An Integer Linear Programming Approach to Database Design

Stratos Papadomanolakis   Anastassia Ailamaki
Computer Science Department
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213
{stratos,natassa}@cs.cmu.edu

Abstract

Existing index selection tools rely on heuristics to efficiently search within the large space of alternative solutions and to minimize the overhead of using the query optimizer for cost estimation. Index selection heuristics, despite being practical, are hard to analyze and formally compute how close they get to the optimal solution. In this paper we propose a model for index selection based on Integer Linear Programming (ILP). The ILP formulation enables a wealth of combinatorial optimization techniques for providing quality guarantees, approximate solutions and even for computing optimal solutions. We present a system architecture for ILP-based index selection, in the context of commercial database systems. Our ILP-based approach offers higher solution quality, efficiency and scalability without sacrificing any of the precision offered by existing index selection tools.

1. Introduction

Automating database physical design is a major challenge in building self-tuning database systems. Previously proposed tools for index selection rely on heuristic algorithms, such as greedy search [7, 2, 9], augmented with techniques to reduce the number of candidate indexes they consider and the number of calls to the query optimizer.

A problem with existing techniques is that there is no way to estimate how close they get to the optimal solution. More recent approaches [5] compute a lower bound for a workload's cost by considering the optimal indexes for each query individually and disregarding any storage constraints. Although this approach is helpful, the derived bounds are not realistic for storage-constrained scenarios.

In this paper we present a new framework based on an Integer Linear Program (ILP) formulation for the index selection problem. The ILP formulation allows the application of standard linear optimization techniques to index selection, that remove the shortcomings of existing heuristic techniques. Specifically, through the application of Linear Programming (LP) relaxation, we are able to obtain a tight bound on the quality of the optimal solution for a given problem instance, which we can use to characterize the quality of any solution. The LP relaxation provides useful information about the problem, allowing us for example to find approximate solutions that have optimal performance but exceed the amount of available storage.

To further improve the quality of an approximate solution, we apply a branch-and-bound technique.Branch-and-bound is advantageous in terms of quality: If run to completion, branch-and-bound will return the optimal solution, while if interrupted before completion, it will return a suboptimal solution, along with a bound for its distance from the optimal. Finally, the ILP formulation can be tuned so that the solution algorithm has comparable performance to existing index selection tools while examining a much larger number of alternative solutions.

The contributions of this paper are the following. First, we present an ILP formulation for the index selection problem. Second, we describe a system architecture for efficiently solving large ILP problem instances. Third, we report preliminary results that demonstrate that our approach has good performance and outperforms existing approaches.

The rest of the paper is organized as follows. Section 2 describes our theoretical formulation, while Section 3 describes new techniques applicable to our ILP model and their benefits. In Sections 4, 5 we present our system architecture for solving ILP instances and discuss performance optimizations. Section 6 presents a preliminary experimental evaluation, Section 7 reviews related work and Section 8 concludes the paper.

2. An ILP Framework For Index Selection

This section details our ILP formulation for index selection and its extension for handling update statements.

2.1. Formulation

Consider a workload consisting of m queries and a set of n indexes I_1-I_n, with sizes s_1-s_n. We want our model to account for the fact that a query has different costs depending on the *combination* of indexes it uses. A *configuration* is a subset $C_k = \{I_{k1}, I_{k2}, ...\}$ of indexes with the property that all of the indexes in C_k are used by some query. This definition is equivalent to the *atomic* configuration definition in [7].

Let P be the set of all the configurations that can be constructed using the indexes in I and that can potentially be useful for a query. For example, if a query accesses tables T_1, T_2 and T_3 then P contains all the elements in the set (indexes in I on T_1) $\times$ (indexes in I on T_2) $\times$ (indexes in I on T_3).

The cost of a query i when accessing a configuration C_k is $c(i, C_k)$ and $c(i, \{\})$ denotes the cost of the query on an unindexed database. We define the *benefit* of a configuration C_k for query i by $b_{ik} = max(0, c(i, \{\}) - c(i, C_k))$.

