Constrained Physical Design Tuning

Nicolas Bruno
Microsoft Research
nicolasb@microsoft.com

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

ABSTRACT

Existing solutions to the automated physical design problem in database systems attempt to minimize execution costs of input workloads for a given a storage constraint. In this paper, we argue that this model is not flexible enough to address several real-world situations. To overcome this limitation, we introduce a constraint language that is simple yet powerful enough to express many important scenarios. We build upon an existing transformation-based framework to effectively incorporate constraints in the search space. We then show experimentally that we are able to handle a rich class of constraints and that our proposed technique scales gracefully.

1. INTRODUCTION

In the last decade, automated physical design tuning became a relevant area of research. As a consequence, several academic and industrial institutions addressed the problem of recommending a set of physical structures that would increase the performance of the underlying database system. The central physical design problem statement has been traditionally stated as follows:

> Given a workload W and a storage budget B, find the set of physical structures (or configuration), that fits in B and results in the lowest execution cost for W.

This problem is very succinctly described and understood. Consequently, it has recently received considerable attention resulting in novel research results and industrial-strength prototypes in all major DBMS. Despite this substantial progress, however, the problem statement and existing solutions cannot address important real-life scenarios. Consider, as a simple example, the following query:

```
SELECT a, b, c, d, e
FROM R
WHERE a=10
```

and suppose that a single tuple from R satisfies a=10. If the space budget allows it, a covering index I_C over (a, b, c, d, e) would be the best alternative for q, requiring a single I/O to locate the qualifying row and all the required columns. Now consider a narrow single-column index I_N over (a). In this case, we would require two I/Os to answer the query (one to locate the record-id of the qualifying tuple from the secondary index I_N, and another to fetch the relevant tuple from the primary index). In absolute terms, I_C results in a better execution plan compared to that of I_N. However, the execution plan that uses I_N is only slightly less efficient to the one that uses I_C (specially compared to the naïve alternative that performs a sequential scan over table `R`), and at the same time it incurs no overhead for updates on columns b, c, d, or e. If such updates are possible, it might make sense to "penalize" wide indexes such as I_C from appearing in the final configuration. However, current techniques cannot explicitly model this requirement without resorting to artificial changes. For instance, we could simulate this behavior by introducing artificial `UPDATE` statements in the workload. This mechanism, however, is not general enough to capture other important scenarios that we discuss below.

Note, however, that the previous example does not lead itself to a new "golden rule" of tuning. There are situations for which the covering index is the superior alternative (e.g., there could be no updates on table R by design). In fact, an application that repeatedly and almost exclusively executes the above query can result in a 50% improvement when using the covering index I_C rather than the narrow alternative I_N. A more subtle scenario that results in deadlocks when narrow indexes are used is described in [13].

In general, there are other situations in which the traditional problem statement for physical design tuning is not sufficient. In many cases we have additional information that we would like to incorporate into the tuning process. Unfortunately, it is often not possible to do so by only manipulating either the input workload or the storage constraint. For instance, we might want to tune a given workload for maximum performance under a storage constraint, but ensuring that no query degrades by more than 10% with respect to the original configuration. Or we might want to enforce that the clustered index on a table T cannot be defined over certain columns of T that would introduce hot-spots (without specifying which of the remaining columns should be chosen). As yet another example, in order to decrease contention during query processing, we might want to avoid any single column from a table from appearing in more than, say, three indexes (the more indexes a column appears in, the more contention due to exclusive locks during updates).

The scenarios above show that state-of-the-art techniques for physical design tuning are not flexible enough. Specifically, a single storage constraint does not model many important situations in current DBMS installations. What we need is a generalized version of the physical design problem statement that accepts complex constraints in the solution space, and exhibit the following properties:

Expressiveness. It should be easy to specify constraints with sufficient expressive power.

Effectiveness. Constraints should be able to effectively restrict the search process (e.g., a naïve approach that tests a-posteriori whether constraints are satisfied would not be viable).

Specialization. In case there is a single storage constraint, the resulting configurations should be close to those obtained by current physical design tools in terms of quality.

In this paper we introduce a framework that addresses these challenges. For simplicity, we restrict our techniques to handle primary and secondary indexes as the physical structures that define the search space (extensions to materialized views and other physical structures are part of future work). Specifically, the main contributions of the paper are as follows. First, in Section 2 we present a simple language to specify constraints that is powerful enough to handle many desired scenarios including our motivating examples. Second, we review a previously studied transformation-based search framework (Section 3) and adapt it to incorporate constraints into the search space (Sections 4 and 5). Finally, in Section 6 we report an extensive experimental evaluation of our techniques.

2. CONSTRAINT LANGUAGE

Our design approach has been to provide a simple constraint language that covers a significant fraction of interesting scenarios (including all the motivating examples in the previous section). We also provide a lower-level interface to specify more elaborate constraints as well as more efficient ways to evaluate constraints. In the rest of this section we introduce our language by using examples.

2.1 Data Types, Functions, Constants

Our constraint language understands simple types such as numbers and strings, and also domain-specific ones. Specifically, we natively handle data types that are relevant for physical design, such as database tables, columns, indexes and queries. We also support sets of elements, which are unordered homogeneous collections (e.g., workloads are sets of queries, and configurations are sets of indexes). These sets can be accessed using either positional or associative array notation (e.g., `W[2]` returns the second query in `W`, and `W["QLong"]` returns the query in `W` whose id is `QLong`).

Our language supports a rich set of functions over these data types. As an example, we can obtain the columns of table T using `cols`(T), the expected size of index I using `size`(I), and the expected cost of query q under configuration C using `cost`(q, C). In the rest of this section, we introduce additional functions as needed.

Finally, there are useful constants that can be freely referenced in the language. We use `W` to denote the input workload, and the following constants to specify certain commonly used configurations:

- `C`: denotes the desired configuration, on top of which constraints are typically specified.
- `COrig`: This is the configuration that is currently deployed in the database system.
- `CBase`: The base configuration only contains those indexes originating from integrity constraints. Therefore, it is the worst possible configuration for `SELECT` queries in the workload, and the one with lowest `UPDATE` overhead.
- `CSelectBest`: This configuration is the best possible one for `SELECT` queries in the workload. Specifically, `CSelectBest` contains the indexes resulting from access-path requests generated while optimizing the input workload (see [4] for more details). Intuitively, indexes in this configuration are the most specific ones that can be used in some execution plan for a query in the workload. For instance, the two indexes in `CSelectBest` for query:

```
SELECT a,b,c FROM R
WHERE a<10
ORDER BY b
```

would be `(a,b,c)`, from the access-path request that attempts to seek column `a` for all tuples that satisfy `a<10` followed by a sort by `b`, and `(b,a,c)`, from the access-path-request that scans `R` in `b`-order and filters `a<10` on the fly.

2.2 Language Features

We next illustrate the different features of our constraint language by using examples.

Simple Constraints. We can specify the storage constraint used in virtually all physical design tuning tools as follows:

```
ASSERT size(C) ≤ 200M
```

where `size(C)` returns the combined size of the final configuration. Constraints start with the keyword `ASSERT` and follow the pattern *function-comparison-constant*. As another example, the constraint below ensures that the cost of the second query in the workload under the final configuration is not worse than twice its cost under the currently deployed configuration:

```
ASSERT cost(W[2], C) ≤ 2 * cost(W[2], COrig)
```

Note that, for a fixed query `Q`, the value `cost(Q, COrig)` is constant, so the `ASSERT` clause above is valid.

Generators. Generators allow us to apply a template constraint over each element in a given collection. For instance, the following constraint generalizes the previous one by ensuring that the cost of *each query* under the final configuration is not worse than twice its cost under the currently deployed configuration:

```
FOR Q IN W
ASSERT cost(Q, C) ≤ 2 * cost(Q, COrig)
```

In turn, the following constraint ensures that every index in the final configuration has at most four columns:

```
FOR I in C
ASSERT numCols(I) ≤ 4
```

Filters. Filters allow us to choose a subset of a generator. For instance, if we only want to enforce the above constraint for indexes that have leading column `col3`, we can extend the original constraint as follows:

```
FOR I in C
WHERE I LIKE "col3,*"
ASSERT numCols(I) ≤ 4
```

where `LIKE` performs “pattern matching” on the index columns.

