \quad i = 1, \dots, n. \quad (4.1)$$

When a plan for a query is chosen, the views and indexes it needs must be materialized. A simple way to represent this dependence as a constraint is as follows:

$$x_{ij} \leq y_t \quad \{i, j, t | v_t \in p_{ij}\}.$$

In this expression p_{ij} represents the set of views and indexes in plan j for query i . In our ILP model we use a slightly modified version of this group of constraints. Note, if we sum up the constraints with the same i, t from this group, we get

$$\sum_{\{j | v_t \in p_{ij}\}} x_{ij} \leq k_{it} y_t,$$

where k_{it} is the number of constraints with the same i, t . In the new set of constraints the left part of each constraint contains variables for a subset of plans for a query i , thus, by the constraints in (4.1), their sum cannot be greater than one. Therefore, we get the next group of constraints:

$$\sum_{\{j | v_t \in p_{ij}\}} x_{ij} \leq y_t, \quad (4.2)$$

The constraint is written for all i, t where at least one plan for query i needs view/index v_t . This is most easily understood in the contrapositive: if v_t is not materialized ($y_t = 0$), none of the plans that use it can be chosen (all such x_{ij} where $v_t \in p_{ij}$ must be 0).

Finally, the total size of the materialized views and indexes cannot exceed the input storage limit:

$$\sum_{t=1}^k w_t \cdot y_t \leq B \quad (4.3)$$

These constraints fully define our plan-selection problem PSP.

4.2.2 Branch and bound

Branch and bound (B&B) is a well-known approach, dating back to at least the 1950's. It obtains exact (optimum) solutions to ILP's at the expense of worst-case exponential runtime. Its effectiveness relies on the assumption that the worst case occurs only rarely in practice. For ease of understanding, a few of the details in the following description are specific to the PSP problem.

The basic algorithm starts with the root node of a tree, which represents the (initial) problem instance. Other nodes represent smaller instances based on fixed assignments of variables; for example, if y_t is fixed at 0, the instance has one less view/index; if it is fixed at 1, the corresponding constraints (4.2) go away, but the B in (4.3) is reduced by w_t .

Each interior node has two children, one for each of the two values (0 or 1) of a specific variable. The leaves of the tree arise either when the fixed assignment at a node fails to satisfy the constraints (i.e., is *infeasible*) or when the values of all variables have been fixed (in this case, the assignment is feasible but not necessarily optimal).

With no bounding B&B is an exhaustive search of all feasible solutions. A node (and its descendants) can be eliminated if the best gain it can achieve — its *upper bound* — is no better than the gain of a feasible solution already found, the *global lower bound*. Note that if we are ready to accept a solution which is not optimal,

that is, a solution with a relative error of at most $\alpha \in [0, 1)$ compared to the optimal solution, then we can eliminate a subproblem if

$$upper_bound \cdot (1 - \alpha) \leq lower_bound$$

This allows us to eliminate more nodes compared to the case when we want to find the exact optimal solution.

The success of B&B, in its ability to obtain optimal solutions quickly, relies on the quality of heuristics used to obtain upper and lower bounds. Our approaches to these are discussed in Sections 4.3 and 4.4, respectively.

A node is *processed* when its bounds have been computed and its children, when appropriate (i.e., feasible and upper bound $>$ lower bound), have been created. A node's lower bound, when greater than the current global one, replaces it. A node is *active* when it has been created but not yet processed. Nodes may be processed in any order, but the order is typically depth first based on judicious choices of branching variables and assignments to explore first. At any point in the execution of B&B the *integrality gap* is the difference between the current lower bound and the largest upper bound among active nodes.

Experimental results in Section 4.5 show that our combination of upper- and lower-bound computations yields good scalability with increasing instance size, better solution quality as compared with a well-known heuristic [ACN00a] when terminated early, and promising tradeoffs between runtime and solution quality.

4.3 Finding Upper Bounds

Upper bounds are essential to cut off specific subtrees of a branch-and-bound tree: If, at some node, an upper bound does not exceed the current global lower bound, then the node and its descendants can be eliminated. The two best-known methods for obtaining upper bounds for maximization problems are *linear programming relaxation* (*LPR*), a general technique that applies to all integer programs, and *Lagrangian relaxation* (*LaR*), whose details are specific to the problem and to the constraints the expert wishes to relax.

Linear Programming Relaxation (LPR) is simple: turn the integer program into an ordinary linear program (LP) by relaxing those constraints that force variables to be integers. In our problem statement, the constraints that variables x_{ij} and y_t are 0/1 variables are replaced by $0 \leq x_{ij} \leq 1$ and $0 \leq y_t \leq 1$. If the optimal LP solution at a node has all 0/1 values, a potential lower bound has been found. Otherwise, the value of the objective is an upper bound on the optimum value with 0/1 values. This form of relaxation is used universally by general-purpose ILP solvers such as CPLEX.

Lagrangian Relaxation (LaR) (see, e.g., [HK70]) requires choosing the constraints to relax. The relaxed constraints are then incorporated into the objective function, so that there is a penalty associated with an unmet constraint.

We relax constraints (4.2) and add to the objective function the term

$$\sum_{\forall(i,t)} u_{it} \left(y_t - \sum_{\{j|v_t \in p_{ij}\}} x_{ij} \right),$$

where u_{ti} is the penalty associated with the (t, i) -th constraint in the group. Any choice of non-negative u_{ti} yields an upper bound to the original objective function. To get the best possible upper bound we want to find a choice that minimizes the objective.

The process we use, called *subgradient optimization* [HWC74], is an iterative one. It stops when either (a) all of the relaxed constraints are satisfied and the current solution is an optimal solution to the original instance; or (b) further improvement, i.e., a decreased upper bound, is deemed unlikely.

Every iteration step solves the relaxed optimization using the current u_{it} 's — initially arbitrary — and, if the solution is not optimal, adjusts the u_{it} 's according to a line search in order to increase the penalty for those constraints that are not satisfied. While there is no best way to choose the step size in the line search, it is usually started at a fixed value (we use 2.0) and halved whenever the upper bound fails to decrease after a fixed number of steps (we use 10). There is also a fixed lower limit for the step size (0.01 in our case). These choices are usually deduced from preliminary experiments; our choices appear to work well for the full range of instances of this model.

Our choice of constraints to relax has a useful feature: the relaxed optimization problem can be partitioned into two subproblems, one involving only the x_{ij} 's, the other only the y_t 's. To wit,

- $\max \sum_{i=1}^n \sum_{j=1}^m b_{ij}x_{ij} - \sum_{\forall(i,t)} \sum_{\{j|v_t \in p_{ij}\}} u_{it}x_{ij}$, subject to $\sum_{j=1}^m x_{ij} \leq 1$, for $i = 1, \dots, n$; and $x_{ij} \in \{0, 1\}$, for $i = 1, \dots, n$ and $j = 1, \dots, m$, which can be solved optimally by a simple greedy algorithm – the objective function reduces to $\max \sum_{i=1}^n \sum_{j=1}^m B_{ij}x_{ij}$, where $B_{ij} = b_{ij} - \sum_{\{t|v_t \in p_{ij}\}} u_{it}$, and
- $\max \sum_{\forall(i,t)} u_{it}y_t$ subject to $\sum_{t=1}^k w_t \cdot y_t \leq B$, $y_t \in \{0, 1\}$, for $t = 1, \dots, k$, which is a knapsack problem.

LaR consistently produces better upper bounds than LPR. However, the runtime of LPR scales better than that of LaR. That said, there are several key advantages of LaR in the B&B context: LaR can be used

- as part of an effective lower bound heuristic, as discussed in the next section,
- to fix values of some variables, reducing the size of the B&B tree (see Variable Binding below),
- to significantly decrease runtime when a given approximation ratio is desired instead of the optimum; experimental results illustrating this point are presented in Section 4.5, and
- to make use of computations at the parent of a node as a starting point; in particular, the final u_{it} 's at a node make good choices for starting u_{it} 's at its children.

Variable Binding. Lagrangian relaxation allows us to use one additional trick that can be used in any node of the branch-and-bound tree to reduce the size of the subproblem.

Let's suppose we solved the lagrangian relaxation described above, and let \bar{u} be the penalty vector that minimizes the upper bound, $\{\bar{x}, \bar{y}\}$ be the corresponding solution of the relaxed problem, and \bar{S}_{ub} be the corresponding upper bound. We are using

the fact that for any nonnegative assignment of the penalty vector u the value of the optimal solution to the relaxed problem is an upper bound to the original problem.

Suppose, $\bar{x}_{i'j'} = 1$. It follows, by (4.1), that $\bar{x}_{ij} = 0$ for all $j \neq j'$. The question is: can we bind $x_{i'j'}$ to 1? To answer this question, we try to set $x_{i'j'}$ to 0 and see if a subproblem with $x_{i'j'} = 0$ has a potential of producing a solution which is better than the current lower bound. It is easy to see, that for \bar{u} , the value of the optimal solution for the subproblem with $x_{i'j'} = 0$ is

$$\bar{S}_{ub} - B_{i'j'} + \max \left\{ \max_{j \neq j'} \{B_{ij}\}, 0 \right\}$$

Note, this value is an upper bound to the subproblem with $x_{i'j'} = 0$. It is not necessarily the lowest upper bound for the problem that we can obtain with the lagrangian relaxation. Nevertheless, if this value drops below a known lower bound, it means this subproblem cannot produce a solution which is better than the current lower bound. As you can see, this test can be done in linear in the number of plans time.

In the same way, if in the solution to the lagrangian relaxation $x_{i'j'} = 0$, we can fix it to 0, if setting it to 1 (and thus removing the best plan for this query from the solution) reduces the upper bound so that it is \leq the current lower bound

$$\bar{S}_{ub} + B_{i'j'} - \max \left\{ \max_{j \neq j'} \{B_{ij}\}, 0 \right\} \leq S_{lb},$$

where S_{lb} is the value of the best known solution to the original problem.

