View and Index Selection for Query-Performance Improvement: Algorithms, Heuristics and Complexity

(extended abstract)

Maxim Kormilitsin
Computer Science Dept.
NC State University
Raleigh, NC 27695 USA
mvkormil@ncsu.edu

Rada Chirkova
Computer Science Dept.
NC State University
Raleigh, NC 27695 USA
chirkova@csc.ncsu.edu

Yahya Fathi
Operations Research Program
NC State University
Raleigh, NC 27695 USA
fathi@ncsu.edu

Matthias Stallmann
Computer Science Dept.
NC State University
Raleigh, NC 27695 USA
matt_stallmann@ncsu.edu

ABSTRACT

Selecting and precomputing indexes and materialized views, with the goal of improving query-processing performance in the system, is an important part of database-performance tuning. The complexity of the view- and index-selection problem is significant and 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] and DB2 [4], 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 focus on developing a *unified quality-centered* approach to view and index selection, 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 efficient evaluation plans for the input workload queries. Each plan is represented as a set of views and indexes; thus, the set of plans in the problem input defines the search space of views and indexes whose materialization may benefit the performance of the input query workload. We show that this version of the view- and index-selection problem is NP hard, even when the set of indexes and views mentioned in the input query plans is of relatively small size. In spite of this level of complexity of the problem, we develop efficient methods that deliver user-specified quality (with respect to the *theoretically possible* quality given the input query plans) of the set of selected views and indexes. Our experimental results and comparisons on synthetic and benchmark instances demonstrate the competitiveness of our approach, and show that it provides for a winning combination with the end-to-end view- and index-selection framework of [1].

EDBT '08 Nantes, France

1. INTRODUCTION

This paper addresses 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. Our specific optimization problem 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.

In this section we provide the necessary background (Section 1.1), outline our specific contributions to solving the view- and index-selection problem (Section 1.2), and discuss related work (Section 1.3). In Section 2 we show that our optimization problem *(VISP)* is NP hard. Section 3 presents our integer linear program (ILP) for problem VISP and discusses the standard branch-and-bound (B&B) technique for solving ILP's. We discuss our approaches to finding the upper and lower bounds in B&B in Sections 4 and 5, respectively. Section 6 reports our experimental results, and we conclude in Section 7.

1.1 Background

Database-performance tuning is an important responsibility of database administrators (dba's) in enterprise-class databases. One focus of the tuning process is selecting and creating indexes and materialized views, with the goal of improving query-processing performance in the system. The complexity of this view- and index-selection problem is significant and may result in high total cost of ownership for database systems. In recognition of this challenge, software tools have been developed for suggesting beneficial views and indexes to the dba, ones that would improve the evaluation efficiency of representative workloads of frequent and important queries. The user can specify constraints that must be met by the tool, typically an upper bound on the storage (disk) space or, alternatively, indexes that must be included. Such view- and index-recommender tools are part of commercial SQL database-management systems, including Microsoft SQL Server [1] and DB2 [4].

The ultimate goal of view and index selection for query performance is to minimize the evaluation costs for a given query workload, subject to input constraints such as available storage/disk space. We adopt the standard measure of performance of a query workload, which is the sum — perhaps weighted — of evaluation costs of the workload queries. Query workloads in practice tend to include a variety of query types, such as aggregate queries on one stored relation alongside nonaggregate queries defined on joins of other stored relations. In our project on view and index selection, our goal is to develop a tool for recommending indexes and materialized views that would guarantee a certain user-defined (perhaps optimal) level of evaluation performance for such real-life workloads.

The focus of this paper 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.

1.2 Our contributions

Markl et al. [19] point out that major computational effort directed at query optimization can easily be justified: it pays off when amortized over many queries. In fact, reductions in query time from over an hour to a few seconds have been observed. We take the sting out of the major computational effort, achieving optimum solutions with reasonable computation time. Furthermore, our branch and bound approach allows for (a) excellent tradeoffs between runtime and solution quality;[1] and (b) interactive (online) response to user demand for progressively better quality guarantees.

