Configuration-Parametric Query Optimization for Physical Design Tuning

Nicolas Bruno
Microsoft Research
Redmond, WA 98052
nicolasb@microsoft.com

Rimma V. Nehme
Purdue University
West Lafayette, IN 47907
rnehme@cs.purdue.edu

ABSTRACT

Automated physical design tuning for database systems has recently become an active area of research and development. Existing tuning tools explore the space of feasible solutions by repeatedly optimizing queries in the input workload for several candidate configurations. This general approach, while scalable, often results in tuning sessions waiting for results from the query optimizer over 90% of the time. In this paper we introduce a novel approach, called *Configuration-Parametric Query Optimization*, that drastically improves the performance of current tuning tools. By issuing a *single optimization call* per query, we are able to generate a compact representation of the optimization space that can then produce very efficiently execution plans for the input query under arbitrary configurations. Our experiments show that our technique speeds-up query optimization by 30x to over 450x with virtually no loss in quality, and effectively eliminates the optimization bottleneck in existing tuning tools. Our techniques open the door for new, more sophisticated optimization strategies by eliminating the main bottleneck of current tuning tools.

Categories and Subject Descriptors

H.2.2 [**Physical Design**]: Access Methods; H.2.4 [**Systems**]: Query Processing

General Terms

Algorithms, Performance

Keywords

Parametric Optimization, Physical Design Tuning

1. INTRODUCTION

Database management systems (DBMSs) increasingly support varied and complex applications. As a consequence of this trend, there has been considerable research on reducing the total cost of ownership of database installations. In particular, physical design tuning has recently become relevant, and most vendors nowadays offer automated tools to tune the DBMS physical design as part of their products (e.g., [1, 8, 15]). Although each solution provides specific features and options, all the tools address a common problem (see Figure 1): given a query workload W and a storage budget B, the task is to find the set of physical structures, or configuration, that fits in B and results in the lowest execution cost for W.

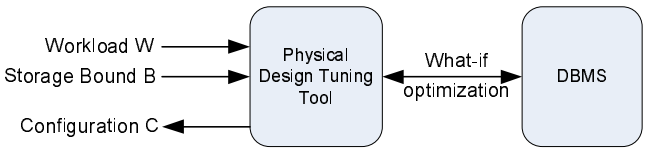


Figure 1: Architecture of existing physical design tools.

The physical design problem as stated above can then be translated into a complex search over a potentially very large space of feasible configurations. Although there are different approaches to conduct this search (e.g., see [3, 6, 14]), a common requirement in all existing solutions is the ability to evaluate the expected cost of a query under a given candidate configuration in the search space. Of course, it has been long established that materializing each candidate configuration in the DBMS and executing queries to obtain their costs is unfeasible in practice. Therefore, existing solutions rely on (i) a *what-if* optimization component [7] that is able to *simulate* a hypothetical configuration C in the DBMS and optimize queries as if C were actually materialized, and (ii) the assumption that the optimizer's estimated cost of a query is a good indicator to the actual execution cost.

Having a *what-if* abstraction enabled the research community to focus on complex search algorithms that relied on this fast-mode of evaluating candidate configurations for the input workload. However, the overhead of a *what-if* optimization call is essentially that of a regular optimization call, which for relatively complex queries can be important. In fact, there is anecdotal evidence that in some cases, over 90% of the tuning time is spent issuing *what-if* optimization calls and waiting for results. Clearly, optimizer calls are a bottleneck in current physical design tools.

A closer inspection of the optimization calls issued by tuning tools reveals that each query in the workload is optimized multiple times for different candidate configurations. Moreover, often these configurations are not so different across each other (e.g., two configurations might share all but a couple of indexes). Intuitively, it would seem that the optimization of a query q under two similar configurations C_1 and C_2 would result in substantial duplication of work (such as query parsing and validation, join reordering, and in general any index-independent process inside the optimizer) and, relatively, just a little amount of configuration-specific work.

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

This situation bears some resemblance to the classical parametric query optimization problem, or PQO [11]. In such context, queries might have unbound parameters at compilation time[1]. Optimizing a parametric query each time it is executed with different parameter values is very expensive. On the other hand, optimizing the query once and reusing the execution plan can be suboptimal if the parameter values are different from those assumed at optimization time. To overcome this problem, PQO optimizes a query once (possibly at a higher overhead than that of a regular optimization call) and obtains back, not a single execution plan, but a structure that encodes a set of candidates that are optimal in some region of the parameter space. Later, when the query is executed with specific parameter values, an appropriate plan is extracted from this structure, which is much faster than re-optimizing the query from scratch.

In this work, we propose a technique inspired by PQO, which we call *Configuration-PQO*, or *C-PQO* for short. The idea is to issue a single optimization call per query (possibly with a larger overhead than that of a regular optimization call), and obtain back a compact representation of the optimization search space that allows us to very efficiently generate execution plans for arbitrary configurations. Then, the modest overhead during the first (and only) optimization call is more than amortized when the same query is re-optimized for different configurations. Our experimental evaluation shows that this approach speeds-up query optimization by 30x to over 450x with virtually no loss in accuracy. We then show that incorporating *C-PQO* into existing physical design tools is straightforward and effectively shifts the overhead away from optimization calls, opening the door for new and more sophisticated search strategies at no perceived additional cost.

We make two assumptions in the rest of this paper. First, we assume that we operate over a top-down, transformational query optimizer (our techniques are in principle applicable to other optimization architectures, but we exploit certain features available in top-down optimizers in our algorithms). Second, we restrict the physical structures in candidate configurations to primary and secondary indexes (but see Section 4.4 for extensions to other structures).

The rest of the paper is structured as follows. Section 2 reviews the *Cascades Optimization Framework*, which is the most well-know example of a top-down transformational optimizer. Section 3 discusses how to extend a Cascades-based optimizer to enable *C-PQO*. Section 4 explains how to infer execution plans and estimated costs for varying configurations in a *C-PQO* enabled optimizer. Section 5 details how to incorporate *C-PQO* into current physical design tools. Section 6 reports an experimental evaluation of our techniques. Finally, Section 7 reviews related work.

2. THE CASCADES FRAMEWORK

In this section we describe the main features of the Cascades Optimization Framework, developed in the mid-nineties and used as the foundation for both industrial (e.g., Tandem's NonStop SQL [5] and Microsoft SQL Server [10]) and academic (e.g., Columbia [2]) query optimizers. Rather than providing a detailed description of all the features in Cascades, we will give a high level overview of the framework followed by a focused description of the components that are relevant to this work (see [9, 13] for more details).

The Cascades Optimization framework results in top-down transformational optimizers that produce efficient execution plans for input declarative queries. These optimizers work by manipulating *operators*, which are the building blocks of *operator trees* and are used to describe both the input declarative queries and the output execution plans. Consider the simple SQL query:

```
SELECT * FROM R,S,T
WHERE R.x=S.x AND S.y=T.y
```

Figure 2(a) shows a tree of logical operators that specify, in an almost one-to-one correspondence, the relational algebra representation of the query above. In turn, Figure 2(c) shows a tree of physical operators that corresponds to an efficient execution plan for the above query. In fact, the goal of a query optimizer is to transform the original logical operator tree into an efficient physical operator tree. For that purpose, Cascades-based optimizers rely on two components: the MEMO data structure (which keeps track of the explored search space) and *optimization tasks*, which guide the search strategy. The following sections discuss these notions in more detail.

2.1 The Memo Data Structure

The MEMO data structure in Cascades provides a compact representation of the search space of plans. In addition to enabling memoization (a variant of dynamic programming), a MEMO provides duplicate detection of operator trees, cost management, and other supporting infrastructure needed during query optimization.

A MEMO consists of two mutually recursive data structures, which we call *groups* and *groupExpressions*. A *group* represents all equivalent operator trees producing the same output. To reduce memory requirements, a *group* does not explicitly enumerate all its operator trees. Instead, it implicitly represents all the operator trees by using *groupExpressions*. A *groupExpression* is an operator having other *groups* (rather than other operators) as children. As an example, consider Figure 2(b), which shows a MEMO for the simple query in the previous section (logical operators are shaded and physical operators have white background). In the figure, *group* 1 represents all equivalent expressions that return the contents of table R. Some operators in *group* 1 are logical (e.g., *Get R*), and some are physical (e.g., *Table Scan*, which reads the contents of R from the primary index or heap, and *Sorted Index Scan*, which does it from an existing secondary index). In turn, *group* 3 contains all the equivalent expressions for $R \bowtie S$. Note that *groupExpression* 3.1, *Join(1,2)*, represents all operator trees whose root is *Join*, first child belongs to *group* 1, and second child belongs to *group* 2. In this way, a MEMO compactly represents a potentially very large number of operator trees. Also note that the children of physical *groupExpressions* also point to the most efficient *groupExpression* in the corresponding *groups*. For instance, *groupExpression* 3.8 represents a hash join operator whose left-hand-child is the second *groupExpression* in *group* 1 and whose right-hand-child is the second *groupExpression* in *group* 2.