Aggregation. Generators allow us to duplicate a constraint multiple times by replacing a free variable in the `ASSERT` clause with a range of values given by the generator. In many situations, we want a constraint acting on an *aggregate* value calculated over the elements in a generator. As a simple example, we can rewrite the original storage constraint used in physical design tools using generators and aggregates as follows:

```
FOR I in C
ASSERT sum(size(I)) ≤ 200M
```

As a more complex example, the following constraint ensures that the combined size of all indexes defined over table `T` is not larger than four times the size of the table itself:

```
FOR I in C
WHERE table(I) = TABLES["T"]
ASSERT sum(size(I)) ≤ 4 * size(TABLES["T"])
```

where `TABLES` is the collection of all the tables in the database, and function `size` on a table returns the size of its primary index.

Nested Constraints. Constraints can have free variables that are bound by outer generators, effectively resulting in nested constraints. The net effect of the outer generator is to duplicate the inner constraint by binding each generated value to the free variable in the inner constraint. As an example, the following constraint generalizes the previous one to iterate over all tables:

```
FOR T in TABLES
    FOR I in C
    WHERE table(I) = T
    ASSERT sum(size(I)) ≤ 4 * size(T)
```

Soft Constraints. The implicit meaning of the language defined so far is that a configuration has to satisfy all constraints to be valid. Among those valid configurations, we keep the one with the minimum expected cost for the input workload. There are situations, however, in which we would prefer a relaxed notion of constraint. For instance, consider a constraint that enforces that every non-UPDATE query results in at least 10% improvement over the currently deployed configuration. In general, there might be no configuration that satisfies this constraint, specially in conjunction with a storage constraint. In these situations, a better alternative is to specify a *soft constraint*, which states that the final configuration should get as close as possible to a 10% improvement (a configuration with, say, 8% improvement would still be considered valid). We specify such *soft* constraints by adding a SOFT keyword in the ASSERT clause. The resulting constraint thus becomes:

```
FOR Q in W
WHERE type(Q) = SELECT
SOFT ASSERT cost(Q, C) ≤ cost(Q, COrig) / 1.1
```

Note that the traditional optimization function (i.e., minimizing the cost of the input workload), can be then specified as follows:

```
FOR Q IN W
SOFT ASSERT sum(cost(Q, C)) = 0
```

If no soft constraints are present in a problem specification, we implicitly add the above soft constraint and therefore optimize for the expected cost of the input workload. In general, however, soft constraints allow significantly more flexibility while specifying a physical design problem. For instance, suppose that we are interested in the smallest configuration for which the cost of the workload is at most 20% worse than that for the currently deployed configuration (as shown in [5], this problem statement is useful to eliminate redundant indexes without significantly degrading the expected cost of the workload). We can specify this scenario using soft constraints as follows:

```
FOR Q IN W
ASSERT sum(cost(Q, C)) ≤ 1.2 * sum(cost(Q, COrig))

SOFT ASSERT size(C) = 0
```

2.3 Generic Constraint Language

In general, a constraint is defined by the grammar below, where bold tokens are non-terminals (and self-explanatory), non-bold tokens are literals, tokens between brackets are optional and "|" represents choice:

```
constraint:=[SOFT] ASSERT [agg] function (≤|=|≥) constant
           | FOR var IN generator
             [WHERE predicate]
             constraint
```

We next show that although our language is simple, it is able to specify all the motivating examples in the previous section. In Section 5 we discuss how we can handle constraints that lie outside the expressive power of the language by using a specialized interface.

2.4 Motivating Examples Revisited

We now specify constraints for the motivating examples in Section 1. The following constraint ensures that no column appears in more than three indexes to decrease the chance of contention:

```
FOR T in TABLES
    FOR col in cols(T)
        FOR I in C WHERE I LIKE "*,col,*"
        ASSERT count(I) ≤ 3
```

The next constraint enforces that the clustered index on table T must have either a, b, or c as its leading column:

```
FOR I in C
WHERE clustered(I)
ASSERT I LIKE "(a,*)|(b,*)|(c,*)"
```

Note that the ASSERT clause is a predicate and does not follow the pattern *"function-comparison-constant"* introduced earlier. We thus implicitly replace a predicate ρ with $\delta(\rho)$=1, where δ is the characteristic function ($\delta(true)$=1 and $\delta(false)$=0).

The constraint below enforces that no SELECT query degrades by more than 10% compared to the currently deployed configuration:

```
FOR Q in W
WHERE type(Q) = SELECT
ASSERT cost(Q, C) ≤ 1.1 * cost(Q, COrig)
```

The last constraint enforces that no index can be replaced by its narrow version without at least doubling the cost of some query:

```
FOR I in C
    FOR Q in W
    ASSERT cost(Q, C - I + narrow(I))/cost(Q, C) ≤ 2
```

where narrow(I) results in a single-column index with I's leading column (e.g., narrow((a,b,c)) = (a)).

3. SEARCH FRAMEWORK

In this section we review the general architecture of our search framework, which we adapted from [4, 5]. For presentation purposes, we address in this section the traditional physical design problem (i.e., we assume that there is a single storage constraint and we optimize for expected cost). Then, in Section 4 we explain how to incorporate multiple constraints into the search framework.

3.1 General Architecture

Figure 1 shows a high-level architectural overview of our search framework. An important component of the framework is the global cache of explored configurations (shown at the bottom of Figure 1). This global cache is structured in three tiers, which respectively contain (i) the best configuration found so far, (ii) the set of non-dominated configurations in case there are multiple soft constraints, and (iii) the remaining suboptimal configurations.

We begin the search from an initial configuration (step 1 in the figure), which becomes the current configuration. After that, we progressively explore the search space until a stopping condition is satisfied (typically a time bound). Each exploration iteration consists of the following steps. First, we evaluate the current configuration and store it in the global cache (step 2 in the figure). Then, we perform a pruning check on the current configuration. If we decide to prune the current configuration, we keep retrieving from the global cache previously explored configurations until we obtain one that is not pruned (this step effectively implements a backtracking mechanism). At this point, we use transformation rules to generate new candidate configurations from the current one (step 3 in the figure). We rank candidate configurations based on their expected promise and pick the best candidate configuration that is not

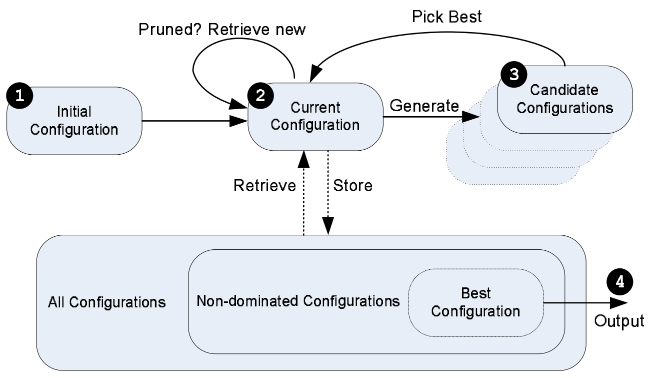


Figure 1: Architecture of the Search Framework.

already in the global cache, which becomes the current configuration. This cycle repeats until the stopping criterium is met, and we output the best configuration(s) found so far (step 4 in the figure).

Looking at the search strategy at a high level, we start with some configuration (either the initial one or a previously explored one) and keep transforming it into more and more promising candidates until a pruning condition is satisfied. At this point we pick a new configuration and begin a new iteration. In the rest of this section we discuss additional details on the search framework.

3.1.1 Configuration Evaluation

Each step in the search process requires evaluating a previously unexplored configuration, which in itself consists of two tasks.

First, we need to determine whether the storage constraint is satisfied, and if not, how close is the current configuration to a viable state. With a storage constraint of B, we simply estimate the size of the current configuration, `size(C)`. If $\texttt{size}(C) \leq B$, the storage constraint is satisfied. Otherwise, the value $\texttt{size}(C) - B$ quantifies how close we are to a valid configuration.

Second, we need to evaluate the optimizing function, that is, the expected cost of the workload under the current configuration. In order to do so, we need to optimize the queries in the workload in a *what-if* mode [9], which returns the expected cost of each query without materializing the configuration. This step is usually the bottleneck of the whole process, since optimizer calls are typically expensive. There are several ways to reduce this overhead. One approach is to use information about previous optimizations to infer, in some cases, the cost of a query under a given configuration without issuing an optimization call (examples of such techniques use atomic configurations [8] or a top-down relaxation approach [4]). A recent approach introduced in [7] results in accurate approximations of the cost of a query at very low overhead (typically orders of magnitude faster than a regular optimization call).