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 [18] — 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.[2]

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 2),
- proof of NP-completeness of the decision version of our problem (Section 2),
- an integer linear program formulation that suggests a natural branch and bound solution strategy (Section 3),
- 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 and 5),
- experimental results on benchmark instances as well as on random instances of increasing size (to illustrate scalability) (Section 6), and
- specification and discussion of a practically important easy-to-solve special case (Section 6.4).

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 $\leq 2\%$ relative error with respect to the optimum on instances with 80 queries within 19 seconds. 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 (online) 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 and 5.

1.3 Related work

Recall (see, e.g., [1]) that in selecting views or indexes that would improve query-processing performance, producing solutions that would guarantee user-specified quality (in

[1]The possibility of such tradeoffs with branch and bound (B&B) has been known for many years — see, e.g., [16], but the traditional approach requires repeating the algorithm multiple times, doubling the number of iterations each time until the desired quality is achieved. In our work, the specification of an error bound actually speeds up the B&B procedure.

[2]Note that defining such "good" query-evaluation plans is not part of our framework; see [1, 8] for possible approaches to this problem.

particular, globally optimal solutions) with respect to all potentially beneficial indexes and views is a computationally hard problem. In general, the authors of the past approaches have concentrated on experimental demonstrations of the quality of their solutions. A notable exception is the line of work [9, 10, 11]. Unfortunately, in 1999 the paper [14] disproved the strong performance bounds of these algorithms, by showing that the underlying approach of [11] cannot provide the stated worst-case performance ratios unless P=NP. In [3] we provide a detailed discussion of past work that centers on OLAP solutions, including [6, 9, 10, 11]. Note that 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.

The state-of-the-art paper [1] 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 [1], implemented in Microsoft SQL Server, is based partly on the authors' previous work [5] on index selection. The contributions stated in [1] are (i) the proposed end-to-end framework for view and index selection in practical systems, and (ii) the module for building the search space of potential views and indexes for a given query workload. Interestingly, the authors of [1] do not recognize as a contribution their heuristic algorithm for selecting views and indexes from the search space built in their framework. In this paper, we experimentally show that our proposed view- and index-selection algorithm fares well compared to the heuristic algorithm of [1], which means that our algorithm is suitable for complementing the overall framework of [1], by providing the user with solution-quality guarantees on the views and indexes to be materialized.

Papers [20, 21] by Prasan Roy and colleagues present projects focusing on multiquery optimization (MQO). Specifically, [21] proposes algorithms for improving query-execution costs in this context, by materializing (as views) and reusing some of their common subexpressions. [20] discusses how to find an efficient plan for the maintenance of a set of materialized views, by exploiting common subexpressions between different view-maintenance expressions. While our approach can be extended to the MQO context, in this paper we focus on improving the evaluation costs of individual queries in presence of materialized views, and consider selecting and materializing indexes (as well as views) for this purpose. Note that even though our approach may not scale up to a very large number of query plans in the problem input, Lohman in [18] argues that for all practical purposes it is enough to consider for each query just a small number of "good enough" (in terms of evaluation costs) query plans.

Finally, in [15] Kratica and colleagues propose a genetic algorithm for minimizing the response time for a given database workload by a proper choice of indexes. While we consider query-performance improvement in presence of views as well as indexes, Kratica and colleagues examine an integer linear programming model that is very similar to ours.

2. COMPLEXITY OF THE PROBLEM

In this section we formally state our problem (2.1) and prove that it is NP-complete (2.2).

2.1 Formal Problem Statement

We first formally define our view- and index-selection problem VISP, 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.

2.2 NP-completeness of VISP

THEOREM 2.1. *VISP is NP-complete.*

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

The decision version VISP-D of VISP is as follows. Given the problem inputs of VISP 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 VISP-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 VISP-D is at least as hard as problem DkSP, i.e., NP-hard.

Now, we need to prove that VISP-D is in NP. It means, we need to show that, given a solution to VISP-D, we can verify it in the polynomial in the size of VISP-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, VISP-D is in NP, and, as we already proved that it is NP-hard, it is NP-complete. □

3. INTEGER LINEAR PROGRAMMING

In this section we formulate our view- and index-selection problem VISP as an integer linear program (ILP), and discuss the standard branch-and-bound (B&B) technique for solving ILP's.[3] The problem-specific heuristics and algorithms we present later use B&B as their basic framework. Subsection 3.1 gives a formal ILP definition of VISP; Subsection 3.2 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 paper.

