Plan-based View and Index Selection for Query-Performance Improvement

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 significant complexity of the 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, 2, 5, 6] and DB2 [4, 32, 34], 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. Our problem inputs include efficient evaluation plans for the input workload queries. This version of the view- and index-selection problem is NP hard [7] and difficult to solve even with a small number of indexes and views appearning in the input query plans. In spite of this, we develop efficient methods that deliver *user-specified quality* (with respect to the *best theoretically possible quality* given the input query plans) of the set of selected views and indexes. Our approach can be extended in a straightforward manner to dealing with index selection in presence of clustered indexes, as well as to handling updates in the input workload. Our experimental results and comparisons on synthetic and benchmark instances demonstrate the competitiveness of our approach, and show that it provides a winning combination with end-to-end view- and index-selection frameworks such as those of [2, 5].

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, which we refer to as *VISP* (for View and Index Selection), is as follows: Given a set of possible plans for each query, choose a subset of plans that provides the greatest reduction in query costs. 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 materialized views and indexes must not exceed a given space (disk) bound. This version of the view- and index-selection problem is NP hard [7], and is difficult to solve optimally even when the set of indexes and views mentioned in the input query plans is small.

Our problem statement for view and index selection does not require any information about the input plans other than (just the IDs of) the views or indexes in the plans, 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, including relational and XML), 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 any cost model that most suits the application.

In this section we provide the background and outline our contributions to solving the view- and index-selection problem VISP (Section 1.1), and discuss related work (Section 1.2). Section 2 presents an integer linear program (ILP) for VISP, and discusses the standard branch-and-bound (B&B) technique for solving ILPs. In Section 2 we also outline extensions of our approach to index selection in presence of clustered indexes (cf. [7, 27]) and to view and index selection under the maintenance-cost constraint [13, 14, 29]. We discuss our methods for finding the upper and lower bounds in B&B in Sections 3 and 4, respectively. Section 5 discusses complements and extensions of our approach that enable its practical use in database tuning, including construction of input query plans and handling of updates in the input workload. Section 6 reports our experimental results.

1.1 Motivation and contributions

Database-performance tuning is an important responsibility of database administrators (dba's) in enterprise-class databases. One focus of the tuning is selecting and creating indexes and materialized views, with the goal of improving query-processing performance in the system. The complexity of this 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, to improve the evaluation efficiency of representative workloads of frequent and important queries. Users 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 database-management systems, including Microsoft SQL Server [1, 2, 5, 6] and DB2 [4, 32, 34].

In our formulation of the problem of view and index selection, the inputs include the query workload of interest and the amount of available disk space for the views and indexes to be materialized. We adopt the standard measure of performance of a query workload, which is the sum — perhaps weighted — of evaluation costs of the workload queries.

We assume that each problem instance specifies one or more evaluation plans for each query. Each plan is viewed by our approach as just a set of candidate views and indexes that provides acceptable — "good enough" in the sense of [25] — 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. (See Section 5 for a discussion of the possible preprocessing algorithms.)

This version of the view- and index-selection problem is known to be NP hard [7]. To mitigate the complexity of the problem, we develop efficient methods that deliver user-specified quality. Here, quality means proximity to the *globally optimum* performance for the input query workload given the input query plans. Note that, while authors of past projects have generally opted for polynomial-time heuristics with no quality guarantees (see Section 1.2), we show experimentally in Section 6 that our methods fare better than well-known past work w.r.t. *scalability,* in addition to providing *solution-quality guarantees* for realistic-size instances.

Query workloads in practice tend to include a variety of query types, such as aggregate queries on one stored relation (familiar from the OLAP setting [8, 12]) alongside nonaggregate queries defined on joins of other stored relations. To the best of our knowledge, we are the first to propose a tool for recommending indexes and materialized views that would guarantee a certain user-defined (perhaps optimum) level of evaluation performance for such real-life workloads. That is, the solutions proposed in this paper are agnostic of the structural properties of the queries, views, indexes, and query plans, and, as such, are appropriate for a variety of practically important query, view, and rewriting languages, including SQL select-project-join queries with arithmetic (inequality) comparisons or with grouping and aggregation. The reason we are able to provide such guarantees is that our approach accepts query plans as inputs and hence does not need to look into the internal structure of queries, indexes and views.

Our main contributions are as follows:

- a problem statement (Section 2) that is flexible in the sense of being adaptable to the full spectrum of data models and query languages, as well as to a variety of constraints, including the storage-limit constraint and the maintenance-cost constraint;
- 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 3 and 4); our solution-quality guarantees hold for all cases of the storage-bounded problem VISP, as well as for some practically important special cases of view and index selection under the maintenance-cost constraint (see Section 2);
- a discussion of the place of our approach in end-to-end frameworks, such as those of [2, 5] (Section 5); and
- experimental results on benchmark instances as well as on random instances of increasing size, to illustrate scalability (Section 6).[1]

Our experiments show that the proposed approach achieves globally optimum solutions with reasonable computation time for realistic-size problem instances. Furthermore, our approach allows for (a) excellent tradeoffs between runtime and solution quality; and (b) interactive (online) response to user demand for progressively better quality guarantees.

