Systematic Exploration of Efficient Query Plans For Automated Database Restructuring

Maxim Kormilitsin[1], Rada Chirkova[1], Yahya Fathi[2], and Matthias Stallmann[1]

[1] Computer Science Department
NC State University
Raleigh, NC 27695 USA
mvkormil@ncsu.edu
chirkova@csc.ncsu.edu
matt_stallmann@ncsu.edu
[2] Operations Research Program
NC State University
Raleigh, NC 27695 USA
fathi@ncsu.edu

Abstract. We consider the problem of selecting views and indexes that minimize the evaluation costs of the frequent and important queries under a given upper bound on the available disk space. To solve the problem, we propose a novel end-to-end approach that focuses on *systematic* exploration of (possibly 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 in this paper 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.

1 Introduction

Selecting and precomputing indexes and materialized views, with the goal of improving query-processing performance, is an important part of database-performance tuning. The significant complexity of this *view- and index-selection problem* may result in high total cost of ownership for database systems. In recognition of this challenge, software tools have been deployed in commercial DBMS, including Microsoft SQL Server [1–4] and DB2 [5–7], for suggesting to the database administrator views and indexes that would benefit the evaluation efficiency of representative workloads of frequent and important queries.

In this paper we propose a novel end-to-end approach that addresses the above view- and index-selection problem. Our specific optimization problem, which we refer to as *ADR* (for Automated Database Restructuring [8]), is as follows: Given a set of frequent and important queries, 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/or indexes, and cannot be executed unless all of the required views and indexes are materialized. The total size of the materialized views and indexes must not exceed a given space (disk) bound. This version of the view- and index-selection problem is NP-hard [9] and is difficult to solve optimally even when the set of indexes and views mentioned in the input query plans is small.

In dealing with the important and extensively studied problem of view and index selection under a storage bound, the novelty of our approach is twofold. First, our *problem statement* concentrates on finding efficient (possibly view- and index-based) query-evaluation plans for the input queries, as opposed to finding individual views or indexes without regard to the efficiency of the plans that they could contribute to. As such, unlike previous approaches (cf., e.g., [2, 3]), our approach quantifies the "benefits" of views or indexes to be materialized precisely by the evaluation costs of the plans they can participate in. Second, in our approach we focus on *systematic* exploration of plans for evaluating the given frequent and important queries. Specifically, we propose a framework (architecture) and algorithms that enable selection of views and indexes that contribute to *the most efficient* plans for the input queries, subject to the space bound.

Our generic 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 output (in the view/index sense[3]) is guaranteed to satisfy the input space bound. We present strong optimality guarantees on this architecture. Notably, the plans in the inputs to and outputs of the second stage are formulated as sets of IDs of views/indexes whose materialization would permit evaluation of the plans in the database. As such, stage one of our architecture encapsulates *all* the problem-specific details, such as the query language for the input queries or for the views/rewritings allowed for consideration in constructing a solution.

The specific algorithms we propose in this paper search for sets of competitive 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 algorithms generate CQAC query-evaluation plans that use CQAC views. Our experimental results demonstrate that (a) our approach outperforms that of [3] when we use our algorithms of Section 4; and (b) a CPLEX [10] implementation of stage two of our architecture is scaleable to very large problem inputs.

In the remainder of this section we discuss related work. In Section 2 we formally state problem ADR. Section 3 presents our architecture for enabling selection of views and indexes that contribute to the most efficient plans for the input queries, subject to the input space bound. Section 4 introduces our algorithms for searching for sets of competitive plans for CQAC queries. In Section 5 we present our experimental results. Finally, Section 6 summarizes our contributions and discusses generalizations of our approach.

Related Work

It is known that in selecting views or indexes that would improve query-processing performance, it is computationally hard to produce solutions that would guarantee user-specified quality (in particular, globally optimum solutions) with respect to all potentially beneficial indexes and views. In general, reports on past approaches, including those for Microsoft SQL Server [1–4] and DB2 [5–7], concentrate on experimental demonstrations of the quality of their solutions.

A notable exception is the line of work in [11–13]. Unfortunately, in 1999 Karloff and colleagues [14] disproved the strong performance bounds of these algorithms, by showing that the underlying approach of [13] cannot provide the stated worst-case performance ratios unless *P=NP*. Please see [15] for a detailed discussion of past work that centers on OLAP solutions, including [11, 13]. In this paper we focus on the problem of view and index selection for query, view, and index classes that are typical in a wide range of practical (either OLTP or OLAP) database systems, rather than limiting ourselves to just OLAP systems.

In 2000, [2] introduced an end-to-end framework for selection of views and indexes in relational database systems; the approach is based partly on the authors' previous work on index selection [16]. We have shown [17] that it is possible to improve on the solution quality of the heuristic algorithm of [2]. In this paper we focus on experimental comparisons of the contributions of this current paper with the approach of [3], which builds on [2] while focusing on a different way of both defining and selecting indexes and views. Our methods can also be combined with the approaches of [4, 18], which consider the problem of evolving the current physical database design to meet new requirements.

Papers [19, 20] by Roy and colleagues report on projects in multiquery optimization. [20] introduced heuristic algorithms for improving query-execution costs in this context, by coming up with query-evaluation plans that reuse certain common subexpressions. [19] developed an heuristic approach to finding plans for maintenance of a set of given materialized views, by exploiting common subexpressions between different view-maintenance plans. The focus of [19] is on efficient maintenance of an existing configuration of views, while we construct optimal configurations of views and indexes to ensure efficient execution of the given queries, by systematic exploration of view- and index-based plans.

Bruno and colleagues [21] proposed an algorithm that continuously modifies the physical database design as a reaction to changes in the query workload. [22] introduced a language for specifying additional constraints on the database schema. The framework proposed in [22] allows a database administrator to incorporate the knowledge of the constraints into the tuning process. Other related work includes genetic algorithms, see [23–25] and references therein. If, for example, our problem replaced the space (disk) bound with a per-unit penalty on space required by each view/index, our problem would be equivalent to that discussed in [24]. With genetic algorithms, hard constraints on feasibility such as the space limit pose one of the following difficulties: (a)

[3] To improve readability, in the remainder of the paper we focus on *view* selection. Extension to *index* selection is straightforward; see Section 6 for a discussion.

random mutations of feasible solutions are not likely to be feasible; or (b) when the constraint is incorporated into the objective function, as is often done, the final optimal solution is probably not feasible and requires an additional heuristic to make it so. Other transformation-based meta-heuristics have similar difficulties, please see [26] for a general discussion. Our problem formulation has an additional feature making it more difficult to solve: the secondary effects of any simple transformation. That is, removal of a view/index eliminates all query plans that use them, and addition of views/indexes may or may not make additional plans possible. Thus, we posit that problems (such as our problem) in presence of hard constraints are not amenable to genetic-algorithm approaches.

