CoPhy: Automated Physical Design with Quality Guarantees

Debabrata Dash[*] **Anastasia Ailamaki**[†]

June 20, 2010
CMU-CS-10-109

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
[†]Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland

This work was partially supported by Sloan research fellowship, NSF grants CCR-0205544, IIS-0133686, and IIS-0713409, an ESF EurYI award, and SNF funds.

Keywords: Convex Programming, Physical Design, Quality Guarantees

Abstract

Physical design requires the selection of an optimal set of design features from a vast search space. The existing tool landscape is dominated by application specific heuristic algorithms, which prune the search space heavily to arrive at a "locally" optimum solution, which may be far from the "globally" optimum. As a result, although the tool runtime is kept at manageable levels, there is no guarantee of the distance of the proposed solution from the optimal. Mature combinatorial optimization tools, on the other hand, provide close to "globally" optimal solutions without pruning the search space, provided that the problem is *convex*. Inspired from the potential of such tools, we develop a combinatorial optimization formulation for the physical design problem, showing that there exists a convex formulation which does not require heuristic pruning of the problem space. Using this formulation, we develop *CoPhy*, an automated physical design tool which suggests near-optimal solutions for physical design problems. We solve the combinatorial problem with an efficient and scalable lagrangian relaxation method, which predicts as well as controls the quality of the final solution. Experiments using TPC-H-like workloads on two commercial systems show that, (a) *CoPhy* is portable across multiple DBMS (b) *CoPhy*'s suggested indexes yield from 20% to 100% better workload speedup compared to greedy approaches, and (c) *CoPhy* scales to workloads composed from thousands of queries.

1 Introduction

Automated physical design is a major challenge in building self-tuning database management systems. The complexity of physical design emanates from the requirement of searching through a potentially huge space containing many design features, such as, indexes, partitions, and materialized views. To make matters worse, each design feature interacts with other features to add a new dimension to the search space. Furthermore, in the real world, the search algorithm needs to satisfy several user-specified constraints, such as requiring the solutions to occupy less than a certain disk budget or requiring low maintenance cost.

Existing commercial physical design tools provide suggestions in a reasonable time by heuristic pruning of the solution space, and limiting interaction between the design features. This, however, prevents them from satisfying two important requirements: predictability and generality. Since the physical design problem is NP-Hard [8], finding the exact optimal solution may require exponential time for any reasonable sized workload. Therefore, the selection tool must predict the quality of the solution, i.e., the distance of the proposed solution from the optimal, allowing the DBA to accept the non-optimal solution in exchange for efficiency. By using simple regression, the DBA can even estimate the execution time required to arrive at a solution of certain quality. This ability to predict the quality of the solution also helps detect infeasible constraints set by the DBA. Therefore, predicting the quality of the solution is an essential requirement in the presence of complex workloads and constraints. The existing approaches lack this quality, since they prune the search space heuristically–limiting their knowledge of the solution quality w.r.t. the globally optimal solution.

Similarly, the physical design tools need to handle multitudes of real-life constraints. The tool must be generic enough to handle new constraint types without a complete rewrite from the ground up. Since the pruning mechanism of the greedy algorithms need to be redesigned to address new constraint types, they are not general.

In the area of Operations Research, the combinatorial optimization problem (COP) formulations and their solvers have been used for efficiently and scalably solving problems where the underlying features interact with each other. They also provide feedback on the distance of the solution from the optimal, thereby allowing the user to terminate the optimization process upon reaching a certain quality guarantee. Unlike the greedy algorithms, COP solvers are generic, i.e. they solve multitudes of new constraint types without changing the code. The usefulness of this approach, however, depends on the "*convexity*" (Section 3.1) property of the COP formulation. If the COP formulation doesn't satisfy this crucial property, the solvers cannot use polynomial time algorithms to solve the sub-problems and gradually arrive at the globally optimal solution.

Modeling the physical design problem as a COP is not straightforward because of the convexity requirement. The state-of-the-art formulations for database physical design achieve the convexity property by enumerating all possible combinations of the candidate indexes–making it impossible to scale without heavy pruning. Any pruning before the actual search process reduces the predictability and efficiency, as the solver can only predict and find solutions only inside the pruned sub-space. The efficiency is also affected, since before pruning away a feature combination, one needs to evaluate its benefits.

Our Approach and Contributions: In this paper, we propose a COP-based tool called *CoPhy*

(**C**ombinatorial **O**ptimization for **Phy**sical Design), which does not prune the search space heuristically, thus allows it to provide quality guarantees. *CoPhy* uses a novel COP formulation to scale the problem size linearly with the candidate features. It then solves the problem efficiently using mature techniques, such as lagrangian relaxation. In this paper, we focus on indexes as the design features, since they are the most commonly used by the DBAs and preserve the difficulty of the complete physical design problem.

The intellectual contribution of the paper is: there exists a compact convex COP formulation for the physical problem, thereby making the problem amenable to the sophisticated and mature combinatorial optimization solvers.

From the point of view of system design, the paper contributes: 1) An efficient and scalable solver for the COP, which combines the existing state-of-the-art techniques from the combinatorial optimization area to achieve its performance goals. It also allows the DBAs to reduce the execution time by trading off quality of the expected solution. For example by allowing 1% difference from the optimal solution, the solver execution time is reduced by an order of magnitude. 2) Demonstrates the design of a *portable* physical designer. The system achieves its portability by using only a very thin layer of interaction between the optimizer and the physical designer. Our experimental results demonstrate the system's superior performance on two different commercial DBMS.

Organization: The rest of the paper is organized as follows: We discuss the related work in Section 2. Section 3 builds the COP for plan selection on a workload and Section 4 adds constraints to the COP. Section 5 details the architecture of *CoPhy* and the optimization methods. We discuss the experimental results in Section 6, and finally, conclude in Section 7.

2 Related Work

Proposed physical design solutions depend heavily on the plan selection mechanism used in the query optimizers. Early research models the query optimizer mathematically, and then suggests the design features. Since early optimizers typically use simple cost models [19], it is relatively straightforward to model the entire optimization process and select appropriate design features accurately. Lum et al. model the selection of secondary indexes as an optimization problem [15]. Esiner et al. improve on that by mapping the index-selection problem to an equivalent network flow problem [11]. Researchers also propose various integer linear program formulations for vertical partitioning of tables [3, 16, 10]. These techniques assume simple cost models for using vertical partitions and build their optimization problem on those models. Modern optimizers, however, use more elaborate cost models which render most previous cost formulations obsolete. Recent work decouples the optimizer design from the problem formulation by modeling the optimizer as the black box and reusing past optimization results [18]. Modeling the optimizer as a black box forces the physical designer to compute the cost of every possible index combination, which is a very expensive process, as such combinations can be exponential in number. We use the same caching approach to model the optimizer, but exploit the internal details of the cache. Our approach allows us to identify useful index combinations without actually enumerating them.

Existing commercial techniques use greedy pruning algorithms to suggest the physical design [12, 1], and use the optimizer directly, there by reducing their efficiency and predictability.

Caprara et al. were the first to propose a COP approach to the index-selection problem, by modeling it as an extension of the *facility-location problem* (FLP) [7], enabling it to exhaustively search the features, instead of greedily searching them. Their formulation, however, assumes that a query can use only a single index. Papadomanolakis et al. extended the formulation to account for queries using more than one indexes and also model index update costs [17]. Kormilitsin et al. propose lagrangian relaxation techniques to solve the FLP formulation [14]. Heeren et al. describe an approximation solution based on randomized rounding, assuming a single index per query [13]. Their solution has optimal performance but requires a bounded amount of additional storage. Zohreh et al. extend the FLP formulation to use views and then provide heuristics to find optimal physical design in OLAP setting [20]. Their algorithm is tuned towards materializing data-cube views and small number of indexes on them. Our approach scales to an index set two orders of magnitude larger than reported in their work. Since these techniques use FLP formulation, they use heuristic pruning to reduce the problem size to a practical level, a limitation we avoid by proposing a new problem formulation.

3 Index Selection As A COP

Formulating the index selection problem to a COP allows us to use the sophisticated combinatorial optimizers. This section briefly discusses the necessary background before deriving the COP for the index selection problem. This is the basis upon which complex physical design constraints are applied and solved efficiently.