3.1 An ILP model for VISP

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

$$\sum_{j=1}^{m} x_{ij} \leq 1 \qquad i = 1, \ldots, n \ . \tag{1}$$

When a plan for a query is chosen, the views and indexes it needs must be materialized:

$$\sum_{\{j \mid v_t \in p_{ij}\}} x_{ij} \leq y_t \ , \tag{2}$$

[3]For a general discussion of B&B, see [16].

In this expression p_{ij} represents the set of views and indexes in plan j for query i, and 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:

$$\sum_{t=1}^{k} w_t \cdot y_t \leq B \tag{3}$$

These constraints fully define our view- and index-selection problem VISP.

3.2 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 VISP 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 (2) go away, but the B in (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*. 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 and 5, 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 6 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 [1] when terminated early, and promising tradeoffs between runtime and solution quality.

4. 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., [12]) 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 (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* [13], 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 $i = 1, \ldots, n$; and $x_{ij} \in \{0,1\}$, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$, 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 $t = 1, \ldots, k$, which is a knapsack problem.

LaR consistently produces better upper bounds than LPR. However, the difference between the two diminishes with increasing problem size. Also, 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 6, 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.

Note that constraints (1) are present in the Lagrangian relaxation and, in fact, are the only constraints on x_{ij}'s. Thus, in the solution to the relaxation we can choose only one plan for each query. Suppose, in the solution to the lagrangian relaxation $x_{ij} = 1$. We can fix $x_{ij} = 1$ if setting it to 0 (and thus taking second best plan into the solution) reduces the upper bound so that it is $\leq$ the current lower bound. This can be done in linear in the number of plans time using our model. In the same way, if in the solution to the lagrangian relaxation $x_{ij} = 0$, 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

5. FINDING LOWER BOUNDS

In this section we discuss our proposed methods for finding lower bounds for the branch-and-bound method (discussed in Section 3) for our view- and index-selection problem VISP.

5.1 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.

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.

Algorithm 1: 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)

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

5.2 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 (5.1). 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 (5.1) to obtain a feasible solution to the problem.

Algorithm 2: 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)

begin
  for *each (i, j) such that $\bar{x}_{ij} = 1$* **do**
    check all constraints from group (2) with $\bar{x}_{ij}$ in them
    if *there is at least one violated constraint* **then**
      set $\bar{x}_{ij} = 0$
  for *each k such that $\bar{y}_k = 1$* **do**
    check all constraints from group (2) with $\bar{y}_k$ in them
    if *there is at least one constraint whose left part evaluates to one for the solution $\{\bar{x}, \bar{y}\}$* **then**
      keep $\bar{y}_k = 1$
    else
      set $\bar{y}_k = 0$
  return Greedy($\{\bar{x}, \bar{y}\}$)
end

6. EXPERIMENTAL RESULTS

In our experiments we pursued two goals. First, we show that our algorithm outputs solutions of significantly better quality than the greedy heuristic of [1] and therefore makes a suitable back end for their view- and index-selection tool. Second, we demonstrate the behavior of our algorithm with respect to runtime versus solution-quality tradeoffs and use in an interactive (online) setting. We demonstrate the former in Section 6.2, and the latter in Section 6.3. For most of our tests we used randomly generated problem instances. We explain how we generate these instances in Section 6.1. Finally, in Section 6.4 we make several additional observations, including a description of a practically important easy-to-solve special case.

For the experiments we used a PC with the Intel Core 2 1.86GHz processor and 1Gb RAM, running the Red Hat Linux operating system.

6.1 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 i to answer query j 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 [9, 11].) 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 be too complex to model.

6.2 Comparison with $Greedy(k, m)$

In this subsection we compare our algorithm with the greedy algorithm $Greedy(k, m)$ described in [1]. This algorithm exhaustively searches for an optimal subset of views and indexes of size k and then greedily 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.