Let y_j be a binary decision variable that is 1 if the index is actually implemented and 0 otherwise. In addition, let x_{ik} be a binary decision variable that is equal to 1 if query i uses configuration C_k and 0 otherwise.

Using x_{ik} and b_{ik}, the benefit for the workload Z is

$$Z = \sum_{i=1}^{m} \sum_{k=1}^{p} b_{ik} \times x_{ik} \tag{1}$$

where $p = |P|$. The values of x_{ik} depend on the values for y_j: We cannot have a query using C_k if a member of C_k is not implemented. Also, we require that a query uses at most one configuration at a time. For instance, a query cannot be simultaneously using both $C_1 = \{I_1, I_2, I_3\}$ and $C_2 = \{I_1, I_2\}$. Finally, we require that the set of selected indexes consumes no more than S units of storage. Thus the formal specification of the index selection problem is as follows.

$$maximize \;\; Z = \sum_{i=1}^{m} \sum_{k=1}^{p} b_{ik} \times x_{ik} \tag{2}$$

subject to

$$\sum_{k=1}^{p} x_{ik} \leq 1 \quad \forall i \tag{3}$$

$$x_{ik} \leq y_j \quad \forall i, \, \forall j, k : I_j \in C_k. \tag{4}$$

$$\sum_{j=1}^{n} s_j \times y_j \leq S \tag{5}$$

Constraints (3) guarantee that a query uses at most one configuration. Constraints (4) ensure that we cannot use a configuration k unless all the indexes in it are built and constraint (5) expresses the available storage. We can also use constraints to restrict the usage of clustered indexes, but we omit the details due to lack of space.

	No index	Single index		Index pairs
Q_1	cost=120	C_1={I_1} S_1=100	C_2={I_2} S_2=100	C_3={I_1,I_2}
		c_{11} = 90 b_{11}= 30	c_{12} = 90 b_{12}= 30	c_{13}= 45 b_{i3} = 75
Q_2	cost=120	C_4={I_3} S_3=100	C_5={I_4} S_4=100	C_6={I_3,I_4}
		c_{24} =85 b_{24}= 35	c_{25} =85 b_{25}= 35	c_{26} = 47 b_{26}= 73

Figure 1. Index selection example.

Figure 1 shows an example with 2 queries and 4 indexes, listing all the relevant configurations for each query. Assume only indexes I_1 and I_2 are relevant to Q_1, whose cost varies depending on whether uses a single-index configuration (c_1 or c_2) or a pair (c_3). The same holds for Q_2 and indexes I_3 and I_4. Assume we want to optimize workload benefit given total storage capacity of S=200 units.

The equivalent ILP instance is as follows:

$$\begin{aligned} minimize \; Z = b_{11} \times x_{11} + b_{12} \times x_{12} + b_{13} \times x_{13} + \\ + b_{24} \times x_{24} + b_{25} \times x_{25} + b_{26} \times x_{26} \end{aligned} \tag{6}$$

subject to

$$\begin{gathered} \sum_{k=1}^{3} x_{1k} \leq 1, \quad \sum_{k=4}^{6} x_{2k} \leq 1, \\ x_{11} \leq y_1, \; x_{12} \leq y_2, \; x_{13} \leq y_1, \; x_{13} \leq y_2, \\ x_{23} \leq y_3, \; x_{24} \leq y_4, \; x_{35} \leq y_3, \; x_{36} \leq y_4, \\ \sum_{j=1}^{4} s_j \times y_j \leq 200 \end{gathered}$$

By inspection we determine the optimal solution

$$\begin{gathered} y_1 = 1, \; y_2 = 1, \; y_3 = 0, \; y_4 = 0, \\ x_{11} = 0, \; x_{12} = 0, \; x_{13} = 1, \; x_{24} = 0, \\ x_{25} = 0, \; x_{26} = 0 \end{gathered}$$

The set of indexes I_1 and I_2 is preferable because their combination has a large benefit for Q_1 and outperforms any other alternative. Notice that the commonly used greedy search would fail to identify the optimal solution. In the first iteration it would pick index I_3 and in the second I_4.