3.1 Optimization Programs

Typically an optimization program minimizes or maximizes an objective function, while satisfying some constraints. Mathematically the programs can be written as:

$$\begin{aligned} &minimize f(x) \\ &\text{such that: } g(x) \leq b\ ,\ x \in R^n \end{aligned}$$

Here f is the objective function and g defines constraints. In general such functions are very hard to optimize. If, however, the function f is a convex function and g defines a convex area, the problem can be solved in polynomial time using well-established techniques [4]. A function f is called convex if the following condition holds:

$$\begin{aligned} &\forall x_1, x_2 \in R^n,\ 0 \leq \lambda \leq 1 \\ &\lambda f(x_1) + (1-\lambda) f(x_2) \geq f(\lambda x_1 + (1-\lambda) x_2) \end{aligned}$$

The constraint $g(x) \geq b$ defines a convex set if:

$$\begin{aligned} &\exists x_1, x_2 \in R^n, 0 \leq \lambda \leq 1 \\ &g(x_1) \geq b \; and \; g(x_2) \geq b \Rightarrow g(\lambda x_1 + (1-\lambda) x_2) \geq b \end{aligned}$$

Note that the *linear programs*, where both f and g are linear functions w.r.t. x, are also convex programs. Figure 1 shows prototypical examples of the linear and convex programs.

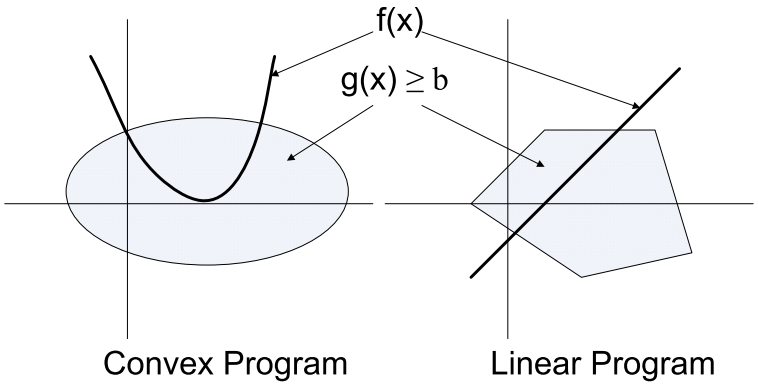

Figure 1: Example of convex and linear programs in 2-D space. The shaded region is defined by the constraint $g(x) \geq b$ and the thick line defines the function $f(x)$.

3.2 Cache-based Query Cost Model

The physical design tools need to determine the cost of the queries, in the presence of the design features. Building a complete white-box model of the optimizer is prohibitively complex, and using the optimizer as a black-box model is prohibitively expensive. We use the optimizer cache model of INUM [18], which takes the middle path, by invoking the optimizer for a few key plans and modeling the complete optimizer using those plans. C-PQO [6] is a similar approach that caches the plans inside the optimizer. We use INUM's model to build the COP, since it caches outside the optimizer; hence, it is portable across DBMSs.

INUM postulates that, although selection tools must examine a large number of alternative designs, the number of different optimal query plans and, thus, the range of different optimizer outputs, is much smaller. Therefore, it makes sense to *reuse* the optimizer output, instead of calling the optimizer to generate similar looking plans. 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 simple numerical calculations and is significantly faster than the complex query optimization code. To explain INUM's postulations, we borrow the following definitions from the literature:

Definition 1 *"Configuration" is a set of indexes. A configuration is called "atomic" with respect to a query, if for each table in the query, at most one index is present in the configuration [9].*

Definition 2 *An "interesting order" is a tuple ordering specified by the columns in a query's join, group-by or order-by clause [19].*

Definition 3 *An "interesting order combination" for a query is the set of interesting orders, where there is at most one interesting order for each table involved in the query [18].*

Definition 4 *An index "covers" an interesting order, if the interesting order is the first column in the index. Similarly an atomic configuration covers an interesting order combination [18].*

An interesting order of a table is a column, which, if ordered, reduces the query cost. For instance, in the query "`select` A, B `from` T `order by` A", A is an interesting order for table T. If there are two tables T_1 and T_2, and their interesting orders are A, C respectively, then possible interesting order combinations are (A, ϕ), (A, C), (ϕ, C), and (ϕ, ϕ). We denote lack of interesting order on a table as ϕ. The atomic configuration $\{T_1(A), T_2(C)\}$, consisting of indexes on A and C, covers the interesting order combination (A, C). Let D be a non-interesting order column in T_2, the atomic configurations $(T_1(A))$ and $(T_1(A), T_2(D))$, cover the interesting order combination (A, ϕ).

Using these definitions, the most important observations from INUM are:

1. If a query involves only Merge-Join and Hash-Join plans, the cost of join and aggregation *does not* depend on the cost of accessing data from the table or indexes. The total cost of the query depends linearly on the cost of accessing data for each table. The cost of accessing data includes the cost of accessing all required rows and columns from indexes, tables, or a combination of them.

2. If a query involves only Merge-Join and Hash-Join plans, then caching one plan per interesting order combination is sufficient to find the plans for all possible atomic configurations.

3. For queries involving all join methods including Nested-Loop Joins, it caches more than one plan per interesting order combination and achieves reasonable cost approximation for the optimal plan cost.

INUM separates the total cost of the query into "internal" join-aggregation costs, and the "leaf" data access costs. The internal costs, determined by join methods and join orders, are only allowed to change between different cached plans. In a given cached plan, the internal cost remains constant, and the variations in the query cost comes from the variation of the data access costs. Therefore, the cost of the plan is linearly dependent on the data access costs, which allows easy determination of the optimal plans and the optimal cost of a query in the presence of atomic configurations. Currently INUM supports only atomic configurations, which excludes plans involving index intersections. Anecdotal evidence suggests that such index intersection plans do not improve the workload performance substantially, while increasing the complexity of the physical designer [6].

We exploit the linearity property of the query costs to build the COP. We first build the COP for index selection for a single query, then extend it for a workload.

3.3 Index Selection for a Query

Plan selection and index selection are mutually complementary processes, as selecting the optimal plans involve selecting the optimal indexes and vice versa. To find the COP for plan and index selection, consider a query Q with O possible interesting order combinations. For each interesting order combination, INUM caches multiple plans. Let P_{op} be the p^{th} plan cached for o^{th} interesting order. The plan can be used only if the atomic configuration covers the plan's interesting order

combination. Let indexes $I_1, \cdots, I_t$ cover these interesting orders on tables $T_1, \cdots, T_t$. Using the first observation of INUM, the cost for P_{op} is:

$$Cost(P_{op}) = IC(P_{op}) + \sum_{i=1}^{t} w_{opi} AC(I_i) \tag{1}$$

The internal cost function $IC(P_{op})$ represents the cost of join and aggregation, $AC(I_i)$ is the access cost function that determines the cost of accessing an index I_i, and w_{opi} is the constant coefficient for $AC(I_i)$. Typically $w_{opi} = 1$, to make the equations simpler to understand, we drop the coefficient in future equations.

The plan cost is the minimum cost using indexes that cover the interesting orders on the table, hence:

$$Cost(P_{op}) = \arg\min_{\alpha_i} (IC(P_{op}) + \sum_{i=1}^{t} \sum_{I_i \in CI(P_{op}, T_i)} \alpha_i AC(I_i)) \tag{2}$$

$$\text{such that:} \sum_{I_i \in CI(P_{op}, T_i)} \alpha_i = 1, \quad \alpha_i \in \{0, 1\} \forall i \tag{3}$$

The covering-index function $CI(P_{op}, T_i)$ finds the set of indexes which *cover* the *interesting order* required by P_{op} on table T_i. The variable α_i is a binary indicator associated with index I_i. If $\alpha_i = 1$, then the index I_i is used in the plan. Constraints in Eq. 3 ensure that only one index is used in the query plan for each table. ***CoPhy*** models the table scans as "empty" index scans, therefore, the equation does not consider table scans as special cases. This is a convex program, since the objective and the constraints are linear. Note that, this problem can be solved with a standard linear program solver, i.e., without the binary constraints. The optimal solution for the program always provides binary α_i variables.