During the variable bidding process we can also use the technique we explained in 4.2.2. Mainly, if we are ready to accept a solution with a relative error of at most α compared to the optimal solution, we can bind a variable to 0, if

$$\left[\bar{S}_{ub} + B_{i'j'} - \max \left\{ \max_{j \neq j'} \{B_{ij}\}, 0 \right\} \right] \cdot (1 - \alpha) \leq S_{lb},$$

Same is true for the binding variables to 1.

4.4 Finding Lower Bounds

In this section we discuss our proposed methods for finding lower bounds for the branch-and-bound method (discussed in Section 4.2.2) for our plan-selection problem PSP.

4.4.1 Lagrangian Heuristics

In this subsection we describe the lagrangian heuristics we use to obtain lower bounds. This algorithm takes on input a solution to the Lagrangian relaxation and builds a feasible solution using the greedy heuristics described in subsection (4.4.2). The idea of the Lagrangian heuristics is to take a solution to the Lagrangian relaxation of the original problem, which is not in general a feasible solution, and modify it as little as possible to get a feasible solution.

To this end, we examine every query-plan assignment obtained after solving the Lagrangian relaxation (i.e., determine every pair i, j for which $x_{ij} = 1$). For each such assignment we consider the collection of required views and indexes in plan j , and if any one of these views or indexes is not materialized (i.e., corresponding $y_t = 0$), we simply remove the assignment of plan j to query i (i.e., $x_{ij} = 0$). Obviously, at the end of this operation we obtain a feasible solution to the original problem. We then remove every unused view/index by setting its corresponding $y_t = 0$ and use the available space according to the greedy algorithm described in subsection (4.4.2) to obtain a feasible solution to the problem.

4.4.2 Greedy Algorithm

To explain this algorithm it is easier to talk in terms of views/indexes and plans. On the input to this algorithm we get a feasible solution $\{\bar{x}, \bar{y}\}$. In this solution, \bar{x} corresponds to the set of chosen plans and \bar{y} corresponds to the set of chosen views and indexes. It is possible that both of these sets are empty. We want to greedily fill in the available space maximizing the total benefit of the plans that can be executed using the chosen views and indexes.

Algorithm 10: Lagrangian Heuristics

Input : Solution to the Lagrangian Relaxation $\{\bar{x}, \bar{y}\}$,
ILP problem formulation of the original problem
Output: feasible solution to the original problem (candidate lower bound)

```

10.1 begin
10.2   for each  $(i, j)$  such that  $\bar{x}_{ij} = 1$  do
10.3     check all constraints from group (2) with  $x_{ij}$  in them
10.4     if there is at least one violated constraint then
10.5       set  $\bar{x}_{ij} = 0$ 
10.6   for each  $k$  such that  $\bar{y}_k = 1$  do
10.7     check all constraints from group (2) with  $\bar{y}_k$  in them
10.8     if there is at least one constraint whose left part evaluates to one for the
        solution  $\{\bar{x}, \bar{y}\}$  then
10.9       keep  $\bar{y}_k = 1$ 
10.10    else
10.11      set  $\bar{y}_k = 0$ 
10.12  return Greedy( $\{\bar{x}, \bar{y}\}$ )
10.13 end

```

Let V be the set of views and indexes corresponding to \bar{y} . For a set of views and indexes chosen for materialization we can find a set of plans for the queries that maximizes the total benefit. To do this, we, for each query, take the best plan that is based on a view/index set that is a subset of V . Let $P(V)$ be the set of plans that maximizes the total benefit of using V and $B(V)$ be the benefit of $P(V)$. Let $S(V)$ be the total weight of the views and indexes in V .

4.5 Experimental Results

In our experiments we pursue three goals. First, we demonstrate the behavior of our algorithm with respect to runtime versus solution-quality tradeoffs and use in an interactive setting. We talk about it in Section 4.5.2. Second, we present a subclass of problem instances corresponding to OLAP database systems and show that this class of problems is relatively easy for our method. Here, we, also, compare our branch-and-bound implementation with CPLEX — an industry-strength tool for solving ILP

Algorithm 11: Greedy Algorithm

Input : ILP problem formulation of the original problem,
a (possibly trivial) feasible solution to this problem $\{\bar{x}, \bar{y}\}$
Output: feasible solution to the original problem (candidate lower bound)

```

11.1 begin
11.2   Let  $k$  be the maximum number of views and indexes a plan can have in its
      definition
11.3   Let  $W$  be the set of all views and indexes
11.4   while we can add views/indexes to  $V$  without violating the space constraint
      do
11.5     find a subset  $U \subset W \setminus V$  of size at most  $k$  that has maximum
      (  $B(V \cup U) - B(V)$  ) / (  $S(V \cup U) - S(V)$  )
11.6      $V := V \cup U$ 
11.7   return  $\{P(V), V\}$ 
11.8 end

```

problems. We present these results in Section 4.5.4. Third, we compare our approach with an existing greedy heuristics of [ACN00a] for the same problem and show that our algorithm outputs solutions of significantly better quality and therefore makes a suitable back end for the view- and index-selection tool proposed in this work. We talk about it in Section 4.5.5. For most of our tests we used randomly generated problem instances. We explain how we generate these instances in Section 4.5.1.

4.5.1 Random Generation of Problem Instances

We generated problem instances based on the values of several parameters. These can be grouped into (i) structural parameters, and (ii) numerical parameters. The structural parameters are as follows:

- N , total number of queries;
- M , the number of plans per query;
- T , the total number of views/indexes; and
- K , the maximum number of views/indexes per plan.

The numerical parameters are as follows:

- W_{min} and W_{max} , the minimum and maximum view/ index weights;
- C_{min} and C_{max} , the minimum and maximum query costs; and
- P , the minimum plan cost.

We generated problem instances randomly using the following process (all numerical values are chosen at random, uniformly distributed over a specified interval).

Structural properties — queries, plans, and views/indexes: For each query i , we first choose the number of views or indexes relevant to the query, a random integer r in $[K, KM/2]$. Then we randomly choose a subset V_i of r views/indexes from the collection of all views and indexes. This subset constitutes the collection of views and indexes used in all plans for query i . For each of the M plans for query i we choose a random number s of views/indexes in $[1, K]$ and a subset of size s from V_i . For an instance that has N queries we assume there are $T = 2N$ views and indexes. We scale problem-instance size by increasing N . These choices determine all the relationships among the input queries, plans, and views and indexes.

Weights and costs: The weight of each view/index is a random value in the interval $[W_{min}, W_{max}]$. The cost of query i is C_i , a random value in the interval $[C_{min}, C_{max}]$. For each plan j for a query i , its cost c_{ij} is a random value from $[P, C_i]$. Thus, the benefit b_{ij} of using plan j to answer query i is $(C_i - c_{ij})$.

Space bound. We chose a space bound that is a fraction of the sum of the weights of the views and indexes. A value too close to 1 makes the problem trivial, and a value too close to 0 makes it likely that no views or indexes can be chosen, again yielding a trivial problem. Our preliminary experiments showed that taking the space bound to be one third of the total view/index weights yields difficult problem instances.

An additional feature of our problem structure is that we order all views and indexes in a list for which we presume neighboring views and indexes to have more in common than others, making them more likely to be usable for the same query. (This property of related views and indexes occurs in practice, e.g., in the OLAP context [GHRU97,

HRU96a].) Therefore, when choosing random views and indexes for a query, we choose contiguous sublists of this list.

Our uniform choice of view/index weights and plan costs/ benefits ensures that, as is common in practice, these factors will vary from very small to very big. The costs of plans for the same query might not be independent as they are in our random generation, but the dependencies among them are likely to be too complex to model.

4.5.2 B&B behavior

In this subsection we demonstrate the behavior of our algorithm with respect to runtime versus solution-quality tradeoffs and interactive use, for a variety of input parameters. To remind the reader, on input our algorithm takes an ILP formulation of a problem and a maximum allowed error, which is a maximum relative difference between an optimal solution and a solution a user is ready to accept. This error allows the algorithm to prune more B&B tree branches and more effectively reduce problem size by binding more variables.

In our first set of experiments we test the behavior of our method for various values of the maximum allowed error on the input, which is the maximum allowed relative difference between the optimal solution and the solution that we can accept. Figure 4.1 shows the scalability of our algorithm for different maximum allowed errors. The X-axis corresponds to the problem size represented by the number of queries in the workload. The Y-axis represents the runtime of the algorithm. A point on this plot corresponds to the average for 30-instance runtime for a given problem size. Table 4.1 shows the values of parameters of runtime distributions for different maximum allowed errors.

On this picture, we can see that the runtime for $error = 1\%$ drops dramatically compared to the time needed to solve a problem optimally, and the curve for $error = 2\%$ goes even lower. This can be explained by the fact that our pruning and reduction techniques are more effective when we are allowed to return a suboptimal solution.

This means that, for a given problem size, we can always find a quite small error that we can accept, so that the problem with this error can be solved in reasonable

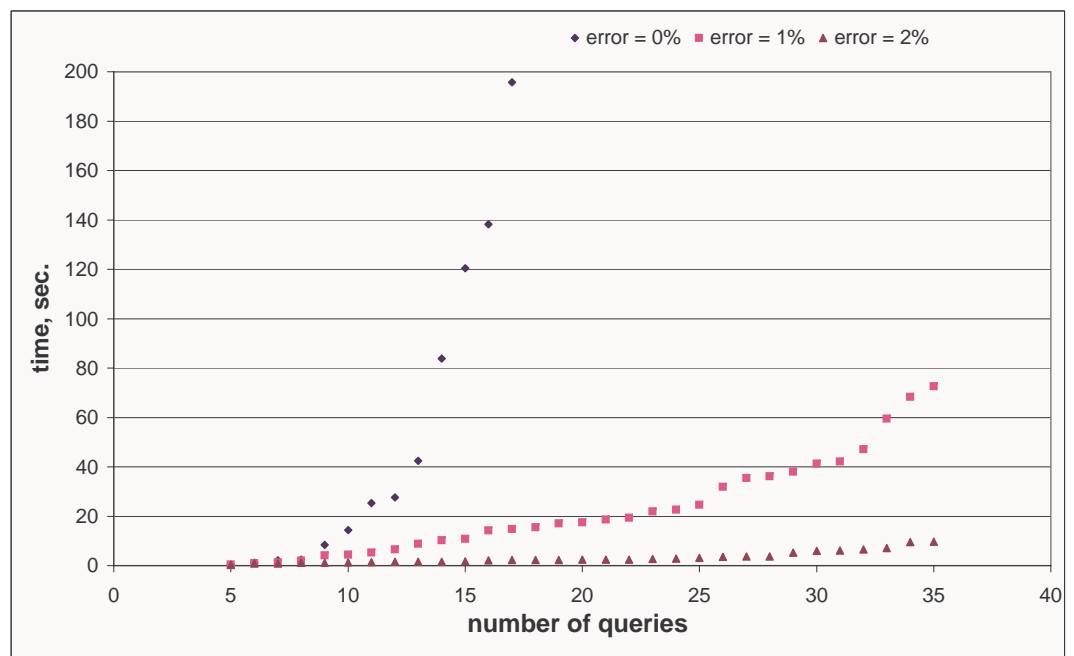


Figure 4.1: Scalability of $B\&B$ for various input errors.