3.1.2 Transformations

After evaluating the current configuration, we apply transformation rules to generate a set of new, unexplored configurations in the search space. For that purpose, we use the *merge-reduce* family of transformations introduced in [5]. Specifically, the transformations that are considered for the current configuration are as follows:

Merging rules: Merging has been proposed as a way to eliminate redundancy in a configuration without losing significant efficiency during query processing [5, 10]. The (ordered) merging of two indexes I_1 and I_2 defined over the same table is the best index that can answer all requests that either I_1 and I_2 do, and can efficiently seek in all cases that I_1 can. Specifically, the merging of I_1 and I_2 is a new index that contains all the columns of I_1 followed by those in I_2 that are not in I_1 (if one of the original indexes is a clustered index, the merged index will also be clustered). For example, merging (a, b, c) and (a, d, c) returns (a, b, c, d). Index merging is an asymmetric operation (i.e., in general merge(I_1,I_2) $\neq$ merge(I_2,I_1)). Let C be a configuration and (I_1, I_2) a pair of indexes defined over the same table such that $\{I_1, I_2\} \subseteq C$. Then, the merging rule induced by I_1 and I_2 (in that order) on C, denoted $\texttt{merge}(C, I_1, I_2)$ results in a new configuration $C' = C - \{I_1, I_2\} \cup \{\texttt{merge}(I_1, I_2)\}$.

Reduction rules: Reduction rules replace an index with another that shares a prefix of the original index columns. For instance, the reductions of index (a, b, c, d) are (a), (a, b), and (a, b, c). A reduction rule denoted as $\texttt{reduce}(C, I, k)$, where k is the number of columns to keep in I, replaces I in C with its reduced version $\texttt{reduce}(I, k)$.

Deletion rules: Deletion rules, denoted $\texttt{remove}(C, I)$, remove index I from configuration C. If the removed index is a clustered index, it is replaced by the corresponding table heap.

The number of transformations for a given configuration C is $\mathcal{O}(n \cdot (n + m))$ where n is the number of indexes in C and m is the maximum number of columns in an index in C. Of course, in real situations this number is likely to be much smaller, because indexes are spread throughout several tables (and therefore merging is valid for only a subset of the possible cases), and also because not all reductions need to be considered. To clarify the latter point, consider index I on (a, b, c, d, e) and the single-query workload:

```
SELECT a,b,c,d,e
FROM R
WHERE a=10
```

In this situation, the only useful reduction for the index is I' on (a), since any other prefix of I is going to be both larger than I' and less efficient for answering the query.

3.1.3 Candidate Configuration Ranking

After generating all valid transformations for the current configuration, we need to rank them in decreasing order of "promise", so that more promising configurations are chosen and explored first. For that purpose, we estimate both the expected cost of the workload and the expected size (i.e., the storage constraint) of each resulting configuration. While estimating sizes can be done efficiently, estimating workload costs is much more challenging. The reason is that often there are several candidate configurations to rank (typically one per transformation), and the cost of optimizing queries (even using the optimizations described earlier) is too costly. To address this issue, we use the local transformation approach of [4, 6] and obtain upper bounds on the cost of queries for each candidate transformation. Consider a query q and a configuration C' obtained from C. The idea is to analyze the execution plan of q under C and replace each sub-plan that uses an index in $C - C'$ with an equivalent plan that uses indexes in C' only.

As an example, consider the execution plan P at the left of Figure 2 under configuration C. Index I on (a, b, c) is used to seek tuples that satisfy $a < 10$ and also to retrieve additional columns b and c, which would eventually be needed at higher levels in the execution plan. Suppose that we are evaluating a configuration C' obtained by reducing I to I' on (a, b). We can then replace the small portion of the execution plan that uses I with a small compensating plan that uses I' (specifically, the replacement sub-plan uses I' and additional rid-lookups to obtain the remaining required c column). The resulting plan P' is therefore valid and at most as efficient as the best plan found by the optimizer under C'.

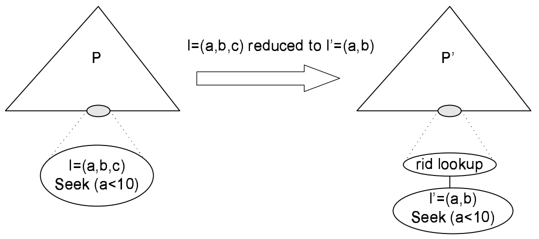


Figure 2: Local transformations to obtain upper-bound costs.

Once we obtain estimates for both the optimizing function and the deviation from the storage constraint for each of the alternative configurations, we need to put together these values to rank the different candidate transformations. In the context of a single storage constraint, reference [4] uses the value $\Delta_{cost}/\Delta_{size}$ to rank transformations, where Δ_{cost} is the difference in cost between the pre- and post-transformation configuration, and Δ_{size} is the respective difference in required storage (reference [4] adapts this metric from the greedy solution for the fractional knapsack problem).

3.1.4 Configuration Pruning

As explained in Figure 1, we keep transforming the current configuration until it is pruned, at which point we backtrack to a previous configuration and start another iteration. Consider a single storage constraint B, and assume a `SELECT`-only workload. Suppose that the current configuration C exceeds B, but after transforming C into C' we observe that C' is within the storage bound B. In this case, no matter how we further transform C', we would never obtain a valid configuration that is more efficient than C'. The reason is that all the transformations (i.e., merges, reductions and deletions) result in configurations that are less efficient for the input workload. Therefore, C' dominates the remaining unexplored configurations, and we can stop the current iteration by pruning C'. When there are multiple rich constraints, the pruning condition becomes more complex, and is discussed in Section 4.

3.1.5 Choosing the Initial Configuration

Although any configuration can be chosen to be the starting point in our search, the initial configuration effectively restricts the search space. Specifically, our search framework is able to eventually consider any configuration that is a subset of the closure of the initial configuration under the set of transformations. Formally, let C be a configuration and let C_i ($i \geq 0$) be defined as follows:

- $C_0 = C$
- $C_{i+1} = C_i \cup \{\texttt{merge}(I_1, I_2)$ for each $I_1, I_2 \in C_i\}$

  $\cup \{\texttt{reduce}(I, K)$ for each $I \in C_i, K < |I|\}$

We define *closure*$(C) = C_k$, where k is the smallest integer that satisfies $C_k = C_{k+1}$. The closure of a configuration C is then the set of all indexes that are either in C or can be derived from C through a series of merging and reduction operations. For that reason, if no subset of the *closure* of the initial configuration satisfies all the constraints, the problem is unfeasible. Unless a specific initial configuration is given, the default starting point is `CSelectBest`, which contains the most specific indexes that can be used anywhere by the query optimizer for the input workload, and thus should be appropriate to handle all but non-standard constraints[1].

[1] An example of such constraint would be the requirement that some index not useful for any workload query be present in the final configuration.

Constraint	Objective
$F(C) \leq K$	$\max(0, F(C) - K)$
$F(C) = K$	$\|F(C) - K\|$
$F(C) \geq K$	$\max(0, K - F(C))$

Table 1: Converting constraints into c-objectives.

4. CONSTRAINED PHYSICAL TUNING

In the previous sections we introduced a constraint language and reviewed a general transformation-based strategy to traverse the space of valid configurations. In this section we explain how to integrate constraints into the search framework. In short, we convert constraints into objective functions and avoid directly comparing multiple objectives together by using Pareto optimality concepts.

4.1 From Constraints to C-Objectives

Constrained physical design is a multi-constraint multi-objective optimization problem (recall that soft-constraints naturally lead to more than a single optimization function). A common approach to handle such problems is to transform constraints into new objective functions (we call these *c-objectives* for short) and then solve a multi-objective optimization problem. Note that the *function-comparison-constant* pattern for `ASSERT` clauses enables us to assign a non-negative real value to each constraint with respect to a given configuration. It is in fact straightforward to create a *c-objective* that returns zero if the constraint is satisfied and positive values when it is not (and moreover, the higher the value the more distant the configuration to one that satisfies the constraint). Table 1 shows this mapping, where $F(C)$ and K denote, respectively, the function (of the current configuration) and the constant in the `ASSERT` clause. For constraints that iterate over multiple `ASSERT` clauses, we sum the values of the individual `ASSERT` clauses[2].