Figure 1 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.

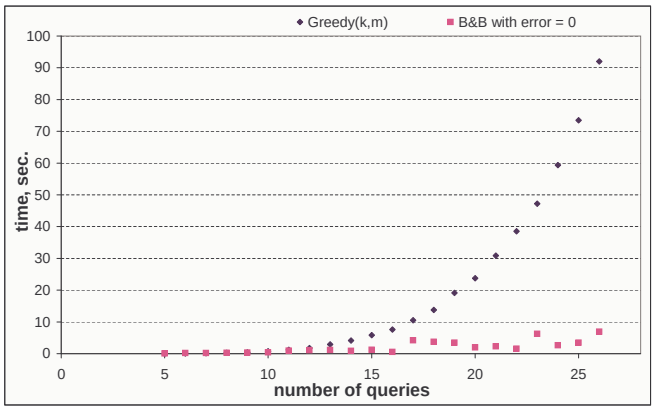

Figure 1: Scalability of $B\&B$ and $Greedy(k, m)$.

In Figure 1, 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 form 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 2 shows the average error of the solution returned by $Greedy(k, m)$. As in Figure 1, 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 2 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 3 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

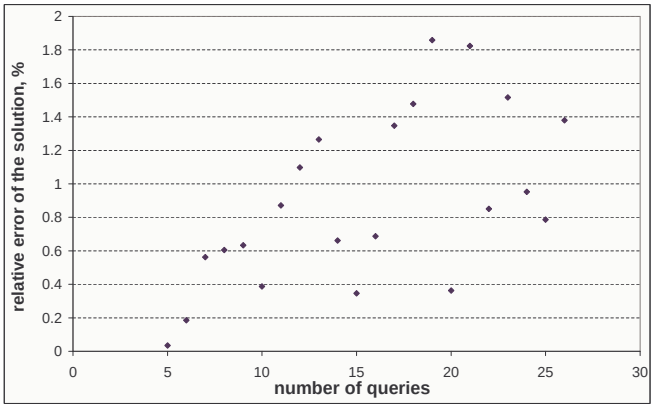

Figure 2: Quality of the $Greedy(k, m)$ solution.

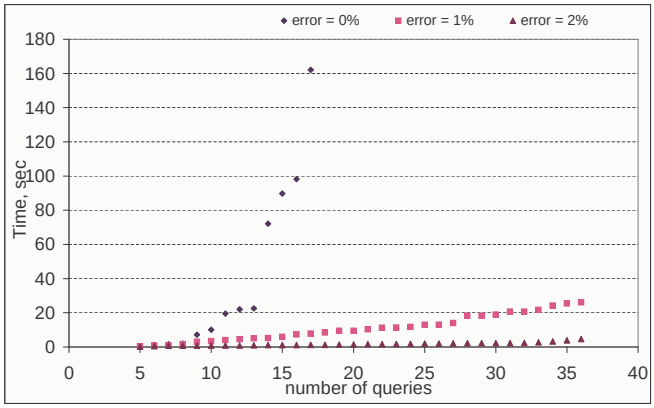

Figure 4: Scalability of $B\&B$ for various input errors.

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 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.

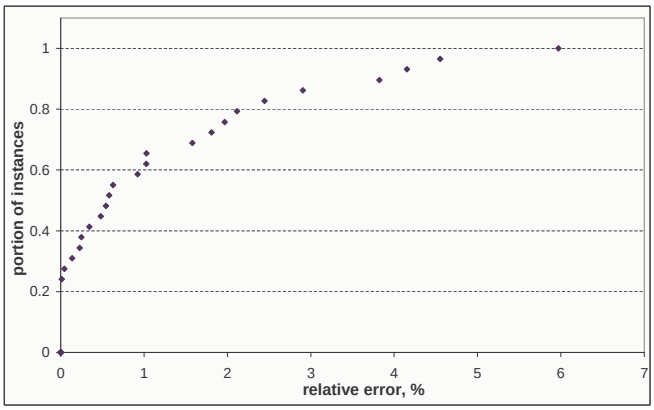

Figure 3: Error distribution for $Greedy(k, m)$.