The *runtime versus solution quality tradeoff is extremely important.* While no approach can guarantee optimum solutions in reasonable time with increasing instance size unless $P = NP$, we are able to guarantee $\leq 2\%$ relative error with respect to the optimum on realistic-size instances, with acceptable runtimes. 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. Also note that we allow users to 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 [19] on the line of work in [12, 13, 15], please see Section 1.2 for a short discussion.)

Alternatively, our algorithm can be run in an *interactive (online) setting,* where it behaves as follows. It begins under the assumption that it is seeking an optimum solution. As soon as it finds a feasible solution, it reports how close that solution is to an optimum one (based on known bounds), 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 3 and 4.

1.2 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 concentrate on experimental demonstrations of the quality of their solutions. A notable exception is the line of work in [12, 13, 15]. Unfortunately, in 1999 Karloff and colleagues [19] disproved the

[1]In our experiments we have solved problem instances with up to 200 queries for the error-free version of the general case, and with up to 1000 queries for either special cases or for those instances where the user-defined error is greater than zero while staying in the 1-3% range.

strong performance bounds of these algorithms, by showing that the underlying approach of [15] cannot provide the stated worst-case performance ratios unless $P=NP$. Please see [3] for a detailed discussion of past work that centers on OLAP solutions, including [12, 15].

The problem VISP considered in this paper is related to the problem of index only (as opposed to index *and* view) selection, with the focus on selecting clustered as well as nonclustered indexes for the stored data, see [7, 27] and references therein. Given the theoretical complexity of each problem, we believe that solving the two problems independently (i.e., selecting first the clustered/nonclustered indexes on the stored data, and then selecting any auxiliary views and indexes to further improve the processing efficiency of the representative queries) improves the odds of developing algorithms with better quality guarantees for each problem. Please see Section 2.2 for a further discussion, and Section 6 for comparisons of our approach to those of [7, 27].

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 [9]. The contributions stated in [2] include (i) an end-to-end framework for practical view and index selection, and (ii) a module for building the search space of candidate views and indexes for the input queries. Note that the authors of [2] do not recognize as a contribution their heuristic algorithm for selecting views and indexes from that search space. We show (Section 6) that our proposed algorithm fares well in comparison to the heuristic algorithm of [2]. Thus, our algorithm is suitable for complementing the framework of [2], by providing the user with solution-quality guarantees on the views and indexes to be materialized. In Section 5 we discuss a potential use of our methods within the approach of [5], 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 [6, 11], which consider the problem of evolving the current physical database design to meet new requirements.

Papers [26, 30] by Prasan Roy and colleagues report on projects in multiquery optimization (MQO). Specifically, [30] proposes algorithms for improving query-execution costs in this context, by materializing (as views) and reusing certain common subexpressions. [26] discusses how to find an efficient plan for the maintenance of a set of materialized views, by exploiting common subexpressions between different view-maintenance plans. While our approach can be extended to the MQO context, we focus on improving the evaluation costs of *individual queries* (as opposed to MQO) in presence of materialized views, and consider selecting and materializing indexes (as well as views) for this purpose.

Genetic algorithms have had some success in situations where there are no hard constraints (see, e.g., [18, 21, 23] 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 [21]. With genetic algorithms, the difficulty posed by a hard constraint is that either (a) when feasibility is taken into account, the solutions arrived at by transforming the current solutions are mostly infeasible, or (b) when the constraint is built into the objective function, the resulting genetic-algorithm solution is highly likely to be infeasible, requiring a heuristic similar to that proposed in Section 4 to make it feasible. Other transformation-based meta-heuristics have similar difficulties, please see [28] 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 VISP) in presence of hard constraints are not amenable to genetic-algorithm approaches.

2. INTEGER LINEAR PROGRAMMING

This section establishes the theoretical context of our work. We begin with a specification of our view- and index-selection problem VISP, which can be shown to be NP hard via a straightforward reduction from the problem of [7]. Then we reformulate VISP as an integer linear program (ILP, Section 2.1), and outline the modifications (Section 2.2) needed to use the ILP to solve the problem of index selection in presence of clustered indexes, as well as the problem of view and index selection under the maintenance-cost constraint. We also discuss (Section 2.3) the branch-and-bound (B&B) technique [22] for solving ILPs. The problem-specific heuristics and algorithms that we present in Sections 3 and 4 use B&B as their basic framework.

In our view- and index-selection problem VISP, inputs include (efficient) evaluation plans for the input workload queries.[2] Each plan is represented as a set of views and (or) indexes; thus, the set of plans in the problem input defines the search space of candidate views and indexes. Each plan has an associated evaluation cost. Each input view or index has an associated size, measured as the amount of disk space required to materialize this view or index. (The plan costs and view/index sizes can be obtained by a what-if optimizer [2, 5].) Finally, the problem input includes two constraints: (1) a storage limit (the amount of disk space available to store the output views or indexes), and (2) a user-defined error bound on the quality of the output. The output of VISP is a set of query plans that minimize the evaluation costs for the input query workload subject to the constraints; this output defines the views and indexes to be materialized. We adopt the standard measure of performance of a query workload, which is the sum (perhaps weighted) of evaluation costs of the workload queries. A formal problem statement for VISP can be found in [20].