2 Preliminaries

Recall that our optimization problem *ADR* (for Automated Database Restructuring [8]) is as follows: 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 cannot be executed unless all of the required views are materialized. The total size of materialized views must not exceed a given space (disk) bound.

Formally, an instance of the problem ADR 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 $P = \{p_1, \ldots, p_n\}$ 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 $\Sigma_{i=1}^{n} c(p_i)$ 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., [2, 3]) 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., [27, 28]) and that the what-if optimizer module used in our architecture uses the same algorithms as the target optimizer in the database system.

For the algorithms that we introduce in Section 4, 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., [29].) Further, we make the common assumption (see, e.g., [30]) of no cross products in query- or view-evaluation plans. 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:

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

In our approach, we consider only chain views — views corresponding to answers to chain queries. Although, there are works (e.g., [30]) 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 1 clarifies the use of CQAC queries, views, and rewritings. Please see Section 6 for more general classes of the problem inputs that are covered by generalizations of the approach of Section 4.

Example 1. Suppose we are given a database with tables $A(a, b), B(b, c), C(c, d)$ and chain query

$$
\begin{array}{l}
\texttt{SELECT } A.a, B.c \\
\texttt{FROM } \quad A, B, C \\
\texttt{WHERE} \\
\qquad A.b = B.b \texttt{ AND} \\
\qquad B.c = C.c \texttt{ AND} \\
\qquad C.d \leq 100
\end{array}
$$

Possible chain views for this query would be

$$
\begin{array}{l}
\texttt{CREATE TABLE } V1 \texttt{ AS} \\
\texttt{SELECT } A.a, B.c \\
\texttt{FROM } \quad A, B \\
\texttt{WHERE} \\
\qquad A.b = B.b
\end{array}
$$

or

$$
\begin{array}{l}
\texttt{CREATE TABLE } V2 \texttt{ AS} \\
\texttt{SELECT } B.b, B.c \\
\texttt{FROM } \quad B, C \\
\texttt{WHERE} \\
\qquad B.c = C.c \texttt{ AND} \\
\qquad C.d \leq 100
\end{array}
$$

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:

$$
\begin{array}{l}
\texttt{SELECT } V1.a, V1.c \\
\texttt{FROM } \quad V1, C \\
\texttt{WHERE} \\
\qquad V1.c = C.c \texttt{ AND} \\
\qquad C.d \leq 100
\end{array}
$$

and

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

In this paper we use the term "automated database restructuring" to point out the possibility of "inventing" new views (see [8]) 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 4, our proposed algorithms (presented in that section) *are* complete in that sense.

3 The Architecture

As discussed in Section 1, 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 [31].

Our problem statement ADR, see Section 2, 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. An optimal combination of the given plans can be selected by a general Integer Linear Program (ILP) problem solver such as CPLEX [10], as will be discussed below.

As shown in Figure 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 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 [17]. While a large variety of exact algorithms and heuristics can be employed in stage two (see [17] for one approach), 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 [10], COIN-OR [32], and SAS/OR [33]. Their superiority over problem-specific heuristics and algorithms has been demonstrated in various domains, including design automation [34]. 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.[4]

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.

Our work strives to exploit the performance of CPLEX and other ILP solvers by transforming ADP 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

[4] Our collective experience bears this out. The [34] 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 [17] outperformed CPLEX 9 but not CPLEX 11, as the latter added a powerful presolver and randomized heuristics for finding feasible solutions.