Table 4.1: Mean and standard deviation of runtime distribution (in seconds) for different values of maximum allowed error.

number of queries	mean of runtime for $error = 0\%$	mean of runtime for $error = 1\%$	mean of runtime for $error = 2\%$	st.dev. of runtime for $error = 0\%$	st.dev. of runtime for $error = 1\%$	st.dev. of runtime for $error = 2\%$
5	0.4914	0.38117	0.30597	0.65114	0.54659	0.23693
6	1.1573	0.99227	0.8078	1.1284	1.0447	0.47762
7	2.0678	1.3302	0.8176	2.9821	2.4193	0.50009
8	2.4506	1.9396	1.1135	3.4071	2.9675	0.76092
9	8.3868	4.0695	1.2084	12.071	7.6732	0.91006
10	14.322	4.4042	1.2107	27.055	8.3177	1.5512
11	25.291	5.3306	1.4245	28.622	8.6364	1.8453
12	27.662	6.5317	1.5299	64.745	10.992	1.9921
13	42.355	8.8331	1.625	104.77	19.874	2.3344
14	83.845	10.28	1.6344	109.72	20.941	2.3349
15	120.48	10.762	1.7444	214.26	31.801	2.5471
16	138.18	14.274	2.0942	251.42	33.178	2.6224
17	195.72	14.87	2.3156	303.41	36.79	2.6243
18		15.584	2.3287		37.412	4.5643
19		17.054	2.3374		37.664	4.8539
20		17.567	2.4075		42.657	5.0604
21		18.61	2.4487		50.283	5.5582
22		19.357	2.4803		53.042	6.0984
23		21.86	2.6866		63.765	7.3873
24		22.701	2.8922		64.324	7.6775
25		24.627	3.0794		67.171	7.7759
26		31.884	3.4891		82.663	8.1748
27		35.411	3.673		91.85	8.2292
28		36.157	3.6777		92.055	14.002
29		37.987	5.2584		94.448	14.239
30		41.279	6.003		110.34	14.784
31		42.117	6.1753		167	21.244
32		47.083	6.5536		186.36	22.076
33		59.445	7.1468		224.77	29.735
34		68.378	9.5552		287.69	33.005
35		72.527	9.7062		296.78	37.267

time. For example, in our experiments we could solve problems with 200 queries and maximum allowed error of 5% in less than one second, see Figure 4.2.

The next experiment is designed to further demonstrate this dependence. For this experiment, see Figure 4.3, we found an instance with 10 queries in it with a big difference between the initial upper and lower bounds, and tested the runtime for this problem instance against various maximum allowed errors on the input. The X-axis in Figure 4.3 represents the error that we give our algorithm in the input. The Y-axis shows the runtime of our algorithm. A point on this graph shows how much time it took the program to get a solution within allowed relative error.

Note that the runtime drops not only because of the easier satisfiable stopping criterion, but also because we bind more variables and prune more subproblems during

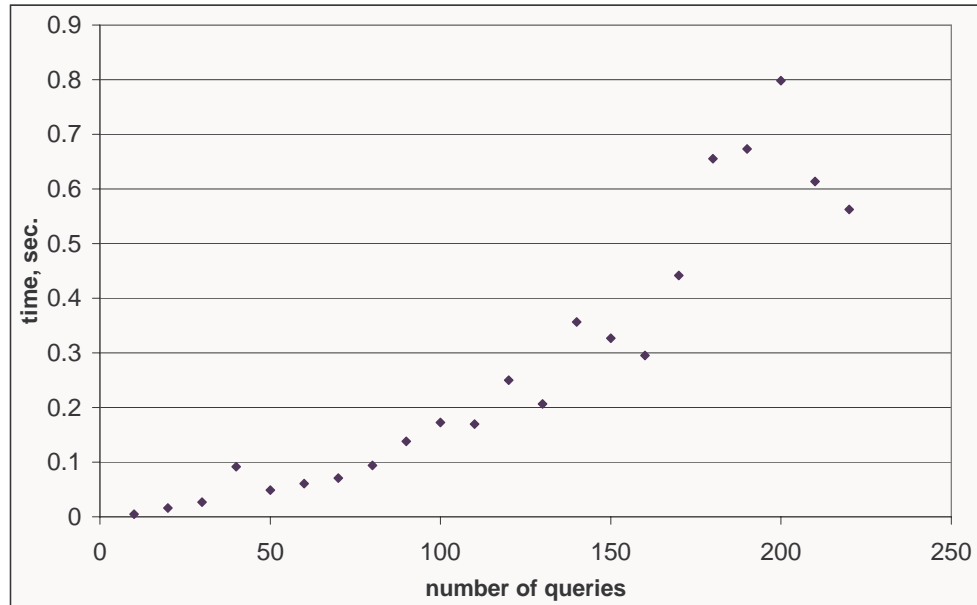


Figure 4.2: Analysis of the runtime of $B\&B$.

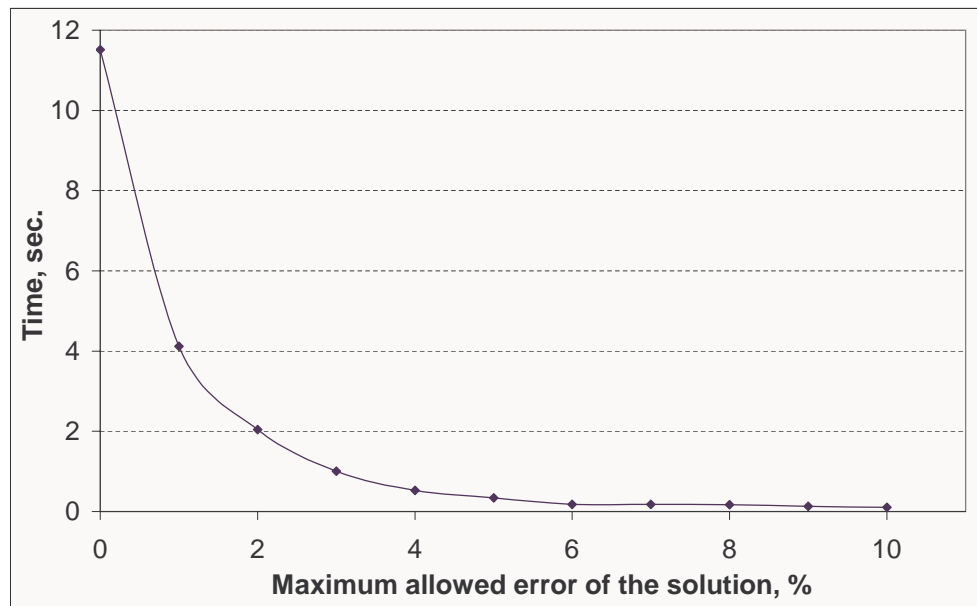


Figure 4.3: Runtime of $B\&B$ on a fixed instance with various input errors.

the exploration of the branch and bound tree. This is an important point given that the traditional runtime versus solution-quality tradeoffs for branch and bound are achieved using successively longer runs. In other words, in the traditional approach, if our goal is to find a solution of a given quality, we run a branch and bound algorithm for a certain amount of time and then check the quality of the solution obtained during this run. If we are not satisfied with the quality, we run the algorithm again, but for longer time period, and we iterate like this until we achieve the desired quality of the solution. Contrary to this approach, in our branch and bound algorithm we use the maximum allowed error for *more effective pruning*.

The goal of the next experiment, see Figure 4.4, is to demonstrate the interactive property of our algorithm. That is, we can ask the program to report a best know solution together with its relative gap (relative difference with a maximum upper bound) every time the latter is improved. At this point, a user can decide to stop the program, if he is satisfied with the quality of the solution, or to continue running the program waiting for a better result. In this experiment, we ran our algorithm with the maximum allowed error set to zero and measured the relative difference between upper and lower bounds during the program execution. A point on this plot says that for a given time point there was a problem instance that had this relative difference between a known feasible solution and a maximum upper bound.

Thus, in the next experiment we run our program with the maximum allowed error set to zero, and record time and relative gap between the upper and lower bounds every time this gap is improved. The plot in Figure 4.4 is based on the experimental results on 181 random instances of the same size. On this plot, the X-axis represents the runtime, and the Y-axis represents the relative difference between an intermediate solution (lower bound) and a maximum upper bound. A point on this plot says that for a given time point there was a problem instance that had this relative difference between a known feasible solution and a maximum upper bound. For any given run there are multiple points on the plot, one for each interaction with the user. Table 4.2 shows the results of four runs from the experimental set.