2.1 An ILP model for VISP

The ILP model presented in this paper continues our line of work [3, 24] on formal systematic exploration of view- and index-selection problems. Similar ILPs have been developed for index selection in presence of clustered indexes [7, 27]. Our ILP model uses the following 0/1 variables: x_{ij} is 1 when the j-th plan (p_{ij}) is chosen for query i, 0 otherwise; y_t is 1 if the t-th view or index (v_t, of size w_t) is materialized, 0 otherwise. Here, each input plan p_{ij} (jth plan

[2]A discussion of the place of VISP in a practical end-to-end database-tuning framework can be found in Section 5.

for query i) is represented by a set of all views and indexes required to execute query i. 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-evaluation cost when plan p_{ij} is chosen for query i, subject to

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

$$\sum_{\{j|v_t \in p_{ij}\}} x_{ij} \leq y_t \qquad \forall\,(i,t) \tag{2}$$

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

Here, constraint (1) says that a query can have at most one plan. When plan p_{ij} is chosen for query i, all views/indexes in p_{ij} must be materialized. One way to state this is $x_{ij} \leq y_t$ for all i, j, t. However, taking (1) into account, we only need a single constraint (2) for each query and view/index. Recall that $v_t \in p_{ij}$ means that the j-th plan for query i requires view/index v_t. That is, if v_t is not selected ($y_t = 0$), none of the plans using v_t can be chosen (all $x_{ij} = 0$ where $v_t \in p_{ij}$). Constraint (3) says that the total size of the selected views/indexes cannot exceed the input storage limit B.

These constraints fully define our view- and index-selection problem VISP. Note that while not being an essential part of the problem formulation, the option of having the user-defined error bound in the input does make our algorithms more powerful than some of the previous approaches.

2.2 Extensions to related view- and index-selection problems

While our focus in this paper is on selecting and precomputing indexes and materialized views for the case where the layout of the original stored data (including clustered indexes) is already in place, our ILP can be modified to model the problem of index *only* selection, with the focus on selecting clustered as well as nonclustered indexes. Please see [7, 27] for a discussion of the constraints needed to modify the ILP. While the required modification of the ILP is not a contribution of this paper, we argue that the close relationship between the ILPs for the two problems permits an exploration of explicit solution-quality guarantees of the index-selection problem of [7, 27], akin to the guarantees that we provide in this paper for VISP. Providing such solution-quality guarantees for index selection in presence of clustered indexes is a direction of our current work.

While our proposed ILP would have to be modified as discussed above to model index selection in presence of clustered indexes, we do *not* have to modify our ILP to model another class of problems of practical interest. We are referring here to a family of special cases of the problem of view and index selection under the maintenance-cost constraint, see [5, 13, 33] and references therein. In this problem class, our ILP[3] correctly models all special cases for which there are no "update-cost dependencies" between the candidate views or indexes to be selected.These special cases include the practically important problems of selecting, under the maintenance-cost constraint, (1) (nonclustered) indexes only, or (2) views only for the cases where it is not effective to merge (in the sense of [2, 5]) the definitions of different views. The fact that we can use our ILP to model all these cases of view and index selection under the maintenance-cost constraint means that we extend to all these cases our solution-quality guarantees provided in this paper. (This guarantee is in contrast with the quality guarantees of the approach of [13], see Section 1.2 for a discussion of [13, 19].)

[3]Note that the input "sizes" of views/indexes would need to be interpreted as update costs, ditto for the storage limit.

2.3 Branch and bound

Branch and bound (B&B) is a well-known approach that obtains optimum solutions to ILPs at the expense of worst-case exponential runtime. Its effectiveness relies on the assumption that the worst case occurs only rarely in practice. The algorithm starts with the root node of a tree, which represents the initial problem instance. Other nodes represent smaller instances based on fixed variable assignments. (E.g., if $y_t = 0$, the instance has one fewer view/index; if $y_t = 1$, 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 values 0/1 of a fixed variable. The leaves of the tree arise when the node assignment violates the constraints (i.e., is *infeasible*) or when all variable values have been fixed, in which case the assignment is feasible but not necessarily optimal.

With no bounding, B&B is a search of *all* feasible solutions. A subtree can be pruned if the best gain its root can achieve (its *upper bound*) is no better than the gain of a known feasible solution, the global *lower bound.* The success of B&B in quickly obtaining optimal solutions relies on the quality of heuristics for obtaining upper and lower bounds. Our approaches to these are discussed in Sections 3 and 4.

A node is *processed* when its bounds have been computed and its children (when feasible and upper bound > lower bound), have been created. A node's lower bound, when greater than the current global bound, 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 the active nodes. This integrality gap bounds the *relative error* of the current best solution. If *current* is the gain of the current solution and *opt* is that of the optimal solution, the relative error is defined as $(opt - current)/opt$.

Our experimental results (Section 6) show that our upper- and lower-bound computations yields good scalability with increasing instance size, better solution quality as compared with the heuristic [2] when terminated early, and promising tradeoffs between runtime and solution quality.