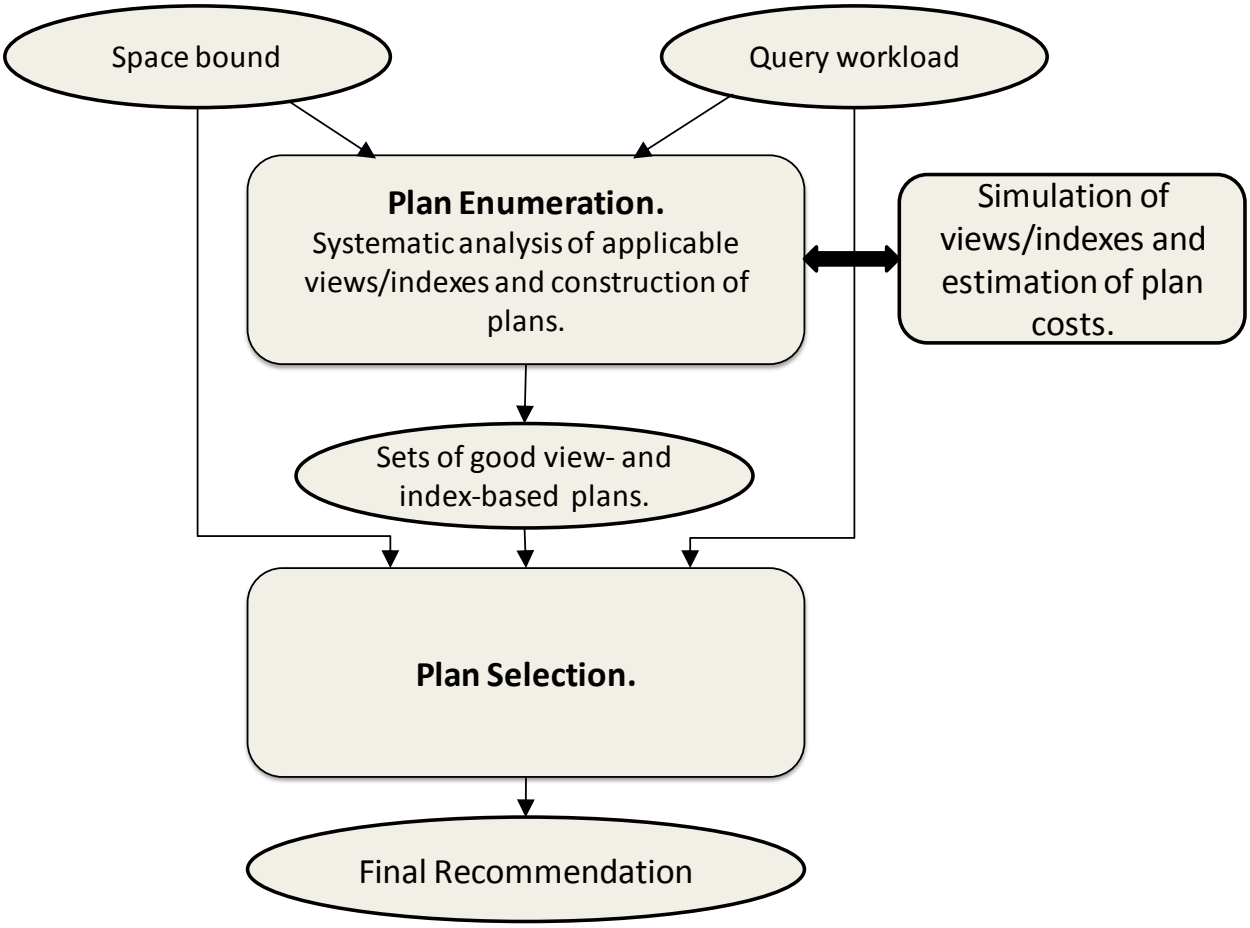


Fig. 1. Our two-stage architecture.

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 ADP 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.

Most of the remainder of the paper is devoted to implementations of stage one of our architecture for variants of the CQAC problem (Section 4), and to experiments confirming the competitiveness of our approach (Section 5). Finally (Section 6), we discuss extensions of the CQAC problem that can be handled by straightforward generalizations of our algorithms of Section 4.

4 Efficient Evaluation Plans for CQAC Queries

We present two variations of a specific algorithm implementing stage one of our architecture of Section 3. 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 5 show, the solution quality of the second approach is still quite good, optimal or almost so for small instances, superior to those of [3] for larger ones (for which the optimum is not known). Please see Section 6 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 4.1 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 4.2 we deal with the complications arising with multiple queries, and show that our algorithms still maintain optimality. At the end of Section 4.2 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 5.

4.1 Finding Efficient Plans for a Single CQAC Query

The standard System-R-style optimizer [35] 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 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* relevant[5] 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.

[5] Even 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 4.2.

We introduce our approach by way of Algorithm SingleQueryPlanGen in Figure 2. The algorithm (as well as the algorithm of Section 4.2) 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.

Algorithm 1: SingleQueryPlanGen

```
   Input: database statistics (see Section 2), CQAC query Q, space bound B
   Output: a set of all plans for Q that require at most B additional space
1  for each subquery q of Q in the order of increasing length do
2      for each split of q into two smaller subqueries q1 and q2 do
3          for each pair of plans p1 and p2 of q1 and q2 do
4              if total space required by simulated views of p1 and p2 is at most B then
5                  create plan p by joining p1 and p2;
6                  save p into the set of plans for q;
7      if the size of the answer to q is at most B then
8          simulate view v_q with the answer to q;
9          create plan p based on v_q;
10         save plan p into the set of plans for q;
11     PrunePlans(q);
12 return set of plans for Q
```

Fig. 2. Constructing (possibly view-based) evaluation plans for a single CQAC query.

In the algorithm above, 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 technics: (1) joining of plans for smaller subqueries (lines 2-6); (2) creating plans that use only one view containing the answer to the subquery (lines 7-10).

We assume that the operations in lines 4-6 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 3, 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 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 7-10, 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 proportionate to

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

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, attributes, and so on. 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 (possibly 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 PrunePlans in Algorithm 2 removes plan p from the list of plans for a subquery whenever p is dominated by another plan p'.

Example 2 illustrates the work of Algorithm SingleQueryPlanGen.

Example 2. 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 1 and suppose the database is populated as in Figure 3.

A	
a	**b**
1	2
2	4
3	4
3	5
4	6

B	
b	**c**
2	2
2	5
3	4
4	5
4	3

C	
c	**d**
2	50
3	100
4	150
5	200

Fig. 3. Sample database.

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 $C(c, d)d \leq 100$: 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 4 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.

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

Fig. 4. Plans for the 2 subqueries of length 2.

Fig. 5 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. 6 shows the actual tables and views for these three plans, ranked by increasing space bound, with brackets indicating the join order.

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

Fig. 5. 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

Fig. 6. Ranking of view-based plans for query Q.

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 8-10. 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 3 of the algorithm.

■

4.2 Finding Efficient Plans for Multiple CQAC Queries

In this subsection we discuss how to adjust the algorithm of Section 4.1 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 [36] 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 6), 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 4.1. 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 3 describes such a situation.

Example 3. Consider two queries $\texttt{Q}_1$:-ABC and $\texttt{Q}_2$:-BCD. We omit attributes, because they are not relevant here. Suppose, in the node corresponding to subgoals ABC, we have two plans $\texttt{p}_1$:-$\texttt{V}_1 \bowtie \texttt{C}$ and $\texttt{p}_2$:-$\texttt{A} \bowtie \texttt{V}_2$, where $\texttt{V}_1$:-AB and $\texttt{V}_2$:-BC; and in node BCD we have $\texttt{p}_3$:-$\texttt{V}_2 \bowtie \texttt{D}$ and $\texttt{p}_4$:-$\texttt{B} \bowtie \texttt{V}_3$, where $\texttt{V}_3$:-CD. Let

$$cost(p_1) < cost(p_2)$$
$$cost(p_4) < cost(p_3).$$

and

$$weight(V_1) < weight(V_2)$$
$$weight(V_3) < weight(V_2).$$

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:

$$cost(p_1) + cost(p_4) < cost(p_2) + cost(p_3)$$
$$weight(V_1) + weight(V_3) \ ? \ weight(V_2)$$

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 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 7, 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 $i, i+1, \ldots, j-1, j$.)

