Towards Adaptive, Flexible, and Self-tuned Database Systems

Ioannis Alagiannis
DIAS, I&C, EPFL

***Abstract*—Modern database management systems (DBMSs) are used to answer complex queries on large data sets. The available physical design features along with the complexity of the modern workloads have turned the database physical design into a very complicated task. Manual design is no longer an option and the need for automating physical design tools has become more demanding than ever. DBMSs provide automated physical design in order to maximize performance and reduce the total cost of ownership.**

In this paper, first we study the steps that a typical automated physical designer [1] follows to propose a physical database design. We identify performance bottlenecks in the procedure namely the high number of expensive calls to the optimizer during the evaluation of different configurations and the extensive use of heuristic pruning to reduce the search space of alternative configurations. Then, we present C-PQO [6] an approach that generates a compact representation of the optimization space with a single optimization call per query. C-PQO eliminates the overhead of calling the optimizer again and again during the evaluation of different configurations and improves performance by 30x to over 450x. We also present CoPhy [13]. CoPhy introduces a combinatorial optimization formulation for the index selection problem. Thanks to the convex property of the proposed formulation, CoPhy solves the problem without heuristic pruning of the search space and can predict the quality of the final solution.

***Index Terms*—Automated Physical Design Tuning, C-PQO, CoPhy**

Proposal submitted to committee: July 5th, 2010; Candidacy exam date: July 12th, 2010; Candidacy exam committee: Prof. Pierre Dillenbourg, Prof. Anastasia Ailamaki, Prof. Christoph Koch.

This research plan has been approved:

Date: ________________

Doctoral candidate: ________________
(I. Alagiannis) (signature)

Thesis director: ________________
(A. Ailamaki) (signature)

Doct. prog. director: ________________
(R. Urbanke) (signature)

I. Introduction

DATABASE Management Systems (DBMSs) have been widely deployed in the last decades and they support a vast number of complicated and sophisticated applications. Those applications along with the explosion of available data have increased the importance of database physical design since the selection of correct physical structures (e.g. index) may improve by orders of magnitude the query execution time. Selecting indexes, materialized views, horizontal and vertical partitions that can enhance performance is a challenging optimization problem especially if storage resources are limited. More formally, the Physical Design Problem can be described as follows: Given a workload W and a space budget B, find the set of physical structures or *configuration*, that fits in the B and results in the lowest execution cost for the queries in W.

Manual physical design is both time consuming and very tedious, as the database administrator (DBA) needs to find the benefits of different individual design features that can possible interact with one another. Most large databases are managed by DBAs who are responsible for the good performance of the database. Tuning a database requires experience and combined skills since it involves choices both in hardware and software level. Needless to say, well qualified DBAs are scarce and expensive and with this, human staff becomes a dominating factor of the total cost of ownership (TCO) of databases in large companies. Motivated not only by the difficulty of tuning but also from the need to reduce the total cost of ownership in their products, several commercial DBMS vendors offer automated physical design tools with several features [1][23][18]. These tools provide different options, however, all of them try to solve the aforementioned problem.

The physical design problem involves searching a potentially very large space of different candidate configurations. Searching the space of alternative configurations is impractical. Therefore, recent physical design tools are based on greedy heuristics that prune the search space. These greedy heuristics make the existing design tools feasible. They, however, prune away large fractions of the search space and often suggest locally optimal solutions instead of the globally optimal one. Furthermore, physical designers allow the DBA to estimate the benefit of new physical design features by simulating the design features efficiently and thus, avoid the cost of materialization. To achieve that they base their decisions on ”what-if” questions [14] [10] instead of actually materializing the candidate structures. Nevertheless, the repeated calls to

the optimizer each time we want to evaluate a query under a different configuration impose a serious bottleneck in the execution of physical designers. Based on experimental results [6] 90% of the tuning time is spent on waiting results from the optimizer instead of evaluating potentially promising configurations.

In this paper, we focus on automated physical database design. Initially, we present an overview of a typical physical database designer based on a system evolved for Microsoft SQL Server 2005 [1]. Then we present the configuration-parametric query optimization (C-PQO) [6] technique which reduces the overhead of repeated calls to the optimizer. C-PQO generates a compact representation of the optimization space with a single optimization call per query and based on this representation C-PQO can optimize queries for several configurations without invoking the optimizer. This approach improves query optimization under different candidate configurations by 30x to over 450x and thus, the performance of a physical design tool. We also present CoPhy [13], a physical design tool that provides quality guarantees to the final solution. CoPhy introduces a combinatorial optimization formulation for the index selection problem and takes advantage of the convex property of the proposed formulation to solve the problem without heuristic pruning of the search space.