The exact solution provided by the ILP formulation is optimal *for the given initial selection of indexes*. If we were to include all the possible indexes that are relevant to the given workload, it would give us the globally optimal solution. Considering the set of all the possible indexes is prohibitively expensive and thus a candidate selection module is necessary. The ILP approach is flexible in that we can use it with an arbitrary candidate index set.

2.2. Handling Updates

The ILP formulation of Section 2.1 can be extended to handle updates in the workload (SQL *INSERT*, *UPDATE* or *DELETE* statements). We model an update statement as a sequence of two sub-statements, “select” and ’“modify”. The “select” part is just another query selecting the set of rows to be modified or deleted and thus is handled by the formulation of Section 2.1 (*INSERT* statements do not have a selection part and thus get a zero benefit value for all configurations). The “update” part is a statement that simply updates the set of rows returned by the “select” part. It has a different behavior, because an index configuration C_k can have a *negative benefit* for the update part, due to the additional cost for updating the relevant indexes in C_k. Specifically, the benefit b_{lk}^U of configuration C_k for the update sub-statement U_l is

$$b_{lk}^U = cost_u(l, \{\}) - (cost_u(l, \{\}) + cost_u(l, C_k)) \qquad (7)$$

The $cost_u(l, \{\})$ value represents the cost of the update statement U_l on a table with no indexes. When one or more indexes exist (configuration C_k) the cost of updating the indexes in C_k is added to $cost_u(l, \{\})$. Equivalently, the benefit of configuration C_k is negative:

$$b_{lk}^U = - \sum_{I_j \in C_k} cost_u(l, I_j) \qquad (8)$$

Generally, for every index I_j we can associate a (negative) benefit value $-f_j$ which corresponds to the total overhead introduced by I_j and is computed by summing all the $-cost_u(l, I_j)$ values over all the update statements U_l. To model updates, we only need to modify the objective function of Section 2.1 (Equation (2)) to take into account the negative benefit values f_j.

$$maximize\ Z = \sum_{i=1}^{m+m_1} \sum_{k=1}^{p} b_{ik} \times x_{ik} - \sum_{j=1}^{n} f_j \times y_j \qquad (9)$$

Equation (9) describes the workload benefit in the presence of m queries and m_1 update statements. The second term simply states that if index I_j is constructed as part of the solution, it will cost f_j units of benefit to maintain it in the presence of the m_1 update statements.

3. Using the ILP Formulation

The ILP model enables the application of new solution techniques to the index selection problem. First, it allows us to compute a tight upper bound Z^* on the maximum benefit (Equation 2) achievable for a specific problem instance. The upper bound indicates how far any solution is from the optimal without actually having the optimal solution, thus allowing us to provide *quality guarantees*. The upper bound derived from an ILP formulation is more accurate compared to previous techniques that completely eliminate the storage constraint, thus significantly overestimating the maximum benefit achievable.

Second, the ILP formulation allows for the efficient computation of an initial approximate solution. Although this solution is not necessarily optimal, the tool characterizes its benefit by comparing it to the Z^* bound and present the system administrator with the choice of keeping it (if the benefit is acceptable) or investing more time in improving it. A system can also utilize the ILP formulation to compute an initial solution that is optimal in terms of benefit, but violates storage constraints. The system administrator is then presented with the option of deploying the additional storage and getting the performance, or looking for a better solution. This option is particularly attractive in cases where focus is more on exploring the design space rather than reaching strictly optimal solutions.

Finally, it is possible to improve the benefit offered by the initial solution through a branch-and-bound search algorithm, that can terminate when a solution of acceptable benefit is reached, or execute until the optimal solution is found.