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 2), 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

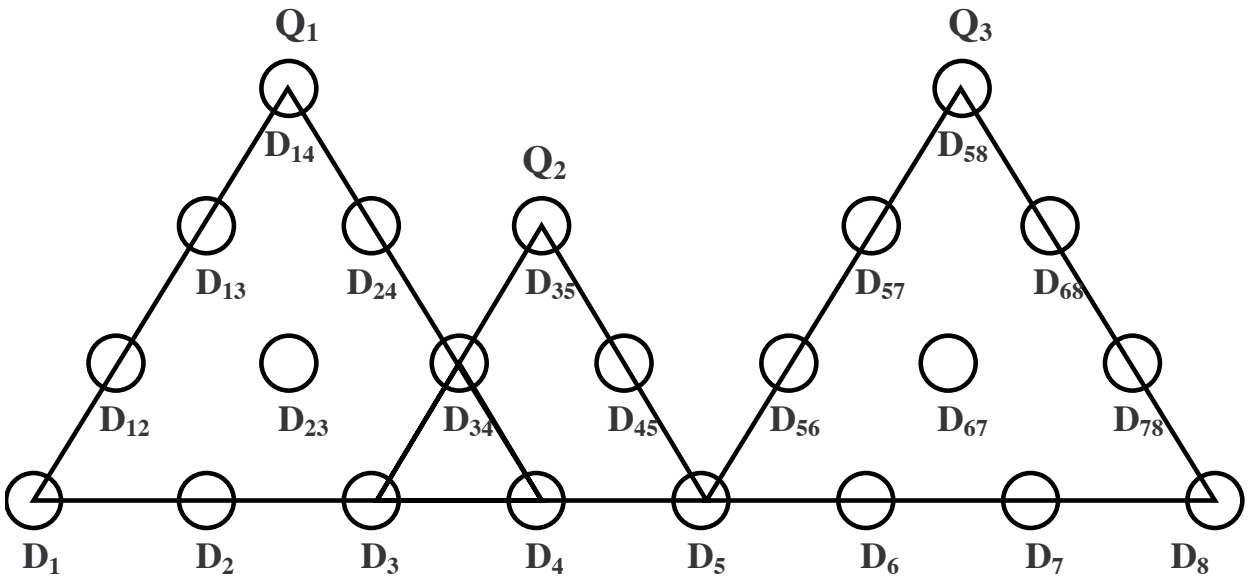


Fig. 7. Example of DP lattice for multiple chained queries. (D_{ij}, with $i < j$, denotes a chain of subgoals $i, i+1, \ldots, j-1, j$.)

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.

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 (Fig. 8). 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.

Algorithm 2: MULTIQUERYPLANGEN

Input: database statistics (see Section 2), set of CQAC queries Q that together form chain H, space bound B
Output: a set of plans for each query in Q, containing optimal solution to ADR

```
1  for each sub-chain q of H in the order of increasing length do
2      for each split q into two smaller sub-chains q1 and q2 do
3          for each pair of plans p1 ∈ plans(q1) and p2 ∈ plans(q2) do
4              if queries(p1) ∩ queries(p2) ≠ ∅ AND total weight of p1 and p2 is at most B then
5                  create plan p by joining p1 and p2;
6                  queries(p) = queries(p1) ∩ queries(p2);
7                  save p into plans(q);
8      let Q' be the set of queries for which q is the subset of tables;
9      for each k ⊆ Q' do
10         simulate view v which is the result of the join of tables in q with the disjunction of the sets of constraints
           of queries in k applied to it;
11         if the size of v is at most B then
12             create plan p based on v;
13             initialize queries(p) with indexes of queries in k;
14             save plan p into plans(q);
15     PRUNEPLANS(q);
16 return set of plans for each query in Q
```

Fig. 8. Constructing (view-based) evaluation plans for multiple CQAC queries.

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 common for m queries subchain, in the worst case, we can create 2^m different views. Thus, the overall complexity of Algorithm 8 is $O(n^3 2^{nm})$.

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 MULTIQUERYOPT. 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 MULTIQUERYOPT.

■

Theorem 2 is a very important result. It means that Algorithm MULTIQUERYPLANGEN 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 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.

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.

In Example 3, we demonstrate that the pruning rule that we used in our single-query algorithm (see Section 4.1) 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 5) suggest that the single-query rule, even if applied to the case of *multiple queries,* does not significantly reduce the quality of the solution.

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 $profit * queries/size$, 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 used by the plan.

5 Experimental Results

The experiments reported in this section address two questions: (a) How does our two-stage approach compete with a *relaxation-based approach (RBA),* such as that of [3]? (b) Can we safely assume that stage two is not the bottleneck when using CPLEX [10]? Our extensive and thorough investigation gives a positive response to each question.

We begin in Section 5.1 with a discussion of how we generate random problem instances with characteristics typical of those arising in practice. In Section 5.2 comparisons with the RBA are reported for a large number of instances. Scalability of the stage two computation using CPLEX is demonstrated in Section 5.3.

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.

5.1 Instance Generation

Query instances derived from benchmarks such as TPC-H [36] 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 on[6]; 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.

[6] Size 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 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.

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.

5.2 Comparisons with RBA

The RBA algorithm we use for comparison purposes is the one described in [3]. 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 [3] we examined the quality achieved by each approach within a specified time period. Our own implementation of [3] was used since we did not have access to the code for it.[7] 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 statistics approximation calls. Although size estimations are not the only operations performed by the optimizer, they tend to dominate the runtime of query optimization.