The rest of the paper is organized as follows: In Section II, we describe a typical automated physical designer based on a system evolved for Microsoft SQL Server 2005. Section III examines the configuration-parametric query optimization technique and Section IV presents CoPhy. Finally, we present the research proposal in Section V and we conclude in Section VI.

II. Overview of a Physical Database Designer

In this section, we present an overview of a typical automated physical database designer. The description is based on the architecture of the Database Tuning Advisor (DTA) for Microsoft SQL Server 2005 [1]. Other commercial database vendors such as IBM [23] and Oracle [18] also provide physical design tools following similar approaches. Nevertheless, we choose DTA because it incorporates state-of-the-art algorithms, provides unique features and is the result of continuous research and development of one of the most active research groups in physical database tuning [16].

Some of the aspects one should consider while designing a physical design tool are the following: the presence of physical structures with different characteristics (e.g. a materialized view is more complex than an index since can involve selection, projection, join and group by), constraints restrictions (e.g. limited budget), manageability requirements (e.g. easy backup), how to estimate the quality of a physical design, scalability of the tool while the database and the workload size increases etc. In the next paragraphs, we present the cost model based on which the designer evaluates the different candidate configurations, the architecture of a typical automated physical database designer and the importance of integrated physical design recommendations.

A. Cost Model

Modern physical design tools perform a quantitative analysis of the impact of a candidate configuration on the input workload using hypothetical (”what-if”) physical structures [14]. The what-if structures allow for simulating the potential benefit from the presence of physical structures such as partitions, indexes and materialized views. The presence of a what-if structure is simulated inside the optimizer by creating meta-data and statistical information in the system catalogs that describe the definition and the distribution of values of the hypothetical structure. The support of what-if structures is very important since the alternative solution of creating and dropping different candidate configurations is costly and impractical. The database physical designers use the what-if structures to evaluate different configurations for the given workload and then they recommend the one with the lowest cost, based on the received feedback (e.g. space consumption, expected execution cost) from the optimizer.

The aforementioned evaluation model keeps the tuning process connected with the query optimizer's cost. This approach has several advantages: (a) The selection of a physical structure is successful only if the optimizer uses it. For example, if the optimizer does not consider a particular index for a query, then there is no point to materialize the index even if we are aware of its usefulness. So, keeping the decision in-sync with the optimizer's cost ensures that if we finally select a candidate configuration it will be used by the query optimizer; (b) The procedure can benefit from the fact that the query optimizer models different aspects of performance (e.g. available memory, number of processors) and the optimizer's cost model evolves over time; (c) Analyzing a potential physical change can be performed without interrupting normal database operations. Nevertheless, it is important to bear in mind that even an advanced query optimizer does not model all the aspects of query execution. Thus, the estimated improvement may be different from the actual improvement if we examine the execution time.

B. Architecture

A typical physical database designer takes as input a database, a workload (e.g. insert, queries, updates), a set of constraints such as storage budget and provides a set of physical design structures which can improve the performance of the given workload. Although implementation details may vary among different automated physical designers, a high-level architecture of a typical physical database designer is illustrated in Figure 1. The basic steps of the architecture are the following:

Column-Group Restriction: Selecting an optimal set of physical structures is computationally hard since it involves searching a vast space of possible configurations. Therefore, it is crucial to prune the search space early. The column-group pre-processing step examines the input workload and eliminates column-groups that occur infrequently in the workload and thus, we expect they will not have significant impact on the quality of the final recommendation. The elimination is performed using a variation of the frequent itemset technique

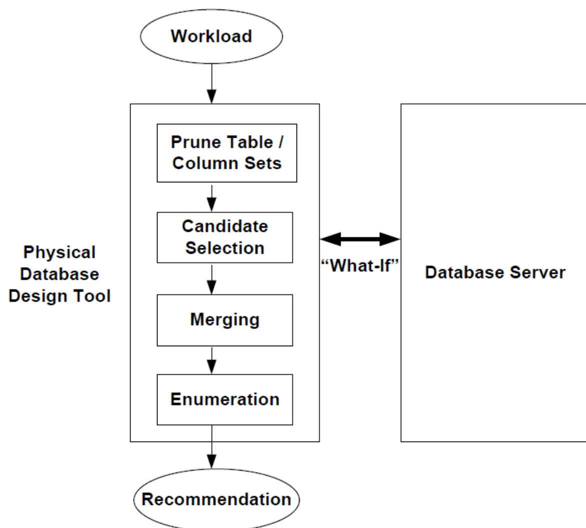


Fig. 1. Overview of the Architecture of a Typical Automated Physical Database Design Tool

[4] [2]. The output of this processing is a set of "interesting" column-groups and is considered during the candidate selection step.