By proceeding as before, each configuration is now associated with $n_s + n_h$ values for n_s soft constraints and n_h hard (i.e., non-soft) constraints. Minimizing the n_h *c-objectives* down to zero results in a valid configuration that satisfies all hard constraints, while minimizing the n_s *c-objectives* results in the most attractive configuration (which might not satisfy some hard constraint). Usually, the n_h *c-objectives* are in opposition to the n_s *c-objectives* and also to each other, and therefore our search problem is not straightforward.

A common approach to address multi-objective problems is to combine all *c-objectives* together into a new single objective function. In this way, the resulting optimization function might become:

$$\textit{singleObjective}(C) = \sum_{i=1}^{n} w_i \cdot \alpha_i(C)$$

where $\alpha_i(C)$ denotes the $i\text{-}th$ *c-objective* and w_i are user-defined weights. While this approach is universally applicable, it suffers from a series of problems. The choice of weights is typically a subtle matter, and the quality of the solution obtained (or even the likelihood of finding a solution whatsoever) is often sensitive to the values chosen. A deeper problem arises from the fact that usually *c-objectives* are *non-commensurate*, and therefore trade-offs between them range from arbitrary to meaningless.

For this reason, we do not reduce the original problem to a single-optimization alternative. Instead, we rely on the concept of *Pareto optimality*, which in general does not search for a single solution but instead the set of solutions with the "best possible trade-offs". We next explain this notion and how we use it to reason with configurations within our search strategy.

[2] Instead, we could consider each `ASSERT` within a generator individually. Our experiments show that this alternative results in additional complexities without improving the effectiveness of the search strategy.

4.2 Pareto Optimality for Configurations

The concept of Pareto optimality can be explained by using the notion of *dominance*. We say that vector $x = (x_1, \ldots, x_n)$ dominates vector $y = (y_1, \ldots, y_n)$ if the value of each dimension of x is at least as good as that of y, and strictly better for at least one dimension. Therefore, assuming that smaller values are better:

$$x \text{ dominates } y \iff \forall i : x_i \leq y_i \wedge \exists j : x_j < y_j$$

An element $x \in X$ is said to be *Pareto Optimal* in x if it is not dominated by any other vector $y \in X$. (The *Pareto Optimal* elements of a set are also said to form the *skyline* [3] of the set).

In our scenario, each configuration is associated with a vector of size $n_s + n_h$ for n_s soft constraints and n_h hard constraints, and thus we can talk about dominance of configurations. If there is a single soft constraint and all hard constraints are satisfiable, there must be a unique *Pareto optimal* solution. In fact, for a configuration to be valid, each of the n_h *c-objectives* must be zero, and thus the valid configuration with the smallest *c-objective* value for the soft-constraint dominates every other configuration. (Even for a single soft constraint, however, there can be multiple *Pareto optimal* configurations among the explored ones during the search.)

4.3 Configuration Ranking

Using the notion of dominance, we can obtain a total ranking of configurations in two steps. First, we assign to each configuration a "rank" equal to the number of solutions which dominate it[3]. As an example, Figure 3(b) shows the rankings of all the two-dimensional vectors shown in Figure 3(a). This ranking induces a partial order, where each vector with ranking i belongs to an equivalence class L_i, and every element in L_i goes before every element in L_j for $i < j$ (see Figure 3(c) for a graphical illustration of such equivalence classes). The final ranking is then obtained by probabilistically choosing a total order consistent with the partial order given by equivalence classes L_i (see Figure 3(d) for an example)[4]. The pseudo-code below implements this idea.

```
RankConfigurations ($C$=$c_1, c_2, \ldots, c_n$:configurations)
Output R: a ranked list of configurations
01  for each $c_i \in C$
02     rank($c_i$) = $|\{c_j \in C : c_j$ dominates $c_i\}|$
03  R = []
04  for each $i \in \{1..n\}$
05     $L_i$ = $\{c \in C :$ rank$(c) = i\}$
06     $LP_i$ = random permutation of $L_i$
06     append $LP_i$ to R
07  return R
```

Our search strategy relies on the ability to rank configurations at two specific points. First, in Step 3 in Figure 1 we need to pick the transformation that would result in the most promising configuration. Second, after pruning the current configuration in Step 2 in Figure 1, we need to pick, among the partially explored configurations, the most promising one to backtrack to. Whenever we require to rank a set of configurations, we proceed as follows. First, we evaluate (or approximate) the values of all the *c-objectives* as explained in Sections 3.1.1 and 3.1.3. Then, using the pseudo-code above we obtain a partial order and probabilistically choose a ranking consistent with this partial order.

4.4 Search Space Pruning

In Section 3 we described a mechanism to prune a given configuration, which relied on identifying when future transformations were not able to improve the current configuration. We now extend this technique to work with multiple, rich constraints. We introduce a function $\mathcal{D}$ that takes a configuration and the left-hand-side function F of an `ASSERT` clause, and returns one of four possible values (which intuitively represent the "direction" on which F moves after applying transformations to the input configuration). Thus,

$$\mathcal{D} :: \textit{configuration} \times \textit{function} \rightarrow \{\uparrow, \downarrow, \leftrightarrow, ?\}$$

Recall that, for any given configuration instance C_0, we evaluate the value $F(C_0)$ by binding the free variable `C` in F (i.e., the desired configuration on top of which constraints are defined) with C_0. The semantics of $\mathcal{D}(C, F)$ are as follows:

$$\mathcal{D}(C,F) = \begin{cases} \uparrow & \text{if } F(C') \geq F(C) \text{ for all } C' \in closure(C) \\ \downarrow & \text{if } F(C') \leq F(C) \text{ for all } C' \in closure(C) \\ \leftrightarrow & \text{if } F(C') = F(C) \text{ for all } C' \in closure(C) \\ ? & \text{otherwise} \end{cases}$$

As an example, consider the following constraint:

```
ASSERT size(C) - size(COrig) ≤ 200M
```

In this situation, $\mathcal{D}(C, F) = \downarrow$ for any C because any sequence of transformations starting with C will result in a smaller configuration, and therefore the value of function F always decreases. Although the definition of $\mathcal{D}$ is precise, in practice it might be unfeasible to evaluate $\mathcal{D}$ for arbitrary values of F. We adopt a best-effort policy, and try to infer $\mathcal{D}$ values. If we cannot prove that $\mathcal{D}(C, F) \in \{\uparrow, \downarrow, \leftrightarrow\}$ we return the unknown value "?". Operationally, we evaluate $\mathcal{D}$ in an inductive manner. We first assign $\mathcal{D}$ values for the base numeric function calls, such as, for instance:

$\mathcal{D}(C, size(\texttt{C})) = \downarrow$
$\mathcal{D}(C, size(Tables[\text{"}R\text{"}])) = \leftrightarrow$
$\mathcal{D}(C, cost(Q, \texttt{C}))$ = if type(Q) is `SELECT` then $\uparrow$ else ?

and propagate results through operators using standard rules, such as for instance (i) $\uparrow$ + $\uparrow$=$\uparrow$, (ii) $\uparrow$+$\downarrow$=?, and (iii) $\max(\uparrow, \leftrightarrow)$=$\uparrow$. (We handle constraints with generators and aggregation similarly, but omit details for simplicity.)

Constraint template	Instance	$\mathcal{D}(C,F)$
$F \leq K, F \neq K$	$F(C) > K$	$\uparrow$ or $\leftrightarrow$
$F \geq K, F \neq K$	$F(C) < K$	$\downarrow$ or $\leftrightarrow$

Table 2: Sufficient pruning conditions for hard constraints.

Using the definition of $\mathcal{D}$, Table 2 specifies sufficient conditions to prune the current configuration for a given hard constraint. Consider the constraint below:

```
ASSERT cost(W[1], C) / cost(W, COrig) ≤ 0.1
```

In this case, $\mathcal{D}(C, F) = \uparrow$ if `W[1]` is a `SELECT` query. The reason is that $\mathcal{D}(C, \texttt{cost(W[1], C)})$=$\uparrow$, $\mathcal{D}(C, \texttt{cost(W, COrig)})$=$\leftrightarrow$, and finally $\uparrow$ / $\leftrightarrow$=$\uparrow$. If, during the search procedure, the current configuration C satisfies $F(C) > 0.1$ (i.e., C violates the constraint), we can guarantee that no element in closure(C) obtained by transforming C would ever be feasible, because values of $F(C')$ are always larger than $F(C)$ for any C' transformed from C. Therefore, pruning C is safe (see Figure 4 for an illustration of this reasoning).