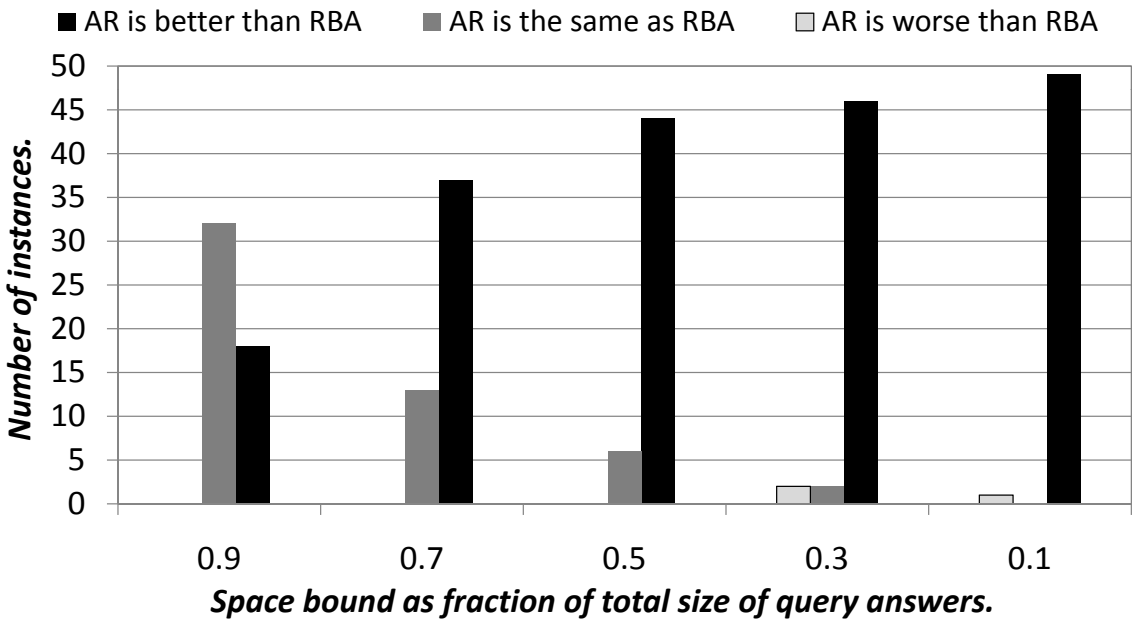
We ran the RBA algorithm on each instance until it had done as many size estimations as our MultiQueryPlanGen (Section 4) using the single-query pruning rule; we refer to this algorithm as *AR,* aggressive (pruning) rule. We then compared the solution quality of the two algorithms.[8] Fig. 9(a) shows 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. 9(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.

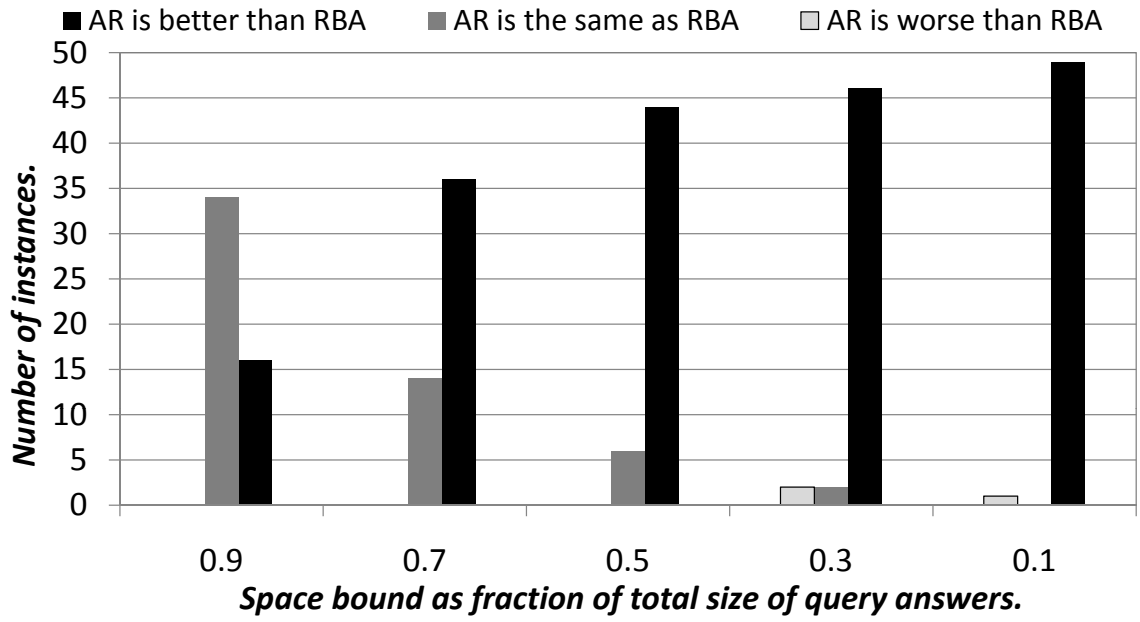
In both experiments the results differ depending on the relationship between the space bound and the total size of the query results. What both parts of the figure show is that our approach gets better relative to RBA as

[7] Microsoft Research is currently unable to distribute the research prototype externally due to IP considerations.

[8] Stage two using CPLEX is also executed following our MultiQueryPlanGen of Section 4. 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.

Fig. 9. Comparing solution quality of our AR (Section 4) versus the RBA of [3].

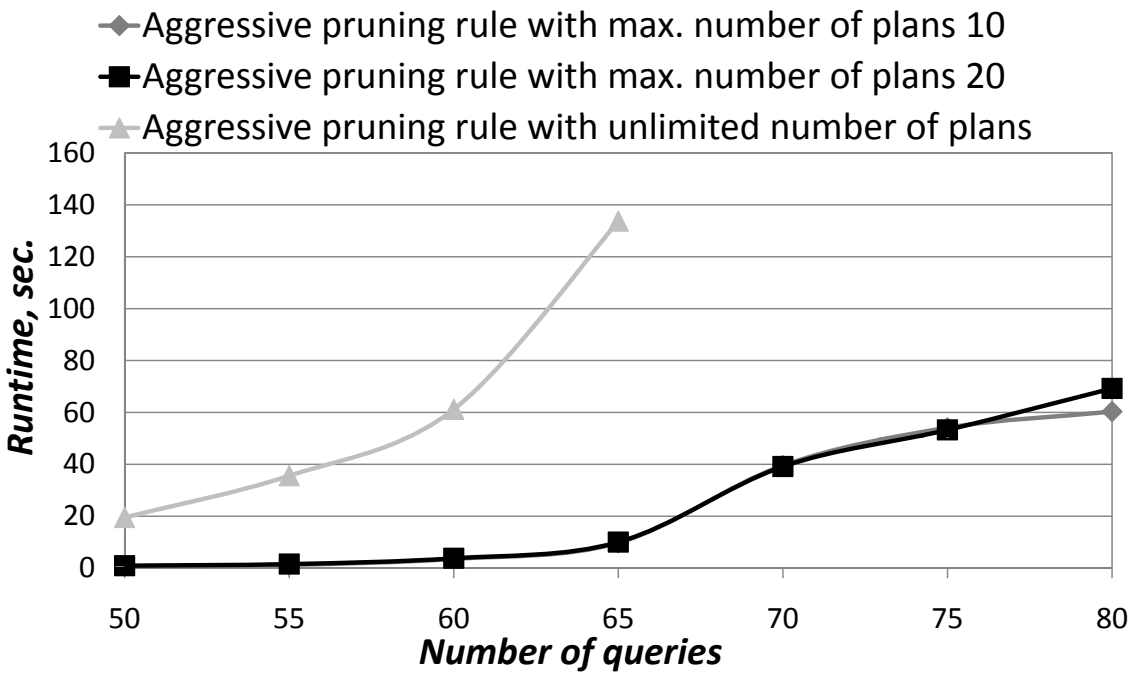


Fig. 10. Scalability of aggressive pruning rules.