Candidate Selection: During the Candidate Selection step the best configuration for each query is selected (*query-specific-best-configuration*) using a greedy search [9]. The union of these best configurations is used as the candidate set for the next processing steps. The idea behind this choice is that a physical structure that is not part of the best configuration for one of the queries, is unlikely to be part of the best configuration for the whole input workload.

Merging: The candidate set from the Candidate Selection step provides a close to optimal configuration for the input workload. However, this set can lead to an over-specialized physical design which might not be beneficial in case of limited storage budget or update intensive workloads. During the Merge Step, new "merged" physical structures with lower storage budget and update overhead are augmented in the candidate set. The intuition behind this idea is that we should consider not only the structures that are useful for an individual query but also other candidates from the search space that might be useful for multiple queries and as a result optimal for the workload. Merging indexes, partitions and materialized views is a challenging task and specialized algorithms and techniques are needed based on the characteristics of each physical structure [3][2][9].

Enumeration: In the enumeration phase, the set of candidate structures from the two previous steps is used to compute the physical design configuration with the lowest cost for the given input workload. It has been shown that the index selection problem is NP-Hard [11] and adding more physical structures (e.g. materialized views, partitions) makes the selection even more difficult. So, the quality of the final recommendations and the scalability of the enumeration highly depends on the heuristic algorithms which try to prune the search without sacrificing potential useful solutions. The enumeration is performed using greedy search algorithms (bottom-up or top-down search) while different constraints such as storage budget are also considered. The bottom-up search strategies begin with an empty configuration and greedily add new structures. On the other hand, the top-down search strategies begin with an optimal configuration which is progressively refined to be in accordance with storage constraints [5].

C. Integrated Physical Design

There is no doubt that the combination of the right indexes, materialized views and partitions can significantly enhance the performance of a given workload. Nevertheless, the presence of all these physical structures makes the selection problem more challenging for the following reasons. First, physical structures can interact with one another (e.g. an index can depend on how a table is partitioned). Second, different structures can have similar result when they are applied (e.g. a clustered index and a materialized view can both be candidates to reduce the execution cost of the same query). Additionally, it is not easy to decide how to distribute the storage budget among the different physical structures. Finally, these physical structures can have different characteristics when it comes to storage and behavior in updates. For example, a non-clustered index can have higher storage and update cost than a clustered index.

Solutions that stage the selection of the physical design structures (e.g. select partitions first, then indexes and finally, materialized views) can lead to poor recommendations [1]. Thus, an integrated physical design recommendation that considers both performance and manageability is needed. A physical design tool that provides an integrated recommendation has to perform intelligent pruning since the space of possible alternative configurations significantly increases. A detailed description of such a tool can be found in [3].

D. Summary

In this section, we presented the basic characteristics of a typical automated physical database designer based on the Database Tuning Advisor (DTA) for Microsoft SQL Server 2005. For every database there is an optimal physical design but it is very difficult to find it. The alternative configurations are endless and the evaluation cost is too high, making it impossible to explore them all while pruning the search space leads to suboptimal recommendations. The goal of automated physical design tools is to make database tuning and performance analysis manageable. Hand-tuned physical design can have comparable results with automated physical design in simple cases, however, when the complexity increases (more physical structures, updates, larger workload etc.) manually finding a "good enough" physical design becomes really difficult [1]. In the next two sections, we focus on two approaches that improve the performance of the aforementioned architecture. The first reduces the overhead of optimizer invocation without aggressive premature pruning during the evaluation of different configurations and the second solves the problem of selecting indexes using a combinatorial optimization formulation of the problem.

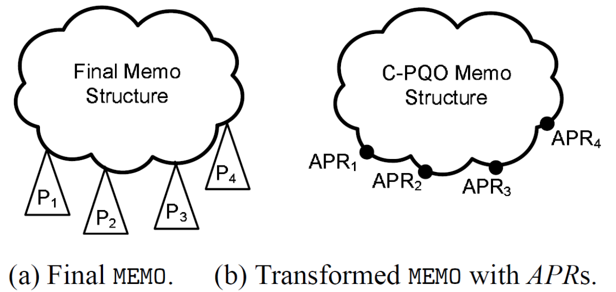
III. Configuration-Parametric Query Optimization for Physical Design Tuning

As we mentioned in the previous section, state-of-the-art physical designers do not materialize alternative physical candidate configurations. Instead, they use what-if structures to simulate the behavior of candidate configurations in DBMSs and evaluate the candidate configurations using the cost model of the query optimizer. The query optimizer can provide reliable estimations of the query execution time. However, query optimization is an expensive process and the repeated optimizer invocations are a bottleneck that increases the execution time, especially when we examine a large set of different configurations. Based on experimental results of an index selection algorithm [5], on average **90%** of its execution time is spent on query optimizer calls. Obviously, the design tool spend most of its time on optimization calls instead of evaluating "promising" candidate configurations.