In addition to representing operator trees, the MEMO provides basic infrastructure for management of *groupExpression* properties. There are two kinds of properties. On one hand, *logical* properties are shared by all *groupExpressions* in a *group* and are therefore associated with the *group* itself. Examples of logical properties are the cardinality of a *group*, the tables over which the *group* operates, and the columns that are output by the *group*. On the other hand, *physical* properties are specific to physical *groupExpressions* and typically vary within a *group*. Examples of physical properties are the order of tuples in the result of a physical *groupExpression* and the cost of the best execution plan rooted at a *groupExpression*.

We introduce additional functionalities of the MEMO data structure as needed in the rest of the paper.

2.2 Optimization Tasks

The optimization algorithm in Cascades is broken into several *tasks*, which mutually depend on each other. Intuitively, the opti-

[1]Traditionally, parameters can be either system parameters, such as available memory, or query-dependent parameters, such predicate selectivity.

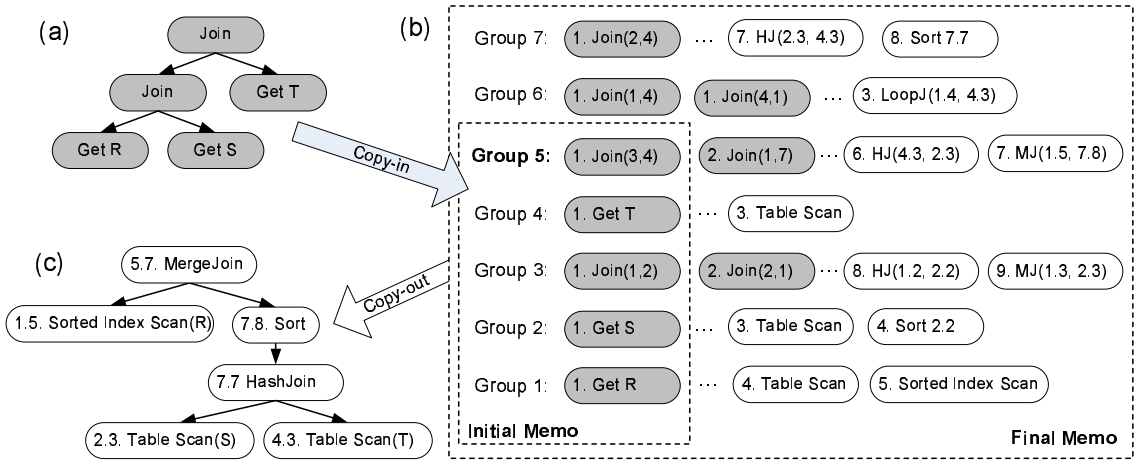


Figure 2: The MEMO data structure in a Cascades-based optimizer.

mization of a query proceeds as follows. Initially, the logical operator tree describing the input query is *copied into* the initial MEMO (see Figure 2(b)) for an example. Then, the optimizer schedules the optimization of the *group* corresponding to the root of the original query tree (*group* 5 in the figure). This task in turn triggers the optimization of smaller and smaller operator sub-trees and eventually returns the most efficient execution plan for the input query. This execution plan is *copied out* from the final MEMO and passed to the execution engine. Figure 3 shows the five classes of tasks in a Cascades-based optimizer, and the dependencies among them. We next describe each of these tasks in some detail.

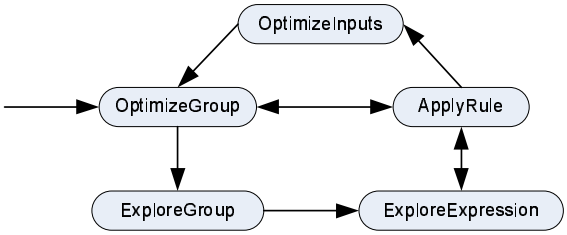


Figure 3: Optimization tasks in a Cascades-based optimizer.

OptimizeGroup: This task takes as inputs a *group* G, a cost bound UB, and a set of *required physical properties* RP. It returns the most efficient execution plan (if exists) that implements *group* G, costs no more than UB and satisfies the required properties RP. This task implements memoization by caching the best execution plan for a given set of required properties in the *winner circle* of the *group*. Subsequent calls to *OptimizeGroup* with the same required physical properties would return immediately with the best plan or a failure (depending on the value of UB). Operationally, optimizing a *group* entails exploring the *group* (see *ExploreGroup* task), and then applying all implementation rules (see *ApplyRule* task) to produce all the candidate physical operators that implement the logical counterparts in the *group*.

ExploreGroup: A *group* is explored by iteratively exploring each logical *groupExpression* in the *group* (see *ExploreExpr* task).

ExploreExpr: Exploring a logical *groupExpression* generates all logically equivalent alternatives of the input *groupExpression* by applying exploration rules (see *ApplyRule* task). This task also uses memoization to avoid repeated work. Consider a "join-commutativity" rule that transforms *groupExpression* G_1 into G_2. When eventually exploring G_2, it would not make sense to apply "join-commutativity" again, since we would obtain G_1 back. *ExploreExpr* uses a bitmap, called *pattern memory*, that keeps track of which transformation rules are valid and which ones should not be applied.

ApplyRule: In general, each rule is a pair of an antecedent (to match in the MEMO) and a consequent (to generate and introduce back in the MEMO). An example is the *join-associativity* rule "$\texttt{JOIN}(g_1, \texttt{JOIN}(g_2, g_3)) \rightarrow \texttt{JOIN}(\texttt{JOIN}(g_1, g_2), g_3)$"[2]. The *ApplyRule* task can be broken down into four components. First, all the bindings for the rule's antecedent are identified and iterated over one by one (for complex rules, there can be different ways of matching the antecedent of the rule with operator trees in the current *group*). Second, the rule is applied to each binding generating one or more new expressions (for the rule above, there is a single substitute per rule application, but in general there might be more than one). Third, the resulting expressions are integrated back into the MEMO, possibly creating new, unexplored *groups* (as an example, applying the join associativity rule to expression 5.1 in Figure 2(b) results in *groupExpression* 5.2, which points to a newly created *group* 7). Finally, each new *groupExpression* triggers follow-up tasks, which depend on its type. If it is a logical operator, the optimizer was exploring the *group* and thus an *ExploreExpr* task is scheduled for the new *groupExpression*. Otherwise, the expression inputs are optimized and the cost of the physical plan is calculated (see *OptInputs* task).

OptInputs: This task optimizes the inputs of a given physical operator p and computes the best execution plan rooted at p. For each input p_i, it first calculates the required properties of p_i with respect to p and then schedules an *OptimizeGroup* task for the *group* of p_i. As an example, suppose that the root of the tree is a `MergeJoin` operator. Since `MergeJoin` expects the inputs in a specific order, the current task generates a required sort property for each of the inputs and optimizes the corresponding *groups* under this new optimization context. The *OptInputs* task also implements a cost-based

[2]This is a very simple exploration rule. More complex rules, especially implementation rules, have right sides that cannot be expressed as succinctly.

```
OptimizeGroup (group G, properties RP, double UB)
returns groupExpression
01 p = winnerCircle[G, RP]
02 if ( p is not NULL )
       if (p.cost < UB) return p
       else return NULL
03 bestP = NULL // No precomputed solution, enumerate plans
04 for each enumerated physical groupExpression candP
05     candP.cost = localCost(candP)
06     for each input p_i of candP
07         if (candP.cost≥UB) go back to 4 // out of bound
08         G_i = group of p_i
           RP_i = required properties for p_i
09         bestP_i= OptimizeGroup(G_i, RP_i, UB − candP.cost)
10         if (bestP_i = NULL) break // no solution
11         candP.bestChild[i] = bestP_i
12         candP.cost += bestP_i.cost
       // Have valid solution, update state
13     if (candP.cost < UB and candP satisfies RP)
           bestP = candP
           UB = candP.cost
14 winnerCircle[G, RP] = bestP
15 return bestP
```

Figure 4: Simplified pseudocode for the *OptimizeGroup* task.

pruning strategy. Whenever it detects that the lower bound of the expression that is being optimized is larger than the cost of an existing solution, it fails and returns no plan for that optimization goal.