The cost of the query, is the minimum cost of all the query's cached plans. Therefore, the cost of the query Q_q can be found as:

$$Cost(Q_q) = \min Cost(P_{op}) \tag{4}$$

Here each instance of the minimization problem $Cost(P_{op})$ has an independent set of constraints. Hence, to find the minimum cost, we iterate over all cached plans P_{op} for the query and minimize $Cost(O_{op})$.

3.4 Index Selection for a Workload

The index selection for a workload looks similar to index selection for a query, but using the same formulation can lead to errors in the presence of constraints. The reason is that for a single query, the plans are independent from each other, i.e., there can be only one optimal plan selected. In a workload, however, each query has a plan, and the costs have to be minimized simultaneously, thus requiring a new formulation.

Consider a workload W, Eqs. 2 and 3 can be extended in a straightforward manner to select plans for each query in the workload.

$$Cost(W) = minimize \sum_{Q_q \in W} Cost(Q_q) \tag{5}$$

$$= \arg\min_{\alpha_{iq}} \sum_{Q_q \in W} (\arg\min_{\alpha_{iq}} IC(P_{opq}) + \sum_{i=1}^{t} \sum_{I_i \in CI(P_{opq}, T_i)} \alpha_{iq} AC(I_i)) \tag{6}$$

$$\text{such that: } \sum_{I_i \in CI(P_{opq}, T_i)} \alpha_{iq} = 1 \;\; \alpha_{iq} \in \{0,1\} \forall i \tag{7}$$

The plan P_{opq} is the p^{th} plan cached for o^{th} interesting order for query Q_q. The indicator variable α_i is changed to α_{iq}, otherwise selecting an index to use in a query forces it to be used in all other queries. Since minimizing over another set of minimizations is not a convex function, this program is not a convex program. To convert it to a convex program, we introduce a new indicator variable p_{opq} which is set to 1 if the plan P_{opq} is selected for query Q_q. Since the second term in the query cost does not change in the following equations, it is denoted by the term CL_{opq} for improved readability.

$$CL_{opq} = \sum_{i=1}^{t} \sum_{I_i \in CI(P_{op}, T_i)} \alpha_{iq} AC(I_i) \tag{8}$$

$$Cost(W) = \arg\min_{\alpha_{iq}, p_{opq}} \sum_{Q_q \in W} p_{opq}(IC(P_{opq}) + CL_{opq})) \tag{9}$$

$$\text{such that: } \sum_{p} p_{opq} = 1 \text{ and } p_{opq} \in \{0,1\} \forall o, p, q \tag{10}$$

The minimization along with the constraints converts the plan selection problem for a workload into a special form of a convex program called "quadratic program"; because the objective function becomes a quadratic function involving variables p_{opq} and α_{iq}. Although convex programs can be solved efficiently in polynomial time, we improve efficiency by converting them to a linear program, thereby removing the quadratic term $p_{opq}\alpha_{iq}$. This is achieved by adding constraints as follows:

$$\arg\min_{\alpha_{iq}, p_{opq}} \sum_{Q_q \in W} (p_{opq}\; IC(P_{opq}) + CL_{opq}) \tag{11}$$

$$\text{such that: } \sum_{I_i \in CI(P_{op}, T_i)} \alpha_{iq} = p_{opq} \;\; \forall T_i \tag{12}$$

Eq. 12 ensures that the plan P_{opq} is selected for query Q_q only if at least one index covering the interesting order requirement for each table has α_{iq} set to 1. Eqs. 11 and 12 form a linear program that selects plans for the entire workload and the indexes required for those plans. Similar to index selection for a query, this program also does not require the binary constraints, as it always finds the binary solutions for the optimal objective values.

Summarizing, Eq.11 defines the objective function for the index selection problem of a workload, and Eqs. 7, 10, and 12 define the constraints. Analyzing the size of this problem, observe that for a workload of $|Q|$ queries and $|P|$ cached plans, if there are $|I|$ indexes in the candidate set, then the size of the generated program is $O((|Q| \times |I|) + |P|)$. For a typical workload $|P| << |I|$, since the number of columns used in the workload is much larger than the interesting orders. Also, $|Q| << |I|$, since every query can contribute to a large number of indexes. Therefore, the program size grows linearly with $|I|$. If $|P|$ is large for a workload, ***CoPhy*** uses approximation techniques [17] to reduce the cache size with a small sacrifice in INUM's accuracy.

4 Adding Constraints

Without constraints, the plan selection and index selection problems are relatively straightforward. The problem, however, becomes much harder when the DBA requires the tool to satisfy some constraints on the selected indexes. Traditionally, the index storage space has been the only constraint for the index selection tools. We discuss the index size constraints in this section, and provide a more complete discussion on translating other types of constraints to the COP in Appendix A.

Size constraint ensures that the final size of the indexes does not exceed a threshold M, or similar conditions based on the index sizes. To translate these constraints, a condition on the "per-query" index indicator variable α_{iq} is not sufficient, since the same index can be used in multiple queries. Using these variables alone causes the problem formulation to double-count the index size.

Therefore, we derive a new variable α_i which indicates the presence of the index in any of the query plans. The α_i variable is the *logical-or* of all the individual α_{iq} variables for all i. The logical-or operation is translated by adding the following constraint to the generated program:

$$\alpha_{iq} \leq \alpha_i \quad \forall i, q \quad \alpha_i \in \{0, 1\} \quad \forall i \tag{13}$$

To prove that these constraints actually work, we show that if an index I_i is used in any query Q_q, then the α_i variable is set to 1. Since $\alpha_{iq} = 1$, and $\alpha_i > \alpha_{iq}$, the constraint $\alpha_i \in \{0, 1\}$ forces α_i has to be 1.

The size constraint is represented using the new variable α_i as:

$$\sum_i \alpha_i size(I_i) \leq M \tag{14}$$

Sometimes the DBAs also specify constraints on the update cost of the indexes. We model this situation by adding the update cost to the objective of the optimization algorithm. Let μ_i be the update cost for the index I_i, then we add the following term to the objective in Eq.11.

$$\sum_i \mu_i \alpha_i \tag{15}$$

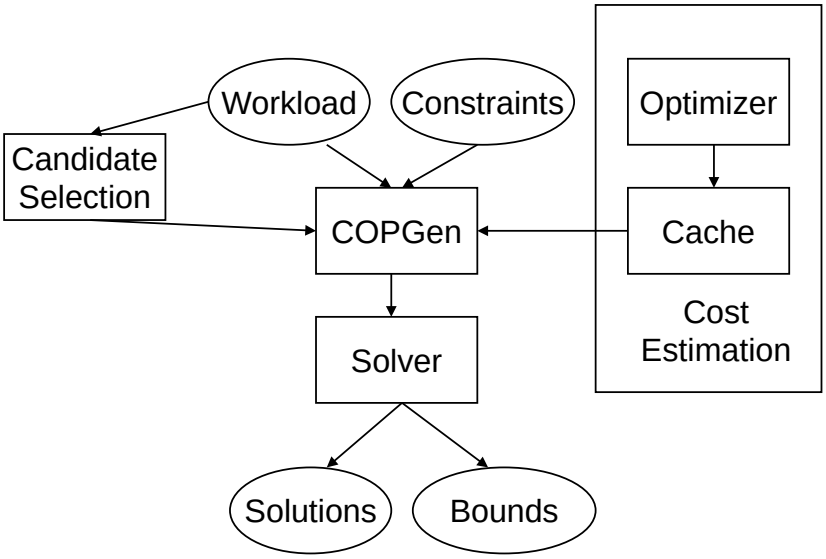

Figure 2: Architecture of *CoPhy*

5 CoPhy System Design

In this section, we discuss the complete architecture for the index selection tool - *CoPhy*, including the process of selecting candidate indexes and techniques to solve the COP developed in the earlier sections.

Figure 2 shows the most important modules of *CoPhy*. All modules except the *Solver* are used for building the COP. These modules decide the α_{iq} and the p_{opq} variables, and then compute the IC and the AC functions. Once the COP is constructed, the *Solver* is used to determine the solution.