Motivated by the aforementioned problem, N. Bruno and R. Nehme [6] propose a "cache-and-reuse" technique that can drastically reduce the number of optimizer calls. The technique exploits the fact that distinct query plans do not change significantly across several configurations and hence, can be cached and reused for estimating query cost. The intuition behind this idea is that the input queries are evaluated multiple times across slightly different configurations (e.g. the same configuration except for an index) which leads to repeating the same steps such as query parsing, validation, join reordering etc. multiple times. To avoid repeating the same steps, Configuration-PQO (inspired by the parametric query optimization problem - PQO) issues a single optimization call per query and obtains a representation of the optimization search space. The single optimization call costs more than the regular optimization call, however, it allows to generate alternative execution plans for arbitrary configurations without invoking the optimizer again and again.

C-PQO operates over a top-down, transformational query optimizer having functionality similar to the *Cascade Optimization Framework* optimizer [15]. The top-down optimizer uses a variant of dynamic programming, called memoization to find an optimal plan. Memoization allows the optimizer to ask a MEMO structure if an expression has been already generated. It achieves efficiency by using the principle of optimality [1] which allows an optimizer to restrict the search space. Cascades-based optimizers receive as input an expression of logical operators and transform it to an optimal plan [15] [21]. To achieve that they rely on two components: (1) the MEMO data structure which offers a compact representation of the search space of plans. Each node in the MEMO structure is a *group*, which represents an equivalence class of expressions producing the same output. To reduce memory requirements, a group does not explicitly contain all its operator trees. Instead, expressions in the MEMO are related to one another by parent-child relationships (*multiexpression*: parent expression having as input child group expressions). (2) The optimization tasks which guide the search strategy by applying exploration rules.

[1]Every subplan of an optimal plan is itself optimal for the requested physical properties.



(a) Final MEMO. (b) Transformed MEMO with *APR*s.

Fig. 2. Conceptual Description of *C-PQO*

The intuition of the C-PQO approach is that only a subset of rules, which deals with access path selection in a plan, might be different when we optimize the same query under different configurations. Let us assume that we have computed the MEMO structure for an input query. If we identify the nodes in MEMO that were produced by applying rules that deal with access path selection then, we can recognize the sub-graphs that depend on the configuration. For example, in Figure 2a nodes $P_1, ..., P_4$ are subgraphs that were a result of access path selection rules while everything that is above those sub-graphs is independent of the configuration. Finally, each P_i is replaced with a physical operator APRi (Access Path Request) which contains relevant information about the logical operator tree that triggered the implementation rule. So, when we want to obtain the execution plan and the estimated cost for a given query under an arbitrary configuration we only have to focus on the APRi descriptions in the MEMO$_{C-PQO}$ and infer execution plans for the new configuration. This is possible since MEMO$_{C-PQO}$ contains enough information to derive configuration-specific execution subplans.

Integrating C-PQO into an existing physical design tool is pretty straightforward and it does not change the designer's architecture. C-PQO can be a new component in the already existing architecture. When a request for evaluating a query issued by the tool, the C-PQO component comes into play. If it is the first time we examine the query, then, a unique C-PQO call is made to the DBMS to obtain the search space for the query (MEMO structure). The MEMO structure is used to compute the execution plan and cost for the current request and is cached for future reference.

In order to confirm the efficiency of the proposed technique, the authors examine the overhead of the initial C-PQO optimization call, the speedup and the accuracy of the subsequent optimization calls using a 22 TPC-H [17] query workload. They show that the first optimization call of C-PQO is no more than 3 times than that of a regular optimization call and in many cases around 1.5 times. The subsequent optimization calls for each query in C-PQO are cheaper and as a result the additional overhead of the first expensive call is completely amortized. Figure 3 shows the average speedup of C-PQO over the regular optimizer call under 280 different configurations varies from 34x to 450x. Additionally, the authors analyze the accuracy of the subsequent optimization calls using C-PQO by comparing the estimated execution cost of the plans computed