As we can see, the five optimization tasks are non-trivial and depend on each other. For clarity purposes, we now present a conceptually simplified version of the *OptimizeGroup* task that incorporates the portions of the remaining tasks that are relevant for this work. Figure 4 shows a pseudocode for *OptimizeGroup*, which takes as inputs a *group* G, required properties RP, and a cost upper-bound UB. *OptimizeGroup*(G, RP, UB) returns the most efficient physical *groupExpression* that satisfies RP and is under UB in cost (otherwise, it returns `NULL`). Initially, line 1 checks the winner circle (implemented as an associative array) for a previous call compatible with RP. If it finds one, it returns either the best plan found earlier (if its cost is below UB) or `NULL` otherwise. If no previous task is reusable, line 4 iterates over all enumerated physical *groupExpressions* in G (note that, strictly speaking, line 4 encapsulates *ExploreGroup*, *ExploreExpr*, and *ApplyRule*). For each such root *groupExpression* p, line 5 estimates the local cost of $candP$ (i.e., without counting its inputs' costs). Then, line 8 calculates the input *group* and required properties for each of $candP$'s inputs and line 9 recursively calls *OptimizeGroup* to optimize them (note that the upper bound in the recursive call is decreased to UB-$candP.cost$). After each input is successfully optimized, lines 11 and 12 store the best implementation for each of $candP$'s children and update its partial cost. Note that, if at any moment the current cost of $candP$ becomes larger than UB, the candidate is discarded and the next one is considered (line 7). Otherwise, after $candP$ is completely optimized, line 13 checks whether $candP$ is the best plan found so far. Finally, after all candidates are processed, line 14 adds the best plan to the winner circle for G and RP, and line 15 returns such plan.

2.2.1 The Rule Set

One of the crucial components in a Cascades-based optimizer is the rule set. In fact, the set of rules that are available to the optimizer is one of the determining factors in the quality of the resulting plans. On one side, *exploration* rules transform logical operator trees into equivalent logical operator trees, and can range from simple rules like join commutativity to more complex ones like pushing aggregates below joins. On the other side, *implementation* rules transform logical operator trees into hybrid logical/physical trees by introducing physical operators into the `MEMO`. Again, implementation rules can range from simple ones like transforming a logical join into a physical hash join, to much more complex ones. In the remainder of this section we provide additional details on a small subset of implementation rules that produce access path alternatives, since these are relevant to our work.

One of such rules transforms a logical expression consisting of a selection over a single table[3] into a physical plan that exploits the available indexes (candidate plans include index scans, rid intersections and lookups among others). After binding the logical operator tree, this rule identifies the columns that occur in *sargable* predicates, the columns that are part of a required sort property, and the columns that are additionally referenced in non-sargable predicates or upwards in the query tree. Then, it analyzes the available indexes and returns one or more candidate physical plans for the input sub-query.

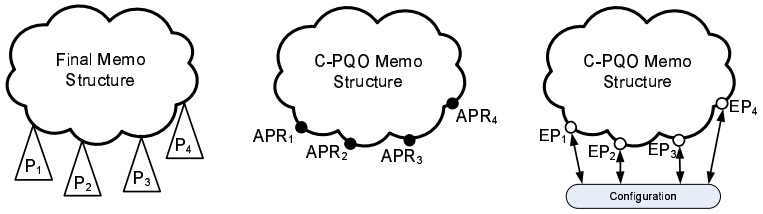
Consider the application of such a rule for a *groupExpression* that represents $\Pi_c(\sigma_{a=10}(R))$, and further suppose that column b is a required sort order. In this case, the rule identifies column a in a sargable predicate, column b as a required order, and column c as an additional column that is either output or referenced upwards in the tree. This information allows the optimizer to identify the available indexes that might be helpful to implement an efficient sub-plan for the sub-query. Suppose that an index on column a is available. The optimizer can then generate a physical operator tree that uses the index to retrieve all tuples satisfying a=10, fetches the remaining columns from a primary index, and finally sorts the resulting tuples in b order. If an index on columns (b, a, c) is also available, the optimizer might additionally generate an operator tree that scans the index in b order and filters on the fly the tuples that satisfy a=10. Depending on the selectivity of a=10, one alternative would be more efficient than the other, and eventually the optimizer would pick the one that results in the smallest execution cost.

Note that the same mechanism is used in another equally important rule that transforms logical joins with single-table right-hand-sides into index-nested-loops execution plans (which repeatedly access an index on the right-hand-side's table for each tuple produced by the left-hand-side input). In this case, the same procedure is conducted with respect to the inner table only, and the joined column in the table is considered as part of a sargable (equality) predicate. For instance, suppose that the logical sub-plan is $(\mathcal{Q} \bowtie_{\mathcal{Q}.x=T.y} T)$, where $\mathcal{Q}$ represents an arbitrary complex expression. Conceptually, the rule produces an index-nested-loop plan and considers the right-hand-side as a single-table-selection $\sigma_{T.y=?}(T)$ as before (where $T.y$ is a column in a sargable predicate with an unspecified constant value).

3. C-PARAMETRIC OPTIMIZATION

As explained in the previous section, among the large set of rules in a Cascades-based optimizer there is a small subset that deals with access path selection. Intuitively, these are the only rules that might make a difference when optimizing the same query under different configurations. Figure 5 shows a high level, conceptual illustration of our approach for *C-PQO*. Consider the final `MEMO` structure after optimizing an input query q. If we only consider physical *groupExpressions*, this final `MEMO` is simply a directed acyclic graph, where each node is a *groupExpression* and edges going out of a

[3]The antecedent *groupExpression* is typically obtained after applying earlier rules, such as pushing selections under joins.



(a) Final MEMO. (b) Transformed MEMO with *APR*s. (c) Inferring execution plans.

Figure 5: Conceptual description of *C-PQO*.

node G connect G with its best children *groupExpressions*. Now suppose that we identify the nodes in the MEMO that were a result of applying some rule that deals with access path selection (these sub-graphs are marked as $P_1, \ldots, P_4$ in Figure 5(a)). Then, everything that is above the P_i sub-graphs is independent of the configuration. We can then modify the final MEMO structure by replacing each P_i with a concise description of the logical operator tree that produced such physical execution plan (we denote such descriptions as *APR*$_i$ in Figure 5(b) and formally define them in Section 3.1). Whenever we want to re-optimize the query under a different configuration we can just focus on the *APR*$_i$ descriptions and infer small execution plans that satisfy their requirements under the new configuration (denoted EP_i in Figure 5(c)). We can then extract the best execution plan from the resulting MEMO (whose bulk was pre-computed) in a much more efficient manner.

Although the conceptual idea behind *C-PQO* is simple, there are, however, significant challenges to address. First, we need a simple and accurate description of the properties of each physical operator tree P_i that is efficient to generate (see Section 3.1). Second, the final MEMO should consider execution plans that can be generated by arbitrary configurations (also see Section 3.1). To better understand this point, consider the rule that implements index-nested-loop joins for logical joins with single-table right hand sides, which we described in Section 2.2.1. If at optimization time there is no index on the right-hand-side table, the optimizer would not even generate an execution plan P_i and we would miss this alternative when later optimizing under a configuration that contains such index. Third, as described in Figure 4, optimizers implement branch-and-bound techniques to prune the search space. In our situation, however, we should not eliminate alternatives from consideration unless we can guarantee that no possible configuration might result in the alternative being useful (see Section 3.2). Finally, current optimizers only keep the most efficient *groupExpression* for every child group of a given *groupExpression*. In the context of *C-PQO*, there is no single "most efficient" *groupExpression*, since this would depend on the underlying configuration. Thus, we should be able to track all possible alternatives that might become part of the final execution plan (see Section 3.3). In the remainder of this section, we explore each of these challenges.

3.1 Intercepting Access-Path Selection Rules

To isolate the variable portion of the optimization process with respect to varying configurations, we instrument the optimizer as follows, extending the techniques in [3, 4]. We create a new physical operator, which we call *APR* (for access path request). Then, we modify the generation of substitutes for all the rules that are related to access path selection (see Section 2.2.1) so that they *always* return an *APR* singleton operator tree rather than an actual physical execution tree. In this way, we make access-path-selection rules configuration-independent, and we do not miss any potential execution sub-plan. The *APR* physical operator contains the relevant information about the logical operator tree that triggered the implementation rule. Specifically, we store in the *APR* node the tuple (S, O, A, N), where S is the set of columns in sargable predicates and the predicate cardinalities, O is the sequence of columns for which an order has been requested, A is the set of additional columns used upwards in the execution plan, and N is the number of times the sub-plan would be executed[4]. Note that (i) S, O, and A may intersect, and (ii) N is greater than one only if the sub-plan is the right-hand-side of an index-nested-loop join. Intuitively, the information in *APR* nodes encodes the properties that any physical execution sub-plan must satisfy in order to implement the logical operator tree that triggered the corresponding implementation rule.