The *Candidate Selection* module determines the candidate indexes used for the α_{iq} variables using the workload's structural properties. The *COPGen* module takes these candidates and the constraints as input and generates the COP. *Cost Estimation* determines the cost model for the queries, and our tool uses INUM as the cost model. This model caches the plans, the internal cost functions (ICs), and the access cost functions (ACs). The complete optimization problem is then input into the *Solver*. The overall performance of the tool depends on the numbers of α_{iq}, and p_{opq} variables, which control the problem size and, consequently, the solver's execution time.

We use a very simple *Candidate Selection* module, which generates indexes for every subset of the columns referenced in a query, eventually producing a large set of candidate indexes. From this large candidate set, the module prunes out indexes that never help in reducing query costs; it then identifies these indexes by finding the minimum access costs for all tables in the query. If the minimum cost of the query using the index along with best possible indexes on all other tables is more than the cost of the query without any index, then the index will never be used by the optimizer. Generally, the module produces thousands of candidate indexes. In comparison, the commercial physical design tools consider only up to hundreds of candidate indexes.

Now we focus on the *Solver*, which is critical for *CoPhy* as it provides the desired efficiency and scalability.

5.1 Solving the COP

The *Solver* module of Figure 2 takes as input the optimization formulation corresponding to an index selection problem and computes the near optimal solution. While optimization problems with integer variables are NP-hard in the worst case, mature solving techniques, in practice, efficiently optimize very large problem instances. This subsection discusses the details of the algorithms that enable a fast solution for such large problems. First, a fast greedy algorithm is proposed, then the complete solution is described.

5.1.1 Greedy Algorithm

We extend the eINUM iterative greedy algorithm proposed in [18] as a baseline solver for the COP. Each iteration of the greedy algorithm determines the index which decreases the workload cost the most and adds it to the solution set. The algorithm terminates when there are no more indexes to add or the storage constraint is violated. The eINUM algorithm however is impractical as a physical designer, since it takes about 3.5 hours to suggest indexes. We therefore extend the algorithm by using the structure of INUM cache to speed it up by a factor of 300! The details of the eINUM algorithm and the optimizations are discussed in Appendix B.

5.1.2 Lagrangian Relaxation Algorithm

Generally the combinatorial programs are solved using the *branch-and-bound* method. In this method, the solver starts with an initial upper bound on the program (in our case the solution of the greedy algorithm). For variable x in the problem, two branches are created, one with $x = 0$ and the other $x = 1$. Before exploring a branch, the solver tests to see if the lower-bound of the branch is higher than the current upper bound. If the lower bound is higher, then that branch is safely pruned and the search continues in other branches. If it has pruned out all the branches or all variables are integral variables, it backtracks to search the branches in the parent nodes.

Commercial COP solvers such as CPLEX use linear programming (LP) relaxation to find the lower and upper bound of the branch-bound process. In LP relaxation, the binary constraints are ignored and the problem is solved to find the lower bound of the objective. While LP relaxation works well with many problem formulations, including FLP [17], in our formulation, the linear relaxation of the α_{iq} variables allows them to be set to very small values. This effectively estimates the lowest possible cost for the workload without any constraint. For tight constraints, this lower-bound is far off the desired optimum. Therefore, CPLEX, which uses only LP relaxation, runs for hours before converging to the optimal solution for our COP. In this section, we improve the bound computation by using lagrangian relaxation (LR) to compute the upper bound and a randomized-rounding method on LP relaxation to compute the lower bound.

To understand the LR method to compute the upper or lower bounds, consider the example problem shown below

$$minimize\ f(x)$$
$$\text{Such that: } g_1(x) \leq A \;\; g_2(x) \leq B$$

This problem has $f(x)$ as the objective function and includes many constraints. The lagrangian relaxation identifies the "tough" constraint among the list of constraints and adds that to the objective function with a multiplier. Intuitively, it punishes the solver by a factor θ, if it does not satisfy the constraint. For example, if the first constraint is the toughest constraint in the problem, then the relaxed problem becomes

$$minimize\ f(x) + \theta(g_1(x) - A)$$
$$\text{Such that: } g_2(x) \leq B\ \ \theta > 0$$

This modified problem may not satisfy the constraint $g_1(x) \leq A$, since it has been moved to the objective. But from the solution found using this problem, we cascade the constraints, similarly to the technique in [14], to find the upper bound on the original problem. By finding an appropriate value of θ, the upper bound is made tighter and the solver converges to the solution faster.

There is no systematic method to find the tough constraints for a specific problem, since it depends on the problem structure. For COP, the constraint in Eq. 13 is the toughest one to solve. Removing that constraint splits the problem into two components, each of which can be solved efficiently. The constraint, however, makes even sophisticated solvers take considerable time to solve the problem. Hence, we use that constraint as the "tough" constraint for the LR technique and then operate in the branch-bound method to solve the problem.

To find the lower bound on a node, *CoPhy* considers the solutions to the LP relaxed problem. If $0 < \alpha_{iq} = r < 1$, and for a random value $0 < s < 1$, we set $\alpha_{iq} = 1$ if $s > r$. This randomized rounding creates a solution that uses more indexes than allowed by the constraints, but has been shown to provide a tight lower bound on the cost function [13].

Since at each node *CoPhy* knows the upper and the lower bound on the objective value, we can gain speed by trading off the accuracy of the final solution by looking at the difference between the upper and lower bounds. This allows the DBA to terminate the optimization problem when the difference goes below a given percentage and accept the results from that search step as the solution.

Furthermore, This allows the DBA to predict when a certain quality guarantee will be achieved by using simple regression. Thereby, she can estimate the quality of the solution after an estimated period of time, and intelligently predict when to terminate the optimization session. Since the COP is based on INUM's cost model, errors in INUM's estimation can limit the solver's knowledge of the exact optimal value. In our experiments, INUM has about 7% error in cost estimation, which we argue to be reasonable for the scalability it provides. Moreover, it is an orthogonal problem to improve INUM's accuracy using more cached plans.

6 Experimental Results

This section discusses *CoPhy*'s results on a TPC-H-like workload and on two mainstream commercial DBMSs. We first discuss the experimental setup and then study the workload in detail. We start with index storage constraints, and then discuss the effect of the quality guarantee on the execution time.

6.1 Experimental Setup

We implement our tool using Java (JDK1.6.0) and interface our code to the optimizer of a popular commercial DBMS, which we call *System1*.

We experiment with a subset of TPC-H benchmark containing 15 queries (our experimental parser, for the time being, does not support the following queries: 7, 8, 11, 15, 20, 21, and 22). Since experiments with larger databases show trends similar those in a 1GB database, we use this database on all the workloads for ease and speed of result of verifications. For these 15 queries, INUM caches 1305 plans and the candidate selection module generates 2325 indexes. We also show the behavior of the system on other real-world and synthetic benchmarks in Appendix C.

We use a dual-Xeon 3.0GHz based server with 4 gigabytes of RAM running Windows Server 2003 (64bit), but for our experimental purposes we limit the memory allocated to the database to only 1GB. We report both the selection tool execution time and quality of the recommended solutions computed using optimizer estimates.

We compute solution quality similarly to [9, 2], by comparing the total workload cost on an unindexed database $c_{unindexed}$ vs. the total workload cost on the indexes selected by the design tool $c_{indexed}$. We report percent workload speedups according to:

$$\%\ \textit{workload speedup} = 1 - c_{indexed}/c_{unindexed}$$

We compare the quality of the indexes suggested by *CoPhy* against the greedy algorithm described in Section 5 and Appendix B, and the FLP-based index selection tool. Since the solution to the problem given by Papadomanolakis et al. is within 0.2% of the optimal solution on *System1* for the pruned candidate set, we present the results from their method and identify them as *FLP*. The scalability of *CoPhy* is demonstrated by varying the workload cardinality.

6.2 Solution Quality Comparison

In this experiment, if D is the size of the database, we allocate xD space to indexes in the final solution. We increase x gradually to observe the improvement in the quality of the selected indexes. For *CoPhy*, we stop the search process when the solution reaches within 5% of the optimal solution. Figure 3 compares the solution quality for TPCH15 using the selection tools on the commercial system *System1*. On the x-axis we increase x, and on the y-axis we show the workload speedup after implementing the selected indexes.