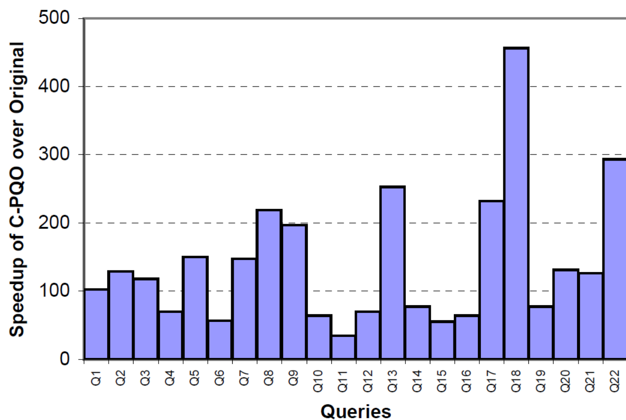


Fig. 3. Speedup of Optimizations in *C-PQO* (after the initial call)

by CPQ-O and the regular optimizer. They found that for the 80% of the calls C-PQO and original optimizer cost differ less than 2.5% while for the 98% of the calls, the error is below 10%.

In this paper, the authors present a technique that significantly reduces the computationally expensive optimizer calls by extracting the search space of all feasible execution plans for a given query. CPQ-O speedups query re-optimization by 30x to over 450x and allows a physical design tool to examine more ”promising” configurations (90% time on search instead of optimizer calls) in less time with virtually no loss in the quality of the results. However, C-PQO binds the process on the SQL Server's top down optimizer. Therefore, it is not straightforward how to port C-PQO to another query optimizer (e.g. the bottom-up optimizer of PostgreSQL). Additionally, the C-PQO technique requires non-trivial changes to the optimizer and is also limited to configurations of primary and secondary indexes without considering partitions.

IV. CoPhy

As we have already mentioned, there are several commercial tools that offer automating tuning with several features [1][23][18]. These tools are based on greedy heuristics. Although these greedy heuristics make the existing design tools practical, they prune away large fractions of the search space and often suggest locally optimal solutions instead of the globally optimal one. Motivated by the shortcoming of existing tools, the authors propose CoPhy [13]. CoPhy formulates the physical design problem of index selection as a combinatorial optimization problem (COP) and shows that there exists a convex formulation which does not require heuristic pruning of the search space and solves the problem using an efficient and scalable lagrangian relaxation method. CoPhy can suggest near-optimal solutions for the physical design problem. Additionally, it can predict the distance of the proposed solution from the optimal solution the tool can compute. Using this information the DBA can trade efficiency (faster execution time) for quality of the suggested solution, estimate the execution time required to arrive at a solution of certain quality and even detect infeasible constraints. Finally, CoPhy can handle user-specified constraints.

A. Formulation as a Combinatorial Optimization Problem

CoPhy avoids the extensive pruning of greedy algorithms thanks to the formulation of the problem as a COP which allows the use of powerful COP solvers. A typical optimization problem tries to minimize (or maximize) an objective function while satisfying some constraints. Lets assume that f is an objective function and g defines the constraints, then an example optimization will be:

$$minimize f(x)$$

$$\text{such that: } g(x) \leq b, x \in R^n$$

Generally speaking, optimizing functions like the one above is difficult. However, if f is convex and g defines a convex area then the problem can be solved in polynomial time. CoPhy's formulation of the problem is convex and as a result polynomial time algorithms can be used to find the optimal solution. Typical physical design tools try to achieve the convex property that CoPhy has thanks to its formulation by enumerating all possible combinations of candidate configurations. However, enumerating all possible combinations leads to huge search space and inevitably to heavy pruning.

The authors present the formulation of the index selection problem as a combinatorial optimization problem in two steps. Initially, index selection for a query and then index selection for the whole workload. CoPhy exploits the idea that plan selection and index selection are mutually complementary since selecting the optimal plans involves selecting the optimal indexes and vice versa. The objective function in this case tries to find the plan with the minimum cost for the query. During this procedure CoPhy uses INUM [19]. INUM similarly to C-PQO caches plans from the optimizer in order to avoid expensive calls. INUM calls the optimizer more than once and caches only few key plans. Using those plans it can model the optimizer's behavior. The key idea of INUM is that even though the space of alternative designs is huge, the number of different optimal query execution plans and, thus, the different plans that the optimizer can return is much lower. Therefore, it makes sense to cache and reuse the optimizer's plans instead of invoking the optimizer multiple times to compute similar plans. Furthermore, in a cached plan, the internal cost (joins-aggregations) remains constant while the query cost depends linearly on how the data are accessed (table scan or indexes).