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. 10 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 are competitive when it comes to solution quality.

Fig. 11 shows the performance of AR20 vs. RBA using the same setting as that of Fig. 9. 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. 12 – 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.

5.3 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. 13 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 [37], 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

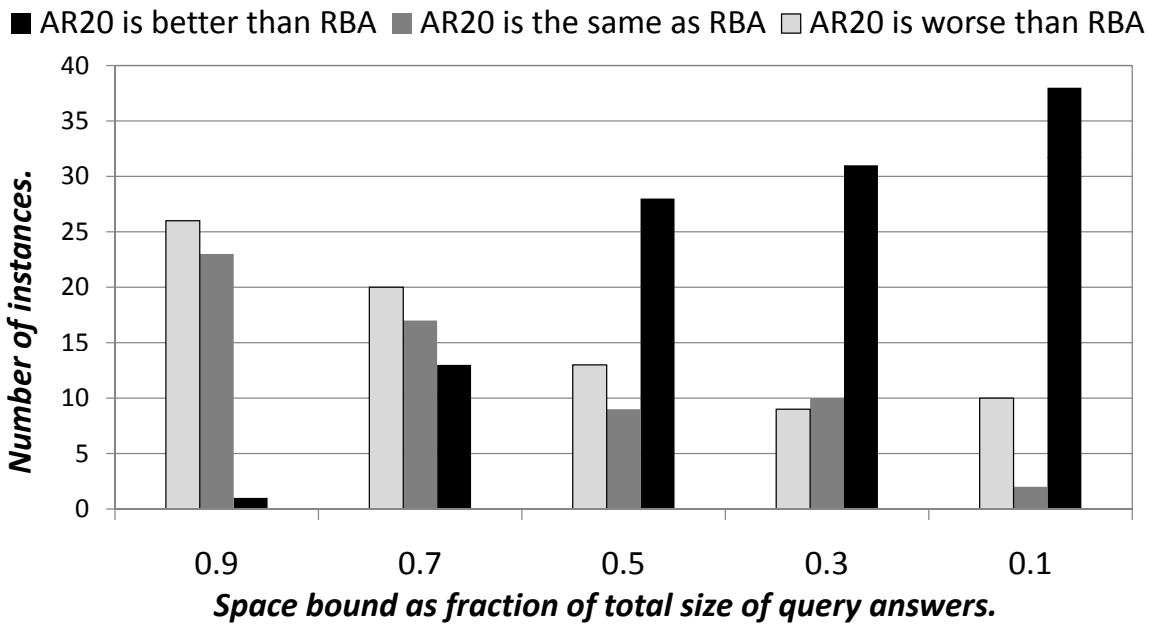


Fig. 11. AR20 versus RBA for different space bounds.

queries can be solved in a few seconds.[9] The largest instance that we were able to solve within 10 minutes using CPLEX had 800 queries, 1074 plans per query, and 6886 views.

5.4 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 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:

$$region - nation - supplier - partsupp - lineitem - orders - customer - nation - region$$

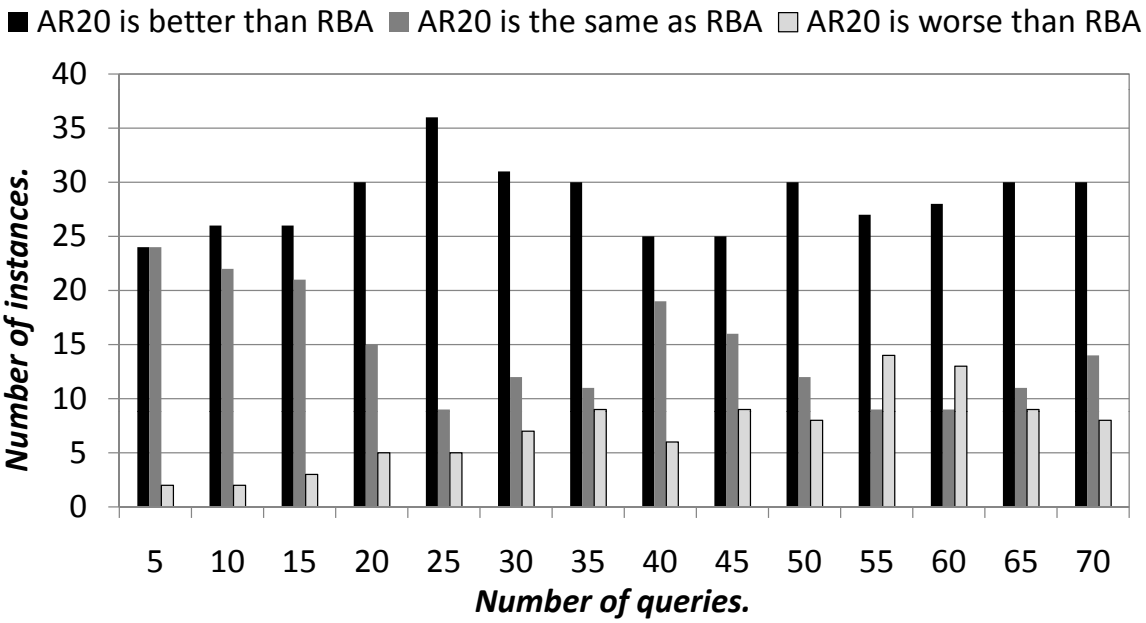
Note that this chain has two occurrences of tables *region* and *nation*. For our model, it is not important, because we treat them as different tables.

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.