Figure 3 shows that *CoPhy* suggests indexes with the highest workload speedup for all values of x on *System1*. As the space allocated for the indexes grows, all tools converge towards selecting indexes with similar workload speedup. *CoPhy* improves on *Greedy* by fully searching the index combination space, instead of looking at one index at a time. *Greedy* prefers large indexes on tables such as LINEITEM, and misses indexes on the smaller PART and CUSTOMER tables. *CoPhy*, however, selected a more balanced set of indexes on both large and small tables to achieve the best solution quality. FLP needs to prune many candidate configurations to keep the problem size under control, hence it misses some plans that *CoPhy* identifies.

The results demonstrate that *CoPhy* finds is significantly superior to the Greedy algorithm and FLP algorithm running on the same set of candidate indexes. The benefit of *CoPhy* is more pronounced when less space is available for indexes.

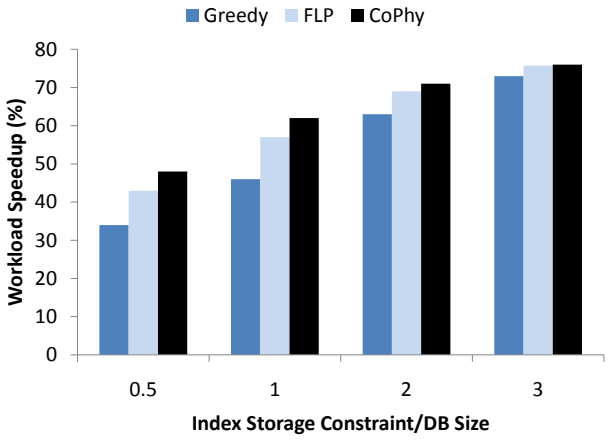

Figure 3: Results for index selection tools for TPCH15

6.3 Execution Time Comparison

Figure 4 compares the execution time of index selection algorithms with $x = 1$. On x-axis, we vary the workload size and on y-axis we show the execution times for different algorithms. To observe the execution time on larger workloads, we create workloads of 250, 500, 750, and 1000 TPC-H-like queries using TPC-H's *QGen* on TPCH15's query templates, and name them TPCH250, TPCH500, TPCH750, and TPCH1000 respectively. To scale INUM computation for these larger workloads, we use INUM approximations [17]. The approximation method reduces the number of cached plans from 90 on average to 2 for each query. We run the experiment with an index size multiplier $x = 1$, and with 5% quality relaxation for *CoPhy*. Since the query templates remain the same, the benefit of using the selection tool for these larger workloads remains similar to Figure 3, except that the cost approximation for larger workload reduces the solution qualities for both *CoPhy* and greedy algorithms by about 3%. Note that, even though TPCH15 does not use approximations, we compare the running times in the same charts in order to save space. The time

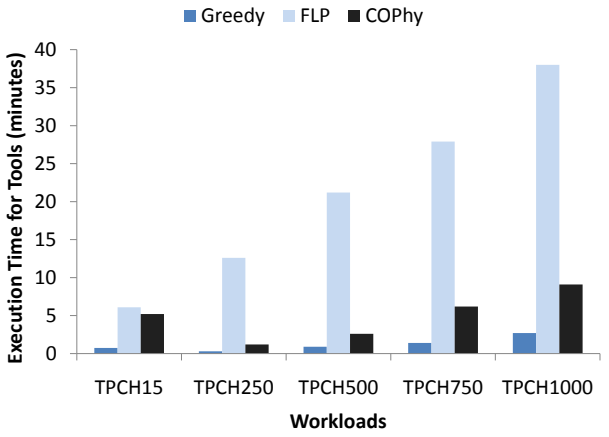

Figure 4: Execution time of algorithms for varying query sizes.

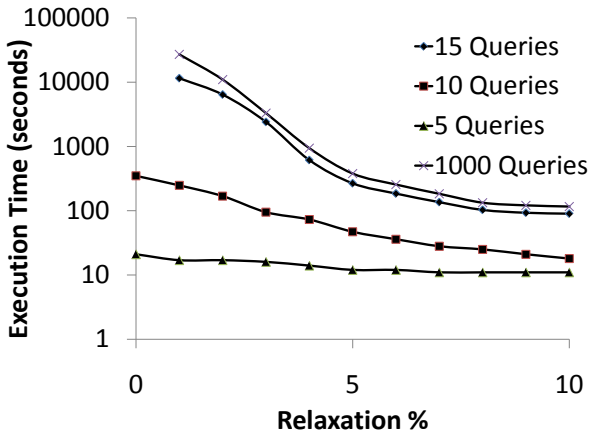

Figure 5: Execution time of *CoPhy* at different relaxations from the optimal

taken to build the INUM cache model the workloads in Figure 4 are 24, 2, 4, 9, and 17 minutes respectively. Using workload compression along with the INUM approximation reduces the INUM cache construction overhead to less than one minute for the generated workloads.

We first focus on TPCH15. *Greedy* is the fastest selection tool as it *Greedy* benefits from the index structure described in Appendix B in detail. In this experiment, building the in-memory structures for *Greedy* takes approximately 18 seconds, and running the greedy algorithm takes about 25 seconds. In comparison, the reported execution time for a similar greedy algorithm is about 3.5 hours [18], hence using the index structure speeds up the algorithm by a factor of nearly 300. *CoPhy* spends about 5.2 minutes to find the solutions, of which about 20 seconds were spent in building the problem from the index variables, and about 45 seconds to find the starting greedy solution and remaining time in solving the problem with the lagrangian algorithm. FLP spends more than 5 minutes pruning the configurations and building the combinatorial problem. FLP, however, solves faster than *CoPhy* since the pruning reduces the problem size and complexity.

Now we consider the generated workloads and observe the scaling behavior of the tools. *Greedy* and FLP scale almost linearly with the problem size. *CoPhy*'s execution time goes up to about 9 minutes for the largest workload, however, about 2.5 minutes are spent initializing the greedy solution, hence the LR method scales well with increasing workload cardinality as well.

We conclude from the experiments that the *CoPhy*'s solver is capable of scaling to thousands of queries, and scales almost linearly with the number of queries.

6.4 Quality Relaxation vs. Execution Time

We gradually increase the relaxation distance from the optimal value and observe the improvement in execution time for the LR based solver. We use *System1* with index space constrained to be the same as the DB size. We report only the time it takes for the LR algorithm to run, as the greedy algorithm's execution time does not change with the relaxation values. In Figure 5, we vary the distance from the optimal value on the x-axis, and on y-axis we show the execution time of the solver. The three different lines in the figure show the behavior on 3 subsets of the TPCH15 queries, containing 5, 10, and 15 queries respectively. We also compare against the 1000 TPC-H

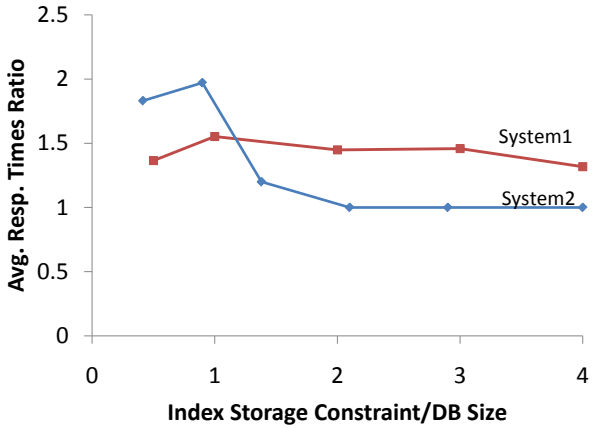

Figure 6: Quality comparison against commercial systems

queries generated in the previous experiment.

Figure 5 shows that even for a small workload with 15 queries, achieving the exact optimal value is not feasible. Even after running the solver for more than a day, we could not converge to the optimal solution. When relaxing the solution to be within 1% from the optimal value, the solver converges after a long time. For a 15 query workload, 5% relaxation provides the best balance between proximity to optimal solution and execution time. When the workload size is reduced to just 5 queries, *CoPhy* find the optimal solution within a minute; this is possible since the problem search space is fairly small for the solver to explore every possible combination of indexes. Similarly, for 10 queries the solver converges to the optimal solution, but takes more time. Consequently, even with small workloads, the ability to relax the targeted solution helps to dramatically reduce the execution time of the solver.