3.1. Linear Program Relaxation

The Linear Program (LP) relaxation of an ILP has the same objective function and constraints as the original ILP, but its solution is no longer required to be integer. For the ILP formulation of the previous section, the LP relaxation accepts fractional values for the y_j, x_{ik} variables, as long as $0 \leq y_j, x_{ik} \leq 1$.

Without the requirement for integer solutions, linear optimization problems can be solved efficiently by polynomial algorithms. In the context of index selection, the LP relaxation quickly provides useful information about the optimal solution of the ILP and its storage requirements.

Let Z^* be the optimal benefit value (Equation (2)) for the LP relaxation and Z_{opt} be the optimal benefit value for the original ILP problem. It can be shown that

$$Z_{opt} \leq Z^* \qquad (10)$$

Thus the optimal value of Equation (2) is an upper bound for the benefit of the optimal ILP solution. In the general case, it is difficult to determine how close the solution of the LP relaxation (Z^*) is to the solution of the original ILP (Z_{opt}).

Certain classes of ILP problems, however, are "well-behaved" in that their LP relaxation provides a very good indicator for the objective function value for the optimal ILP solution. We are currently actively working on theoretically and experimentally characterizing the relationship between the ILP and LP relaxation solutions for index selection. Our preliminary results, presented in Section 6, suggest that the best benefit value computed through the LP relaxation is actually very close to that of the optimal ILP solution (within 15%). In any case, the LP relaxation provides a much tighter bound for the optimal solution compared to previous approaches, that compute bounds by eliminating the storage constraint (as Section 6 verifies).

The LP relaxation can also be used to derive an "initial" solution that has a benefit of at least Z^*, but violates the storage constraint, requiring $S' \geq S$ storage. Consider the solution x^*_{ik},y^*_j of the LP relaxation. We derive an integer solution by first rounding the values y^*_j so that every non-zero variable y^*_j is set to 1. Given the new y^*_j values, new x^*_{ik} can be selected so that each query has a maximum benefit under the (3)-(4) constraints of Section 2.1.

If the rounded solution requires only a small increase in storage, it might still be of use to the system administrator. Rounding works best if most of the fractional y^*_j variables are either 0 or close to 1, in which case the increase in storage by setting them to 1 is small. Our experiments suggest that this is the case in practice: An intuitive explanation is that y^*_j variables that are non-zero but close to zero don't offer much in terms of benefit (due to the constraints (4)) of Section 2.1 and therefore will not often occur in solutions.

3.2. Branch and Bound

Branch-and-bound algorithms are commonly used in the solution of ILPs. Although they make use of the LP relaxation method, they are guaranteed to derive the optimal integer solution, if run to completion. In addition, the branch and bound algorithm can be interrupted before completion, in which case a sub-optimal solution is returned along with a bound from the optimal.

The general branch-and-bound approach for ILP is to choose a decision variable and fix it to be either 0 or 1, thus generating two subproblems with one fewer decision variable (branching). The LP relaxation technique is applied to the subproblems, computing bounds for their benefit values. The branching is repeated for the subproblems and can be performed in a depth-first or breadth-first order. Once a first integer solution is reached (where all the variables have been fixed) or if an initial valid solution already exists (for example through manipulating a rounded LP relaxation result) its value Z can be used to prune those subproblems that have an upper bound $Z^* \leq Z$, as no integer assignment resulting from the subproblem will be better than the existing integer solution. The algorithm terminates when it runs out of time or all the variables have been assigned and there are no new subproblems.

Combining the ILP model with a branch-and-bound algorithm has several advantages over the existing heuristic search framework. Quality-wise, branch-and-bound avoids the problem of missing "interactions" between indexes, as long as the interactions are captured in the ILP model. Furthermore, branch and bound does not have the problem of being "trapped" in locally optimal solutions. If ran to completion, it is guaranteed to find the optimal solution.