Index selection for a query: In order to find the optimal configuration for the given query, CoPhy tries to find the optimal plan. Thus, using INUM's properties it formulates the cost of the P_{op} plan (the pth cached plan for the oth interesting order[2]) as the sum of the internal cost and access data cost:

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

[2]An ”interesting order” is a tuple ordering specified by the columns in a query's join, group-by or order-by clause [20].

where $IC(P_{op})$ returns the internal cost plan, $AC(I_i)$ returns the access cost using index I_i and w_{opi} is the constant coefficient for $AC(I_i)$ (for simplicity $w_{opi} = 1$). The cost of P_{op} will be minimal if indexes that cover the interesting orders on the table are used:

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

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

The function $CI(P_{op,T_i})$ returns the indexes that cover the interesting order required by P_{op} on table T_i and the variable a_i is a binary variable that takes 1 if the index I_i is used in the plan. The above formulation is convex since the objective and the constraint function are both linear. To determine the cost for the given input query, we have to find the cached plan for the query with the minimum cost since this plan would be returned by the optimizer.

Index selection for a workload: While examining the problem of index selection for a workload, we should consider that in contrast with the case of a single query, now the plans we select are not independent from each other. Thus, the cost has to be minimized across the whole workload. The problem formulation is the following:

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

$$= \arg\min_{a_{iq}} \sum_{Q_q \in W} (\arg\min_{a_{iq}}(IC(P_{opq})$$

$$\sum_{i=1}^{t} \sum_{I_i \in CI(P_{opq,T_i})} a_{iq} AC(I_i))$$

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

where W is the workload, P_{opq} is the pth plan cached for the oth interesting order of Query Q_q and a_{iq} is a binary variable that indicates when indexes are selected for the whole workload. The current formulation of the problem is not convex (minimization over a set of minimizations). In order to transform this formulation into a convex one, the authors introduce another indicator variable p_{opq} which is 1 if the plan P_{opq} is selected for query Q_q. Now:

$$CL_{opq}{}^{3} = \sum_{i=1}^{t} \sum_{I_i \in CI(P_{opq,T_i})} a_{iq} AC(I_i)$$

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

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

The new constraint indicates when plans are selected for the workload. The current formulation of the the problem is convex. However, the objective function is now quadratic due to the term $p_{opq} a_{iq}$. By adding constraints as follows the formulation becomes linear:

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

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

The last constraint 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 a_{iq} set to 1.

Adding constraints: CoPhy can handle sophisticated constraints as long as they are linear constraints and thus, they do not violate the convexity property. For example, the constraint that at least one index on a table has to contain the column C_c can be expressed by just adding the following constraint function to the program:

$$\sum_{contains(C_c)} a_{iq} \geq 1$$

where a_{iq} is a binary variable having value 1 if an index I_i is used in a query Q_q and *contains* a function that returns the indexes which contain the column C_c. The authors show that the COP formulation has similar expressiveness with the physical constraint language presented by N. Bruno et al. [7].

Solving the problem: CoPhy uses the lagrangian relaxation (LP) technique to relax the problem and a branch-and-bound method to solve it. Branch-and-bound is a method which is based on the observation that the enumeration of all solutions has a tree structure. Each node in the tree is a partial solution. Branch-and-bound examines only the most promising nodes at each step by computing the lower-bound of the branch first. If the lower bound of a branch is higher than the upper bound of the current solution then this branch can be pruned safely. To compute the upper bound and a lower bound CoPhy uses the lagrangian relaxation technique. The lagrangian relaxation technique divides the constraints into two sets. The first set contains "good" constraints with which the problem is easily solvable and the second set contains "bad" constraints that make solving the problem difficult. The LR tries to relax the problem by removing the "bad" constraints and putting them into the objective function with weights (penalty for not satisfying the particular constraint). CoPhy uses as initial upper bound for the branch-and-bound method the solution of a greedy algorithm. In each step it examines the lower bound on a node using the LP technique and if the brand below that node should be examined, it divides the problem into subproblems which tries to solve following the same procedure. In each step, CoPhy is aware of the lower and upper bound of the objective value and so it can estimate the quality of the solution by examining the difference between the upper and the lower bound.

B. Experimental Results

CoPhy provides better solution when it is compared with a Greedy and a FLP [8] algorithm (the advantage of using CoPhy is higher in case of tighter space constraint for indexes).

[3]The term CL_{opq} is introduced to improve readability.