Consider as an example the following query:

$$\Pi_{R.c,S.b}\big(\sigma_{R.a=5}(R) \bowtie_{R.x=S.y} S\big)$$

Figure 6 shows a partial MEMO when optimizing the above query. There are four *APR* nodes resulting from different implementation rules. As an example, *APR*$_2$ was generated by the rule that implements index strategies for single-table selections. The information contained in this operator specifies that (i) there is one sargable column $R.a$ returning 25,000 tuples, (ii) there is no order requested, (iii) the columns that are required are $R.c$ and $R.x$, and (iv) the sub-plan would be executed once. Similarly, *APR*$_4$ in *group* 5 was generated as part of the implementation rule that transforms joins with single-table right-hand-sides into index-loop joins. It specifies that $S.y$ is a sargable column which would be sought with 25,000 bindings. Finally, *APR*$_1$ and *APR*$_3$ are generated as part of the rule that implements physical scans over tables.

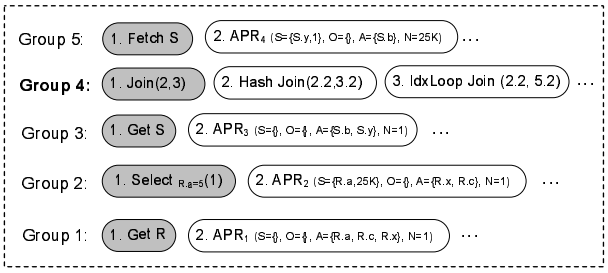


Figure 6: Partial MEMO for a simple input query.

Note that this mechanism results in the following two crucial properties. First, the leaf nodes in the MEMO are always *APR* physical operators. Second, there are no remaining rules in the optimizer that depend on the specific configuration, so the optimization of queries is truly configuration-independent. Also note that we still have not shown how to calculate the cost of an *APR* physical operator. We next address this issue.

[4]We store additional details in *APR* nodes, but we omit such details to simplify the presentation.

3.2 Relaxing the Search Space

The original Cascades formulation uses a variant of branch-and-bound pruning to avoid exploring alternatives that are guaranteed to be sub-optimal. Specifically, line 7 in Figure 4 completely discards a *groupExpression* –without even optimizing all its children– if it costs more than the upper bound UB. While this pruning makes sense in a traditional optimizer, it would be wrong to apply it for *C-PQO* since the ultimate cost of an expression in fact depends on the actual configuration, and therefore making any aggressive pruning would result in potential loss of optimal plans for some valid configurations.

A simple solution for this problem would be to simply remove the pruning altogether (i.e., remove line 7 from Figure 4. While this approach is correct, it might result in much longer optimization calls. Instead, we next show how we can improve this simple approach by eliminating all the candidates that cannot be part of any solution under any configuration. At the same time, we illustrate how to calculate the cost of an *APR* physical operator.

Extending the Cost Model

One of the difficulties of handling *APR* physical operators is that they are really a specification of the required properties that any sub-execution plan must satisfy in order to be a valid alternative. Therefore, there is no precise notion of the cost of an *APR*, since it depends on the actual configuration. However, there are precise bounds on the cost of such *APR*. On one hand, there exists a configuration with the right indexes that makes a given *APR* execute as fast as possible. Conversely, the configuration that contains no indexes is guaranteed to result in the worst possible implementation of any given *APR*. Therefore, instead of having a single cost for each physical operator in the tree, we propose to maintain two costs, denoted *bestCost* (which is the smallest possible cost of any plan implementing the operator tree under any configuration), and *worstCost* (which is the largest smallest possible cost of any execution plan over all configurations, and effectively is the smallest possible cost under the configuration that contains no indexes).

Values of *bestCost* and *worstCost* are calculated very easily for non-leaf nodes in the `MEMO`. In fact, the *bestCost* of an operator is the local cost of the operator plus the sum of the minimum *bestCost* of each child (*worstCost* values are calculated analogously). We next describe how to obtain *bestCost* and *worstCost* for the leaf *APR* physical operators.

Consider an *APR* node with information *(S,O,A,N)* where each element in S contains a column, the predicate type (i.e., equality or inequality), and the cardinality of the predicate. Similarly to the work in [4], we obtain the index-based execution plan that leads to the most efficient implementation of the logical sub-tree as follows:

1. Obtain the best "seek-index" I_{seek} containing (i) all columns in S with equality predicates, (ii) the remaining columns in S in descending cardinality order, and (iii) the columns in $(O \cup A) - S$.
2. Obtain the best "sort-index" I_{sort} with (i) all columns in S with single equality predicates (they would not change the overall sort order), (ii) the columns in O, and (iii) the remaining columns in $S \cup A$.
3. *bestCost*(*APR*) is defined as the minimum cost of the plan that implements *APR* with either I_{seek} or I_{sort} (see Section 4.2 for a mechanism to calculate the cost of a plan that uses a given index to implement an *APR* operator).

Obtaining the value of *worstCost* for a given *APR* operator is simpler. We need to evaluate the cost of implementing the *APR* operator with just a heap (i.e., without any index). The following lemma establishes the correctness of our approach[5]:

LEMMA 1. *Let α be a physical* APR *operator. The procedure above returns the minimum possible cost of any execution plan implementing α under any valid configuration and the largest cost of the optimal execution plan for α under any valid configuration.*

As an example, consider an *APR* operator α with information: $[S{=}(a_{[eq,300]}, b_{[lt,200]}, c_{[gt,100]}), O{=}(d), A{=}(e), N{=}1]$. Then, the value of *bestCost* is the minimum cost of the plan that uses either $I_{seek} = (a, c, b, d, e)$ or $I_{sort}{=}(a, d, b, c, e)$. Similarly, the value of *worstCost* is the cost of the plan that uses a heap to implement α.

Pruning the Search Space

With the ability to calculate *bestCost* and *worstCost* values for arbitrary physical operators in the `MEMO` structure, a relaxed pruning rule can be implemented as follows:

Relaxed pruning rule: Every time that the partial *bestCost* of a *groupExpressions* g is larger than the *worstCost* of a previous solution for the group under the same optimization context, eliminate g from consideration (we evaluate the effectiveness of this pruning step in the experimental evaluation).

LEMMA 2. *The relaxed pruning rule does not eliminate any execution plan that can be optimal for some valid configuration.*

3.3 Putting it all Together

We now show the modified version of the *OptimizeGroup* procedure of Figure 4 with all the changes required to support *C-PQO*. Figure 7 shows a pseudocode of the new procedure, which we call *OptimizeGroup*$_{C\text{-}PQO}$. We next describe this new procedure and contrast it with the original version.

The first difference in *OptimizeGroup*$_{C\text{-}PQO}$ is the input/output signature. The new version does not accept an upper bound UB as input, since this is used in the original branch-and-bound pruning strategy, which we do not rely on anymore. Also, the output of the new procedure consists not of a single *groupExpression* but of the full set of *groupExpressions* for group G and required properties RP. The second difference in the new version is that we replace the original *winnerCircle* structure with the more general *alternativePool* associative array, which returns the set of all valid *groupExpressions* for a given group and set of required properties.

For a given input group G and required properties RP, algorithm *OptimizeGroup*$_{C\text{-}PQO}$ first checks the *alternativePool* data structure for a previous call compatible with RP. If it finds one, it returns the set of *groupExpressions* previously found, effectively implementing memoization. Otherwise, line 4 iterates over all enumerated physical *groupExpressions* in G (note that this line encapsulates the changes to the rules that select access paths and now return *APR* operators). For each such root *groupExpression* $candP$, line 5 estimates the local values of *bestCost* and *worstCost* for $candP$ (in case of *APR* operators, we use the procedure of Section 3.2, while in the remaining cases both *bestCost* and *worstCost* values are equal to the original local cost of the operator). Then, line 8 calculates the input *group* and required properties for each input of $candP$ and line 9 recursively calls *OptimizeGroup*$_{C\text{-}PQO}$ to optimize them. After each input is successfully optimized, lines 11 and 12 store the set of candidates for each of $candP$'s children in the array *allChildren* and update its partial values of *bestCost*