3. FINDING UPPER BOUNDS

Upper bounds are essential to cut off subtrees of the 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) turns the ILP into an ordinary linear program (LP) by relaxing the constraints that force variables to be integers. In our problem, the constraints that x_{ij} and y_t are 0/1 variables are replaced by $0 \le x_{ij} \le 1$ and $0 \le y_t \le 1$. An optimal LP solution at a node that has all 0/1 values is a potential lower bound. Otherwise, the value of the objective is an upper bound on the optimum value with 0/1 values. This relaxation is used by general-purpose ILP solvers (e.g, CPLEX).

Lagrangian Relaxation (LaR) (see, e.g., [16]) 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 on the original objective function. To get the best possible upper bound we want to find a choice that minimizes the objective.

We use an iterative process called *subgradient optimization* [17]. 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 problem using the u_{it}'s (initially arbitrary) and, if the solution is not optimal, adjusts the u_{it}'s to increase the penalty for the unsatisfied constraints. While there is no best way to choose step size in the adjustment, it is usually started at a fixed value (we use 2.0) and halved when 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, deduced from our preliminary experiments, 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 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 $\sum_{j=1}^{m} x_{ij} \le 1$, 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 $\sum_{t=1}^{k} w_t \cdot y_t \le B$, $y_t \in \{0, 1\}$, 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:

- in an effective lower-bound heuristic (Section 4),
- to fix values of some variables, thus reducing the size of the B&B tree (see Variable Binding below),
- to significantly decrease runtime when a given approximation ratio is desired, see Section 6, and
- to use 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 initial u_{it}'s at its children.

Variable Binding. Lagrangian relaxation allows us to use one additional trick, applicable in any node of the B&B tree to reduce subproblem size. 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 that in the solution to the lagrangian relaxation $x_{ij} = 1$. We can fix $x_{ij} = 1$ if setting it to 0 (and thus taking the second best plan into the solution) reduces the upper bound so that it does not exceed the current lower bound. This can be done in time linear in the number of plans. 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 does not exceed the current lower bound.

4. FINDING LOWER BOUNDS

In this section we discuss our proposed methods for finding lower bounds for the branch-and-bound method (discussed in Section 2) for our view- and index-selection problem VISP.

4.1 Greedy algorithm

To explain this algorithm, it is easier to talk in terms of views/indexes and plans. In 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, to maximize 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 query plans that maximizes the total benefit. To do this, for each query we 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: Algorithm *Greedy*($\{\bar{x}, \bar{y}\}$)

```
Input  : ILP formulation of the original problem;
         a (possibly trivial) feasible solution $\{\bar{x}, \bar{y}\}$
Output : feasible solution to original problem (candidate lower bound)
begin
    Let $k$ be the maximum number of views and
    indexes that 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
```

4.2 Lagrangian heuristics

We now describe the Lagrangian heuristics that we use to obtain lower bounds. This algorithm transforms a solution to the Lagrangian relaxation into a feasible solution, using the heuristic of Section 4.1. The idea of the Lagrangian heuristics is to take a solution to the Lagrangian relaxation of the original problem (in general not a feasible solution) and to 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., the corresponding $y_t = 0$), we simply remove the assignment of plan j to query i (i.e., $x_{ij} = 0$). This process yields a feasible solution to the original problem. We then remove every unused view/index by setting its $y_t = 0$, and use the available space according to the heuristic of Section 4.1 to obtain a feasible solution to the problem.

Algorithm 2: Lagrangian Heuristics

```
Input  : Solution to the lagrangian relaxation $\{\bar{x}, \bar{y}\}$;
         ILP formulation of the original problem
Output : feasible solution to original problem (candidate lower bound)
begin
    for each $(i, j)$ such that $\bar{x}_{ij} = 1$ do
        check all group (2) constraints with $\bar{x}_{ij}$;
        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 group (2) constraints with $\bar{y}_k$;
        if there is at least one constraint whose
        left-hand side evaluates to 1 for solution $\{\bar{x}, \bar{y}\}$
        then
            keep $\bar{y}_k = 1$;
        else
            set $\bar{y}_k = 0$;
    return Greedy($\{\bar{x}, \bar{y}\}$)
end
```

5. USING OUR APPROACH IN PRACTICE

Our problem formulation is a useful abstraction of an important component of the view- and index-selection problem, for which we can deliver explicit quality guarantees on the solutions provided by our approach. In this section we discuss how our methods can be built into an end-to-end framework for selection of views and indexes for efficient query processing in database systems in practice.

5.1 Obtaining input query workloads

In all past work that we know of on view or index selection, the authors assume knowledge of the representative query workloads, including knowledge of the frequency of updates on the stored data. (Please see Section 5.3 for a more detailed discussion of the issue of updates.) It is generally assumed that a workload of representative queries and updates can be developed by a skilled dba; please see [6] for a range of approaches. When it is not practical to use our algorithm to recompute from time to time beneficial view/index configurations from scratch, the methods of [6, 11] can be used to adapt the view and index definitions and contents to the changes in system requirements over time.

5.2 Obtaining candidate query plans

Note that even though our proposed algorithm may not scale up to an extremely large number of query plans in the problem input, Lohman in (the slides for) [25] argues that for all practical purposes it is enough to consider for each query just a small number of "good enough" (w.r.t. evaluation costs) query plans. The methods of [2, 5, 10] can be used to generate good-quality [25] query plans for the query workloads in our problem inputs. (See note in the beginning of Section 6 on using [2] in our TPC-H [31] experiments.) In addition, in our ongoing work we have developed algorithms for ranking large numbers of view- and index-based query plans based on the plan costs. Our current results show that it is possible to efficiently obtain a small number of competitive evaluation plans for workloads involving a large practically important class of SQL queries.