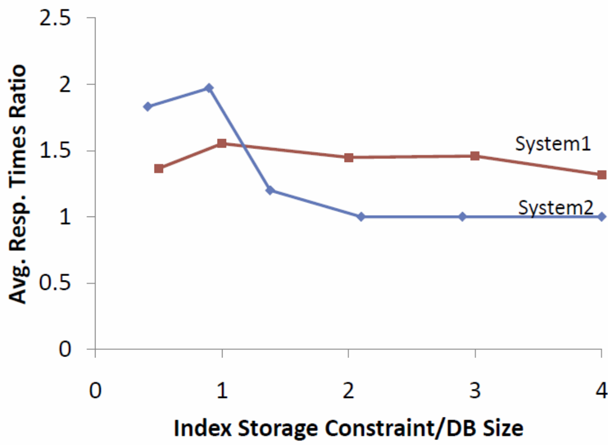


Fig. 4. Quality Comparison of CoPhy vs. Commercial Systems

Additionally, it is important to point out the comparison of CoPhy with two commercial physical design tools. The comparison is made using a workload of 15 TPC-H queries and the relative workload speedup $C_{system-X}/C_{CoPhy}$ is reported. CoPhy always performs better than the first physical designer and better than the second when the available space budget is low (Figure 4).

C. *Conclusions*

CoPhy formulates the problem of index selection as a combinatorial optimization problem, avoids heuristic pruning that is used from greedy approaches, can express interesting constraints and can provide the DBA with information about the distance of the current solution from the optimal one. CoPhy shows better results than existing commercial physical designers, especially in cases of tight constraints and scales almost linearly with the number of queries. The current presented formulation is restricted to index selection and no hints are provided for extending this formulation for other physical structures. Finally, CoPhy's solution is affected by the error on underlying cost model (7% in case of INUM).

V. RESEARCH PROPOSAL

Database Management Systems (DBMSs) have become extremely popular since they first appeared in the 1970's. Through decades of research DBMSs have evolved in a complex and sophisticate software system working over expensive hardware that allows for organizing, analyzing and storing data. They support a large number of different applications, constitute part and parcel of the information technology (IT) industry and a big percentage of today's data is managed using DBMSs. However, the development, deployment and maintenance of a conventional database are not naive tasks and can impose serious restrictions on the adoption of databases in new and existing application areas. Below, we refer to some characteristic examples:

- **Scientific databases:** Scientific fields such as astronomy and biology generate vast volumes of highly complex data at an increasing rate that have to be managed and analyzed. Nevertheless, most scientific databases do not use DBMSs. One reason is that designing and tuning an efficient database physical schema is a challenging optimization problem and definitely not an easy one for a non-expert user. Additionally, scientific databases evolve. New data are received on daily basis and the usage patterns may change as well. So, scientists instead of investing time every day and money on configuring their database, they prefer custom solutions such as Unix-based tools which many times are faster and user-friendlier.
- **Real-time applications:** Conventional DBMSs follow the model of "process-after-store". The data is initially loaded and after a low level tuning is available to the user for query processing. However, the aforementioned model adds significant delay which is not acceptable on real-time systems such as the one used in financial institutions for instant decisions (e.g. exchange in stock market [22]).
- **Simple user applications:** Nowadays, not only the big companies but also the users have their own quickly proliferating data. Music, photos, e-books and movie collections are some typical examples. For this kind of data no user is willing to tune a database (and learn SQL) in order to manage his/her data, so they organize them manually in a file hierarchy.

These are few of the already existing and newly-emerged applications that are different from the traditional OLTP (Online Transaction Processing) and DSS (Decision Support System) applications. In the applications above, we observe that loading the whole data set is not always desirable. For example, in scientific applications in which data is of the size of several Gigabytes per day, it is possible that I am interested only in a small fraction of the new data and it is not clear when or whether I would need the rest of the data. So, why should we spend time to load the whole data set? Additionally, processing the whole data set it is not always the case. Many times, a quick feedback on whether the data is important or not is needed. So, why should I spend time and resources on loading and tuning a system for data that might be useless? Would it not be better if I could run my queries and have the feedback immediately without the initialization overhead? Moreover, DBMSs are a riddle hard to solve that deters non-expert users from using DBMSs and directs them to custom solutions.

In this research proposal, we argue about a new data management system suitable for the aforementioned applications which provides the user data management as a service. The user gives her data and she can immediately start executing queries on the input data without any preparation steps or tuning phases. The whole process remains invisible to the user and the main copy of the data remains outside of the DBMS. Subsets of the data according to the under execution queries is extracted from the input files using scripting languages (awk, perl etc.) and is brought into the database during query processing. It is important to point out that query processing is not performed on data owned by the DBMS but on data that comes into the system when we start executing a query.