[5]We omit proofs due to space constraints, but note that the results depend on current query processing models. New access methods or implementation strategies would require extending the lemma.

```
OptimizeGroup_C-PQO (Group G, Properties RP)
returns Set of groupExpressions
01 allP = alternativePool[G, RP]
02 if (allP is not NULL) return allP
// No precomputed solution, enumerate plans
03 candPool = ∅
   UB = ∞
04 for each enumerated physical groupExpression candP
05    candP.bestCost = localCostForBest(candP)
      candP.worstCost = localCostForWorst(candP)
06    for each input p_i of candP
07       if (candP.bestCost ≥ UB)
            continue back to 3 // out of bounds
08       G_i = group of p_i
         RP_i = required properties for p_i
09       allP_i = OptimizeGroup_C-PQO(G_i, RP_i)
10       if (allP_i = ∅) break // no solution
11       candP.allChildren[i] = allP_i
12       candP.bestCost += min_{c_i ∈ allP_i} c_i.bestCost
         candP.worstCost += min_{c_i ∈ allP_i} c_i.worstCost
      // Have valid solution, update state
13    candPool = candPool ∪ {candP}
14    if (candP.worstCost < UB)
         UB = candP.worstCost
15 alternativePool[G, RP] = candPool
16 return candPool
```

Figure 7: Modified *OptimizeGroup* task for *C-PQO*.

and *worstCost*. Note that, if at any moment the current *bestCost* of $candP$ becomes larger than the *worstCost* of a previously calculated solution, the candidate is discarded and the next one is considered (line 7). Otherwise, after $candP$ is completely optimized, line 13 adds it to the candidate pool for the G and RP and line 14 updates the upper bound with the *worstCost* value of $candP$ if applicable. After all candidates are explored, line 15 updates the *alternativePool* array and line 16 returns the set of candidate *groupExpressions*.

As we can see, even though several changes are required to enable *C-PQO* in a Cascades-based query optimizer, the main structure of the optimization tasks remain similarly organized.

The Output of a *C-PQO* Optimization

After executing *OptimizeGroup*$_{C\text{-}PQO}$ on the root node of the initial MEMO with the global set of required properties, we need to prepare the output of a *C-PQO* optimization. We do so by extracting an AND/OR subgraph of physical operators from the final MEMO[6]. Each node in the output graph can be of one of two classes. On one hand, AND nodes contain the actual physical *groupExpressions* along with their *bestCost* and *localCost* values (the *localCost* value is the one defined in line 5 of Figure 7 and can be obtained back by subtracting the *bestCost* values of the best children of a *groupExpression* from its own *bestCost* value). Each outgoing edge of an AND node represents a child of the corresponding *groupExpression* and goes into an OR node. OR nodes, in turn, correspond to candidate choices, represented by outgoing edges into other AND nodes. Additionally, each OR node distinguishes, among its choices, the one that resulted in the minimum *bestCost* value (i.e., the child that contributed to `candP.bestCost` in line 12 of Figure 7). Finally, we add a *root* OR node that has outgoing edges towards every AND node that corresponds to a *groupExpression* in the root node of the final MEMO satisfying the original required properties (since there might be many alternatives to implement the query, this root OR node represents the highest-level choice). Figure 8 shows a sample AND/OR graph induced by a partial MEMO structure (the notation $7.\{1,2,3\}$ in physical operators refers to the set of *groupExpressions* $\{7.1, 7.2, 7.3\}$, and the best alternative for an OR node is shown with a bold arrow). In the rest of the paper, we use MEMO$_{C\text{-}PQO}$ to denote the AND/OR graph induced from the final MEMO produced by a *C-PQO* optimizer.

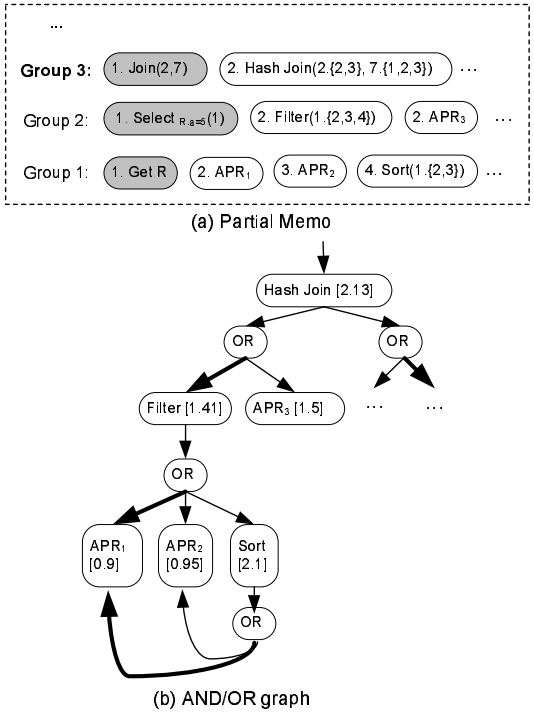


Figure 8: AND/OR graph induced from a MEMO.

4. FAST RE-OPTIMIZATION IN C-PQO

The MEMO$_{C\text{-}PQO}$ described in the previous section encapsulates all the optimization state that is possible to obtain without knowing a specific configuration instance. Additionally, it contains enough information to derive configuration-specific execution sub-plans. In this section we describe how to obtain an execution plan and the estimated cost for a query given its MEMO$_{C\text{-}PQO}$ under an arbitrary configuration. In Section 4.1 we describe a one-time post-processing pass on the MEMO$_{C\text{-}PQO}$ that simplifies subsequent re-optimizations. In Section 4.2 we describe how we deal with leaf *APR* nodes. Next, in Section 4.3 we describe the re-optimization algorithm. Finally, in Section 4.4 we discuss extensions to our basic approach.

4.1 Initialization

Before the first re-optimization, we perform a couple of quick transformations in the MEMO$_{C\text{-}PQO}$ to reduce subsequent work. Since *APR* nodes are produced by modified implementation rules, there might be several instances of such rules that produce *APR* nodes that are indistinguishable. Since our techniques do some work for each *APR* in the MEMO$_{C\text{-}PQO}$ on a re-optimization, we collapse all identical *APR* nodes into a single representative. We do it efficiently by using a hash table of *APR* nodes. Since the final number of distinct *APR* nodes is fairly small (see the experimental evaluation), this step is very efficient. Additionally, we perform a quick consolidation of OR nodes in the MEMO$_{C\text{-}PQO}$. Recall that OR nodes represent a subset of choices in a given *group*. We therefore analyze each OR node and collapse those that are defined on the same

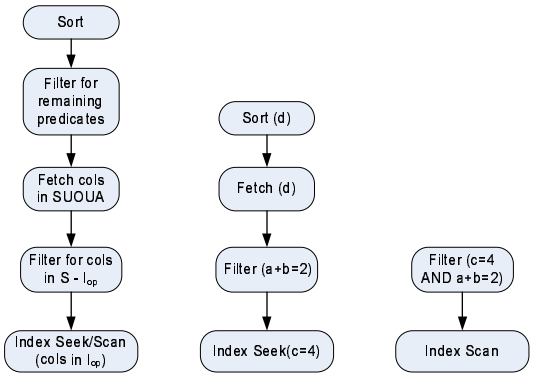
[6]We use a normalization phase that relies on pointer unswizzling to efficiently produce a heavily compressed serialization of the MEMO that is then exported to the client.

group and agree on all the alternatives. Since *group* ids are consecutive integers, we can do this step efficiently by using an array indexed by the OR node *group* id. Comparing whether two OR nodes are identical is also efficient, because the MEMO$_{C\text{-}PQO}$ returns *groupExpressions* in the OR node sorted by *groupExpression* id, and therefore the set-comparison can be done in linear time. Finally, we perform a bottom-up traversal of the resulting graph and eliminate all OR nodes that have no AND children (e.g., because there were no successful implementations during optimization due to contradictory required properties), and all AND nodes for which there is at least one child for which we eliminated the corresponding OR node.

4.2 Leaf Node Calculation

A crucial component in our technique is the ability to calculate the cost of the best plan for a physical *APR* operator for a given input configuration. Consider a physical *APR* operator α =(S, O, A, N). Suppose that we want to calculate the cost of an alternative sub-plan that uses an index I over columns $(c_1, \ldots, c_n)$ to implement α. In order to do so, we simulate the implementation rules that produced α in the first place, and approximate what the optimizer would have obtained under the hypothetical configuration that contains index I. Let I_α be the longest prefix $(c_1, \ldots, c_k)$ that appears in S with an equality predicate, optionally followed by c_{k+1} if c_{k+1} appears in S with an inequality predicate. We can then implement α by (i) seeking I with the predicates associated with columns in I_α, (ii) filtering the remaining predicates in S that can be answered with all columns in I, (iii) looking up a primary index to retrieve missing columns in $\{c_1, \ldots, c_n\}$ - S - O - A, (iv) filtering the remaining predicates in S, and (v) optionally sorting the result if O is not satisfied by the index strategy. Figure 9(a) shows the generic pattern for single-index execution plans that implements this logical tree.