Performance-wise, running to completion and determining the optimal solution might not be feasible for large problems. *We propose using the branch-and-bound framework only for a few iterations and stop the search once a solution with an acceptable distance from the optimal has been reached.* The bounds computed during the branch-and-bound process can directly be used to determine when a solution of acceptable quality has been reached and terminate the algorithm as early as possible.

Finally, it is possible to "customize" branch-and-bound search by incorporating all the search heuristics developed in the literature. For instance, a greedy search algorithm can be used to derive an initial integer solution that can be used to prune subproblems and improve algorithm performance. In a sense, the branch-and-bound approach can provide at least the quality and performance of existing algorithms, while having additional quality benefits and the added advantage of a bound from the optimal solution.

4. A System for Index Selection Using ILPs

Figure 2 shows the architecture of our ILP-based approach. The *ILP Model* module takes as input the query workload and a storage constraint and produces an ILP specification. The candidate indexes used in the ILP formulation are provided by the *Candidate Selection* module. Our ILP approach is flexible in that it can accept any set of indexes as its input, thus allowing the use of existing candidate selection techniques. The input index set is processed by the *ILP Model* module to produce a set of configurations for each query (represented by the x_{ik} variables).

The benefit b_{ik} for each configuration is computed by the Index Usage Model (INUM), that efficiently provides query cost estimates with the same precision as the query optimizer. The INUM itself is initialized by performing a small number of key optimizer calls, the results of which are used in query cost estimation. Once the ILP model is complete, the LP and branch-and-bound solver modules are used to derive solutions. The solver result is either the optimal so-

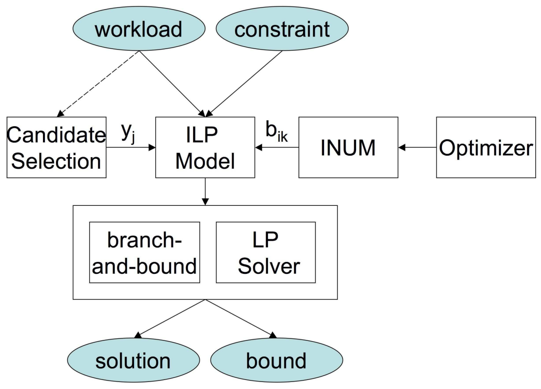

Figure 2. Architecture for an ILP-based index selection algorithm.

lution or an approximate one, along with a tight bound for its distance from the optimal.

The Index Usage Model [11] is a framework for efficient cost estimation. Given a query and an index configuration, it computes a value for the query cost that is equal (or in the worst case very close) to the value returned by the optimizer, without actually performing an optimizer invocation. Since the optimizer is not involved, the INUM can achieve 3 orders of magnitude faster cost estimation while maintaining optimizer precision. The INUM is independent of the index selection algorithm used, however it is a direct match for our ILP-based approach that requires evaluating a large number of configurations.

The intuition behind the INUM is that although design tools must examine a large number of alternative designs, the number of different optimal query execution plans and thus the range of different optimizer outputs is much smaller. Therefore it makes sense to *reuse* the optimizer output, instead of repeatedly computing the same plan. The INUM works by first performing a small number of key optimizer calls per query in a *precomputation* phase and caching the optimizer output (query plans along with statistics and costs for the individual operators). During normal operation, query costs are *derived* exclusively from the precomputed information without any further optimizer invocation. The derivation involves a simple calculation (similarly to computing the value of an analytical model) and thus is significantly faster compared to the complex query optimization code.

Besides it accuracy and performance benefits, an aspect of the INUM that is relevant to our ILP-based approach is the time consumed by its *precomputation* phase. The INUM can efficiently provide cost estimates for tens of thousands of configurations, but it must first be initialized by performing a small number of key optimizer calls. The duration of the initialization phase is therefore part of the ILP solving time and must be taken into account when comparing with existing index selection tools. We therefore include INUM setup timing results in our experimental section.