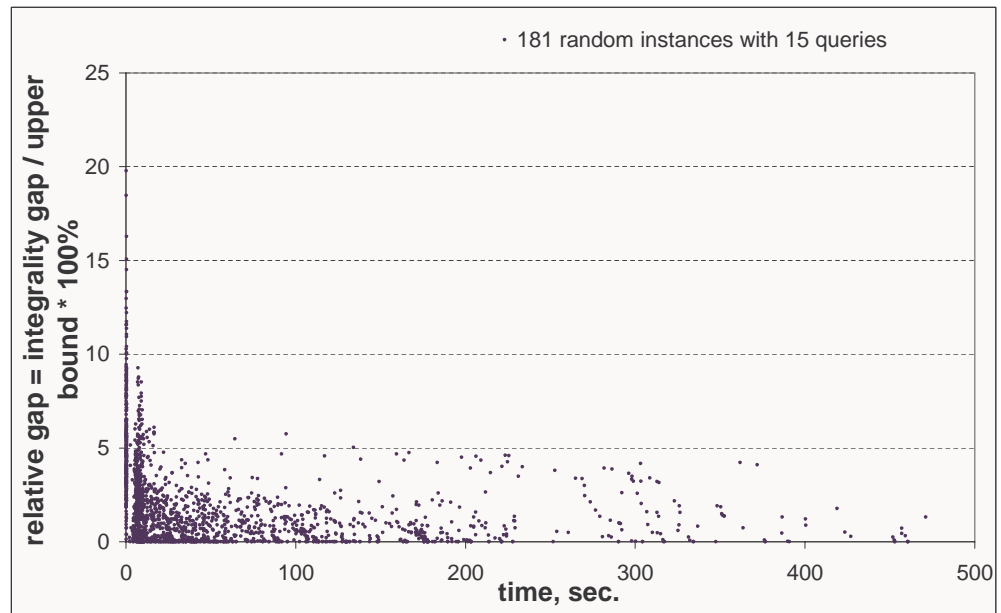


Figure 4.4: Gap between lower and upper bounds.

Table 4.2: Example output of our algorithm in the interactive mode.

	Lower bound	Upper bound	Relative gap	Time, sec.
Instance 1	96600443	102587138	5.84E-02	0.25
	98551309	102587138	3.93E-02	0.278
	98551309	102512180	3.86E-02	4.964
	101875267	102512180	6.21E-03	5.032
	101973924	102512180	5.25E-03	5.336
	101973924	102473305	4.87E-03	5.601
	101973924	102441958	4.57E-03	9.488
	101973924	102408181	4.24E-03	9.876
	101973924	102150217	1.73E-03	9.989
	101973924	101973924	0	14.279
Instance 2	77275038	81911556	5.66E-02	0.29
	80111547	81911556	2.20E-02	6.888
	80111547	81868853	2.15E-02	7.214
	81340468	81868853	6.45E-03	7.809
	81751917	81868853	1.43E-03	9.021
	81751917	81847284	1.17E-03	25.257
	81751917	81751918	1.19E-08	26.177
	81751917	81751917	0	26.265
Instance 3	94154782	97678592	3.61E-02	0.238
	94154782	97156221	3.09E-02	16.582
	94416240	97156221	2.82E-02	33.056
	94416240	96780048	2.44E-02	35.426
	94482547	96780048	2.37E-02	36.043
	94482547	96524132	2.12E-02	55.438
	94482547	95817264	1.39E-02	99.472
	94482547	95625588	1.20E-02	104.383
	94482547	94985612	5.30E-03	108.124
	94482547	94668700	1.97E-03	112.068
	94482547	94482547	0	115.986
Instance 4	57627943	63217526	8.84E-02	0.262
	60655306	63217526	4.05E-02	0.263
	61070305	63217526	3.40E-02	7.382
	61122162	63217526	3.31E-02	7.902
	62472860	63217526	1.18E-02	8.422
	62472860	62918941	7.09E-03	44.928
	62472860	62472861	1.44E-08	49.682
	62472860	62472860	0	49.737

Although the setting is different here (the error bound adjusted interactively versus being part of the input), there is a clear relationship between this plot (Figure 4.4)

and the one showing scalability for different errors (Figure 4.1). In Figure 4.4, the majority of points drops below the 2% level after 10–20 seconds, correspond to the point for 15 queries in Figure 4.1.

4.5.3 OLAP-alike Problem Instances

In OLAP environment it is often the case that a query can be answered using only one view. Thus, we run a set of experiments for randomly generated instances in which every plan has only one view in it. For this case, we design a slightly modified random instance generator. To make sure that every view is used at least once, we, first, for each view, choose a plan. After that, for all plans that did not get a view during the first stage, we choose one.

In our experiments we noticed that the complexity of the problem — program runtime to solve it — depends on the *overlap level* of plans. We define *overlap level* as the average number of plans a view appears in, or $OL = \frac{\#plans}{\#views}$. The idea of this measure is to reflect how much plans from a workload have in common. Our intuition is the larger this number is the more time we need to solve the problem. Note that $OL \geq 1$, and the case when $OL = 1$ corresponds to problem instances with one-to-one correspondence between plans and views.

Following all said above, we run a set of experiments in which we measure the algorithm runtime against different values of OL . Figure 4.5 shows the results. For this set of experiments, all problem instance have the same total number of plans (100), but varying number of views (from 38 to 100). Each point on Figure 4.5 corresponds to the average of 30 experiments on instances with the same number of queries, plans, and views. The X -axis represents the average number of plans a view appears in. The Y -axis represents runtime in seconds. Table 4.3 reports results of the same experiments. For a fixed value of OL , it gives a mean and a standard deviation of the runtime distribution.

From the results of these experiments we can see that for the range $1 \leq OL \leq 1.7$ the problem is quite simple for our approach. In fact, the quality of upper and lower bounds is good enough to be able to solve these problems in the first node of the

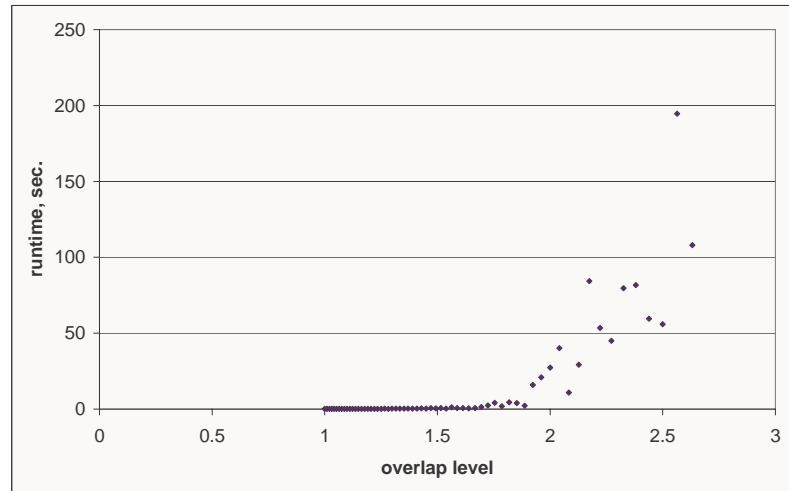


Figure 4.5: Complexity of the problem for different values of OL .

branch-and-bound tree.

In the experiment described above, we showed that we found a class of problem instances for which our algorithm performs well. In the next subsection we compare our approach with CPLEX on this class of problem instances and present one more subclass of problems on which our branch and bound algorithm behaves better than CPLEX.

4.5.4 Comparison with CPLEX

The goal of this section is to experimentally compare our branch-and-bound approach with CPLEX [CPL00] and to check limits of our algorithm.

CPLEX is an industry-strength tool for solving mathematical programming problems. To solve ILP problems it uses a very general and robust branch-and-cut algorithm. This algorithm is tuned to achieve good performance on a wide variety of Mixed Integer Programming (MIP) models. A great deal of algorithmic development effort has been devoted to construct this tool and to choose its settings. That is why it is hard to compete with it in general case. But, in this section, we present a class

Table 4.3: Mean and standard deviation of runtime distribution (in seconds) for a fixed value of OL .

OL	mean runtime	standard deviation
1.00	0.17	0.004
1.11	0.19	0.006
1.20	0.28	0.008
1.32	0.37	0.282
1.41	0.43	0.375
1.52	0.45	0.416
1.61	0.90	0.409
1.72	3.71	2.325
1.82	19.50	3.900
1.92	22.21	17.927
2.00	60.31	26.523
2.13	57.07	15.010
2.22	76.84	54.563
2.33	263.40	60.536
2.44	102.92	45.656
2.50	108.02	149.820
2.63	316.52	80.197

of problem instances for which our branch-and-bound algorithm shows better results, and show that our algorithm can be used as a presolver for CPLEX to improve its performance on our ILP model.

In our first set of experiments, we compare our approach with CPLEX on the class of problem instances presented in the previous subsection. For these experiments, we fix OL to a value from $1 \leq OL \leq 1.7$ and vary the problem size — total number of queries. Then, we measure runtime of our B&B and CPLEX. Figure 4.6 shows the results.

Here, we fix $OL = 1.4$, fix the number of plans per query to 5 and vary the number of queries from 40 to 200. The number of views for each instance is set to $\frac{\text{total_number_of_plans}}{OL}$ to keep the ratio the same for all instances. Again, each point on the plot corresponds to the average of 30 experiments.

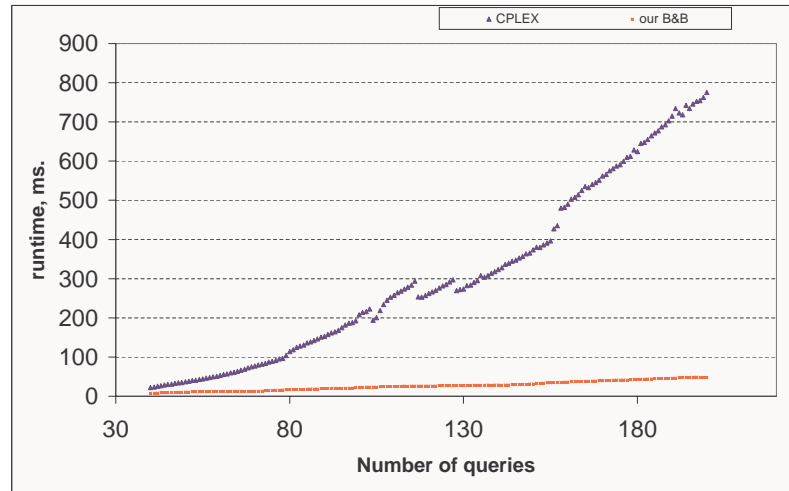


Figure 4.6: Scalability of the algorithms for $OL = 1.4$ (problem instances in which a view appears in 1.4 plans in average).