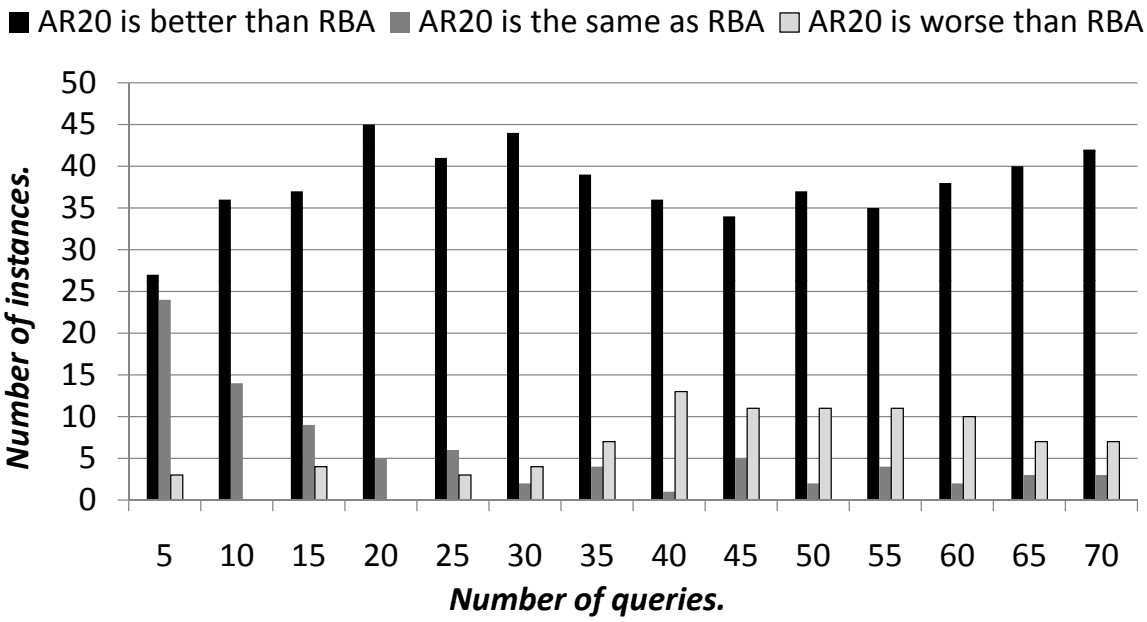
TPC-H workload consists of 22 queries. As we showed in the sections above, such instances are relatively small for our proposed algorithm. Therefore, we were able to run an experiment on 1000 TPC-H-shaped instances. The average runtime of our algorithm was 3.4 seconds. We allowed RBA to do the same number 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 14 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 14, 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.

[9] The 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.

Fig. 12. AR20 versus RBA as a function of number of queries.

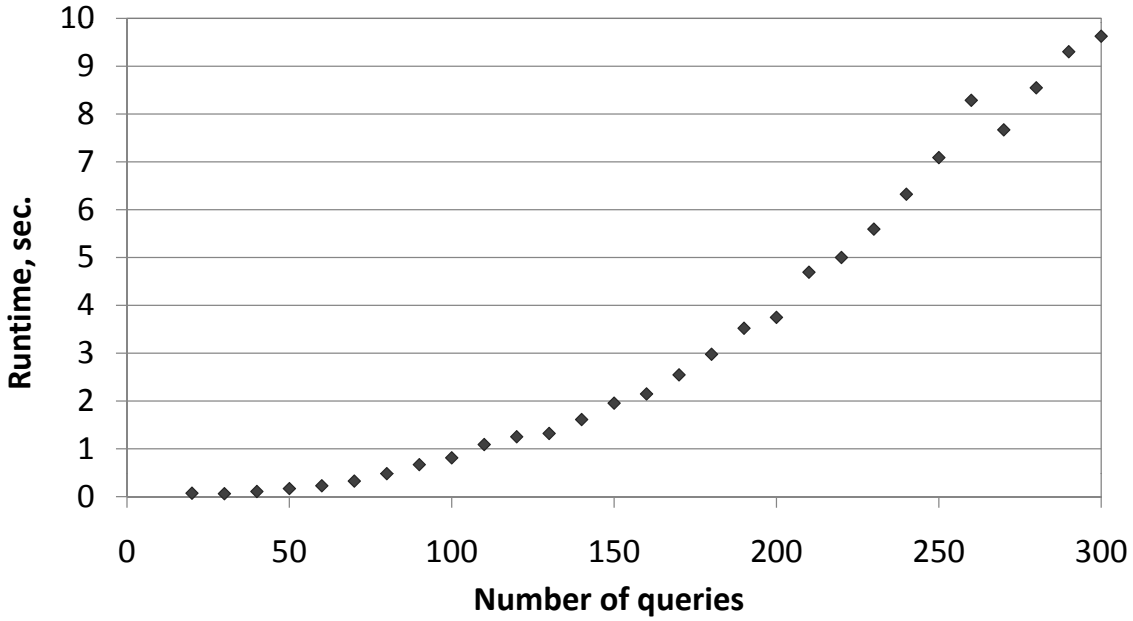


Fig. 13. Scalability of CPLEX when used to implement stage two.

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

Fig. 14. 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 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.

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

Fig. 15. Relative difference between the solution values returned by AR44 and RBA, $(S_{RBA}-S_{AR44})/max(S_{RBA}, S_{AR44})$, for randomly generated instances

Out of 1000 instances, AR44 found a better solution in 883 cases and RBA found a better solution in 106 cases.

Although Figure 15 shows the general relationship between the solutions returned by the two algorithms, the actual distribution is not clear. Figure 16 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 16(a) is based on 883 instances for which AR44 found a better solutions, Table 16(b) is based on 106 instances on which RBA was better.

relative difference, %	percentrank
0.06	1.0
4.33	0.9
8.88	0.8
12.67	0.7
16.20	0.6
20.09	0.5
23.69	0.4
27.99	0.3
33.43	0.2
40.18	0.1
73.36	0.0

(a) AR44 is better than RBA.

relative difference, %	percentrank
0.04	1.0
0.70	0.9
1.51	0.8
2.12	0.7
4.04	0.6
6.39	0.5
8.53	0.4
10.21	0.3
13.98	0.2
18.66	0.1
52.88	0.0

(b) RBA is better than AR44.

Fig. 16. Distribution of the solution relative difference between AR44 and RBA.

From Figure 16, 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.

5.5 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.

6 Discussion and Extensions

In this paper 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) and algorithms (Section 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 5) 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 4 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 4.