6.3 B&B behavior

In this subsection we demonstrate the behavior of our algorithm with respect to runtime versus solution-quality tradeoffs and interactive (online) use, for a variety of input parameters.

Figure 4 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.

In Figure 4, the curve corresponding to $error = 0\%$ goes up much faster than the same curve on the plot where we compare the scalability of our algorithm with that of $Greedy(k, m)$. Recall that for the sake of comparison with $Greedy(k, m)$, we used instances with unit view/index weights, making the resulting problem instances easier. It is worth mentioning that for the errors of 1% and 2% we were able to run experiments on instances having up to 80 queries and to achieve nearly linear runtime growth.

For the next experiment, see Figure 5, 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 5 represents the error that we give our algorithm in the input, that is, the maximum allowed relative difference between the optimal solution and the solution that we can accept. 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.

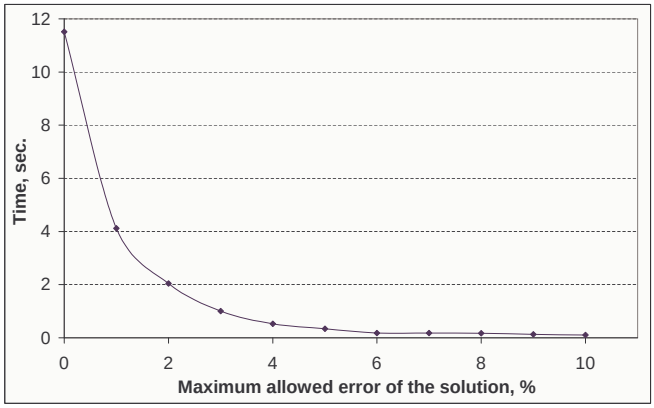

Figure 5: Runtime of $B\&B$ on a fixed instance with various input errors.

Note that the runtime drops not only because of the eas-

ier satisfiable stopping criterion, but also because we bind more variables and prune more subproblems during 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 instead of *more effective pruning.*

The goal of the next experiment, see Figure 6, is to demonstrate the interactive (online) property of our algorithm. That is, at any point of program execution we can ask it to report the solution it has at this point, together with its quality (relative difference with the 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 6 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.

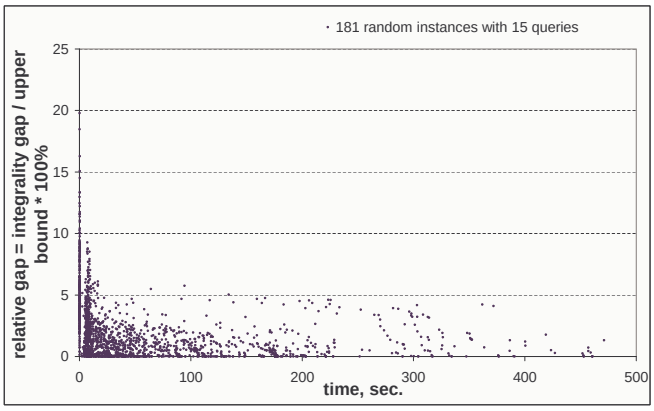

Figure 6: Gap between the lower and upper bounds.

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 6) and the one showing scalability for different errors (Figure 4). In Figure 6, the majority of points drops below the 2% level after 10–20 seconds, correspond to the point for 15 queries in Figure 4.

6.4 Other Observations

We make here a few additional observations on our algorithm and experments. First, our preliminary experiments on the TPC-H benchmark dataset [22] 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 [14] on the line of work [9, 10, 11]). Finally, we did preliminary experiments where the input plans for our algorithm were formed by applying to the TPC-H data the module of [1] for building the search space of potential views and indexes. Based on these experiments we observe that it may be likely in practice to have, as an input to our algorithm, a workload in which every view is used in only one plan. With such problem inputs the view- and index-selection problem degenerates to (well-studied) multiple-choice knapsack [7], and our algorithm is guaranteed to find a globally optimal solution in the first branch-and-bound node.