From these results, we can see that the runtime of our program grows much slower than that of CPLEX for the increasing problem size. As before, this can be explained by the fact that the quality of upper and lower bounds we obtain in our algorithm is good enough to solve problems from this class in the first node. This fact inspired our next set of experiments.

For our next set of experiments we use our initial random instance generator, the one that generates general database-oriented instances with more than one view per plan allowed. Note that such instances are, in general, harder, as having more than one view in a plan brings more complicated structure. To balance the increased complexity we allow the solution to be within 5% of the optimum.

As in the previous set of experiments, we fix the number of plans per query to 5 and vary the number of queries from 40 to 200. Note that for this random instance generator we do not have a direct control over the total number of views/indexes used by plans, we can only set a maximum on this number. Thus, we set this maximum to $\frac{\text{total_number_of_plans}}{1.4}$, but, as the experiment results showed, the actual value of OL varied from 1.42 to 1.75. This happens because during random instance generation

plans randomly choose subsets of views and indexes they use. Thus, there can be some views and indexes that are not chosen by any plan. That is why, the actual number of views/indexes in a problem instance is less than a preset maximum.

Note that OL in this experiment is not the same as in the case of OLAP — with one view per plan. To remind, we defined OL as the average number of plans a view can appear in, and for the case where each plan has exactly one view in it we define $OL = \frac{\#plans}{\#views}$. But, for the case when we allow more than one view/index in a plan it must be $OL = \frac{\#plans \cdot A}{\#view}$, where A is the average number of views/indexes in a plan. For this set of experiments, we set the maximum number of views/indexes per plans to 3. Thus, in average, each plan has 2 views/indexes in it. Therefore, the real value of OL lies between 2.82 and 3.5 for this set of experiments. The results are shown on Figure 4.7.

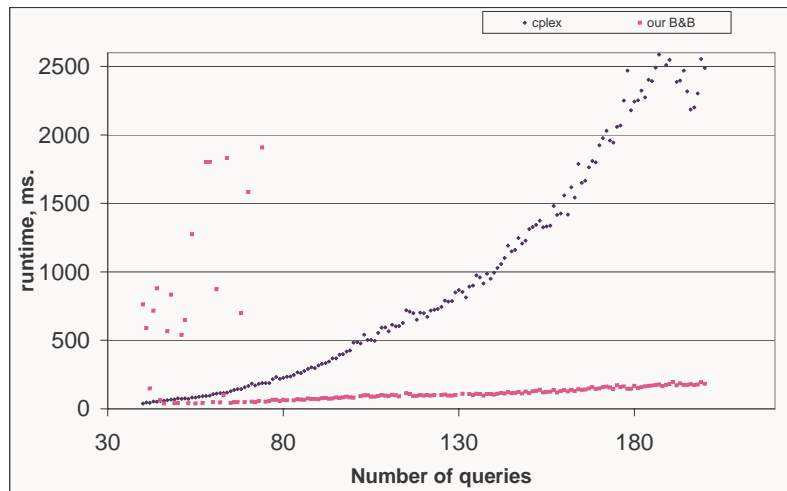


Figure 4.7: Scalability of the algorithms for database-oriented instances.

As you can see, the general trend is that runtime of our algorithm grows much slower than that of CPLEX. Again, as for the case of OLAP, this is because the quality of our upper and lower bounds is good enough to solve the problems of this class in the first node.

Thus, we found two subcases of PSP where our algorithm behaves better than CPLEX. But, as we said before, CPLEX still beats us on some other subclasses of the problem. This can be explained by the fact that the main contributions of our work are the methods for finding good upper and lower bounds, and the variable binding techniques. Note that our method benefits from all three techniques even if we run only one iteration of our branch and bound algorithm. To remind the reader, at each iteration, we, first, find an upper bound of a subproblem using the Lagrangian relaxation, see 4.3; second, we find a lower bound using the Lagrangian heuristic, see 4.4; third, we try to bind variables of the subproblem using the solution to the Lagrangian relaxation, see 4.3.

At the same moment, we do not address some other questions related to branch-and-bound procedure, such as tree exploration strategy. CPLEX follows a different strategy that allows it to perform well on a wide variety of problems. Mainly, for the speed-quality tradeoff of the methods that find bounds, CPLEX is more oriented towards the speed component if compared to our approach. Also, it uses a whole lot of different techniques that speed up the optimization process very well. Among them, cuts generation methods, probing, different heuristics with very carefully chosen settings.

Having that our method performs better when a problem can be solved in the first node of branch-and-bound tree, and CPLEX has a good set of algorithms for branch-and-bound tree exploration, we propose to use a mix of these two solvers for our problem. Mainly, we propose to use our algorithm as a presolver of CPLEX. Given a problem instance, we, first, find initial upper and lower bounds on the optimal solution to this problem using our method. Then, we reduce the size of the problem using our variable binding technique. Finally, we give the reduced problem and a good feasible solution for it to CPLEX. For such mixed solver, there can be two possible scenarios. First, on the presolve stage, our algorithm finds such upper and lower bounds that their relative difference is less than maximum allowed error. In this case, we return the solution corresponding to the lower bound and we are done. Second, if the relative difference between upper and lower bounds is greater than maximum allowed error, we run our variable binding module and solve the remaining problem

with CPLEX.

Preliminary experiments showed that the mixed solver solves random problem instances faster than CPLEX on its own. In our future work, we plan to run more tests to justify this point.

4.5.5 Comparison with $Greedy(k, m)$

In this subsection we compare our algorithm with the greedy algorithm $Greedy(k, m)$ described in [ACN00a]. This algorithm exhaustively searches for an optimal subset of views and indexes of size k and then greedily (based on the additional benefit accrued given the current configuration) adds views and indexes to this subset until it has m views and indexes in it. It is worth mentioning that $Greedy(k, m)$ assumes that all views and indexes have weight 1, while our algorithm can work with any weights. Thus, for the experiments in this subsection all views and indexes have weight 1. Note, the fact that all views and indexes have unit weight makes the problem easier for us, because, in this case, the knapsack problem that we get after the Lagrangian relaxation can be solved greedily.

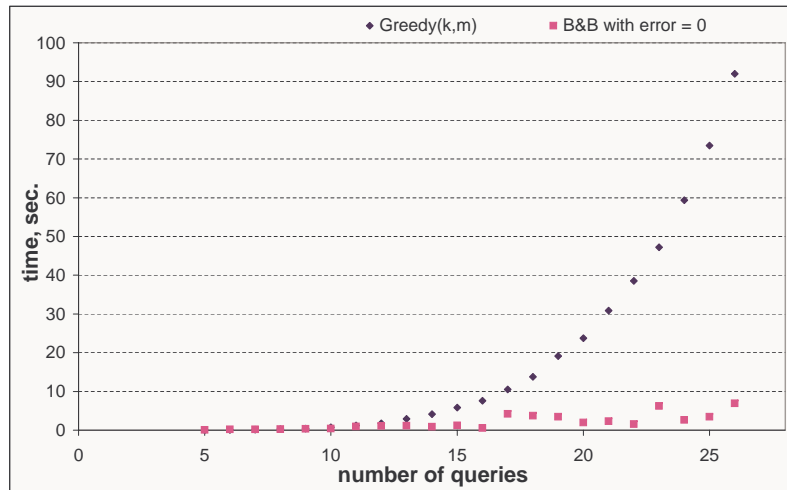


Figure 4.8: Scalability of $B\&B$ and $Greedy(k, m)$.

Figure 4.8 shows the scalability of $Greedy(k, m)$ when compared to our algorithm

with input error 0% (optimal solution). For $Greedy(k, m)$ we set k to 3 and set m to the input storage limit. Our choice of $k = 3$ comes from the fact that a larger value will increase the already prohibitive runtime, while a smaller value (the only other choices are 2 and 1) increases the error of the solution.

In Figure 4.8, the X-axis corresponds to the problem size represented by the number of queries in the workload, and the Y-axis corresponds to the runtime of the algorithms, measured in seconds (which is the unit of runtime in all the other figures as well). A point on this plot represents the average runtime for 30 experiments for a given problem size. It can be observed that the runtime of $Greedy(k, m)$ grows fast with the problem size. This can be explained by the fact that the runtime is proportional to C_V^k (number of different ways to choose k elements from a set of size V). In contrast, the runtime of our algorithm grows much slower. Note that in this set of experiments our algorithm is always getting an optimal solution, while $Greedy(k, m)$ is getting a solution with no guarantee on quality.

Figure 4.9 shows the average error of the solution returned by $Greedy(k, m)$. As in Figure 4.8, the X-axis corresponds to the problem size represented by the number of queries in the workload. The Y-axis corresponds to the average relative error compared to an optimal solution. A point on this graph represents the average relative error for 30 experiments of a given problem size.

Figure 4.9 is based on our experiments on 22 different problem sizes, with between 5 and 26 input queries, for the total of 660 experiments. In this set of experiments, $Greedy(k, m)$ returned an optimal solution in only 40% of the cases. The maximum relative error was 29%, and in 1% of all experiments the relative error was more than 12.5%.

Figure 4.10 shows the relative error distribution from another point of view. It is based on 30 experiments on instances with 24 queries each. The X-axis represents the relative error of the solution returned by $Greedy(k, m)$. The Y-axis represents the fraction of problem instances. A point on this plot, for a given relative error, shows the fraction of problem instances that have a solution with at most the relative error on the X-axis.