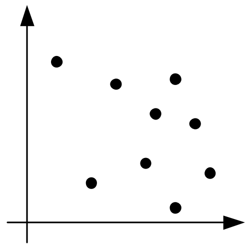
Soft Constraints. In addition to the conditions stated in Table 2, pruning a configuration C based on a soft constraint additionally requires that C satisfy all the hard constraints (since any value of the *c-objective* associated with the soft constraint is acceptable, we might otherwise miss overall valid solutions).

4.4.1 Additional Pruning Guidance

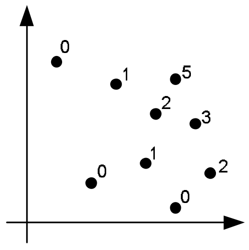
Although the above technique safely prunes configurations guaranteed to be invalid, there are certain situations in which we require

[3]A variation of this approach is used in [14, 17] in the context of constrained evolutionary algorithms.

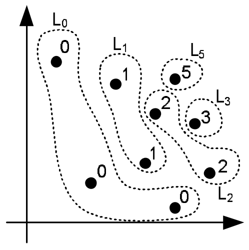
[4]We shuffle element ranks in each equivalence class to decrease the chance of getting caught in local minima due to some arbitrary ordering scheme.



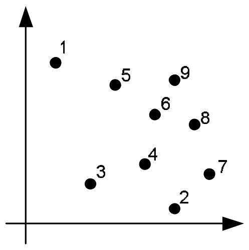
(a) Original Points.



(b) Pareto ranking.

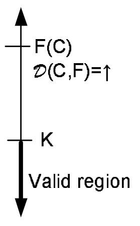


(c) Pareto layers.



(d) Instance ranking.

Figure 3: Inducing a partial order from the dominance relationship.



Constraint: $F \leq K$
$F(C) > K$
$\mathcal{D}(C, F) = \uparrow$
Result: Prune

Figure 4: Sample pruning condition.

additional support. Suppose that we want to minimize the cost of a workload with updates using the constraint below:

```
SOFT ASSERT cost(W, C) ≤ 0
```

Since the workload has updates, $\mathcal{D}(C, cost(W, C))$=?. However, suppose that the initial configuration does not contain any index on table R, and all updates queries refer exclusively to table R. In this situation we *know* that the cost of the workload would always increase as we apply transformations, but our system cannot infer it. To address such scenarios, we augment the constraint language with annotations that override the default pruning behavior. Specifically, by adding the keyword `MONOTONIC_UP` (respectively, `MONOTONIC_DOWN`) before the `ASSERT` clause, we specify that the respective constraint function F satisfies $\mathcal{D}(C, F) = \uparrow$ (respectively $\mathcal{D}(C, F) = \downarrow$). Of course, our framework has no way to verify whether the annotation is correct (otherwise it would have used this knowledge upfront!) and implicitly trusts the annotation as being correct. The example above can then be augmented as follows:

```
SOFT MONOTONIC_UP ASSERT cost(W,C) ≤ 0
```

4.4.2 Heuristic Pruning

To allow for additional flexibility in defining the search strategy, in this section we present annotations that heuristically restrict the search space. In contrast to the previous section, these annotations result in a trade-off between search space coverage and the efficiency of the search procedure, and are interesting when at least one constraint satisfies $\mathcal{D}(C, F) = ?$. Recall that our search strategy keeps applying transformation rules to the current configuration with the objective to obtain the best configuration that satisfies all constraints. Since *c-objectives* are usually conflicting, a configuration that improves some objectives might move away from others. However, if the transformed configuration does not improve any objective, there might not be an incentive to continue exploring beyond that point (of course, this is a heuristic and as such it might prune valid solutions). Instead, we might consider the configuration an end-point and backtrack to a previously seen configuration. This pruning condition can be succinctly expressed using the notion of dominance. Suppose that the current configuration, C was obtained by using some transformation over configuration C_p. Then, whenever C_p dominates C we prune C and backtrack. We can enable this heuristic pruning by annotating the global constraint specification with the value `USE_DOMINANCE_PRUNING`.

To provide even additional flexibility into the search strategy, we enable two annotations that modify how pruning is handled for individual constraints that satisfy $\mathcal{D}(C, F) = ?$. Specifically, we can specify the following behaviors:

`HILL_CLIMB`: If a constraint is marked as `HILL_CLIMB`, any transformation from C_p to C that results in a value of the constraint in C that is worse than that of C_p gets pruned, even though C_p does not dominate C.

`KEEP_VALID`: Values of a constraint marked as `KEEP_VALID` can go up or down from C_p to C. However, if C_p satisfies the constraint and C does not, we prune C.

The annotations discussed in this section effectively change the search strategy and require a non-trivial understanding of the search space, its relationship with constraints, and even the internal workings of the framework. Providing guidance to assist users or even propose the usage of such annotations is a very important problem that lies outside the scope of this work.

4.4.3 Transformation Guidance

Suppose that we want an existing index $goodI$ to appear in the final configuration. We can achieve this by using a constraint:

```
FOR I in C
WHERE name(I) = "goodI"
ASSERT count(I) = 1
```

This is such a common situation that we provide an alternative and more direct approach to achieve the same goal:

```
AVOID delete(I) WHERE name(I)="goodI"
```

would mechanically ignore any transformation that matches the specification above. In general the syntax of such specification is:

```
AVOID transformations [WHERE predicate]
```

As a less trivial example, to avoid merging large indexes we can use the following fragment:

```
AVOID merge(I1,I2)
WHERE size(I1)≥100M OR size(I2)≥100M
```

As with other heuristic annotations, the usage of these alternatives should be guided by special knowledge about the search space and its impact on the input constraints.

5. IMPLEMENTATION DETAILS

In this section we provide some implementation details of a prototype built using the constraint optimization framework described earlier. We also explain some extensions that enable additional flexibility and performance. Figure 5 illustrates the different required steps to go from a problem specification to a `SQL` script that deploys the resulting physical design. Initially, we provide a specification

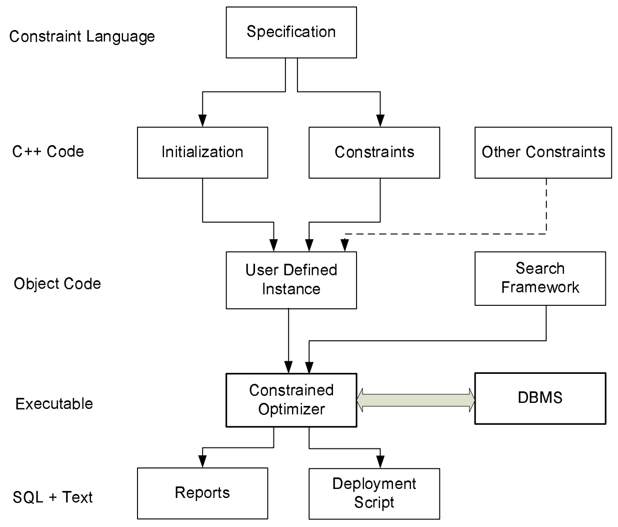


Figure 5: From Problem Specification to Results.

for the constrained optimization problem. A full specification contains a header, which includes database and workload information (e.g., the location to find the DBMS and the workload), and the main body, which includes the initial configuration and all the constraints specified in the language of Section 2. A special-purpose compiler consumes the specification and produces two C++ files. One file provides the necessary plumbing mechanism to initialize the search framework and perform the optimization and the other specifies each of the constraints by using C++ classes (more details are discussed in Section 5.1). Note that it is possible to directly specify constraints in C++, which provides more flexibility at the expense of simplicity. After all constraints are translated into C++ classes, the next step compiles this intermediate code and links the result with the search framework library. This step produces a program that connects to the database system and attempts to solve the constrained optimization problem. Upon completion, the executable returns a SQL script, which can be used to deploy the best configuration, and additional reports that provide details on the configuration to be deployed and the overall search process[5].