7. CONCLUSIONS

In this paper we proposed a *unified quality-centered* approach to the view- and index-selection problem. To the best of our knowledge, we are the first to adopt the solution-quality focus for this generic practical problem setting. We showed that the view- and index-selection problem in this context is NP hard, even for the small sets of input queries and plans. Despite of the level of complexity of the problem, we developed an efficient approach that finds a solution with user-specified quality, with respect to *globally optimal* quality for this query workload. Our approach is based on an integer linear programming formulation of the problem that suggests a natural branch and bound strategy with effective upper and lower bounding techniques that lead to attractive tradeoffs between time and solution quality and to interactive quality control by the user. Our experimental results corroborate the competitiveness of our approach, demonstrate a practically important easy-to-solve special case, and show that our algorithm provides for a winning combination with the end-to-end view- and index-selection framework of [1].

This project, together with our other work [2, 3], lays the foundation for studying view and index selection in a systematic principled way. (The project reported in this paper is complementary to our systematic studies [2, 17] of the OLAP view-selection problem.) In addition, our contributions make it possible, in *practical* settings, to quantify the "goodness" of specific view- and index-selection solutions with respect to the best possible (that is, *globally optimum*) counterparts, rather than just with respect to the base line where the system does not use any views or indexes. In our current and future research, we study a problem setup where the search space of relevant plans and views or indexes can be larger, possibly exponential in the size of the original problem input.

8. ACKNOWLEDGMENTS

The authors' work has been partially supported by NSF grants DMI-0321635, 0307072, and 0447742.

9. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, pages 496–505, 2000.

[2] Z. Asgharzadeh Talebi, R. Chirkova, and Y. Fathi. Exact and inexact methods for solving the problem of

view selection for aggregate queries. Technical Report TR-2007-27, NC State University, 2007.

[3] Z. Asgharzadeh Talebi, R. Chirkova, Y. Fathi, and M. Stallmann. Exact and inexact methods for selecting views and indexes for OLAP performance improvement. Technical report, NC State University, 2007.

[4] C. M. Broughton. IBM DB2 cube views and DB2 materialized query tables in a SAS environment. `http://www.sas.com/partners/directory/ibm/cubeviews.pdf`, 2005.

[5] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL server. In *VLDB*, pages 146–155, 1997.

[6] C. I. Ezeife. A uniform approach for selecting views and indexes in a data warehouse. In *IDEAS*, pages 151–160, 1997.

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[8] G. Gou, M. Kormilitsin, and R. Chirkova. Query evaluation using overlapping views: Completeness and efficiency. In *SIGMOD Conference*, pages 37–48, 2006.

[9] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for OLAP. In *ICDE*, pages 208–219, 1997.

[10] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. In *ICDT*, pages 453–470, 1999.

[11] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD Conference*, pages 205–216, 1996.

[12] M. Held and R. M. Karp. The traveling salesman problem and minimum spanning trees. *Operations Research*, 18:1138–1162, 1970.

[13] M. Held, P. Wolfe, and H. P. Crowder. Validation of subgradient optimization. *Mathematical Programming*, 6:62–88, 1974.

[14] H. J. Karloff and M. Mihail. On the complexity of the view-selection problem. In *PODS*, pages 167–173, 1999.

[15] J. Kratica, I. Ljubic, and D. Tosic. A genetic algorithm for the index selection problem. In *EvoWorkshops*, pages 280–290, 2003.

[16] E. Lawler and D. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14:699–719, 1966.

[17] J. Li, Z. A. Talebi, R. Chirkova, and Y. Fathi. A formal model for the problem of view selection for aggregate queries. In *ADBIS*, pages 125–138, 2005.

[18] G. M. Lohman. Is (your) database research having impact? In *DASFAA*, pages 3–5, 2007.

[19] V. Markl, G. M. Lohman, and V. Raman. LEO: An autonomic query optimizer for DB2. *IBM Systems Journal*, 42(1):98–106, 2003.

[20] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *SIGMOD Conference*, pages 307–318, 2001.

[21] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD Conference*, pages 249–260, 2000.

[22] TPC-H:. TPC Benchmark H (Decision Support). Available from `http://www.tpc.org/tpch/spec/tpch2.1.0.pdf`.