5.3 Handing updates in the input workload

In general, the class of space-bounded problems (e.g., our VISP here) and the class of problems under the maintenance-cost constraint are not interchangeable, and each problem class is known to be hard to solve formally [13, 19, 29]. In practice, it is typical to consider "compromise" approaches (see, e.g., [2, 5, 33]). Our problem VISP can be modified in this spirit, by including update statements into the input workload and by adding the maintenance-cost constraint into the objective function of our ILP model of Section 2.1 (cf. [27]). Determining the solution-quality guarantees for this modification of the problem is part of our current work.

5.4 Incorporating our approach into end-to-end database-tuning frameworks

In Section 6 we show that our proposed algorithm fares well in comparison to the heuristic algorithm of [2], which means that our algorithm is suitable for complementing the overall framework of [2] by providing the user with solution-quality guarantees on the selected views and indexes. We now discuss how using our approach can enhance the quality guarantees of database tuning in the framework of [5].

In the line of work of [1, 2, 9, 5, 6, 7], Bruno and Chaudhuri in [5] focus on the following problem: As heuristic-based methods for physical database tuning become more complex, it is increasingly difficult to analyze, evolve, and add new algorithmic features without risk of regression. [5] addresses this problem by proposing a new framework for physical design, where the search space of views and indexes is produced by a "what-if" optimizer,[4] and then search is performed in this space for an output view/index configuration. Under the constraints on both (1) the runtime of the search algorithm of [5], and (2) the input storage limit and update costs for the solution, the algorithm outputs a configuration that provides the best workload-performance quality among the

[4]The search space includes each view or index that the optimizer deems "interesting" for any workload query.

configurations explored so far. The search is performed by iterative dropping or merging some views/indexes, possibly with backtracking to states that have already been explored.

To the best of our understanding, [5] does not list *global* solution quality among the contributions. On the other hand, our proposed algorithm has excellent global quality guarantees and, moreover, is directly compatible with the desirable principles (of what-if APIs and dependence on the optimizer) listed in [5]. Thus, we believe that the practical benefits of the framework of [5] on very large problem inputs could be greatly enhanced by using our approach. Specifically, our algorithm can be used instead of the search strategy of [5], as soon as the size of the view/index configuration can be handled by our approach.[5] (Sections 5.2–5.3 provide a discussion of the required preprocessing.)

6. EXPERIMENTAL RESULTS

Our experiments focus primarily on the advantages of Lagrangian relaxation, both in the context of B&B and as part of a heuristic. Direct comparison with other work in the literature is not possible or even appropriate in most cases.[6] Often (as pointed out in Sections 1.2 and 5) there are either fundamental differences in problem formulation, or the other work complements and is orthogonal to ours. Other times, the instances used in related work are not available or are too small to demonstrate the scalability of our approach. We return to these points in Section 6.5 and draw what conclusions we can. Note that our comparisons with those past projects may be partly relevant despite the differences.

We did preliminary experiments on the TPC-H benchmark dataset [31]. Our B&B algorithm was able to obtain an optimum solution in 0.2 seconds on instances with 22 input queries (the actual TPC-H queries) and 32 views. Note that the input query plans for these experiments were formed using the module of [2] that builds the search space of potential views and indexes.

A major theme in our experiments is that of scalability. Note that, with our approach, approximate solutions are needed only for very large query loads, see footnote 1 in Section 1. To draw conclusions about scalability we must rely on randomly generated instances that emulate real databases and exhibit increases in difficulty with increasing size. We posit that our way of generating random instances could allow fair and relevant comparison with future work.

The generation of random instances is discussed in Section 6.1. The advantages of our B&B algorithm with respect to the quality-runtime tradeoffs are demonstrated in Section 6.2. A direct comparison between or B&B algorithm and the greedy heuristic of [2] is given in Section 6.3. These results show that our algorithm (which, unlike the heuristic of [2], guarantees optimal solutions) is competitive with the heuristic. In Section 6.4 we compare our LaR-based heuristic (Algorithm 2 in Section 4.2) to the greedy heuristics of [2] and [27]; we conclude with Section 6.5.

For the experiments we used an Intel 1.86GHz processor with 1GB RAM, running Red Hat Linux.

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

The numerical parameters are as follows:

- W_{min} and W_{max}, the min/max view/index weights;
- C_{min} and C_{max}, the min/max 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 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 that covers 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 with N queries we assume there are $T = 2N$ views and indexes. Finally, we scale instance size by increasing N.

Weights and costs: The weight of each view/index is a random value from $[W_{min}, W_{max}]$. The cost of query i is C_i, a random value within $[C_{min}, C_{max}]$. For each plan j for 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 choose a space bound that is a fraction of the sum of the weights of the views and indexes. Our preliminary experiments show that the value of 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 where we assume that neighboring views and indexes have more in common than others, making them more likely to be usable for the same query. (This property occurs in, e.g., the OLAP context [3, 12, 15, 24].) Thus, 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 query-plan costs might not be independent (as in our random generation), but the dependencies are likely be too complex to model.