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 [4]: 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 4. 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 [35], 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 4.2 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 4.2, 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 4 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 4. The extension is based on the formal results in the previous work [27, 38] 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 4. Such a view may also have grouping and aggregation in it, see [38] for the local conditions for building aggregate views for subqueries of an aggreate 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.

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 [39], where one expects only aggregate queries to be asked on the stored data. When only aggregate queries are expected, then the framework of [13] 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 paper.

References

1. Agrawal, S., Chaudhuri, S., Kollár, L., Marathe, A.P., Narasayya, V.R., Syamala, M.: Database tuning advisor for Microsoft SQL Server 2005. In: VLDB. (2004)
2. Agrawal, S., Chaudhuri, S., Narasayya, V.R.: Automated selection of materialized views and indexes in SQL databases. In: VLDB. (2000) 496–505
3. Bruno, N., Chaudhuri, S.: Automatic physical database tuning: A relaxation-based approach. In: SIGMOD. (2005) 227–238
4. Bruno, N., Chaudhuri, S.: Physical design refinement: The merge-reduce approach. ACM Transactions on Database Systems **32**(4) (2007) 28–43
5. Balmin, A., Özcan, F., Beyer, K.S., Cochrane, R., Pirahesh, H.: A framework for using materialized XPath views in XML query processing. In: VLDB. (2004)
6. Valentin, G., Zuliani, M., Zilio, D.C., Lohman, G.M., Skelley, A.: DB2 advisor: An optimizer smart enough to recommend its own indexes. In: ICDE. (2000)
7. Zilio, D.C., Zuzarte, C., Lightstone, S., Ma, W., Lohman, G.M., Cochrane, R., Pirahesh, H., Colby, L.S., Gryz, J., Alton, E., Liang, D., Valentin, G.: Recommending views and indexes with IBM DB2 design advisor. In: ICAC. (2004)
8. Chirkova, R.: Automated Database Restructuring. PhD thesis, Stanford U. (2002)
9. Chaudhuri, S., Datar, M., Narasayya, V.R.: Index selection for databases: A hardness study and principled heuristic solution. IEEE TKDE **16** (2004) 1313–1323
10. ILOG: CPLEX Homepage (2004) `http://www.ilog.com/products/cplex/`.
11. Gupta, H., Harinarayan, V., Rajaraman, A., Ullman, J.D.: Index selection for OLAP. In: ICDE. (1997)
12. Gupta, H., Mumick, I.S.: Selection of views to materialize under a maintenance cost constraint. In: ICDT. (1999) 453–470
13. Harinarayan, V., Rajaraman, A., Ullman, J.D.: Implementing data cubes efficiently. In: SIGMOD. (1996)
14. Karloff, H.J., Mihail, M.: On the complexity of the view-selection problem. In: PODS. (1999)
15. Asgharzadeh Talebi, Z., Chirkova, R., Fathi, Y., Stallmann, M.: Exact and inexact methods for selecting views and indexes for OLAP performance improvement. In: EDBT. (2008)
16. Chaudhuri, S., Narasayya, V.R.: An efficient cost-driven index selection tool for Microsoft SQL server. In: VLDB. (1997) 146–155
17. Kormilitsin, M., Chirkova, R., Fathi, Y., Stallmann, M.: View and index selection for query-performance improvement: Quality-centered algorithms and heuristics. In: CIKM. (2008)
18. Gupta, A., Mumick, I.S., Rao, J., Ross, K.: Adapting materialized views after redefinitions: techniques and a performance study. Inf. Sys. **26**(5) (2001) 323–362
19. Mistry, H., Roy, P., Sudarshan, S., Ramamritham, K.: Materialized view selection and maintenance using multi-query optimization. In: SIGMOD. (2001) 307–318
20. Roy, P., Seshadri, S., Sudarshan, S., Bhobe, S.: Efficient and extensible algorithms for multi query optimization. In: SIGMOD. (2000) 249–260
21. Bruno, N., Chaudhuri, S.: Online approach to physical design tuning. In: ICDE-07
22. Bruno, N., Chaudhuri, S.: Constrained physical design tuning. PVLDB **1** (2008)
23. Horng, J.T., Chang, Y.J., Liu, B.J., Kao, C.Y.: Materialized view selection using genetic algorithms in a data warehouse system. In: Congr. Evol. Comp. (1999)
24. Kratica, J., Ljubic, I., Tosic, D.: A genetic algorithm for the index selection problem. In: EvoWorkshops. (2003) 280–290
25. Lawrence, M.: Multiobjective genetic algorithms for materialized view selection in OLAP data warehouses. In: GECCO. (2006) 699–706
26. Reeves, C.R., ed.: Modern Heuristic Techniques for Combinatorial Problems. John Wiley & Sons (1993)
27. Gou, G., Kormilitsin, M., Chirkova, R.: Query evaluation using overlapping views: Completeness and efficiency. In: SIGMOD. (2006) 37–48
28. Chaudhuri, S., Krishnamurthy, R., Potamianos, S., Shim, K.: Optimizing queries with materialized views. In: ICDE. (1995) 190–200
29. Klug, A.: On conjunctive queries containing inequalities. JACM **35** (1988) 146–160

30. Ono, K., Lohman, G.: Measuring the complexity of join enumeration in query optimization. In: VLDB. (1990) 314–325
31. Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., Marchetti-Spaccamela, A., Protasi, M.: Complexity and Approximation. Springer Verlag (1999)
32. COIN-OR. http://www.coin-or.org
33. SAS-OR. http://www.sas.com/technologies/analytics/optimiz- ation/or/
34. Li, X.Y., Stallmann, M.F., Brglez, F.: Effective Bounding Techniques For Solving Unate and Binate Covering Problems. In: Design Autom. Conf. (2005)
35. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: SIGMOD. (1979)
36. TPC Benchmark H. http://www.tpc.org/tpch/ spec/tpch2.1.0.pdf
37. Kormilitsin, M., Chirkova, R., Fathi, Y., Stallmann, M.: View and index selection for query-performance improvement. Technical report, NCSU (2007)
38. Afrati, F.N., Chirkova, R.: Selecting and using views to compute aggregate queries (extended abstract). In: ICDT. (2005) 383–397
39. Chaudhuri, S., Dayal, U.: An overview of data warehousing and OLAP technology. SIGMOD Record **26**(1) (1997) 65–74