As a simple example, consider a single-table logical operator tree representing the query $\Pi_d\big(\sigma_{a+b=2 \wedge c=4}(R)\big)$ and the associated *APR* operator α = (S=$\{c\}$, O=$\{d\}$, A=$\{a, b, d\}$, N=1) (note that there is a sort requirement of column d). Figure 9(b) shows the resulting physical tree for an index I_1=(c, a, b) (this execution plan seeks on I_1 for tuples satisfying c=4, filters tuples satisfying $a + b$=2, fetches the remaining columns and performs a final sort to satisfy the required order). Analogously, Figure 9(c) shows the execution plan for the same logical operator tree and index I_2=(d, c, b, a) (this execution plan scans the covering index I_2 and filters on the fly all the predicates, but does not explicitly sort the output since it is already ordered in the right way).



(a) Original Pattern (b) $I_1(c, a, b)$ (c) $I_2(d, c, b, a)$

Figure 9: Plan Pattern.

Multiple Indexes

A configuration generally contains multiple indexes defined over the table of a given request. In principle, we could use more than one index to obtain a physical sub-plan that implements a request (e.g., by using index intersections). We decided not to consider these advanced index strategies since they would increase the complexity with generally modest gains in quality. However, we note that this is just a matter of convenience, and exactly the same principles can be used to incorporate these less common strategies. We therefore calculate the best execution plan for a given *APR* node as the minimum cost alternative for each of the indexes in the configuration defined over the same table as α.

4.3 Overall Cost Calculation

We now present the overall algorithm to re-optimize a query under an arbitrary configuration. For a given configuration C and MEMO$_{C\text{-}PQO}$ M, function `bestCostForC(`$root(M)$`, `C`)` returns the cost of the best execution plan for the query represented by M under configuration C as follows:

```
bestCostForC(Node n, Configuration C) =
switch(n)
  case AND(APR_i, {}):
       return leafNodeCalculation(APR_i, C) (Section 4.2)
  case AND(op, {g_1, g_2, ..., g_n}):
       return localCost(op) + Σ_i bestCostForC(g_i, C)
  case OR({g_1, g_2, ..., g_n}):
       return min_i bestCostForC(g_i, C)
```

The function above operates depending on the type of input node n. If n is a leaf node (i.e., an *APR* node), we estimate the cost of the best configuration as explained in Section 4.2. Otherwise, if it is an internal AND node, we calculate the best cost by adding to the *localCost* of the *groupExpression* in n the sum of the best costs of each of n's children (calculated recursively). Finally, if n is an OR node, we return the minimum cost among the choices.

Additional Details

In addition to the straightforward implementation of this functional specification, we perform the following optimizations, which we omitted above to simplify the presentation:

Memoization: Note that the same node can be a child of multiple parents. To avoid unnecessary recomputation, we use memoization and therefore cache intermediate results so that we operate over each node at most once.

Branch-and-Bound pruning: `bestCostForC` induces a depth-first search strategy. We then maintain the cost of the best solution found so far for each node in the MEMO$_{C\text{-}PQO}$ and discard alternatives that are guaranteed to be sub-optimal.

Execution plans: In addition to calculating the best cost for each node, we also return the operator tree that is associated with such a cost. Therefore, the same algorithm returns both the best execution plan and its estimated cost.

Note that the first two optimizations above are analogous to those in the Cascades Optimization Framework.

4.4 Extensions

We now discuss some extensions to the techniques described in the paper that take into account important factors such as query updates and materialized views, but we omit a detailed treatment of these issues due to space constraints.

Update Queries

So far we implicitly discussed SELECT-only workloads. In reality, most workloads consist of a mixture of "select" and "update" queries, and *C-PQO* must take into consideration both classes to be useful. The main impact of an update query is that some (or all) indexes defined over the updated table must also be updated as a side effect. To address updates, we modify the configuration-dependent implementation rules that deal with updates, and replace them with (non-leaf) *UAPR* nodes that encode the relevant update information. At re-optimization time, we calculate the cost of updating all relevant indexes in the configuration for each *UAPR* node.

Materialized Views

Although indexes are the most widely used redundant data structure to speed-up query execution, materialized views are also a valuable alternative. Similar to the access-path-selection implementation rules described in 2.2.1, query optimizers rely on view-matching related rules that, once triggered within the context of a *groupExpression*, return zero or more equivalent rewritings of such *groupExpression* using an available view in the system. To incorporate materialized views into a *C-PQO* optimizer, we need to instrument such rules in a similar manner to what we did in the case of indexes. Specifically, every time a view-matching rule is triggered, we analyze the expression and return a *VAPR* node that encodes the logical operator-subtree. These *view APR*s are more complex than regular *APR*s, since we have to encode the view expression itself, which might contain joins, grouping clauses and computed columns. However, the idea is still the same, and at the end of query optimization we return a $\texttt{MEMO}_{\textit{C-PQO}}$ that contains both *APR*s and *VAPR*s. A subtle complication of dealing with materialized views is that the optimizer might trigger index-based implementation rules over the newly used materialized views. In such situation, *APR*s are defined over *VAPR*s rather than base tables, but the underlying principles remain the same.

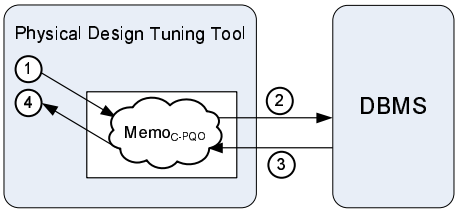


Figure 10: Integrating *C-PQO* into a physical tuning tool.

5. C-PQO-BASED TUNING TOOLS

As explained before, current physical design tuning tools traverse the search space by repeatedly optimizing queries under different candidate configurations. It turns out that integrating *C-PQO* into existing physical design tools is straightforward. Figure 10 shows a generic architecture to achieve this goal. A new *C-PQO* component intercepts each optimization request (C, Q) for query Q and configuration C issued by the tuning tool (step 1 in the figure). If Q has not been seen before, the *C-PQO* component issues a unique *C-PQO* call to the DBMS (step 2 in the figure), obtaining back a $\texttt{MEMO}_{\textit{C-PQO}}$ (step 3 in the figure). Then, it calculates the execution plan and cost for Q using the $\texttt{MEMO}_{\textit{C-PQO}}$ as described in Section 4 and returns the result to the caller (step 4). The $\texttt{MEMO}_{\textit{C-PQO}}$ is cached locally so that future calls with the same query are served without going back to the DBMS. In this way, the tuning tool is not aware that the optimization calls are actually being served by a *C-PQO* component, and proceeds without changes regarding its search strategy.

Deeper Integration with Tuning Tools

Although the architecture described above is enough to dramatically boost the execution times of tuning tools, there might be additional opportunities to leverage *C-PQO* for physical design tuning. Consider for instance re-optimizing a multi-query workload. If the workload queries share some structure, rather that operating over each individual $\texttt{MEMO}_{\textit{C-PQO}}$ structure for the workload queries, we can create a combined $\texttt{MEMO}_{\textit{C-PQO}}$ based on the individual query $\texttt{MEMO}_{\textit{C-PQO}}$ structures by simply adding a new AND root node. Additionally, we can collapse identical sub-graphs into a single representative, obtaining a compressed representation that would be re-optimized much faster. Furthermore, suppose that we want to re-optimize a query under a configuration C_{new} that is slightly different from a previously optimized configuration C_{old}. We can reuse the $\texttt{MEMO}_{\textit{C-PQO}}$ computation for C_{old} by (i) recalculating all *APR* leaf nodes that can be influenced by the differences between C_{old} and C_{new} (e.g., *APR*s over tables that have the same indexes in both C_{new} and C_{old} do not need to be recalculated), and (ii) recalculating bottom-up the cost and plans based on the (small) number of changes in the *APR* leaf nodes.

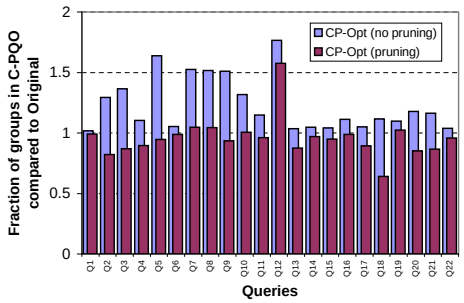
More generally, *C-PQO* eliminates the overhead of optimization calls in the tuning process (after the first *C-PQO* call). Therefore, many architectural choices that were required in previous designs should be revisited and perhaps modified. As a simple example, consider the relaxation-based approach in [3]. The idea is to progressively "shrink" an initial optimal configuration using transformations that aim to diminish the space consumed without significantly hurting the expected performance. For that purpose, such techniques estimate the expected increase in execution time for a large set of candidate transformations over the current configuration (e.g., the effect of replacing two indexes with a merged one). However, with *C-PQO* we can obtain the precise increase in execution time at the same overhead as the original approximation in [3], and therefore avoid doing guess-work during the search. We believe that *C-PQO* can enable a new generation of optimization strategies by exploiting directly the representation of $\texttt{MEMO}_{\textit{C-PQO}}$, instead of just using *C-PQO* as a sophisticated caching mechanism.