5.1 Compilation into C++ classes

An important extensibility mechanism results from using C++ as an intermediate language to specify constraints. In fact, we can use C++ to directly specify constraints that are too complex to be handled inside the constraint language, or constraints that require specific extensions for performance. We now describe the compilation step from the original specification language into C++. Each constraint is translated into a class derived from the base Constraint class, which is defined as follows:

```
class Constraint {
protected:
  typedef enum {TNONE, TUP, TDOWN, ...} TPruning;
  virtual TPruning pruning(Conf* conf) {return TNONE;}
  virtual double score(Conf* conf) = 0;
  virtual double estScore(Conf* fromConf,
                          Conf* toConf,
                          Transformation* t);
...
}
```

[5]Reports additionally describe suboptimal configurations, present tradeoffs in terms of constraint violation, and allow DBAs to analyze in relative depth the benefits of a particular configuration.

The base Constraint class exposes three virtual methods. The first one, pruning, returns the value $\mathcal{D}(C, F)$. By default it always returns TNONE (i.e., corresponds to $\mathcal{D}(C, F) = ?$) and its definition implements the inference mechanism and the heuristic annotations discussed in Section 4.4. The second one, score, is called every time we need to obtain the value of the *c-objective* associated with the constraint. It takes a configuration as an input and returns a real number. The result value from score should be zero when the constraint is satisfied, and larger than zero otherwise (its magnitude should reflect the degree of constraint violation). Clearly, the simplicity of the constraint language makes the compilation step into derived classes fully mechanical. As an example, consider the following constraint, which enforces that no index is larger than half the size of the underlying table:

```
FOR I in C
ASSERT size(I) ≤ 0.5 * size(table(I))
```

In this case, the generated function would look as follows:

```
class C1:  public Constraint {
...
   double score(Conf* conf) {
      double result = 0;
      for (int i=0; i<conf->numIndexes(); i++) {
        double f = size( conf[i] );
        double c = 0.5 * size( table(conf[i]) );
        double partialResult = MAX(0.0, f - c);
        result += partialResult;
      }
      return result;
    }
...
};
```

The third function in the base Constraint class, estScore, is called every time we need to estimate the *c-objective* for a given transformation. It takes as inputs the original configuration, the transformation, and the resulting configuration, and returns a real number. There is a default implementation of estScore that mimics almost exactly the implementation of score working on the transformed configuration. A subtle point is that the methods that obtain the cost of the workload under a given configuration are automatically replaced in estScore with those that exploit local transformations from the original configuration, and therefore the default implementation is very efficient. We can, however, replace the default implementation estScore with a customized version that further improves efficiency. Consider again the storage constraint:

```
FOR I in C
ASSERT sum( size(I) ) ≤ 200M
```

and suppose that the transformation merges I_1 and I_2 into I_3. Using the following equality:

$$\sum_{I \in \text{toConf}} size(I) = size(I_3) - size(I_1) - size(I_2) + \sum_{I \in \text{fromConf}} size(I)$$

we can compute the size of the transformed configuration in constant time, provided that we have the size of the original configuration available. Note that all transformations follow the same general pattern, i.e., $C_{\text{after}} = C_{\text{before}} \cup I^+ - I^-$, where I^+ and I^- are set of indexes. Therefore, in many situations we can incrementally evaluate ASSERT functions by reusing previously computed values.

6. EXPERIMENTAL EVALUATION

We now report an experimental evaluation of the search framework described in this paper.

6.1 Experimental Setting

Our experiments were conducted using a client prototype that connects to an augmented version of Microsoft SQL Server. The server code-base was extended to support the techniques in [4, 7] to provide what-if functionality and the ability to exploit local transformations. For our experiments we used a `TPC-H` database and workloads generated with the `QGen` utility[6].

6.2 Single Storage Constraint

We first consider the traditional scenario with a single storage constraint, and compare our framework against previous work in the literature. We used a 1GB `TPC-H` data and tuned a 22-query workload with both our framework and the relaxation approach of [4] augmented with the techniques of [7] so that both approaches rely on the same underlying query optimization strategy. We used three minutes for each tuning session, and simulated the approach in [4] with the following constraint specification:

```
Initial = CSelectBest
SOFT ASSERT cost(W,C) = 0
ASSERT size(C) ≤ B
```

where `B` is the storage bound (note that the last line is the only strictly required one, since the other two are always included by default). Figure 6 shows the resulting execution cost of the workload for different values of `B`. We can see that the results are virtually indistinguishable for storage bounds that cover the whole spectrum of alternatives. Figure 7 compares the efficiency of both approaches. We can see that our framework can evaluate roughly half of the number of configurations in the approach of [4, 7], and the trends are similar in both approaches. The additional time per configuration in our approach comes from additional layers of infrastructure required to generalize the approach in [4] to work with arbitrary constraints (in other words, many components are hard-wired in [4]). Considering that our framework is substantially more general and there are many opportunities for performance improvement, we believe that our approach is very competitive.

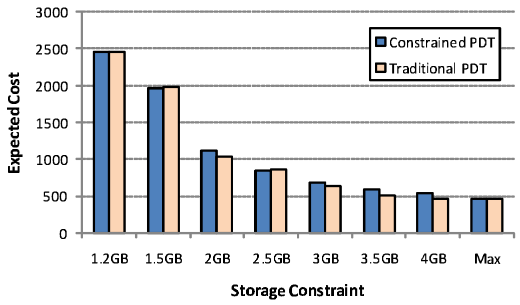


Figure 6: Quality of recommendations for storage constraint.

Figure 8 shows the expected cost of the best explored configuration over time, for different storage constraints (we do not include in the figure the start-up cost required to optimize each query for the first time). We can see that usually the search procedure finds an initial solution relatively quickly, and then it refines it over time. It is important to note that after only 60 seconds, the search strategy converged to very competitive solutions in all cases.

Figure 9 illustrates the six initial iterations/backtracking when tuning the same workload with a storage constraint of 3GB. In many cases, the most promising configuration is not always the best one, and therefore the stochastic backtracking mechanism is crucial in exploring the search space.

[6]Available at `http://www.tpc.org`.

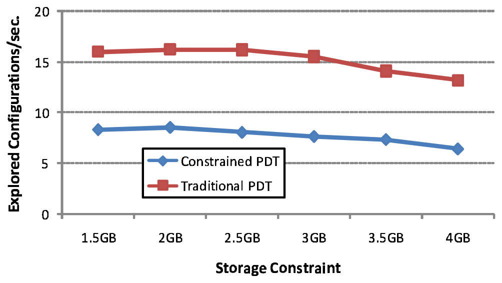


Figure 7: Efficiency of different alternatives.

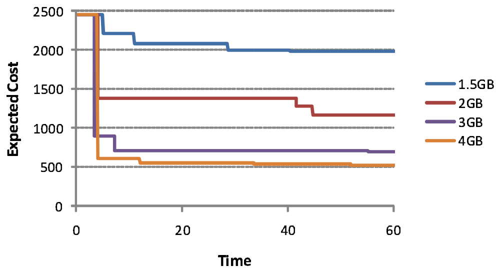


Figure 8: Quality of recommendations over time.

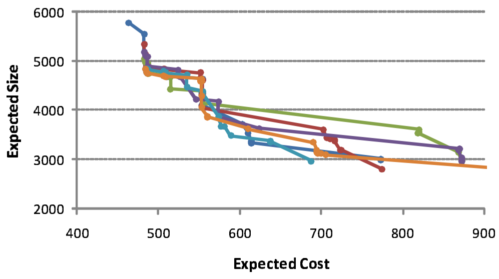


Figure 9: Backtracking to an earlier configuration.

Finally, Figure 10 shows the number of candidate transformations against the number of indexes of the originating configuration for the first 300 configurations evaluated in Figure 9. We can see that the number of candidate transformations is indeed quadratic in the number of indexes (due to the *merge* transformations), but the quadratic coefficient is significantly less than one –0.2 in Figure 10– due to restrictions in the set of feasible transformations (e.g., we cannot merge indexes on different tables).

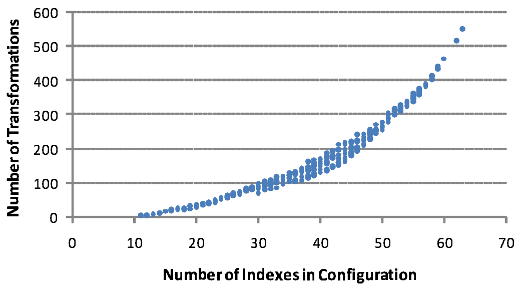


Figure 10: Number of candidate transformations.

