View and Index Selection for Query-Performance Improvement: Quality-Centered Algorithms and Heuristics

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, 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 this paper, we develop efficient methods that deliver user-specified quality of the set of selected views and indexes when given view- and index-based plans as problem inputs. Here, quality means proximity to the globally optimum performance for the input query workload given the input query plans. 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 [1, 2].

Categories and Subject Descriptors:  
G.1.6 [Numerical Analysis]: Optimization; G.4 [Mathematical Software]: Algorithm Design and Analysis; H.2.1 [Database Management]: Logical Design.


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 [5], and is difficult to solve optimally even when the set of indexes and views mentioned in the input query plans is small.

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, as well as to a variety of constraints, including the storage-limit 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;
- a discussion of the place of our approach in end-to-end frameworks, such as those of [1, 2]; and
- experimental results on benchmark and random instances, to illustrate scalability.

2. INTEGER LINEAR PROGRAMMING

Our ILP model uses the following 0/1 variables: \( x_{ijt} \) is 1 if the \( j \)-th plan \( p_{ij} \) is chosen for query \( i \), 0 otherwise; \( y_t \) is 1 if the \( t \)-th view or index \( v_w \) of size \( w_t \) is materialized, 0 otherwise. Here, each input plan \( p_{ij} \) ( \( j \)-th plan for query \( i \)) is represented by a set of all views and indexes required to execute query \( i \). The objective is to maximize \( \sum_{i=1}^{m} \sum_{j=1}^{n} b_{ij} x_{ijt} \), where \( b_{ij} \) is the improvement (gain) in query-evaluation cost when plan \( p_{ij} \) is chosen, subject to

\[
\sum_{j=1}^{m} x_{ijt} \leq 1 \quad i = 1, \ldots, n \quad (1)
\]
\[
\sum_{(j,w) \in P_{ij}} x_{ijt} \leq y_t \quad \forall (i,t) \quad (2)
\]
\[
\sum_{t=1}^{k} w_t \cdot y_t \leq B \quad (3)
\]

Here, constraint (1) says that a query can have at most one plan. When plan \( p_{ij} \) is chosen, all views/indexes in \( p_{ij} \) must be materialized. One way to state this is \( x_{ijt} \leq y_t \) for all \( i,j,t \). However, taking (1) into account, we only

\(^{2}\)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.
need a single constraint (2) for each query and view/index. Recall that \( v_i \in p_{ij} \) means that the \( j \)-th plan for query \( i \) requires view/index \( v_i \). That is, if \( v_i \) is not selected (\( y_i = 0 \)), none of the plans using \( v_i \) can be chosen (all \( x_{ij} = 0 \) where \( v_i \in p_{ij} \)). 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.

### 2.1 Branch and Bound

Branch and bound (B&B) obtains optimum solutions to ILPs at the expense of worst-case exponential runtime. 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. 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 experiments show that our upper- and lower-bound computations yield good scalability w.r.t. instance size, better solution quality as compared with the heuristic [1] when terminated early, and promising tradeoffs between runtime and solution quality. Please see [5] for the details.

### 2.2 Finding Upper Bounds

In our work, we use Lagrangian Relaxation (LaR) (see, e.g., [4]) for obtaining upper bounds. LaR 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. The details of this step can be found in [5]. LaR has several key advantages in the B&B context, as LaR can be used:

- in an effective lower-bound heuristic,
- to fix values of some variables, thus reducing the size of the B&B tree (see [5]),
- to significantly decrease runtime when a given approximation ratio is desired, and
- to use computations at the parent of the B&B tree node as a starting point.

### 2.3 Finding Lower Bounds

We use Lagrangian heuristics to obtain good lower bounds. Our algorithm transforms a solution to the Lagrangian relaxation into a feasible solution. The idea of the Lagrangian heuristics is to take a (infeasible) solution to the Lagrangian relaxation of the original problem and to modify it as little as possible to get a feasible solution. The idea is to first take a feasible part of the solution to the LaR, and to then greedily fill in the remaining free space thus maximizing the benefit of the solution, see [5] for the details.

### 3. EXPERIMENTAL RESULTS

Figure 1 shows the scalability of our algorithm for different maximum allowed errors. 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.

In another experiment, see Figure 2, we used a ten-query instance with a big difference between the initial upper and lower bounds, and tested the runtime against various maximum allowed errors. The runtime decreases not only because of the more easily satisfied stopping criterion, but also because we bind more variables and prune more subproblems during the exploration of the branch and bound tree.

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; the points along the line show where the relative error improves. The remaining points were generated in similar fashion based on the other 29 runs.

### 4. REFERENCES