6. EXPERIMENTAL EVALUATION

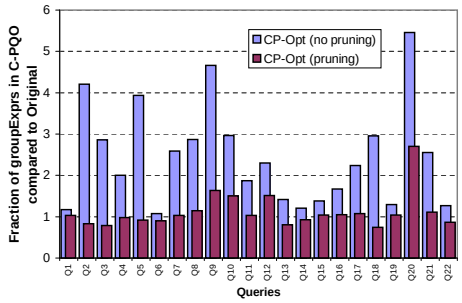
In this section we report an experimental evaluation of the techniques in the paper. We implemented our approach in Microsoft SQL Server 2005 and used a TPC-H database with the 22-query workload generated by the qgen utility[7] (we tried other databases and workloads obtaining similar results). The four questions that we address in the rest of this section are summarized below:

1. What is the overhead of the first (and only) optimization call in *C-PQO*? (Section 6.1)
2. What is the speedup of subsequent optimization calls? (Section 6.2)
3. What is the accuracy of subsequent optimization calls? (Section 6.3)
4. What is the effect of integrating *C-PQO* in existing physical design tools? (Section 6.4)

[7] Available at http://www.tpc.org.



(a) Total *groups*.



(b) Total *groupExpressions*

Figure 11: Space overhead for the initial *C-PQO* optimization.

6.1 Initial Optimization Call: Overhead

To evaluate the overhead of the first optimization call of *C-PQO* we optimized each of the 22 queries with and without *C-PQO* enabled in the DBMS. Figure 11 shows the fraction of *groups* and *groupExpressions* in the *C-PQO* enabled DBMS compared to the original DBMS. The figures distinguish *C-PQO* with and without the cost-based pruning of Section 3.2. We can see that without pruning, the number of groups generated by the *C-PQO* optimizer is between 1x and 1.8x that of the original optimizer and the number of *groupExpressions* is between 1.1x and 5.5x that of the original optimizer. When we use the relaxed pruning rule (i.e., the regular *C-PQO* mode) the fraction of groups in *C-PQO* drops to between 0.64x and 1.6x, and the fraction of *groupExpressions* drops to between 0.75x and 2.7x. The reason for factors smaller than 1x is that *APR* nodes effectively collapse potentially large execution sub-plans into a single node that contains the logical representation of the operator tree. Figure 11 shows that the relaxed pruning rule is effective in cutting down the number of *groups* and *groupExpressions* generated by the optimizer, and also that the space overhead by *C-PQO* is between 1x and 3x of that of the original optimizer.

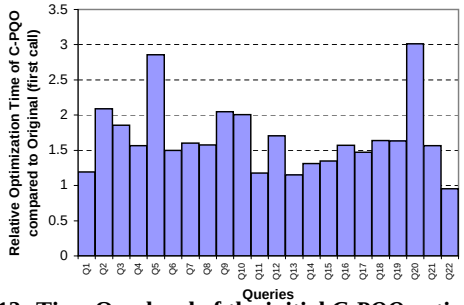


Figure 12: Time Overhead of the initial *C-PQO* optimization.

Figure 12 shows the overheads of *C-PQO* in terms of optimization time. We can see that the first optimization call of *C-PQO* is no more than 3 times that of a regular optimization call (and in many cases around 1.5x). Assuming that subsequent optimization calls for the same query in *C-PQO* are cheap (see next section), the figure shows that after just a couple of optimization calls we can completely amortize the additional overhead of *C-PQO*.

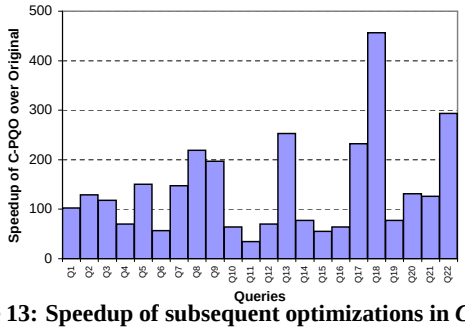


Figure 13: Speedup of subsequent optimizations in *C-PQO*.

6.2 Subsequent Optimization Calls: Speedup

We now measure the average time to produce an execution plan and a cost for subsequent calls with *C-PQO*. For that purpose, we compared the time to optimize each of the 22 TPC-H queries under *C-PQO* against the regular query optimizer. We used the 280 different configurations that were considered by an existing tuning tool for the workload and averaged the results. Figure 13 shows that the average speedup per query when using *C-PQO* varies from 34x to 450x. To put these numbers in perspective, the table below shows the total number of optimizations with *C-PQO* that are possible per regular optimization for a sample of the TPC-H queries, *including the first* C-PQO *call*.

Original	*C-PQO* (Q_{11})	*C-PQO* (Q_{18})	*C-PQO* (Q_{20})
1	0	0	0
2	29	169	0
3	64	625	1
4	99	1081	132
5	134	1537	263

Figure 14 shows the total number of distinct *APR* physical operators for each of the 22 TPC-H queries. Contrasting this figure with Figure 11(b), we see that only a small fraction of the nodes in the $\text{MEMO}_{C\text{-}PQO}$ require a non-trivial amount of processing to obtain execution plans and costs for varying workloads.

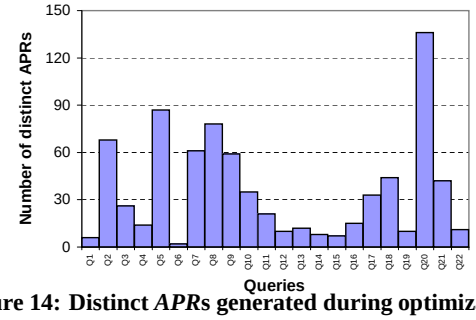
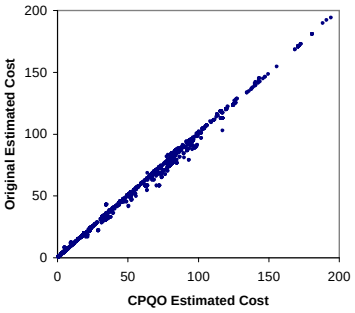


Figure 14: Distinct *APR*s generated during optimization.

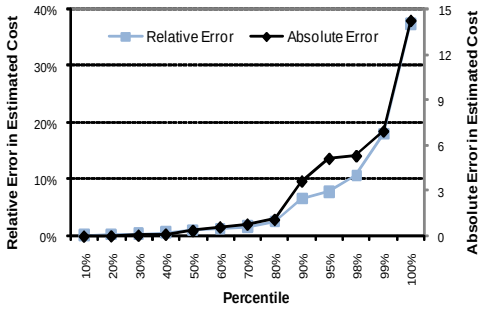
6.3 Subsequent Optimization Calls: Accuracy

We now analyze the accuracy of subsequent optimization calls using *C-PQO*. We first optimized each of the TPC-H queries under the 280 different configurations as in the previous section using both the regular query optimizer and *C-PQO*. We then compare the estimated execution cost of the plans found by the regular optimizer and those found by *C-PQO*. Figure 15(a) shows the results

for the over 6,000 optimization calls, which are heavily clustered around the diagonal. Figure 15(b) shows a different view of the same data, in which we show the maximum relative and absolute errors in cost between the original optimizer and *C-PQO* for different percentiles. We can see that for 80% of the calls, *C-PQO* and the original optimizer differ in less than 2.5%, and for 98% of the calls, the error is below 10%. We analyzed the remaining 2% of the cases and we found that the errors result from either small inaccuracies in our cost model or certain optimizations that the query optimizer performs (e.g., using record-id intersections) and we decided, for simplicity, not to include in our prototype.



(a) Absolute cost estimation error.



(b) Cumulative relative and absolute cost estimation error.

Figure 15: Accuracy of *C-PQO* optimization calls.

6.4 Interaction with Tuning Tools

In this section we evaluate the benefits of *C-PQO* when integrated into an index tuning tool as described in Section 5. We used the 22 query TPC-H workload with different storage and time constraints.