6.3 Multiple, Richer Constraints

We now explore more complex scenarios that require additional constraints. Consider the tuning session with a 3GB storage bound that we described in the previous section. The dark bars in Figure 11 show the number of indexes per table in the resulting configuration. We can see that many tables have 6 or 7 indexes. Suppose

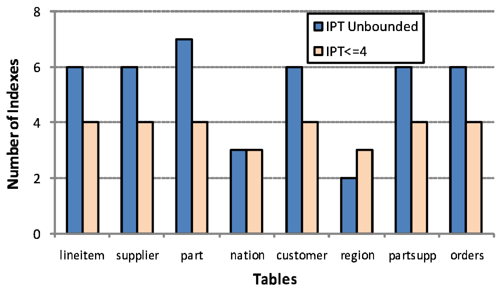


Figure 11: Number of indexes per table in two configurations.

that we want to limit the number of indexes in any given table by four. We can then search for a configuration that additionally satisfies the following constraint, denoted `IPT` for indexes-per-table:

```
FOR T TABLES
    FOR I in indexes(T)
    ASSERT count(I) ≤ 4
```

Since the specification contains a single soft-constraint, there is a single optimal configuration. Figure 12 shows this solution (at the bottom-left of the figure) along with all non-dominated configurations that are cheaper but do not satisfy all constraints. This visualization provides additional insights to DBAs, who might be willing to trade-off efficiency for some slight violation of a constraint.

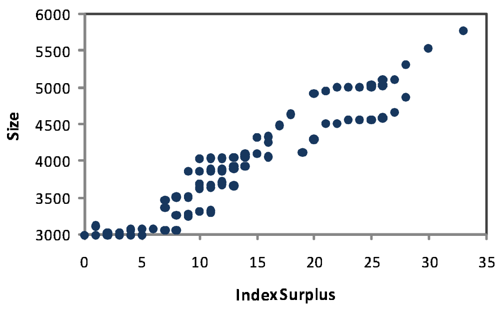


Figure 12: Non-dominated set of configurations for `IPT` $\leq$ 4.

The chosen configuration at the top-left of Figure 12 satisfies the new `IPT` constraint, as shown with the lighter bars in Figure 11. Note that the resulting configuration is not a strict subset of the original one, in which we simply removed indexes until the new constraint was satisfied. This is clearly observed in Figure 13, which depicts the cost of each query under both configurations. For each query in the figure there is a narrow line, which bounds the cost of the query under `CBase` from above, and under `CSelectBest` from below (for `SELECT` queries, any configuration results in an expected cost between these two values). Each query is also associated in the figure with a wider bar, whose extremes mark the cost of the query under the configuration obtained with just a storage constraint, and the configuration obtained by additionally bounding the number of indexes per table to four (i.e., `IPT` ≤ 4). If the configuration obtained with `IPT` ≤ 4 is the cheaper one, the bar is painted black; otherwise it is painted white. Since the figure contains both black and white bars, we conclude that there are queries that are more efficiently executed under either the original configuration and `IPT` ≤ 4. Of course, the *total* cost of the workload under the original configuration (676 units) is smaller than that under the `IPT` ≤ 4 configuration (775 units), because the space of solutions for `IPT` ≤ 4 is more restrictive than that of original specification.

As another example, suppose that we want to find a good configuration under 2GB that additionally satisfies that no query under the final configuration execute slower than 70% the time of the

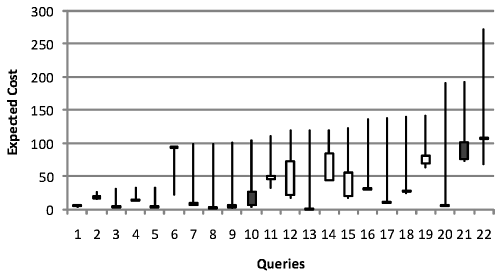


Figure 13: Expected query costs for `IPT` ≤ 4.

query under the currently deployed configuration (we denote that constraint *S70* below). The specification looks as follows:

```
FOR I IN C ASSERT sum(size(I)) ≤ 2G
FOR Q IN W ASSERT cost(Q, C) ≤ 0.7 * cost(Q, COrig)
```

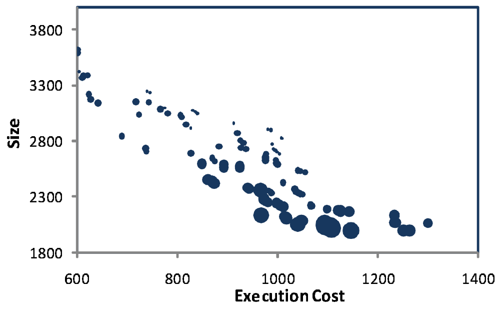


Figure 14: Non-dominated configurations for *S70*.

Running the tool for five minutes produced no feasible solution to this specification. Instead, the search procedure returned the non-dominated unfeasible configurations in Figure 14 (each circle in the figure corresponds to one configuration, and the area of the circle represents the degree of violation of the *S70* constraint). We might infer that the constraints might be too strict. Specifically, the tight storage constraint is preventing simultaneously satisfying the *S70* constraint. To relax the problem, we replaced the hard storage constraint by the following one:

```
FOR I IN C SOFT ASSERT sum(size(I)) ≤ 2G
```

Essentially we transform the problem into a multi-objective problem (reducing execution time *and* storage) with a single *S70* constraint. As there are multiple *soft-constraints*, the search strategy is not guaranteed to return a single solution. Instead, it returns the set of non-dominated configurations shown in Figure 15. These configurations present the best trade-offs between size and execution cost that satisfy the *S70* constraint (it also shows why the original specification resulted in no solutions – the smallest configuration requires 2.4GB).

Suppose that we pick this *smallest* configuration in Figure 15 (after all, our initial hard constraint limited the storage to 2GB). Figure 16 contrasts the execution cost of the queries in the workload under both this configuration and the one obtained when only optimizing for storage (i.e., when dropping the *S70* constraint), but giving the 2.4GB storage bound that the *S70* configuration required. Each query in the figure is associated with a light bar that represents 70% of the cost of the query under the base configuration (i.e., the baseline under the *S70* constraint). Additionally, each query in the figure is associated in the figure with a narrower black/white bar, whose extremes mark the cost of the query under the configuration obtained with just a storage constraint, and the configuration obtained by additionally enforcing *S70*. If the configuration obtained

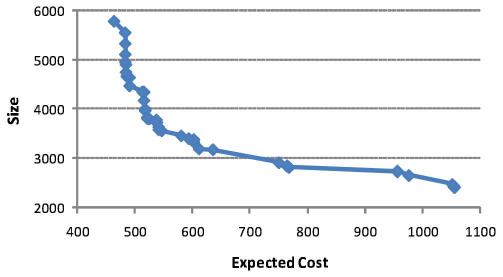


Figure 15: Non-dominated configurations for relaxed *S70*.

with *S70* is the cheaper one, the bar is painted black; otherwise it is painted white. We can clearly see that the configuration satisfying *S70* is always under the baseline (as expected). The figure also helps understand the trade-offs in cost for queries when the *S70* constraint is additionally enforced. As with the previous example, the *S70* constraint is worse than the storage-only constraint overall (901 vs 1058 units) because the search space is more restricted. However, some queries in the "no-*S70*" configuration fail to enforce the 70% bound that is required.

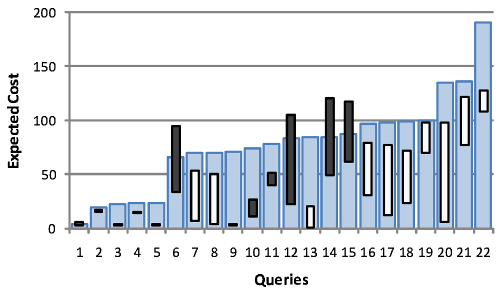


Figure 16: Expected query costs for *S70*.