6.2 B&B behavior

Our B&B algorithm exhibits a lot of flexibility when it comes to tradeoffs between runtime and guaranteed solution quality. For example, Figure 1 shows the scalability of our algorithm for different maximum allowed errors. The X-axis represents problem size via the number of queries in the workload. The Y-axis shows the runtime of the algorithm. Each point on the plot corresponds to the average of the runtimes for thirty instances, all of a given problem size. Each curve corresponds to a relative error bound *specified explicitly by the user;* the algorithm does not terminate unless the current solution is within that bound of optimal.

[5]The scalability results reported in this paper imply that for many realistic-size problem inputs, our approach can be applied directly on the *original* search space produced in [5].
[6]E.g., Microsoft Research is currently unable to distribute the research prototype externally due to IP considerations.

Runtime decreases significantly even when the input relative error is a mere 1–2%. Beyond what is shown in Figure 1, we were able to solve almost all (29 out of 30) 80-query instances with errors of 1% and 2% in less than 15 seconds.

For the next experiment, see Figure 2, we used a ten-query instance with a big difference between the initial upper and lower bounds,[7] and tested the runtime against various maximum allowed errors. The X-axis in Figure 2 represents the user-specified error, and the Y-axis shows the runtimes for our algorithm. A point on the graph shows how long it took the algorithm to get a solution within each relative error. Each point represents a different run with the given error bound as user input. The actual relative error of the solution may be significantly smaller than the input error value. The runtime decreases not only because of the more easily satisfied stopping criterion, but also because we bind more variables (see Section 3) and prune more subproblems during the exploration of the branch and bound tree.

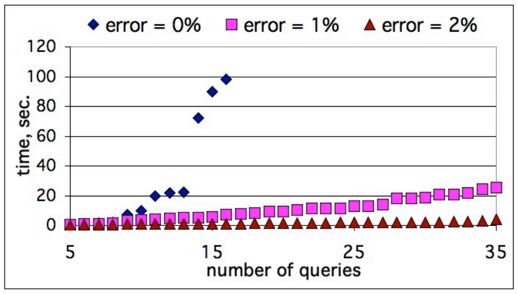

Figure 1: Scalability of *B&B* for various input errors.

The next experiment, in Figure 3, shows the interactive (online) property of our algorithm. During execution the program pauses when the relative error improves. A user can then choose to accept the current solution or have the algorithm look for a better one. The plot in Figure 3 is based on experimental results for 30 random instances, where each instance has 15 queries, 6 plans per query, and 30 views. On this plot, the X-axis represents the runtime, and the Y-axis represents the relative error. Each run yields multiple plot points, one for each time the relative error improved in that run. The highlighted line shows the progress of one particular run — points along that line are points at which the relative error improves. The remaining points were generated in similar fashion based on the other 29 runs.

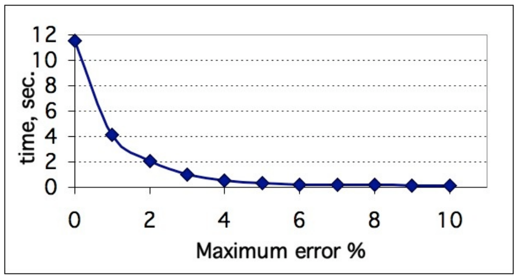

Figure 2: Runtime of *B&B* on a fixed instance with various input errors.

[7]Large differences between the initial upper and lower bounds are likely to make the problem instance hard to solve.

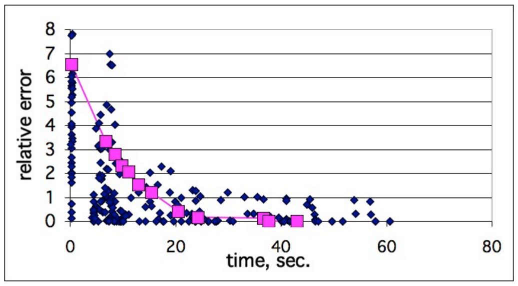

Figure 3: Relative error improvements for multiple runs using our interactive approach.

Results for $Greedy(k, m)$ with $k = 2$				
queries	median	mean	stdev	max
25	10.1	13.1	8.5	32.0
30	11.8	11.4	6.5	23.5
35	9.6	11.2	5.4	23.9
40	11.2	11.1	5.4	22.1
Results for our B&B with 20% error				
queries	median	mean	stdev	max
25	6.5	6.3	3.9	15.9
30	8.7	7.4	4.4	13.9
35	5.6	5.8	3.0	12.7
40	7.5	7.4	3.5	15.9

Table 1: Quality of $Greedy(k, m)$ outputs vs. our B&B: statistics for relative error w.r.t. optimum.

6.3 Comparison with $Greedy(k, m)$ of [2]

In this subsection we show, via direct comparison, that the guaranteed optimum solutions obtained by our B&B heuristic compare favorably with expected (but not guaranteed) quality of solution obtained by algorithm $Greedy(k, m)$ of [2].