The distribution over this set of same-size experiments is exponential, suggesting

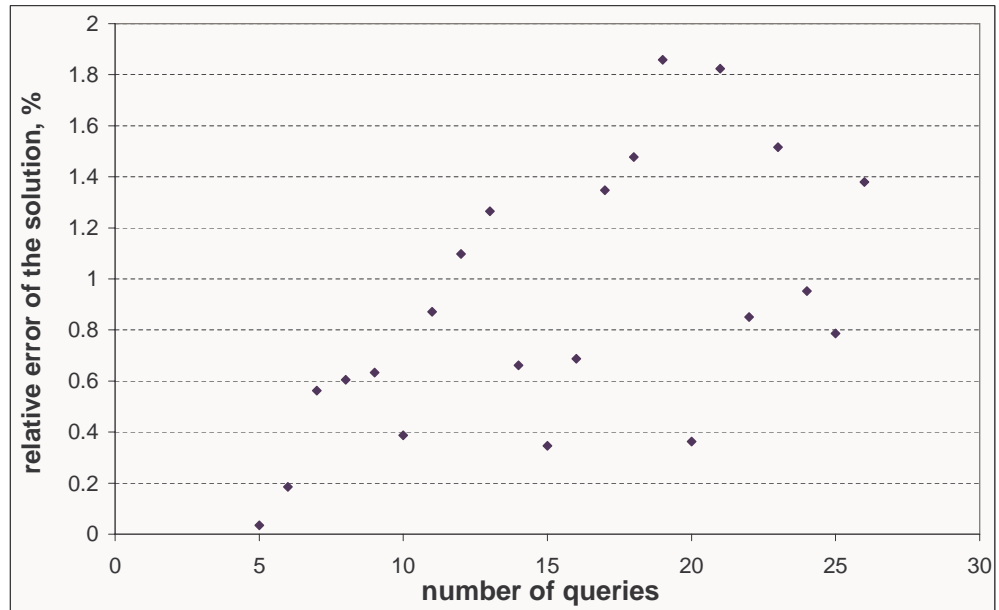


Figure 4.9: Quality of the $Greedy(k, m)$ solution.

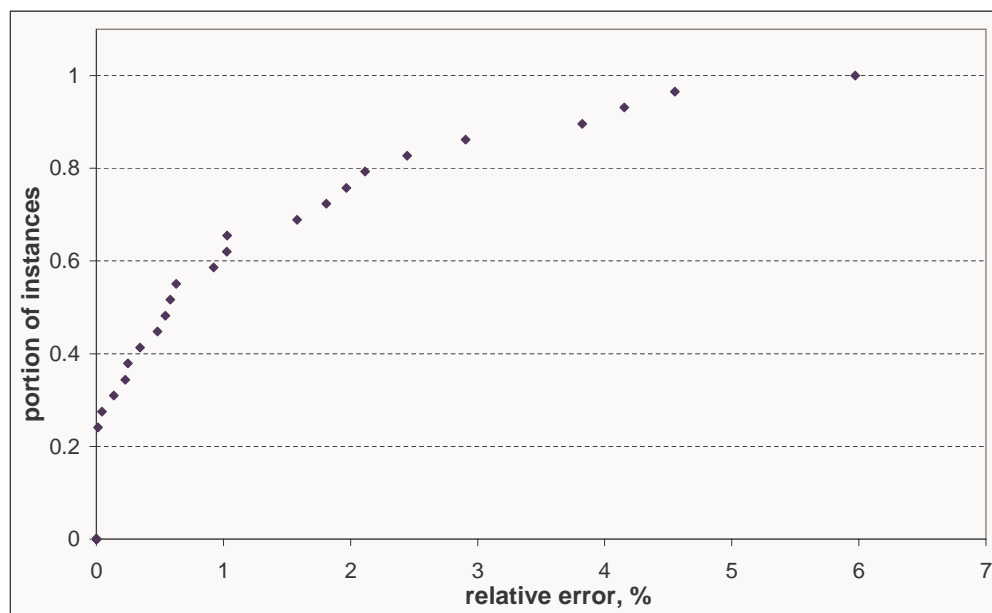


Figure 4.10: Error distribution for $Greedy(k, m)$.

that, while the median (0.5%) and mean (1.3%) are small, the performance can differ wildly from one instance to the next, the extreme opposite of a guarantee.

4.5.6 Other Observations

We make here a few additional observations on our algorithm and experiments. First, our preliminary experiments on the TPC-H benchmark dataset [TH] showed that our algorithm can obtain in 0.2 sec. an optimal solution on instances with 22 input queries (the actual TPC-H queries) and 32 views. Second, users can specify the precision of the output of our algorithm in two ways — maximizing the gain or minimizing the query-evaluation costs — while always obtaining correct solutions (cf. the observation in [KM99] on the line of work [GHRU97, GM99, HRU96a]).

Chapter 5

Extensions and Future Work

5.1 View-Selection Problem for the More General Case of Queries

In this section, we discuss how our algorithm can be extended to a more general class of queries, and how it will affect the complexity of the algorithm. We demonstrate it on the class of *comb* queries.

Definition 7. A comb query is a query of the next form

$$Q(x) : -S_0(x_0)S_1(x_1)T_1(y_1) \dots S_{n-1}(x_{n-1})T_{n-1}(y_{n-1})S_n(x_n),$$

where $S_0, S_1, \dots, S_n, T_1, T_2, \dots, T_{n-1}$ are database tables, $x, x_0, x_1, \dots, y_1, y_2, \dots$ are sets of attributes such that

- $x \subseteq \bigcup_{i=0}^n x_i \bigcup_{j=1}^{n-1} y_j$
- $\begin{cases} x_i \cap x_j \neq \emptyset & , \text{ if } |i - j| \leq 1 \\ x_i \cap x_j = \emptyset & , \text{ if } |i - j| > 2 \end{cases}$
- $x_i \cap y_i \neq \emptyset, \text{ if } \forall i \in \{1, \dots, n-1\}$
- $x_j \cap y_i = \emptyset, \text{ if } i \neq j$

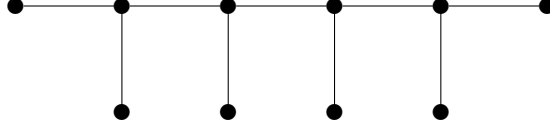


Figure 5.1: Graph representation of a comb query.

- $y_i \cap y_j = \emptyset$, if $i \neq j$

Figure 5.4 shows a graph representation of a comb query. In this graph, vertices are base tables and edges represent join conditions between the tables. A comb graph consists of a chain (we will refer to it as to main or global chain) with teeth dangling from the inner nodes of the chain.

To remind the reader, in the algorithms that we propose in Section 3, we build plans for each node corresponding to a connected subgraph of the chain. The attractiveness of the class of chain queries is in that the total number of connected subgraphs of the chain is quadratic in the length of the chain. Let us now calculate the total number of connected subgraphs for a comb graph. For a given subchain of length k (number of nodes of the chain) of the main chain, there are at most 2^k subgraphs that include this subchain and a subset of teeth connected to it. This number is lower for subchains that includes either leftmost or rightmost nodes of the global chain. If the length of the global chain is s , then there are $s - k + 1$ different subchains of size k . Thus, the total number of connected subgraphs of a comb graph is

$$\sum_{k=1}^s (s - k + 1) 2^k = \sum_{i=1}^s \sum_{k=1}^i 2^k = \sum_{i=1}^s 2^{i+1} \sim O(2^s).$$

Since $s \sim n/2$, where n is the total number of vertices of the comb, the total number of connected subgraphs of a comb graph on n vertices is proportional to $O(2^{n/2})$.

The generalization of our algorithm to the case when queries of the workload can be combined into a tree-shaped graph can be found in Figure 5.2. The main difference

Algorithm 12: GeneralMultiQueryPlanGen

Input: database statistics (see Chapter 3), set of queries Q that together form an acyclic graph $H = (V, E)$, where V is the set of vertices and E is the set of edges, space bound B

Output: a set of plans for each query in Q , containing optimal solution to ADR

```

12.1 for each connected sub-graph  $q = (V', E')$  of  $H$  in the order of increasing  $|V'|$ 
    do
12.2   for each pair of subgraphs  $q_1$  and  $q_2$ , which are obtained from  $q$  by removing
        one edge do
12.3      $\lfloor$  CONSTRUCTJOINPLANS( $q_1, q_2, q, B$ );
12.4   let  $Q'$  be the set of queries for which  $q$  is the subset of tables;
12.5   CONSTRUCTVIEWPLANS( $q, Q, B$ );
12.6   MULTIQUERYPRUNEPLANS( $plans(q)$ );
12.7 return set of plans for each query in  $Q$ 

```

Figure 5.2: Constructing (view-based) evaluation plans for multiple queries.

of this algorithm, compared to the chain-query version, is the conditions in the for-loops in lines 12.1 and 12.2. Instead of considering subchains, we build plans for each connected subgraph of the global graph in the order of increasing number of vertices. For each subgraph q , we remove from it one edge (or, join condition) and obtain two smaller subgraphs corresponding to subqueries q_1 and q_2 . After that, we take all pairs of plans for q_1 and q_2 and join them to obtain candidate plans for q .

Note that the pruning rules that we proposed in Chapter 3 do not set any restrictions on the underlying queries or plans. Therefore, these rules are also applicable for the general case.

5.2 Applicability of Our Approach to the Multiple Query Optimization

In certain database application it is often possible that a group of queries is submitted to the database management system at the same time. A naïve and straightforward approach is to execute these queries separately. But this can be inefficient when the queries are closely related and share a lot of common substructures. Multiple Query Optimization (MQO) consists of identifying such common subexpressions and building a global plan for this group of queries. Paper [Sel88] shows that MQO is NP-hard.

The difference between ADR and MQO is that for the former we try to minimize the query execution time by using view and indexes (for short, tasks) having a restriction on the amount of the tasks that we can use, and for the latter we try to minimize the amount of intermediate results, while answering *all* queries in the workload.

If we look closer at the two problem we can notice the similarity of views in ADR and intermediate results of MQO. In fact, if we think of intermediate results as views, we can use our current algorithm for MQO.

For MQO, the first stage consists of building the sets of base-table plans for each query. Here, the trick is to keep, for each query, not only the best plan, but also its suboptimal plans, because, even if a plan is suboptimal for a given query, it may

share some of the computational step with a plan for another query, and result in a winning combination. Consider Example 10.