6.5 Comparing Against Commercial Tools

Since the commercial designer tools lack a fast cost model like INUM, their runtime is typically much higher than the INUM-based solutions discussed so far. Hence we only show the comparison of the performance of their suggested indexes. We compare *CoPhy* against two commercial system physical designers implemented on *System1* and *System2* using TPCH15. Since the indexes suggested by the tools on these two systems cannot be directly compared, we normalize the performance on each system by the performance of *CoPhy*'s suggested indexes. If c_{system} is the total workload cost on *System* using the suggested indexes and c_{cophy} is the workload cost with *CoPhy*'s suggested indexes on *System*, then we report the relative workload speedup: c_{system}/c_{cophy}. On the x-axis we increase the index constraint size as a multiple of the table size, and on y-axis we report the relative speedup.

Figure 6 shows that *CoPhy* always performs better than *System1*'s designer tool, and performs better than *System2*'s tool when the space allocated to indexes is low. This is a direct result of not pruning the candidate set eagerly. Therefore, we believe that the generic search method of *CoPhy* helps improve the query performance of the commercial systems, when compared to the sophisticated and fine tuned greedy methods employed by the commercial tools.

More Experiments: We discuss more generic constraints on the COP and study their properties in Appendix A. We also provide results on two more workloads in Appendix C.

7 Conclusion

In this paper we present *CoPhy*, a practical and scalable physical design tool. We first demonstrate that we can formulate a reasonably sized COP for the physical design problem, so that the state-of-the-art solvers can be used to solve the efficiently. Unlike existing selection tools, this approach allows DBAs to trade off the execution time against the quality of the suggested solutions. Our experimental results indicate that *CoPhy* suggests significantly better results when compared to existing commercial physical design tools, especially when the constraints are tight. We show the scalability and the portability of *CoPhy* by running on multiple DBMSs. Finally, *CoPhy*'s formulation of the physical design problem can be used with the commercial state-of-the-art cost models, given access to the optimizer internals [6].

References

[1] S. Agrawal et al. Database tuning advisor for microsoft sql server 2005. In *VLDB'04*.

[2] S. Agrawal et al. Automated selection of materialized views and indexes in SQL databases. In *VLDB 2000*.

[3] J. M. Babad et al. A record and file partitioning model. *Commun. ACM*, 20(1):22–31, 1977.

[4] S. Boyd et al. *Convex Optimization*. Cambridge University Press, March 2004.

[5] N. Bruno et al. Constrained physical design tuning. *PVLDB*, 1(1):4–15, 2008.

[6] N. Bruno et al. Configuration-parametric query optimization for physical design tuning. In *SIGMOD '08*.

[7] A. Caprara et al. A branch-and-cut algorithm for a generalization of the uncapacitated facility location problem. *TOP*, 1996.

[8] S. Chaudhuri et al. Index selection for databases: A hardness study and a principled heuristic solution. *IEEE TKDE*, 16(11), 2004.

[9] S. Chaudhuri et al. An efficient cost-driven index selection tool for Microsoft SQL server. In *VLDB'97*.

[10] D. Cornell et al. An effective approach to vertical partitioning for physical design of relational databases. *IEEE Transactions on Software Engineering*, 16(2), 1990.

[11] M. J. Eisner et al. Mathematical techniques for efficient record segmentation in large database systems. *JACM*, 1976.

[12] G.Valentin et al. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of ICDE'00*.

[13] C. Heeren et al. Optimal indexes using near-minimal space. In *PODS*, 2003.

[14] M. Kormilitsin et al. View and index selection for query-performance improvement: quality-centered algorithms and heuristics. In *CIKM*, 2008.

[15] V. Y. Lum et al. An optimization problem on the selection of secondary keys. In *Proceedings of the 1971 26th annual conference*.

[16] S. Navathe et al. Vertical partitioning algorithms for database design. *ACM Trans. Database Syst.*, 9(4):680–710, 1984.

[17] S. Papadomanolakis et al. *Large-scale data management for the sciences*. PhD thesis, CMU, 2007.

[18] S. Papadomanolakis et al. Efficient use of the query optimizer for automated physical design. In *VLDB '07*, pages 1093–1104. VLDB Endowment, 2007.

[19] P. Selinger et al. Access path selection in a relational database management system. In *SIGMOD 1979*.

[20] Z. A. Talebi et al. Exact and inexact methods for selecting views and indexes for olap performance improvement. In *EDBT*, 2008.

A Adding More Constraints

Recently Bruno et al. propose a language to specify constraints [5]. In this section, we demonstrate the COP's generality by translating the proposed sophisticated constraints into the COP. All constraints introduced in this section are linear in nature, hence do not violate the convexity of the COP. We group the constraints into four categories: index constraints, configuration constraints, generators, and soft constraints, and describe their translations.

A.1 Index Constraints

The DBA may restrict the selection of the indexes by specifying conditions on the size of the index, the columns appearing in the index, or the number of columns appearing in indexes. Hence, we sub-group these constraints into three categories: column constraints, index width constraints, and index size constraints.

Column Constraint: This group of constraints specifies the presence or absence of a column in the solution set. For example, one column constraint requires that at least one index built on a table T_t contains a column C_c. This condition can be translated to the COP by adding the following constraint to the program for every such column:

$$\sum_{contains(C_c)} \alpha_{iq} \geq 1 \tag{16}$$

Reusing the notation from Eq. 6, we denote an index I_i being used in a query Q_q as the binary variable α_{iq}. The function $contains(C_c)$ finds the set of indexes which contain the column C_c. There can be many variations of this constraint, such as: only the first column of the indexes is C_c,

or the index has a set of columns etc. All these variations are translated by changing the function $contains(C_c)$.

Index Width Constraint: This group of constraints limits the index search space by the number of the columns they contain. The translation of these constraints is similar to the previous one, as shown below:

$$\alpha_{iq} width(I_i) \geq N \tag{17}$$

Where, N is the limit on the number of columns for an index, and $width(I_i)$ determines the width of the index I_i.

A.2 Configuration Constraints

Although the DBA can specify any constraint on the configuration space, typically she constrains only the *final configuration* or the *result configuration* by limiting the costs of the queries using it. For example, the DBA may want to make sure that the final configuration speeds up all queries in the workload by at least 25% compared to the initial configuration. We translate such constraints into COP by reusing the cost for the query in Eq. 2 and adding the following constraint to COP:

$$Cost(Q_q) \leq 0.5\ InitialCost(Q_q) \tag{18}$$

The function $InitialCost(Q_q)$ represents the cost of the query before running the index selection tool.

A.3 Generators

Generators allow the DBA to specify constraints for each query, index, or table, without specifically mentioning them. It is equivalent to *for-loops* in regular programming languages. For example, if the DBA wants to restrict the final query cost, she specifies the following constraint:

```
FOR Q_q IN W
ASSERT Cost(Q_q) ≤ 0.5 InitialCost(Q_q)
```

The syntax of the constraint is self-explanatory. This constraint is translated by adding constraints shown in Eq. 18 for each query in the workload. The generator can also contain *Filters* to limit the scope of the constraints. For example, following constraint limits the number of columns for indexes only when they contain the column C_3.

```
For All I_i WHERE Contains(I_i, C_3)
ASSERT NumCols(I_i) ≤ 4
```

The "WHERE" condition filters the indexes using a boolean condition. $NumCols(I_i)$ determines the number of columns in I_i and $Contains(I_i, C_3)$ determines that the index I_i contains the column C_3. This filter is translated by generating the constraints in Eq. 16 and 17.

The constraint language also supports aggregates, such as sum and count etc. If the aggregation does not violate the convexity of the constraints, then they are added to the list of constraints generated by the program. Many of the common aggregations such as sum and count do not violate the convexity property and can be translated in this manner.

A.4 Soft Constraints

The constraints discussed so far are termed as *hard* constraints. The final solution proposed by the design tool has to satisfy all hard constraints. Sometimes the DBA may want to specify a *soft* constraint, which should be satisfied to the extent possible, if not completely. The description of the soft constraints is similar to that of an objective function, hence, we convert these conditions into objectives in the COP.