5. Performance Considerations

Since our solution algorithms (described in Sections 3.1, 3.2) rely on solving the LP relaxation of the original ILP formulation once or multiple times, the performance of solving LP problems is critical for the overall feasibility of our approach. The efficiency of solving an LP problem depends on the number of decision variables and the number of constraints. In the context of index selection, efficiency is determined by the number of candidate indexes (y_j) and the number of the x_{ik} variables.

Our approach allows us to view the number of candidates and configurations as separate "tuning knobs" and use them to control performance. Our work departs from existing solutions that provide no guarantees over the quality of their heuristics. The mathematical structure of the ILP formulation allows us to modify the number of candidates or the number of configurations to improve performance, while quantifying the impact on the optimality of the solution. In the next sections we describe *candidate selection* and *configuration selection* in the context of our ILP formulation and present *sensitivity analysis* as a way to estimate solution quality loss.

5.1. Candidate Selection

Candidate selection (shown also in Figure 2 as a distinct module) determines the set of indexes that will be used in the ILP formulation and therefore the number of y_j variables. Candidate selection is necessary because in practice, incorporating all the relevant indexes in an ILP formulation results in huge models that are not practical to solve. Naturally, omitting indexes from consideration can result in suboptimal solutions. It is the job of the candidate selection module to identify a "good" set of indexes that minimizes quality loss. Our formulation does not make any assumptions on the algorithm used and can therefore incorporate previously proposed approaches [5, 7, 9].

Compared to previous approaches, the ILP formulation allows for a much larger number of candidates. Our experiments indicate that a fairly standard high performance linear solver (in MATLAB's Optimization Toolbox) can solve LP relaxations of 10000 variables in under a minute on a high-end server. The available LP solver performance, combined with the use of the INUM (Section 4) for fast cost estimation, allows us to grow the number of candidates (y_j variables) in the thousands, a much larger number compared to previous techniques.

The flexibility in selecting a large number of candidates reduces the chances that some important index will be missed by the candidate selection. In addition, we propose the use of sensitivity analysis techniques (described in Section 5.3 to estimate the loss in solution quality from a particular set of candidates and to guide the design of effective index selection algorithms.

5.2. Configuration Selection

Configuration selection is the process of determining which configurations will be used in the model. In other words, configuration selection determines the x_{ik} variables and the corresponding b_{ik} benefits that need to be considered. Reducing the number of configurations is critical for performance, as it is exponential to the number of candidates. Intuitively, configuration selection corresponds to setting the benefit values for the omitted configurations to zero. Omitting a configuration does not imply omitting its constituent indexes. It only means that the "importance" of certain index combination is underestimated by the model. Again, we propose the use of sensitivity analysis techniques to quantify the impact of configuration selection to performance.

Again, the ILP formulation allows for a much larger number of configurations to be evaluated. Given the performance statistics of the previous section (10000 variables in 1 minute), for a workload of 50 queries, the model can easily accommodate 200 different configurations per query. Existing index selection tools for a similarly-sized workload examine a much smaller number of candidates, as each atomic configuration/query combination requires an optimizer call, and performing more than a few hundreds of calls is prohibitively expensive.

5.3. Sensitivity Analysis

Sensitivity analysis in linear optimization is the process of estimating the stability of the solution to a given LP with respect to changes in the LP's parameters [4]. For instance, there exist sensitivity analysis techniques for computing intervals for LP parameters, such that parameter variations within those intervals do not affect the optimal solution. For cases where optimality is not preserved, there are ways to efficiently compute the new optimal solution in an incremental fashion.

We plan to investigate the use of sensitivity analysis to model candidate and configuration selection. For example, we can use a modern LP solver to obtain intervals for all the benefit (b_{ik}) and index storage (s_j) parameters of an LP. The intervals are actually computed at no extra cost, as they are a product of the LP solution.