Then, this data can be stored using different formats (row-store, column-store etc). Our proposal is a database system which will be flexible in functionality, adaptive to changing requirements both at storage and execution level and capable of maintaining itself as the workload evolves while keeping this process transparent to the user. This project is a challenging step towards a new generation of hybrid DBMSs which are flexible to change their characteristics according to their input. There are different problems that we have to consider while designing this new system, for example:

Query Execution: The first problem involves query execution in an adaptive database system. The way data is stored is critical for the overall performance of a database. A system which is able to use a combination of different formats for storing data (column-store, row-store etc) requires also a specialized execution kernel. For example, column-store and row-store databases use completely different execution strategies so an adaptive kernel should be able to create execution plans which can exploit the characteristics of the underlay storage based on the input queries.

Storage layout: The second and more intriguing problem is the decision of the proper storage layout for a given query workload and how this layout should evolve as the workload changes. This challenging problem involves exploring a combination of alternative physical configurations and dynamically deciding for the appropriate data layout, the proper operators and the execution plans with the highest impact on performance for a given workloads. The goal is a DBMS which will adapt as conditions change without human intervention.

VI. CONCLUSION

Automated physical design is important for achieving high performance while reducing the managing cost of database systems. Improving the physical design automatically has been on the top agenda both in academia and industry. In this proposal, we present a typical automated physical designer and two techniques for improving its efficiency and performance. Finally, we argue for the need of a flexible in functionality, adaptive to changing requirements and capable of maintaining itself as the workload evolves database that will be able to serve new and existing application areas.

REFERENCES

[1] S. Agrawal et al. Database Tuning Advisor for Microsoft SQL Server 2005. In Proceedings of the International Conference on Very Large Databases (VLDB), 2004.

[2] Agrawal, S., Chaudhuri, S., and Narasayya, V. Automated Selection of Materialized Views and Indexes for SQL Databases. Proceedings of VLDB 2000.

[3] Agrawal, S., Narasayya, V., and Yang., B. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In Proceedings of ACM SIGMOD 2004.

[4] Agrawal, R., Ramakrishnan, S. Fast Algorithms for Mining Association Rules in Large Databases. In Proceedings of VLDB 1994.

[5] N. Bruno and S. Chaudhuri. Automatic physical database tuning: a relaxation-based approach. In Proceedings of the SIGMOD Conference, 2005.

[6] N. Bruno and R. Nehme, Configuration-Parametric Query Optimization for Physical Design Tuning, in Proceedings of the ACM International Conference on Management of Data (SIGMOD), 2008

[7] N. Bruno et al. Constrained physical design tuning. PVLDB, 1(1):415, 2008.

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

[9] S. Chaudhuri, S., and Narasayya, V. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. VLDB 1997.

[10] S. Chaudhuri and V. R. Narasayya. AutoAdmin What-if Index Analysis Utility. In SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA, pages 367378, 1998.

[11] S. Chaudhuri, M. Datar, and V. Narasayya. Index selection for databases: A hardness study and a principled heuristic solution. IEEE Transactions on Knowledge and Data Engineering, 16(11):13131323, 2004.

[12] S. Chaudhuri and V. Narasayya, "Self-Tuning Database Systems: A Decade of Progress," in Proceedings of the 33rd International Conference on Very Large Databases, Vienna, Austria, 2007

[13] D. Dash, A. Ailamaki. CoPhy: Automated Physical Design with Quality Guarantees. Technical Report CMUCS-10-109

[14] S. Finkelstein,M. Schkolnick,P. Tiberio: Physical database design for relational databases. ACM ToDS. 1988

[15] G. Graefe. The Cascades framework for query optimization. Data Engineering Bulletin, 18(3), 1995.

[16] The AutoAdmin project. http://research.microsoft.com/dmx/AutoAdmin.

[17] TPC Benchmark H. Decision Support. http://www.tpc.org

[18] Performance Tuning using the SQLAccess Advisor. http://www.oracle.com/technology/products/bi/db/10g/pdf/ twp_general_perf_tuning_using_sqlaccess_advisor_10gr1_1203.pdf

[19] S. Papadomanolakis, D. Dash, A. Ailamaki. Efficient Use of the Query Optimizer for Automated Physical Design. VLDB 2007.

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

[21] L. Shapiro et al. Exploiting upper and lower bounds in top-down query optimization. In Proceedings of IDEAS, 2001.

[22] M. Stonebraker, U. Cetintemel, "One Size Fits All": An Idea Whose Time Has Come and Gone, pp. 2-11, 21st International Conference on Data Engineering, IEEE Computer Society Press, Tokyo, Japan, April 2005, 0-7695-2285-8.

[23] D. C. Zilio, J. Rao, et al. DB2 Design Advisor: Integrated Automatic Physical Database Design. In Proceedings of the 30th Intl. Conf. on Very Large Data Bases, pages 10871097, 2004.