For example, the following constraint requires that the solver should try to achieve the lowest possible cost for the workload, but a solution is still valid even when the cost is not 0. The "SOFT" keyword indicates constraints that can be violated by the solver. `SOFT ASSERT` $Sum(Cost(Q_q)) = 0$ In COP, we add the following objective for the constraint:

$$\min \sum Cost(Q_q) \tag{19}$$

With multiple soft constraints, the optimization program becomes an instance of a *multi-objective* optimization [4] program. In multi-objective optimization, the solver does not search for one optimal point, rather, a set of points in the solution space which are *un-dominated*. A solution is called un-dominated, if no other point exists in the solution space where every objective is smaller. The set of such points is called "pareto-optimal" or "sky-line" points.

When all objectives are convex, the pareto-optimal solution can be determined by using the "scalarization" method [4]: if the solver desires to optimize a vector $O_1, O_2, \cdots, O_n$ of objectives together, it creates a vector $\lambda_1, \lambda_2, ..., \lambda_n$ with $\lambda_i > 0$. Using these constants the solver optimizes the following problem:

$$\min \quad \lambda_1 O_1 + \lambda_2 O_2 + ... + \lambda_n O_n$$

Since we assume each of O_i is convex in the worst case, the scalarized-objective also remains convex. Hence, *CoPhy* solves efficiently using standard techniques. By varying the values of λ_i, the solver finds the pareto-optimal surface of the solutions and the DBA can decide on the optimal point on the surface.

A.5 Extending the Constraint Language

So far, we discuss how to translate the state-of-the-art constraint language into solvable combinatorial optimization programs. It is possible to translate even more difficult constraints into COP, for example, the DBA can add a constraint to restrict the index selection so that the first columns in any two indexes on a table are not the same. The existing language does not allow such constraints, and it is difficult to implement such constraints in a greedy constraint solver, since selection of one index depends on the presence or absence of another. These sophisticated constraints are translated and solved in *CoPhy* by adding the following constraint:

$$1 = \sum_{IsFirstCol(C_c)} \alpha_i \; \forall \; C_c \in T_t \tag{20}$$

The function $IsFirstCol(C_c)$ finds all indexes which have C_c as the first column.

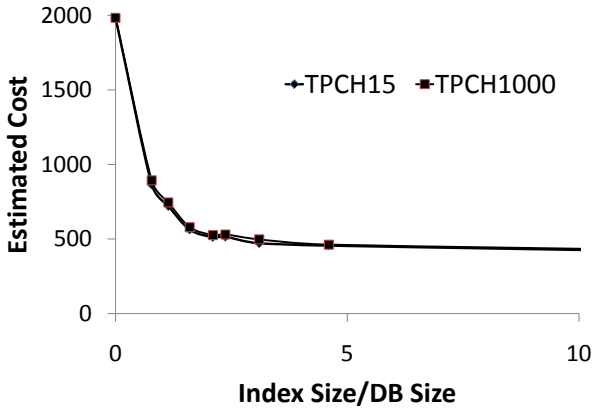

Figure 7: Pareto-optimal curve with the storage constraint and the workload costs as soft constraints

A.6 Experimental Results

This section discusses some experimental results for the COP after adding the new type of constraints. First we discuss the soft constraint results, and then the results for other constraint types.

A.6.1 Adding Soft Constraints

So far we discuss index selection problem with hard index storage constraints and a soft constraint of workload cost being zero. In this experiment we add more soft constraints to the problem to observe what the pareto-optimal surface *CoPhy* generates. We replace the hard constraint of the index storage cost with a soft constraint which tries to minimize the storage cost. Let s be the storage cost, and c be the cost of the workload in the new configuration. Using the scalarization technique discussed in Section A.4, *CoPhy* minimizes the term $\lambda_1 c + \lambda_2 s$. *CoPhy* begins by considering the extreme points where $\lambda_1 = 0, \lambda_2 \neq 0$ and the other extreme point where $\lambda_1 \neq 0, \lambda_2 = 0$. Then it considers the third point with $\lambda_1 c = \lambda_2 s$, and finally explores other points in between using a binary search like technique.

Using these two soft constraints, we run the solver with different values of λ_i settings to achieve the pareto-optimal curve as shown in Figure 7. The the x-axis shows the storage cost of the suggested indexes, and the y-axis shows the estimated cost of the workload. For each point in the graph, the solver takes approximately 25 seconds and we show 10 different combinations of λ_i values. *CoPhy* determines the shape of the curve using the first 3 points, hence *CoPhy* estimates the shape of the pareto-optimal curve faster than manually changing the hard constraints.

A.6.2 Varying the Constraint Size

In this experiment we show that increasing the input program size does not affect the scalability of the solver. We investigate the effect of extra constraints by separating them into three categories, i.e., a) index constraints, b) configuration constraints, and c) storage constraints. For the first category, we generate constraints of the form "`For Index I on LINEITEM I like`

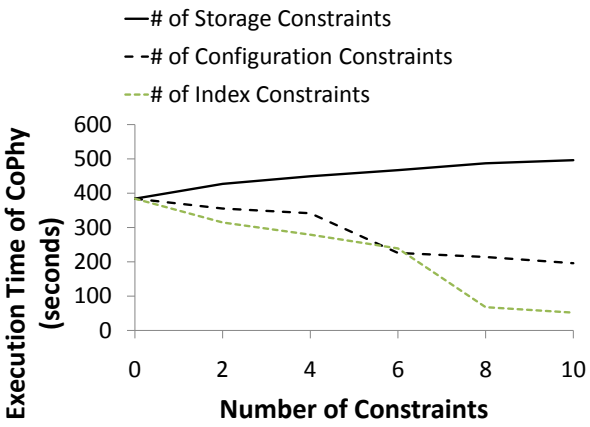

Figure 8: Execution time of *CoPhy*'s solver with varying number of constraints.

`'column,%'`". We generate 10 different constraints for 10 columns of LINEITEM table, and add them to our program. For the second category, we add "`For Query Q in Workload, Cost(Q) < 0.9*EmptyCost(Q)`" to the program for each query Q. For the final category, we add the constraints of the form "the size of the indexes built on a subset of tables must not exceed 90% of the total storage constraint". For example, one of the constraints specifies that LINEITEM's and ORDERS's indexes should not exceed 90% of the total storage space.

Figure 8 shows the execution time of the solver (since other times remains constant for all points) for each of the constraint categories with increasing number of constraints. On the x-axis we increase the number of constraints in steps of two, and on the y-axis we show the execution time of solver for the modified problems. The presence of configuration constraints in the index selection problem restricts us from using the greedy initialization, hence the execution time of the solver suffers slightly when compared to earlier experiments. As we add the index and configuration constraints, the runtime of the solver is reduced, because adding these constraints to the optimization problem, reduces the size of the search space, thus helping the solver to get solutions faster. In TPCH15, LINEITEM generates 809 candidate indexes out of the total 2325 indexes. By removing LINEITEM's indexes, we prune the search space drastically, as it is an important table in the workload.

Similarly, adding the configuration constraints reduces the search space by eliminating many configurations which do not provide sufficient cost improvement. For example, with 10 constraints on the configurations, the number of candidate configurations drops from approximately 55900 to about 6300. Although the search space shrinks drastically, the running time of the solver does not go down proportionally, since the majority of the search happens in the reduced space in the normal case as well. The storage constraints, however, add more processing to the solver, as they are harder constraints compared to the other varieties. As more storage constraints are added, the overhead of doing the lagrangian relaxation increases, consequently the execution time increases–albeit almost linearly with a small slope. Note that the storage constraints we add are artificial constraints and represent the worst case behavior for the solver. Whereas, in the real-world the DBA will more likely add multiple configuration and index constraints rather than storage constraints.

B Greedy Algorithm

In this section, we discuss a fast greedy algorithm to find a candidate final configuration. The greedy algorithm is fast because it always considers only one index at a time, i.e., does not consider the index interaction effects. It still provides satisfying results, however, because it considers a large number of candidates.