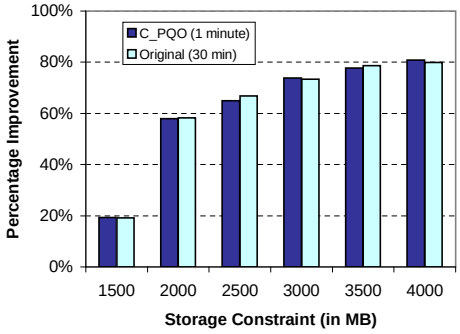


Figure 16: Overall quality of a *C-PQO*-enabled tuning tool.

Figure 16 shows the quality of the final configurations produced by the tuning tool when using regular optimization calls and also with *C-PQO*. We let the original tuning tool run for 30 minutes and the *C-PQO*-enabled tool run for 1 minute (at these points, both tools stabilized and did not improve further the quality of their recommendations). We can see that for all settings, the percentage of improvement[8] of the final configuration for the input workload is almost the same for both systems.

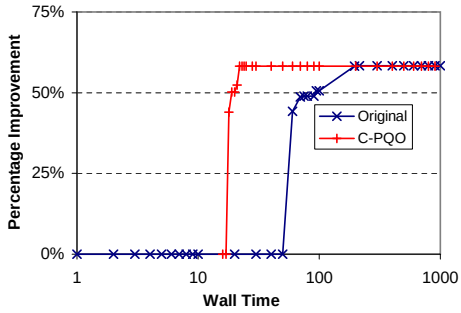


Figure 17: Improvement of configuration quality over time.

Having shown that the quality of recommendations does not suffer by using *C-PQO*, we now report the efficiency aspect of integrating *C-PQO* with tuning tools. For that purpose, we tuned the workload with a storage constraint of 2GB, and measured the quality of the best configuration found by each system over time. Figure 17 shows the results for a tuning session of around 15 minutes (note the logarithmic x-axes). We can see that although both tuners result in the same quality of recommendations, the *C-PQO*-based tuner arrives at such configuration in 22 seconds of tuning, while it takes over 210 seconds to the regular tuner to achieve a comparable result. This difference is further revealed in Figure 18, which shows the number of optimizations per minute for each query in the workload during a 15 minute tuning session. We can see that there is over an order of magnitude improvement in throughput when using a *C-PQO*-enabled tuning tool.

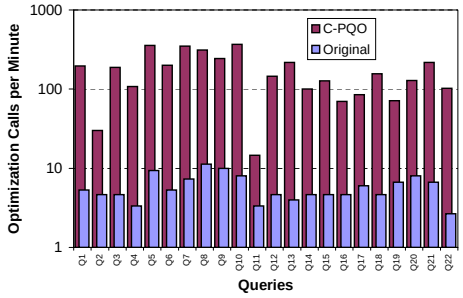


Figure 18: Optimization throughput inside the tuning tool.

Finally, Figure 19 shows a breakdown of the time used in a typical tuning session. For this purpose, we tuned the input workload for 10 minutes and with a space constraint of 2GB. We can see that, as discussed in the introduction, the original tuning tool uses around 92% of the time waiting for results of optimization calls. Consequently, less than 8% of the tuning time is actually spent in the proper search. In contrast, when using the *C-PQO*-enabled tuner, the situation is completely reversed. The tool uses less than 4% of the time doing the first optimization call for each query, and another 5% of the time doing all the subsequent calls, leaving 90% of the time for the search strategy proper. Note that *C-PQO* analyzed almost 10,000 different configuration and performed over 31,000 optimization calls in the 10 allowed minutes, while the original tuning tool managed to process just 355 configurations and below 1,700 optimization calls in the same amount of time.

[8] Percentage of improvement is traditionally defined as $1 - \frac{\text{cost\_recommended}}{\text{cost\_original}}$.

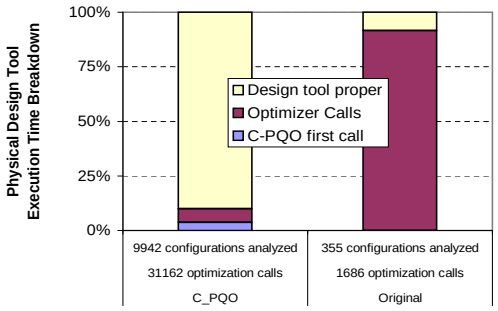


Figure 19: Execution time breakdown during a tuning session.

7. RELATED WORK

There has been considerable research on automating the physical design in DBMSs (e.g., [1, 6, 8, 15]). All this work relies on the *what-if* interface to evaluate candidate configurations, and therefore could be enhanced by the *C-PQO* ideas in this work to boost their performance. References [3, 4] introduce, in a slightly different context, the underlying functionality that we use in the physical *APR* nodes in Section 3. In contrast to our work, reference [4] exploits such technology in the context of local optimizations, by transforming a final execution plan into another that uses different physical structures. Instead, we are able to address the full optimization problem by generating and exploiting $\texttt{MEMO}_{C\text{-}PQO}$ structures rather than just any final execution plan.

Parametric Query Optimization [11] (or *PQO*) studies mechanisms to generate optimal execution plans for varying parameters. Traditionally, *PQO* considered parameters that are either system-related (e.g., available memory) or query-related (e.g., selectivity of parametric predicates). In this work, we address the problem of parametric query optimization when the parameter is the actual physical configuration on top of which the query needs to be optimized.

Very recently, reference [12] introduces INUM, a technique that shares with ours the goal of reducing the bottleneck of optimization calls inside physical design tools. The idea is to extend the local-transformation approach in [3, 4]. During a preprocessing step, several optimization calls are issued for a given query until the resulting plans are enough to infer optimal plans for arbitrary configurations. In contrast to our work, INUM is not fully integrated with the query optimizer. For that reason, it is not clear how to extend the approach for more complex execution plans (such as correlated sub-queries) or other physical access path structures (such as materialized views). Specifically, reference [12] reports an average relative error of 7% for `TPC-H` query 15 with no estimation error above 10%. In contrast, our techniques result in a average relative error of 1.04%, with no estimation error above 1.8% for the same query. INUM also requires hints and assumptions to reduce the number of regular optimization calls per query in the pre-computation phase (which could be exponential in the number of tables in the worst case). As an example, for `TPC-H` query 15, INUM requires 1,358 regular optimization calls before it can start optimizing arbitrary configurations. Our *C-PQO* optimizer required a *single C-PQO* execution call (worth 1.4 regular optimization calls) to arrive at the same state (a difference of roughly three orders of magnitude). For that reason, an experimental evaluation of INUM [12] results in 1.3x to 4x improvement in the performance of tuning tools, where our techniques result in over an order of magnitude improvement over INUM.

8. CONCLUSION

In this work we address the current bottleneck of current physical design tools: large amounts of time waiting for the optimizer to produce execution plans. Inspired by the ideas on parametric query optimization, we designed an infrastructure that produces, with little overhead on top of a regular optimization call, a compact representation of all execution plans that are possible for varying input configurations. We are then able to instantiate this representation with respect to arbitrary configurations and simulate the optimization of queries orders of magnitude more efficiently than traditional approaches at virtually no degradation in quality. Our techniques are straightforward to incorporate into existing physical design tools, and our initial experiments show drastic improvements in performance. Furthermore, we believe that an even more interesting challenge lies ahead. With the main bottleneck of current tools gone, we might be able to focus on more complex optimization strategies by exploiting directly the representation of *C-PQO*-enabled tools, instead of using *C-PQO* as a sophisticated caching mechanism.

9. 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] K. Billings. A TPC-D Model for Database Query Optimization in Cascades. Ms. thesis, Portland State University, 1996.

[3] N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2005.

[4] N. Bruno and S. Chaudhuri. To tune or not to tune? A Lightweight Physical Design Alerter. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2006.

[5] P. Celis. The Query Optimizer in Tandem's new ServerWare SQL Product. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1996.

[6] S. Chaudhuri and V. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1997.

[7] S. Chaudhuri and V. Narasayya. Autoadmin 'What-if' index analysis utility. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1998.

[8] B. Dageville et al. Automatic SQL Tuning in Oracle 10g. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2004.

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

[10] G. Graefe. The Microsoft Relational Engine. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 1996.

[11] Y. Ioannidis, R. Ng, K. Shim, and T. Sellis. Parametric query optimization. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1992.

[12] S. Papadomanolakis, D. Dash, and A. Ailamaki. Efficient use of the query optimizer for automated physical design. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2007.

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

[14] G. Valentin, M. Zuliani, D. Zilio, G. Lohman, and A. Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2000.

[15] D. Zilio et al. DB2 design advisor: Integrated automatic physical database design. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2004.