We can then use the obtained intervals to reason about indexes that are not included in the candidate set: If we can prove that there exist no indexes that can cause LP parameters to exceed the computed intervals (for example, indexes with high benefit values or low storage), we can conclude that the existing candidate set is "good enough". If not, we can estimate the potential benefit loss and decide if additional indexes need to be added in the candidate set. Formalizing the application of sensitivity analysis techniques to index and candidate selection is part of our ongoing work.

6. Comparing with Existing Approaches

In this section we present preliminary results from our ILP-based implementation, using TPC-H queries on a commercial DBMS. We compare the performance and quality of our approach to that of the commercial index selection tool.

6.1. Experimental Setup

Our implementation interfaces to a commercial DBMS that has its own index selection tool. Our implementation uses Java for the INUM, database interfaces and model construction and MATLAB's optimization toolbox for the solvers. We obtain a set of candidate indexes for our input workload through the commercial index selection tool, by observing the virtual indexes it builds using a profiler. Our workload consists of 5 TPCH queries, on a 1GB TPCH database. The storage constraint was set to 60000 pages or 480MB. With this specification, the commercial index selection tool examined 64 candidate indexes. We enumerated all the configurations that could be constructed from the candidate set and generated 1589 configuration decision variables. All experiments were run on an Intel Xeon 3GHz server. Performance improvements are on the form $\%improvement = 1 - performance_{optimized}/performance_{orig}$. The benefit numbers returned by our algorithm were verified by actually implementing the indexes and obtaining real optimizer cost estimates.

6.2. Experimental Results

The solution of the LP relaxation of the ILP had 5 nonzero index (y_j) decision variables. Three of them were integer (equal to 1) and the others had a value of 0.76. Figure 3 compares the maximum benefit value for the LP relaxation (*LP_optimal*) to the benefit bound computed by removing the storage constraint (*unlimited*). We obtain the latter simply by invoking the commercial tool without any storage constraint. *LP_optimal*) provides a much tighter bound to the optimal value compared to *unlimited*.

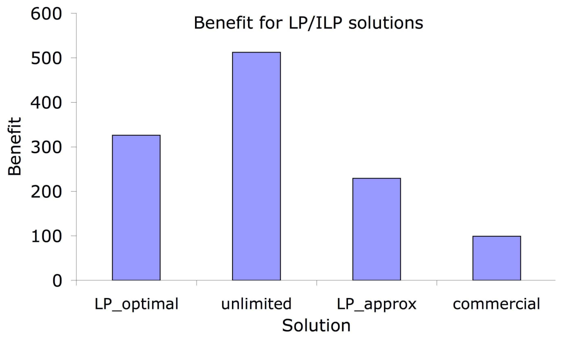

Figure 3. Benefit values for various solutions.

By truncating the LP relaxation result we obtain an integer solution with identical benefit that that requires only 10% more space. (For this example, the truncated solution becomes the optimal if we increase the storage by 10%). Assuming that the original storage constraint is hard, the truncated integer solution is not feasible. We derive a feasible solution, *LP_approx*, by removing one index (setting its y_j value to 0) so that the remaining fit into the available storage. According to Figure 3, *LP_approx* is within 32% of the optimal ILP solution.

We ran the branch-and-bound algorithm in MATLAB's Optimization Toolbox in order to improve the approximate solution, using its benefit value for pruning. The branch-and-bound algorithm returned the optimal solution for the original problem, which is within 15% of the optimal LP relaxation solution, confirming that for this example the optimal value of the LP relaxation is a good approximation for the ILP optimal value. The *commercial* bar in Figure 3 shows the benefit achievable by the solution provided by the commercial index selection tool. The quality of the commercial tool solution is 56% lower than that of the approximate LP solution (*LP_approx*).

MATLAB took 1.3s to solve the LP relaxation, while the commercial tool reported 1 minute of running time. The time for INUM construction (Section 4) was 7s. Overall, the time for constructing the INUM and the ILP model and for obtaining initial solutions through the LP relaxation was much less compared to the commercial tool.