Data: I: The candidate set of indexes
O: The objective function
$sizeConstr$: Index size constraint
Result: S: the solution index set

```
1  initialization:;
2  S_1 = S_2 = φ;
3  while true do
4  |   I_m = arg max_{I_i ∈ I \ S_1} (Cost(W, S_1) − Cost(W, S_1 ∪ I_i));
5  |   if sizeConstr(S_1 ∪ I_m) then
6  |   |   S_1 = S_1 ∪ I_m;
7  |   else
8  |   |   break;
9  |   end
10 end
11 Repeat 3-10 with I_ms = arg max_{I_i ∈ I \ S_1} (Cost(W,S_2) − Cost(W,S_2 ∪ I_i)) / Size(I_i);
12 return S = Max_Benefit(S_1, S_2);
```

Algorithm 1: Greedy algorithm for sub-modular index-selection problem

Alg. 1 shows the algorithm, which consists of two loops, each selecting a candidate solution set. Line 4 finds the index I_m, such that, if added to the set of selected indexes, provides the maximum reduction to the cost of the workload. The function $Cost(W, S)$ determines the cost of the workload with the index set S. The loop terminates when no new index satisfy the $sizeConstr$ constraint. Similarly, in the second loop, Line 11 finds the index with maximum improvement for the objective function per unit storage cost. The candidate solution sets for the indexes are called S_1 and S_2 respectively. Since S_1 does not consider the storage overhead of the indexes, it generally selects larger indexes. Selecting large indexes reduces the cardinality of the set S_1. The S_2 set, on the other hand, gives more weight to the index size and holds more indexes of a smaller size. The algorithm then returns a candidate set with higher improvement.

This algorithm directly selects the α_i variables in Eq. 13. By setting the dependent variables, such as α_{iq}, *CoPhy* determines the p_{opq} variables that are used in the objective function in Eq. 11. Therefore, this algorithm not only solves the index selection problem, but also determines the cost of the final objective by cascading the constraints.

Optimizing the Greedy Algorithm: In Alg. 1, finding I_m and I_{ms} dominates the execution time of the greedy algorithm. A naive implementation, which iterates over all possible configurations to determine the benefit of the index table takes several hours for a 15 query workload [18].

We speed up the performance of this algorithm by building an in-memory lookup structure. The structure consists of two large hash tables. The first hash table, named *CostMap*, contains a

mapping of the form $(P_{opq}, T_t) \rightarrow MinCost$. Reusing the notations in Section 3, P_{opq} represents the p^{th} plan of the o^{th} interesting-order combination for query Q_q, and T_t is a table used in the query. The value $MinCost$ is the current minimum cost for accessing data for the table T_t in the plan P_{opq}. The hash table is initialized with $MinCost = \infty$ for each entry. This hash table allows the greedy algorithm to easily find the current best solution for each cached plan. The index I_i is of interest, only if it lowers the current minimum cost of any of the plans.

The second hash table, named *IntMap*, contains the mapping of the form $C_c \rightarrow Q_q \rightarrow List < Plans >$, where C_c is a column in the table, $List < Plans >$ is the list of plans which benefit by using C_c column as an interesting order, and Q_q is used to group all such interesting orders in query Q_q. For each new index I_i, if it covers the interesting order C_c, the greedy algorithm directly looks up the queries and plans benefiting from that interesting order.

Since the greedy algorithm needs to find the queries which benefit from an index, and the amount of benefit the index provides, using these two structures speeds up both the bottleneck functions by a factor of 300 (as shown in Section 6).

C More Workloads

This section discusses the results for *CoPhy* on two more workloads. The first one, NREF, is a real-world protein workload. It shows that for simple queries, the quality improvement of *CoPhy* is on par with commercial tools. Then we investigate further into the effect of query complexity on the quality of *CoPhy*'s results by using a synthetic benchmark–SYNTH.

C.1 Results for NREF

We now discuss the performance of *CoPhy* and other index selection tools for the NREF workload. The NREF database consists of 6 tables and consumes 1.5 GBs of disk space. The NREF workload consists of 235 queries involving joins between 2 and 3 tables, nested queries and aggregation. Since the queries are simpler and use fewer columns, all selection algorithms converge to the same set of a small index set. In *System1* these indexes provide 28% workload speedup and on *System2* the speedup is about 23%. *Greedy* outperforms all other algorithms by suggesting indexes in 19 seconds, and *System1* completes in 5.6 minutes. Since we use approximation method to estimate the query costs, *CoPhy* uses 470 cached plans for the COP and executes in about 1.4 minutes.

C.2 Results for SYNTH

We use the synthetic benchmark to study the behavior of the physical designer in the presence of increasing query complexity. SYNTH is a 1GB star-schema database, containing one large fact table, and smaller dimension tables. The dimension tables themselves have other dimension tables and so on. The columns in the tables are numeric and uniformly distributed across all positive integers. We use 10 queries, each joining a subset of tables using foreign keys. Other than the join clauses, they contain randomly generated select columns, where clauses with 1% selectivity, and order-by clauses. We generate 4 variants of these queries, with the increasing number of candidate indexes, and interesting orders, hence with increasing complexity. Table 1 shows the details of the queries on the database.

Workload	SYNTH1	SYNTH2	SYNTH3	SYNTH4
candidates	381	746	3277	4933
cache size	20	110	418	592
# of tables	1	4.6	4.8	10.5

Table 1: The details of the synthetic benchmark. The first row lists the candidate index set's size for the workloads, the second one lists the INUM cache sizes, and the last row shows the average number of tables in the query.

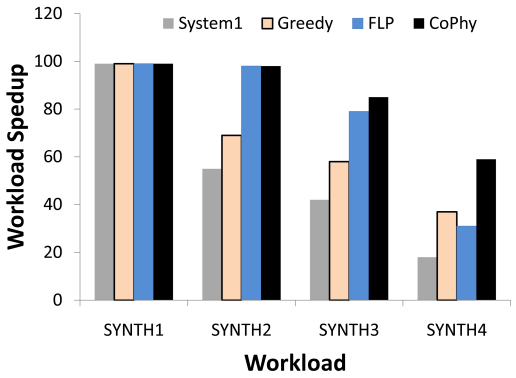

Figure 9: Solution quality comparison for SYNTH workload.

Figure 9 shows the solution quality of the techniques on various SYNTH workloads when the indexes occupy 25% of the space of the tables. On x-axis, we increase the query complexity, and on y-axis we show the solution quality of the techniques on *System1*. When the queries involve

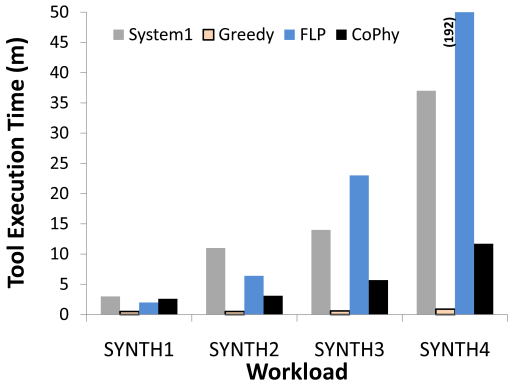

Figure 10: Tool execution time comparison for SYNTH workload.

only 1 table, all techniques perform equally well. The quality of the greedy solutions, however, reduces as the number of tables, hence the interaction between the indexes become important, which the greedy mechanism of *System1* and *Greedy* do not address. FLP performs as well as CoPhy when there are only one or two tables in the query. As the number of tables in the table increases, FLP's solution suffers. Without pruning the FLP formulation creates a COP with about 110 million variables. Solving such a large combinatorial problem is not feasible in today's solvers. Therefore, it prunes away a large fraction of the problem space to produce a problem of the order of tens of thousands. This substantial pruning removes many candidate configurations from the search space, hence reduces the solution quality drastically.

Figure 10 shows the tools' execution times for the different workloads. As expected *Greedy* scales the most, since the number of cached plans are relatively small. The FLP technique, scales the worst, as it spends most of the time pruning away configurations. *System1* scales well with the complexity of the queries, since it uses the optimizer as a black box, the raise in the complexity does not affect its runtime. *CoPhy* scales almost linearly with the INUM cache size as expected from the formulation.