Heuristic $Greedy(k, m)$ proposed in [2] exhaustively searches for an optimal subset of views/indexes of size k, and then greedily adds views and indexes until the subset has m views/indexes. Note that, while our algorithm can work with any weight values, $Greedy(k, m)$ assumes that all views and indexes have weight 1. Any comparisons involving $Greedy(k, m)$ will therefore be on unit-weight instances (with all other aspects of random generation the same). Because it searches exhaustively for a set of view/indexes of size k, the runtime for $Greedy(k, m)$ is exponential in k.

Table 1 shows the behavior of the solution quality for $Greedy(k, m)$ with $k = 2$; the results for $k = 1$ are slightly worse. Each line shows basic statistics for the relative error (w.r.t. the optimum) with 30 experiments of the given problem size. Even though our B&B only guarantees a 20% *input* error bound, the actual errors are much smaller, both objectively and in comparison with those of $Greedy(k, m)$.

Figure 4 shows the scalability of $Greedy(k, m)$ and of our B&B, with input error 20%. The X-axis shows problem size via the number of input queries, and the Y-axis shows the runtimes. A point on the plot represents the average runtime for 30 experiments for given problem size. Our B&B runtime falls between the linear and quadratic curves for $k = 1$ and $k = 2$, respectively, showing that we get better quality *and* a guarantee with competitive runtime. (Recall that to enable the comparison with $Greedy(k, m)$, we use here the unrealistic assumption of unit-weight views and indexes.)

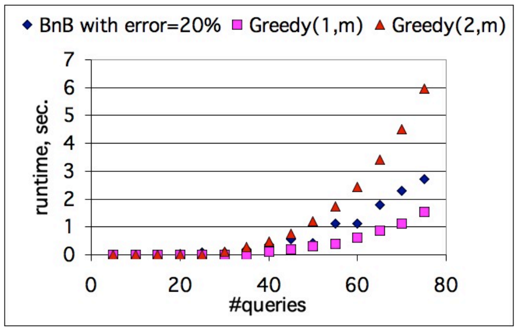

Figure 4: Runtime of $B\&B$ with 20% relative error versus $Greedy(k,m)$ of [2] with $k = 1, 2$.

6.4 Advantages of Lagrangian relaxation

We now compare the solution quality, runtime, and scalability of our LaR heuristic (Algorithm 2 in Section 4.2) versus two other heuristics. One of these others is $Greedy(k, m)$ discussed in Section 6.3. Because of the large size of the instances used for the experiments reported in this subsection, the only feasible choice for k in $Greedy(k, m)$ is 1.

The other heuristic that we consider in this subsection is the approach of [27], based on LP relaxation, to selecting clustered and unclustered indexes (only, as opposed to views), please see Sections 1.3 and 2.2 for an overview and discussion.[8] In this subsection we refer to this heuristic as LPR. To obtain a feasible solution from an LP relaxation we (a) turn all non-zero valued variables into 1's to satisfy the integrality constraint; then (b) greedily remove indexes/views until the space constraint is satisfied.

None of the three heuristics has guarantees on solution quality. Indeed, all the instances were too large to be solved optimally. These randomly generated instances had between 50 and 550 queries, 20 plans per query, and 100 indexes/views. We obtained similar results with a larger or smaller number of plans per query.

Figure 5 illustrates that the solution quality (total benefit) of our LaR approach is significantly better than that of LPR, the difference being nearly a factor of 2. The greedy heuristic yields even higher quality solutions. (Recall that we imposed significant restrictions on problem inputs to make the three heuristics comparable, please see footnote 8 for a description. Thus, the solution-quality results of this subsection may not be representative of the more realistic problem instances that our approach can handle.) As shown in Table 2, the LaR heuristic is significantly faster than the other two.

While the time to solve an LP is faster than the Lagrangian relaxation discussed in Section 3, the time for the LaR and LPR *heuristics* is dominated by the greedy conversion to feasible solutions. In the case of LaR, the number of indexes/views added during the greedy part (Algorithm 1 of Section 4.1) is usually small, thus the minimal increase in runtime with increasing problem size. However, the greedy phase of the LPR heuristic must typically remove a large number of indexes/views to obtain a feasible solution.

[8]For the experimental comparison to be meaningful, we assumed that all three approaches do selection of (unclustered) indexes only. In addition, we had to apply to all the problem instances the unit-weight restriction on the use of $Greedy(k, m)$, see discussion in Section 6.3.

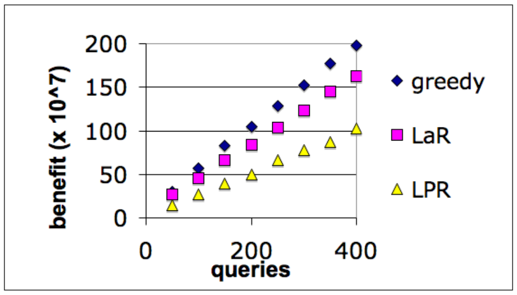

Figure 5: Average benefit for three heuristics on large instances.

queries	LaR	greedy	LPR
100	0.1	2.2	7.6
200	0.3	4.4	17.2
300	0.5	6.5	27.3
400	0.9	8.8	38.3

Table 2: Average runtime for three heuristics on large instances.