6.4 Scalability

We now analyze the scalability of our search strategy with respect to the number and complexity of the input constraints. We first generated specifications with varying numbers of simple storage constraints (strictly speaking, the most restrictive of these implies the rest, but our framework cannot make this inference and considers each one individually). Figure 17 shows the impact of the number of input constraints on the search efficiency. Increasing the number of constraints by 50x only reduces the number of evaluated configurations per second from eight to around two. Even 100 simultaneous constraints result in more than one (specifically, 1.39) configurations being analyzed per second[7]. It is important to note that the approach in [4] without the optimizations in [7] analyzes 1.09 configurations per second for a single storage constraint.

We next explore the scalability of our approach for varying complexity of the constraints. For that purpose, we created a "dummy" constraint, parameterized by (α, β) that is always satisfied but takes α milliseconds to evaluate each configuration (Section 3.1.1) and β milliseconds to estimate the promise of each candidate transformation (Section 3.1.3). Figure 18 shows the number of configurations evaluated per second when varying the values of parameters α and β for the dummy constraint. Clearly, the larger the values of α and β the fewer configurations are evaluated per unit of time. We can see from the picture that it is feasible to have evaluation functions (i.e., α) values in the second range, and our strategy would

[7]Note that a fraction of the overhead arises from using suboptimal code to maintain non-dominated configurations, so the results in a more careful implementation of our prototype would be even better.

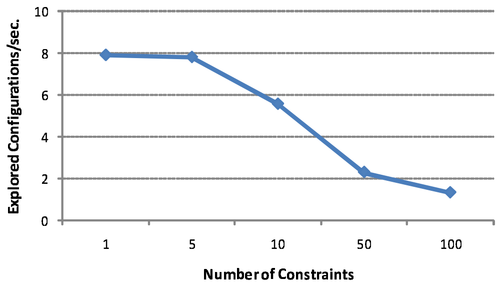


Figure 17: Scalability with respect to number of constraints.

still evaluate one configuration per second, which is similar to the performance in [4]. Higher values of β, however, degrade the efficiency of our strategy much more rapidly, because the estimation function is called multiple times per configuration to rank all the candidate transformations. Therefore, it is crucial to use efficient procedures to estimate configuration promise. We note that all the constraints discussed in this paper result in sub-millisecond α and β values. Specifically, consider the soft constraint that minimizes execution cost. This is a expensive constraint, since it requires performing local transformations to estimate candidate promises and either optimizing queries or using the techniques in [7] to evaluate configurations. Our experiments showed average values of α=9.2 ms and β=0.008 ms for this constraint.

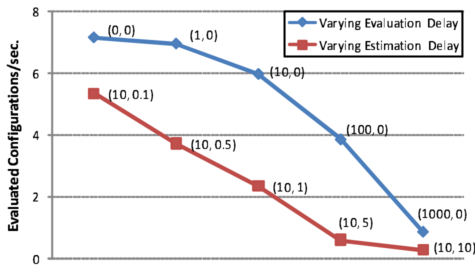


Figure 18: Scalability with respect to constraint complexity.

7. RELATED WORK

With the aim of decreasing the total cost of ownership of database installations, physical design tuning has become an important and active area of research. Several pieces of work (e.g., [2, 8, 10, 15, 18, 20]) present solutions that consider different physical structures, and some of these ideas found their way into commercial products (e.g., [1, 2, 8, 9, 10, 12, 18, 19, 20]). In contrast with this work, most of previous research has focused on a single storage constraint.

References [4, 5, 6, 7] introduce some of the building blocks of our search strategy. Specifically, [4] introduces the concept of a transformational engine and the notion of a `CSelectBest` configuration. Reference [6] exploits the techniques in [4] in the context of local optimizations, by transforming a final execution plan into another that uses different physical structures. Reference [5] considers a unified approach of primitive operations over indexes that can form the basis of physical design tools. Finally, reference [7] introduces *Configuration-Parametric Query Optimization*, which is a light-weight mechanism to re-optimize queries for different physical designs at very low overhead. By issuing a single optimization call per query, [7] is 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.

The field of constrained optimization has been extensively studied in the past, and the approaches vary depending of the nature of both constraints and the optimization function. When variables are continuous and the optimization function and constraints can be expressed as linear functions, the simplex algorithm has proved to be an effective tool. When the unknown variables are required to be integer, the problem is called *integer programming*, which is NP-Hard and can be solved by branch and bound and cutting-plane methods. Non linear but twice differentiable constraints can be solved using the non-linear optimization techniques in [11]. A sub-field more closely related to ours is combinatorial optimization, which is concerned with problems where the set of feasible solutions is discrete. Combinatorial optimization algorithms solve instances of problems that are believed to be hard in general (reference [16] proves that the general physical design problem is NP-Hard). For that reason, usually heuristic search methods (or *meta-heuristic* algorithms) have been studied. Examples of such techniques are simulated annealing, tabu search, or evolutionary algorithms (e.g., see [14, 17]).

8. CONCLUSIONS AND FUTURE WORK

In this paper we introduced the constrained physical design problem and proposed a language that enables the specification of rich constraints easily. As DBMS applications become increasingly complex and varied, we believe that constrained physical design tuning is an important addition to the repertoire of tools of advanced DBAs. As discussed in this paper, many new scenarios can be successfully and efficiently handled by our framework. We also explained how a transformation-based search strategy can be used to solve the constrained physical design problem. There are several open challenges where further work is needed. We mention some of these below:

Analysis of Constraints. The ability to reason about relationships among constraints can result in large benefits in search efficiency. For instance, if we recognize that some constraint is implied by others, or that certain constraints are positively (or negatively) correlated, we can exploit this information to guide the search strategy more effectively.

Monitoring of constraints. In the context of an evolving system, it would be very interesting to devise monitoring mechanisms that can alert whenever a constraint is no longer satisfied due to changes in either the workload or the data distribution, and therefore a tuning session would be required, similar to the work in [6].

Incremental constrained tuning. Suppose that the representative workload or data distribution changes only slightly. In this case, it would be beneficial to incrementally refine the currently deployed configuration rather than re-tune the system from scratch obtaining, perhaps, a configuration that is very different from the current one. The rationale is that DBAs might deeply understand the currently deployed configuration and they will have a high bar before accepting significant changes to the physical design.

Higher level user interaction. Although the constraint language is simple and powerful, it might not always be the preferred alterative to interact with a database system. Novel mechanisms to simplify specification of constraints, through powerful user interfaces or macros (which would then be compiled down into our constraint language) might be beneficial in easing the path to adoption.

9. REFERENCES

[1] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2004.

[2] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2000.

[3] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2001.

[4] 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.

[5] N. Bruno and S. Chaudhuri. Physical design refinement: The "Merge-Reduce" approach. In *International Conference on Extending Database Technology (EDBT)*, 2006.

[6] 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.

[7] 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.

[8] 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.

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

[10] S. Chaudhuri and V. Narasayya. Index merging. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 1999.

[11] A. R. Conn, N. I. M. Gould, and P. L. Toint. Large-scale nonlinear constrained optimization: a current survey. In *Algorithms for continuous optimization: the state of the art*, 1994.

[12] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic SQL Tuning in Oracle 10g. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2004.

[13] B. Duncan. Deadlock Troubleshooting (Part 3). Accessible at `http://blogs.msdn.com/bartd/archive/2006/09/25/deadlock-troubleshooting-part-3.aspx`.

[14] C. M. Fonseca and P. J. Fleming. Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In *Proceedings of the Conference on Genetic Algorithms*, 1993.

[15] S. Papadomanolakis and A. Ailamaki. An integer linear programming approach to database design. In *Workshop on Self-Managing Database Systems*, 2007.

[16] G. P. Shapiro. The optimal selection of secondary indices is NP-Complete. In *SIGMOD Record 13(2)*, 1983.

[17] P. D. Surry, N. J. Radcliffe, and I. D. Boyd. A Multi-Objective Approach to Constrained Optimisation of Gas Supply Networks : The COMOGA Method. In *Evolutionary Computing. AISB*, 1995.

[18] 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.

[19] D. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design advisor: Integrated automatic physical database design. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2004.

[20] D. Zilio, C. Zuzarte, S. Lightstone, W. Ma, G. Lohman, R. Cochrane, H. Pirahesh, L. Colby, J. Gryz, E. Alton, D. Liang, and G. Valentin. Recommending materialized views and indexes with IBM DB2 design advisor. In *International Conference on Autonomic Computing*, 2004.