Example 10. *Suppose we have two queries $Q_1 : -ABC$ and $Q_2 : -BCD$, and the best plans for the individual queries are $P_{11} : -(A \bowtie B) \bowtie C$ and $P_{21} : -B \bowtie (C \bowtie D)$, correspondingly. Consider an alternative solution $P_{12} : -A \bowtie (B \bowtie C)$ and $P_{22} : -(B \bowtie C) \bowtie D$. To evaluate solution (P_{11}, P_{21}) , we need to perform 4 join operations, while to execute plans (P_{12}, P_{22}) we only need to perform 3 joins. Thus, the second solution might be a winning combination for this workload.*

MQO differs from ADR in that all plans of MQO use only base tables, and what matters is the order of joins in each plan. So, we need a method of encoding plans in such a way that later (in the second stage) we can easily identify subexpressions common to all queries. It can be done using a notion that is similar to a view, let us call it an *intermediate result*. An intermediate result is simply a result of a subquery. For each plan p , we keep a list of intermediate results that it went through, $intres(p)$. If a plan contains intermediate result R , it does not matter how this result is calculated (although it still can be deduced from the plan). What matters is that this plan, during the execution, goes through intermediate result R . Instead of weights that we used for views, intermediate results have costs associated with obtaining them, which are calculated during the optimization on the first stage.

Figure 5.3 contains the details of the first stage plug-in for MQO. We start by initializing plans for single-table subqueries, Procedure INITIALIZELEVELONE. It is very similar to multi-query version of CONSTRUCTVIEWPLANS. Mainly, we create one intermediate result of each query subset that can use the table. This creates an opportunity of having plans that can be used by any subset of queries. After that, we follow the same as in ADR dynamic programming algorithm and join the plans for smaller subqueries to create plans for bigger ones. One additional feature of this algorithm, in comparison to ADR, is that, for each plan, we maintain a list of intermediate result of this plan.

Getting back to Example 10, following Algorithm 13, query Q_1 has two plans that can be encoded as follows $P_{11} : -R_A R_B R_{AB} R_C R_{ABC}$, $P_{12} : -R_B R_C R_{BC} R_A R'_{ABC}$, and query Q_2 has

Algorithm 13: MQOPLANGEN

Input: database statistics (see Chapter 3), set of CQAC queries Q that together form chain H
Output: a set of plans for each query in Q

```

13.1 INITIALIZELEVELONE( $Q, H$ );
13.2 for each sub-chain of size at least two  $q$  of  $H$  in the order of increasing length
    do
13.3   for each split  $q$  into two smaller sub-chains  $q_1$  and  $q_2$  do
13.4   |   CREATEJOINPLANS( $q_1, q_2, q$ );
13.5 return set of plans for each query in  $Q$ 
  
```

Procedure INITIALIZELEVELONE(Q, H)

Input: set of CQAC queries Q that together form chain H
Output: set of plans for each single-table subquery of H

```

14.1 for each table  $t$  of  $H$  do
14.2   let  $Q'$  be the set of queries that use  $t$ ;
14.3   for each  $k \subseteq Q'$  do
14.4   |   create intermediate result  $ir$  corresponding to  $t$  with the disjunction of
14.5   |   the sets of constraints of queries in  $k$  applied to it;
14.6   |   create new plan  $p$  and initialize its intres with  $ir$ ;
14.7 return set of plans for each single-table subquery  $t$  of  $H$ ;
  
```

Procedure CREATEJOINPLANS(q_1, q_2, q)

Input: subquery q_1 and a set of plans for it; subquery q_2 and set of plans for it; subquery q , which is a join of q_1 and q_2
Output: a set of plans for subquery q

```

15.1 for each pair of plans  $p_1 \in plans(q_1)$  and  $p_2 \in plans(q_2)$  do
15.2   if  $queries(p_1) \cap queries(p_2) \neq \emptyset$  then
15.3   |   create new plan  $p$ ;
15.4   |   create new intermediate result  $ir$  by joining  $p_1$  and  $p_2$ ;
15.5   |    $intres(p) = intres(p_1) \cup intres(p_2) \cup ir$ ;
15.6   |    $queries(p) = queries(p_1) \cap queries(p_2)$ ;
15.7   |   save  $p$  into  $plans(q)$ ;
15.8 return set of plans for subquery  $q$ 
  
```

Figure 5.3: Constructing plans for MQO.

plans $P_{21} : -R_C R_D R_{CD} R_B R_{BCD}$, $P_{22} : -R_B R_C R_{BC} R_D R'_{BCD}$. Here, the cost of R_{ABC} is the cost of joining R_{AB} and R_C , and the cost of R'_{ABC} is cost of joining R_{BC} and R_A .

From this example, we can see that the algorithm may create several intermediate results for exactly the same subquery. The difference of these intermediate results is in the cost associated with their obtaining.

Note that our pruning rules for ADR are no longer valid for MQO. This issue remains open, for now.

Recall the ILP for the ADR problem:

$$\min \sum_{i=1}^n \sum_{j=1}^m c_{ij} x_{ij} \quad (5.1)$$

$$\sum_{j=1}^m x_{ij} \geq 1 \quad i = 1, \dots, n, \quad (5.2)$$

$$\sum_{\{j|v_t \in p_{ij}\}} x_{ij} \leq y_t \quad \forall i, t \text{ s.t. } \exists j : v_t \in p_{ij}, \quad (5.3)$$

$$\sum_{t=1}^k w_t \cdot y_t \leq B \quad (5.4)$$

where c_{ij} is the cost of plan j for query i (for other details, see Chapter 4). The above model always has a feasible solution, because we always have base plans that do not use any views.

In the MQO problem, we minimize the total cost of intermediate results. Thus, we need to replace the objective function with $\sum_{t=1}^k \omega_t y_t$, where ω_t is the cost of t -th intermediate result, and remove the constraint (5.4). All other constraints remains exactly the same as in the ADR problem.

We can see that the ILP mode for MQO is very similar to the one of ADR. Thus, we can assume that we can achieve the same solution quality and scalability results for the second stage of MQO as for the second stage of ADR.

5.3 ADR Under a Maintenance-Cost Constraint

The problem we try to solve in Chapter 3 has disk space as the restraining factor, but, as we discussed in Section 2.5, in practice, the real constraining parameter is the maintenance cost.

Here is the formal definition of the ADR under the maintenance cost constraint.

- Given:
1. A set of queries.
 2. A function that, for any given set of views and indexes, returns its maintenance cost.
 3. A bound on the maintenance cost.
 4. A function that, for each query and each set of views and indexes, returns the cost of evaluating the query using these views and indexes.

Find: a subset of views and indexes that

- (i) whose maintenance cost does not exceed the bound, and
- (ii) minimizes the response time of the given query set.

We can apply our two-stage architecture for the maintenance-cost version of ADR. On the first stage, we can use the same DP-based algorithm that we proposed in Chapter 3. The only difference is that we can no longer assign weights to plans, because the weight of a plan, in general, depends on other structures that are chosen for materialization. This, in turn, means we cannot apply the pruning rules that we designed for the space-constrained version of the problem. Therefore, some alternative pruning rules are needed for version of ADR.

On the second stage, we can adapt the ILP approach that we used for the space-constrained version of ADR.

To remind the reader, the ILP model for ADR with disk space constraint looks

as follows:

$$\max \sum_{i=1}^n \sum_{j=1}^m b_{ij} x_{ij} \quad (5.5)$$

$$\sum_{j=1}^m x_{ij} \leq 1 \quad i = 1, \dots, n, \quad (5.6)$$

$$\sum_{\{j|v_t \in p_{ij}\}} x_{ij} \leq y_t \quad \forall i, t \text{ s.t. } \exists j : v_t \in p_{ij}, \quad (5.7)$$

$$\sum_{t=1}^k w_t \cdot y_t \leq B \quad (5.8)$$

The objective is to maximize the improvement (gain) in query response. A query can have at most one plan; this is expressed by the constraint (5.6). Constraint (5.7) makes sure that when a plan for a query is chosen, the views and indexes it needs are materialized. Finally, the total size of the materialized views and indexes cannot exceed the input storage limit; this is insured by the constraint (5.8).

We want to show that our branch-and-bound method described in Chapter 4 can be adjusted to be able to solve the maintenance-cost version of the problem. In fact, when we replace the disk space constraint by the maintenance-cost constraint we replace constraint (5.8) by a new constraint

$$f(y_1, y_2, \dots, y_k) \leq M \quad (5.9)$$

where $f(y_1, y_2, \dots, y_k)$ is a function that for a given set of views and indexes, returns its update cost, and M is a given maintenance-cost limit.

If we apply the Lagrangian relaxation described in Section 4.3 to this problem, we, again, get two separable subproblems. First subproblem is identical to the one in disk-space version of the problem and can be solved optimally by a simple greedy algorithm. Second subproblem is a knapsack-like problem and is, in fact, harder than the corresponding subproblem of the disk-space PSP.

$$\max \sum_{\forall(i,t)} u_{it} y_t \text{ subject to } f(y_1, y_2, \dots, y_k) \leq M. \quad (5.10)$$

To solve this problem, we need to know the structure of function f . For this, we introduce the notion of *update lattice*, which is actually a slightly modified version of

AND-OR view graph introduced in Section 2.1. Figure 5.4 shows an example of the update lattice.

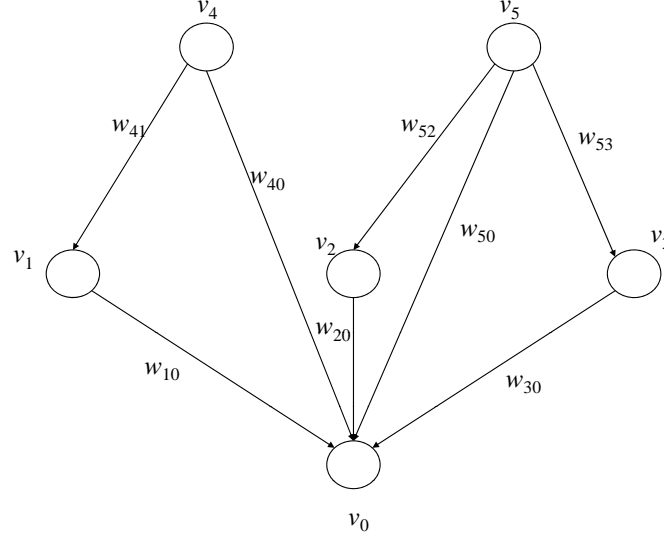


Figure 5.4: Update lattice.