6.5 Other observations

We now discuss the relevance of our experiments to related work. We commented in Section 1.2 on the genetic algorithms proposed in [18, 21, 23] and their need to "soften" the space constraint. The two remaining papers that most closely address our problem are [5] and [7]. (The approach of [27] is extended and compared to ours in Section 6.4.)

As discussed in Section 5.4, the authors of [5] focus on exploring the search space of combinations of views and indexes to find those combinations that have low(er) query-processing costs while satisfying the input space bound. Unlike us, they do not appear to address solution quality with respect to a global optimum. The ideas presented in [5] could be used either to develop a good set of plans (and thus be orthogonal to our work) or to provide a heuristic solution in the setting where there are no predefined plans with arbitrary benefits (and thus be incomparable to our work). Note that our work is applicable no matter how the benefit of a plan is calculated. Finally, we posit that the approach of [5] can be *combined* with our proposed algorithm to improve the quality guarantees of the output view/index configurations; please see Section 5.4 for a detailed discussion.

If we specialize our work to the problem of [7] of selecting clustered and unclustered indexes (but not views), their result may apply in the sense that (as reported in [7]) they achieve roughly the same solution quality as $Greedy(k, m)$ of [2], with somewhat better execution performance than $Greedy(k, m)$. In contrast, we obtain solutions having guaranteed (including optimum) quality, with much faster execution times than $Greedy(k, m)$ in a more general context. We posit that experimental comparisons of the approach of [7] with ours would confirm the conclusions of [7] that the quality and performance characteristics of their approach are comparable to those of $Greedy(k, m)$. Thus, we expect our approach to compare with that of [7] in the same way our approach compares with $Greedy(k, m)$, see Section 6.3.

To summarize, we have shown that: (a) a B&B algorithm based on Lagrangian upper and lower bounds is effective in obtaining optimal solutions, solutions with user-specified relative error, or solutions selected interactively by the user based on reductions in relative error; (b) Lagrangian relaxation has distinct advantages over linear programming relaxation (such as the approach of [27]) as a starting point for computing feasible solutions; and (c) our B&B algorithm obtains optimal solutions faster than non-optimal solutions computed by the greedy heuristic of [2]. The work presented here is clearly competive in a practical/experimental sense.

7. ACKNOWLEDGEMENTS

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

8. REFERENCES

[1] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala. Database tuning advisor for Microsoft SQL Server 2005. In *VLDB-04*.

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

[3] Z. Asgharzadeh Talebi, R. Chirkova, Y. Fathi, and M. Stallmann. Exact and inexact methods for selecting views and indexes for OLAP performance improvement. In *EDBT*, 2008.

[4] A. Balmin, F. Özcan, K. S. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB-04*.

[5] N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *SIGMOD Conference*, pages 227–238, 2005.

[6] N. Bruno and S. Chaudhuri. Physical design refinement: The merge-reduce approach. *ACM Transactions on Database Systems*, 32(4):28–43, 2007.

[7] S. Chaudhuri, M. Datar, and V. R. Narasayya. Index selection for databases: A hardness study and a principled heuristic solution. *IEEE Trans. Knowledge and Data Engineering*, 16(11):1313–1323, 2004.

[8] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, 1997.

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

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

[11] A. Gupta, I. S. Mumick, J. Rao, and K. Ross. Adapting materialized views after redefinitions: techniques and a performance study. *Information Systems*, 26(5):323–362, 2001.

[12] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for OLAP. In *ICDE*, 1997.

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

[14] H. Gupta and I. S. Mumick. Selection of views to materialize in a data warehouse. *IEEE Trans. Knowledge and Data Engineering*, 17(1):24–43, 2005.

[15] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD-06*.

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

[17] M. Held, P. Wolfe, and H. P. Crowder. Validation of subgradient optimization. *Math. Progr.*, 6:62–88, 1974.

[18] J.-T. Horng, Y.-J. Chang, B.-J. Liu, and C.-Y. Kao. Materialized view selection using genetic algorithms in a data warehouse system. In *Congr. Evol. Comp.-99.*

[19] H. J. Karloff and M. Mihail. On the complexity of the view-selection problem. In *PODS*, 1999.

[20] M. Kormilitsin, R. Chirkova, Y. Fathi, and M. Stallmann. View and index selection for query-performance improvement. Technical report, NCSU, 2007.

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

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

[23] M. Lawrence. Multiobjective genetic algorithms for materialized view selection in OLAP data warehouses. In *GECCO*, pages 699–706, 2006.

[24] 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.

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

[26] 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.

[27] S. Papadomanolakis and A. Ailamaki. An integer linear programming approach to database design. In *ICDE Workshops*, pages 442–449, 2007.

[28] C. R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems.* John Wiley & Sons, 1993.

[29] K. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *SIGMOD Conference*, pages 447–458, 1996.

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

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

[32] G. Valentin, M. Zuliani, D. C. Zilio, G. M. Lohman, and A. Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, 2000.

[33] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *VLDB*, pages 136–145, 1997.

[34] D. C. Zilio, C. Zuzarte, S. Lightstone, W. Ma, G. M. Lohman, R. Cochrane, H. Pirahesh, L. S. Colby, J. Gryz, E. Alton, D. Liang, and G. Valentin. Recommending materialized views and indexes with IBM DB2 design advisor. In *ICAC*, 2004.