MATLABs branch-and-bound procedure took several hours to complete. The reason is that generic solvers can not distinguish between the index decision variables (y_j) and the configuration variables (x_{ik}) and do not exploit the fact that the latter depend on the former. Developing a a specialized branch-and-bound procedure to replace MATLAB's generic solver is part of our ongoing work.

7. Related Work

Microsoft's Index Tuning Wizard and Database Tuning Advisor [7, 1] use a "candidate selection" phase to identify promising indexes, a "merging" step for augmenting the candidate set [8] and a greedy search for selecting a locally-optimal solution. The DB2 Advisor [9] uses the optimizer to select an initial set of indexes and formulates a knapsack problem that is solved by a greedy search.

Our ILP formulation is based on [6]. Their formulation accounts for queries using more than one indexes and also models index update costs. They also provide a specialized branch-and-bound procedure for its solution. Our work is concerned with applying the formalism in [6] to real-world problems, that involve commercial database systems and workloads.

In terms of modeling, we integrate the model in [6] with the query optimizer in existing systems. We also deal with the problems of selecting candidates and configurations using sensitivity analysis, so that we derive reasonably-sized ILP instances that can be solved efficiently. In terms of solutions, we do not focus on algorithms for finding optimal solutions, as the model, due to candidate and configuration selection, already incorporates approximations. Our work proposes the use of LP relaxation to find approximate solutions and computing optimality bounds.

Finally, to accurately model real world constraints, we include storage limits in our formulation. [6] does not consider storage and it is unclear if their analysis and proposed algorithms can apply to problem instances resulting from our formulation.

[10] describes an ILP and a solution based on randomized-rounding, assuming a single index per query. Their solution has optimal performance but requires a bounded amount of additional storage. Since the algorithms in [10] assume the use of a single index per query, they ignore "interactions" between indexes and are therefore not directly applicable to commercial relational systems.

Besides index selection, there is work on designing additional features, such as materialized views and table partitions [2, 12, 3]. The proposed algorithms are similar to their index selection counterparts, only they are facing a combinatorial explosion in the number of alternative designs that arises when exploring feature combinations. We believe our modeling approach will be beneficial for problems combining multiple features, because it can better capture the "interaction" between features and it offers higher scalability.

8. Conclusion

In this paper we present an Integer Linear Programming (ILP) model for the index selection problem. We apply standard optimization techniques to compute optimality bounds, derive approximate solutions with known distance for the optimal and improve approximate solutions. We describe an efficient implementation architecture that makes use of optimizer estimates similarly to commercial tools. Our preliminary experiments indicate that ILP-based index selection is efficient efficiently and offers higher quality solutions compared to existing tools.

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 2004*.

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

[3] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the SIGMOD Conference*, New York, NY, USA, 2004. ACM Press.

[4] D. Bertsimas and J. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1997.

[5] N. Bruno and S. Chaudhuri. Automatic physical database tuning: a relaxation-based approach. In *SIGMOD '05*.

[6] A. Caprara and J. Salazar. A branch-and-cut algorithm for a generalization of the uncapacitated facility location problem, 1996.

[7] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL server. In *Proceedings of VLDB 1997*.

[8] S. Chaudhuri and V. R. Narasayya. Index merging. In *Proceedings of ICDE 1999*.

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

[10] C. Heeren, H. V. Jagadish, and L. Pitt. Optimal indexing using near-minimal space. In *PODS '03: Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2003.

[11] S. Papadomanolakis, D. Dash, and A. Ailamaki. Intelligent use of the query optimizer in automated database design. Technical Report CMU-CS-06-151, Computer Science Department, Carnegie Mellon University, 2006.

[12] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. Db2 design advisor: Integrated automatic physical database design. In *VLDB*, pages 1087–1097, 2004.