Definition 8. An update lattice is a DAG $G = (V, E)$, in which nodes correspond to views and edge $e_{ij} \in E$ means that view v_i can be updated using view v_j , if v_j is chosen for materialization, w_{ij} is the cost of such update. G has one sink node v_0 that corresponds to the database. For any view $v_i \in V$ there exists an edge e_{i0} that connects this node to v_0 . For a given subset of views $V' \subset V$ chosen for materialization, the update cost of view $v_i \in V'$ is the weight of a minimum-weight path from this node to any other node from $V' \cup v_0$ in the subgraph induced by $V' \cup v_0$. The update cost of the set V' is the sum of update costs of individual views contained in it.

Thus, the problem is to find a view subset $V' \subset V$ that maximizes the objective function from (5.10) and whose maintenance cost does not exceed limit M .

The subproblem that we get after applying the Lagrangian relaxation to our problem is NP-complete, because it is a generalization of the knapsack problem. But the knapsack problem is NP-complete in the weak sense, as it admits a pseudo-polynomial algorithm. Thus, our subproblem may also have a pseudo-polynomial al-

gorithm. Therefore, our next step is to determine whether our subproblem is weakly or strongly NP-complete. To prove that our subproblem is NP-complete in the strong sense, we need to find an NP-complete problem in the strong sense and reduce it to our subproblem.

From all said above, the next step for this problem is to find the complexity of the problem obtained after the Lagrangian relaxation and, depending on the results of the complexity analysis, to develop a, possibly approximate, algorithm for solving it.

Chapter 6

Conclusion

In this work we showed that the problem of answering queries using views can be divided on a number of subproblems. The first step in the process of view-selection is to identify which view can be used to answer queries from the given set [AGK99, Ull89, PKL02, CG00]. The second step is to determine possible reformulations of the queries from the workload [CM77, CKPS95a, CV93]. The last step is choosing views that can be appropriately maintained and that minimize the processing time of the input query workload. We demonstrated that most of works concentrated on the case of data warehouses, where queries make heavy use of aggregations [GHRU97, HRU96a, Gup97, LACF05, PKL02, GM99], while the general case of the view-selection problem is not studied so good [BDD⁺98, ACN00a].

In our work, we have considered the problem of selecting views or indexes that minimize the evaluation costs of the frequent and important queries under a given upper bound on the disk space available for storing the views or indexes selected to be materialized. (In the remainder of this chapter we will refer to views and indexes collectively as *tasks*.) Thus, to solve the problem, we proposed a novel end-to-end approach that focuses on *systematic* exploration of (possibly task-based) *plans* for evaluating the input queries. Specifically, we proposed a framework (architecture, see Section 3.1) and algorithms (Chapters 3 and 4) that enable selection of those tasks that contribute to *the most efficient* plans for the input queries, subject to the space bound. We presented strong optimality guarantees on the proposed architecture. Our

proposed algorithms search for sets of competitive view-based plans for queries expressed in the language of conjunctive queries with arithmetic comparisons (CQACs). This language captures the full expressive power of SQL select-project-join queries, which are common in practical database systems. Our experimental results on synthetic and benchmark instances (see Section 3.3) corroborate the competitiveness and scalability of our approach.

Bibliography

- [AC05] F. N. Afrati and R. Chirkova. Selecting and using views to compute aggregate queries (extended abstract). In *ICDT*, pages 383–397, 2005.
- [ACG⁺99] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation*. Springer Verlag, 1999.
- [ACN00a] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, pages 496–505, 2000.
- [ACN00b] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, pages 496–505, 2000.
- [AGK99] F. N. Afrati, M. Gergatsoulis, and T. G. Kavalieros. Answering queries using materialized views with disjunctions. In *ICDT*, pages 435–452, 1999.
- [ALU01] F. N. Afrati, C. Li, and J. D. Ullman. Generating efficient plans for queries using views. In *SIGMOD Conference*, pages 319–330, 2001.
- [BC05] N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *SIGMOD*, pages 227–238, 2005.
- [BDD⁺98] R. G. Bello, K. Dias, A. Downing, J. Feenan, J. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Optimizing queries using

- materialized views: A practical, scalable solution. *Proceeding of VLDB*, pages 659–664, 1998.
- [BFP⁺72] M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest, and R.E. Tarjan. Time bounds for selection. *Journal Computer and System Sciences*, 7(4):448–461, 1972.
- [CD97] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, 1997.
- [CG00] R. Chirkova and M. R. Genesereth. Linearly bounded reformulations of conjunctive databases. In *Computational Logic*, pages 987–1001, 2000.
- [Chi02] R. Chirkova. *Automated Database Restructuring*. PhD thesis, Stanford U., 2002.
- [CKPS95a] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceeding of 11th Int. Conference on Data Engineering*, pages 190–200, Los Alamitos, CA, 1995. IEEE Computer Soc. Press.
- [CKPS95b] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, pages 190–200, 1995.
- [CM77] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 77–90, 1977.
- [CN97] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL server. In *VLDB*, pages 146–155, 1997.
- [COI] COIN-OR. <http://www.coin-or.org>.
- [CPL00] ILOG S.A. CPLEX 7.0 software package, <http://www.ilog.com>, 2000.

- [CV93] S. Chaudhuri and M. Y. Vardi. Optimization of *real* conjunctive queries. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 25-28, 1993, Washington, DC*, pages 59–70. ACM Press, 1993.
- [End72] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.
- [FGK94] R. Fourer, D. M. Gay, and B. W. Kernighan. An introduction to the AMPL modeling language for mathematical programming. *Mathematic*, 1(1):49–56, Spring 1994.
- [GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE*, pages 152–159, 1996.
- [GHRU97] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for OLAP. In W. A. Gray and P. Larson, editors, *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K.*, pages 208–219. IEEE Computer Society, 1997.
- [GKC06] G. Gou, M. Kormilitsin, and R. Chirkova. Query evaluation using overlapping views: Completeness and efficiency. In *SIGMOD Conference*, pages 37–48, 2006.
- [GL01] J. Goldstein and P. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD Conference*, pages 331–342, 2001.
- [GM99] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. *Proc. of the International Conference on Database Theory (ICDT)*, 1540:453–470, 1999.

- [GMUW02] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.
- [Gup97] H. Gupta. Selection of views to materialize in a data warehouse. In *ICDT*, pages 98–112, 1997.
- [Hal01] A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal: Very Large Data Bases*, 10(4):270–294, 2001.
- [HK70] M. Held and R. M. Karp. The traveling salesman problem and minimum spanning trees. *Operations Research*, 18:1138–1162, 1970.
- [HRU96a] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD Conference*, pages 205–216, 1996.
- [HRU96b] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, 1996.
- [HWC74] M. Held, P. Wolfe, and H. P. Crowder. Validation of subgradient optimization. *Mathematical Programming*, 6:62–88, 1974.
- [ILO04] ILOG. CPLEX Homepage, 2004. <http://www.ilog.com/products/-cplex/>.
- [JS96] T. Johnson and D. Shasha. Hierarchically split cube forests for decision support: description and tuned design. Technical Report TR1996-727, New York University, November, 1996.
- [KCFS07] M. Kormilitsin, R. Chirkova, Y. Fathi, and M. F. Stallmann. View and index selection for query-performance improvement: Algorithms, heuristics and complexity. Technical report, NC State University, 2007.
- [Klu88] A. Klug. On conjunctive queries containing inequalities. *JACM*, 35:146–160, 1988.

- [KM99] H. J. Karloff and M. Mihail. On the complexity of the view-selection problem. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania*, pages 167–173. ACM Press, 1999.
- [KP83] J. Krarup and P.M.Pruzan. The simple plant location problem: Survey and synthesis. *European Journal of Operational Research*, 12:38–81, 1983.
- [LACF05] J. Li, Z. Asgharzadeh Talebi, R. Chirkova, and Y. Fathi. A formal model for the problem of view selection for aggregate queries. In *ADBIS*, pages 125–138, 2005.
- [LMSS95] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 95–104, San Jose, California, 22–25 May 1995.
- [Loh07] G. M. Lohman. Is (your) database research having impact? In *DASFAA*, pages 3–5, 2007.
- [LSB05] X. Y. Li, M. F. Stallmann, and F. Brglez. Effective Bounding Techniques For Solving Unate and Binate Covering Problems. In *Design Autom. Conf.*, 2005.
- [LW66] E.L. Lawler and D.E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14:699–719, 1966.
- [MC79] J. M. Mulvey and H. P. Crowder. Cluster analysis: An application of lagrangian relaxation. *Management Science*, 25:329–340, 1979.
- [OL90] K. Ono and G.M. Lohman. Measuring the complexity of join enumeration in query optimization. In *VLDB*, pages 314–325, 1990.
- [PH01] R. Pottinger and A. Y. Halevy. MiniCon: A scalable algorithm for answering queries using views. *VLDB Journal*, 10(2-3):182–198, 2001.

- [PKL02] C. Park, M. Kim, and Y. Lee. Finding an efficient rewriting of OLAP queries using materialized views in data warehouses. *Decision Support Systems*, 32(4):379–399, 2002.
- [SAC⁺79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.
- [SAS] SAS-OR. <http://www.sas.com/technologies/analytics/optimization/or/>.
- [Sch86] Schrijver, A. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.
- [Sel88] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13:23–52, 1988.
- [SQL92] ISO/IEC SQL3. (ISO-ANSI Working Draft) Database Language SQL (SQL3). Document ISO/IEC JTC1/SC21 N6931 American National Standards Institute. (Later versions available from Working Group or Rapporteur Group documents.), 1992.
- [TH] TPC-H:. TPC Benchmark H (Decision Support). Available from <http://www.tpc.org/tpch/spec/tpch2.1.0.pdf>.
- [TS97] D. Theodoratos and T. K. Sellis. Data warehouse configuration. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, pages 126–135, 1997.
- [Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press, 1989.
- [ZCL⁺00] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex SQL queries using automatic summary tables. In *SIGMOD Conference*, pages 105–116, 2000.