VLDB Journal manuscript No.
(will be inserted by the editor)

Constrained Physical Design Tuning

Nicolas Bruno · Surajit Chaudhuri

Received: date / Accepted: date

Abstract Existing solutions to the automated physical design problem in database systems attempt to minimize execution costs of input workloads for a given storage constraint. In this work, 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 a previously proposed transformation-based framework to incorporate constraints into 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. Our approach generalizes previous work that assumes simpler optimization models where configuration size is the only fixed constraint. As a consequence, the process of tuning a workload becomes more flexible but also more complex, and getting the best design in the first attempt becomes difficult. We propose a paradigm shift for physical design tuning, in which sessions are highly interactive, allowing DBAs to quickly try different options, identify problems, and obtain physical designs in an agile manner.

Keywords Constrained physical design tuning · Access methods · Interactive design and tuning

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 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 [2–4, 7–10, 12–14, 21, 24, 26] and industrial-strength prototypes in all major DBMS [1, 16, 25]. Despite this substantial progress, however, the problem statement and existing solutions do not address important real-life scenarios, as we discuss next. Consider 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 than 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

N. Bruno
Microsoft Research
E-mail: nicolasb@microsoft.com

S. Chaudhuri
Microsoft Research
E-mail: surajitc@microsoft.com

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 instead of the narrow alternative I_N. A subtler scenario that results in deadlocks when narrow indexes are used is described in [17].

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 the 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 instead 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 work 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. Specifically, the main contributions of this work 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). In Section 6 we report an extensive experimental evaluation of our techniques. We then recognize that the task of tuning the physical design of a database is seldom fully specified upfront, but instead is an interactive process. In Section 7 we present a framework to enable interactive tuning sessions, which we believe represents a paradigm shift in the area of physical design tuning.

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 process them. We next 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 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 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 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. Additionally, 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. For instance, suppose that we have a uniqueness constraint on columns (a, b) in table T, which is enforced by an unique index on such columns. As another example, consider a foreign-key constraint referencing a primary key that is backed by an index. Any physical design must contain such indexes, because their purpose is not improving performance but instead guaranteeing correctness. `CBase` contains only mandatory indexes and is therefore the worst possible valid configuration for `SELECT` queries in the workload, and the one with the lowest `UPDATE` overhead.
- `CSelectBest`: This configuration is the best possible one for `SELECT` queries in the workload. Specifically, `CSelectBest` contains indexes resulting from access-path requests generated while optimizing the input workload (see [7] 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
  ```

  are `(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: To specify the storage constraint used in virtually all physical design tuning tools we use:

```
ASSERT size(C) ≤ 200M
```

where `size(C)` returns the combined size of the final configuration. Constraints start with the keyword `ASSERT` and follow the *function-comparison-constant* pattern. 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` does "pattern matching" on 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. 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 [8], 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 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)).

2.5 Language Semantics

Constrained physical design is a multi-constraint multi-objective optimization problem (soft-constraints naturally lead to more than a single optimization function). A common approach to handle such problems is to transform constraints into objective functions (we call these *c-objectives* for short) and then solve a multi-objective optimization problem.

From Constraints to C-Objectives

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[1].

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

Table 1 Converting constraints into c-objectives.

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, as follows:

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

[1] 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.

where $\alpha_i(C)$ denotes the *i-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. Section 6.3 shows empirically why this is not an easily solvable problem.

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 look for a single solution but instead for the set of solutions with the "best possible trade-offs".

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* [5] 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. For multiple soft constraints, the *Pareto optimal* solutions might not be unique, but instead show the best trade-offs among soft-constraints for the set of valid configurations.

In this section we reduced a specification in our constraint language into a multi-objective optimization problem, without giving an explicit mechanism to perform the optimization. In the following sections we introduce our framework to solve the constrained physical design problem.

3 Search Framework

We now review the general architecture of our search framework, which we adapted from [7,8]. 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). 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.

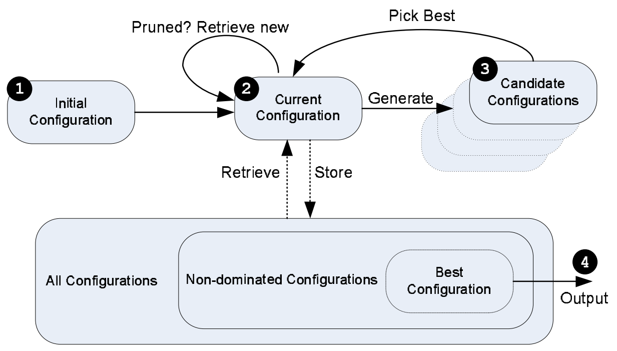


Fig. 1 Architecture of the Search Framework.

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 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. We then 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 search iteration requires evaluating a previously unexplored configuration, which 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, $\texttt{size}(C)$. 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 [13], 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 [12] or a top-down relaxation approach [7]). A recent approach introduced in [11] 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 [8]. 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 [8,14]. 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 $\{I_1, I_2\} \subseteq C$. 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, 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. Then, $\texttt{remove}(C, I)$ returns a new configuration C':

$$C' = C - \{I\} \cup \begin{cases} heap(T) & \text{if } I \text{ is clustered index of } T \\ \emptyset & \text{otherwise} \end{cases}$$

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 processing 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 [7,9] and obtain upper bounds on the cost of queries for each candidate transformation. Consider a query q and a configuration C' obtained from C. We 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.

Consider for instance 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'.

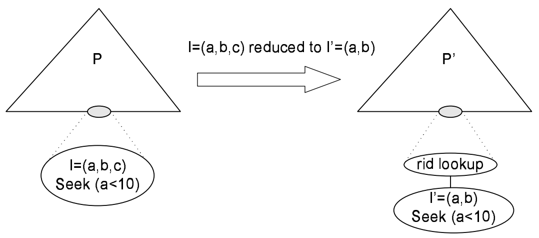


Fig. 2 Local transformations to obtain upper-bound costs.

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 [7] 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. The rationale behind this metric is that it slightly adapts the greedy solution for the fractional knapsack problem [6].

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. Let C be a configuration and C_i ($i \geq 0$) be defined as follows:

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

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[2]. The details on how to obtain `CSelectBest` can be found in [7,11]. Essentially, we instrument the query optimizer to track each logical expression that can be answered with appropriate access paths, and obtain the best possible index that satisfies each of such "access-path-requests". The net effect is that, if all such indexes were materialized, the optimizer would find, at each step, the best possible indexes and therefore result in the best possible query execution plan for varying configurations.

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

4 Constrained Physical Tuning

So far we introduced a small constraint language (Section 2) and a mechanism to transform a specification in our language into a multi-objective optimization problem (Section 2.5). We additionally reviewed a general transformation-based strategy to traverse the space of valid configurations for the case of a single constraint (Section 3). In this section we explain how to generalize this search framework to incorporate multiple objectives and leveraging Pareto-optimality concepts.

4.1 Configuration Ranking

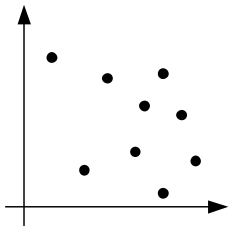
Using the notion of dominance, we can obtain a total ranking of configurations in two steps (a variation of this approach is used in [18,23] in the context of constrained evolutionary algorithms). First, we assign to each configuration a "rank" equal to the number of solutions which dominate it. As an 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 an illustration[3]. 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))[4]. This can be implemented as follows:

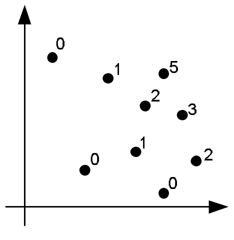
```
**RankConfigurations** ($C$=$c_1, c_2, \ldots, c_n$:configurations)
**Output R:** a ranked list of configurations
01  for each $c_i \in C$
02      rank($c_i$) = $|\{c_j \in C : c_j$ dominates $c_i\}|$
03  R = []
04  for each $i \in \{1..n\}$
05      $L_i$ = $\{c \in C :$ rank$(c) = i\}$
06      $LP_i$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

As explained in Section 3, our search strategy relies on the ability to rank configurations at two specific

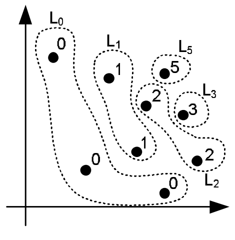
[3] Note that equivalence classes L_i are not the same as "sky-bands," where each class L_i is the skyline of the subset that does not contain any L_j ($j < i$). We experimentally evaluated this "sky-band" alternative, but it was more expensive to compute and also produced slightly worse results than our approach.

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

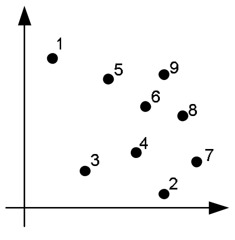

(a) Original Points.



(b) Pareto ranking.



(c) Pareto layers.



(d) Instance ranking.

Fig. 3 Inducing a partial order from the dominance relationship.

points. First, in Step 3 in Figure 1 we pick the transformation that would result in the most promising configuration. Second, after pruning the current configuration in Step 2 in Figure 1, we pick, among the partially explored configurations, the most promising one to backtrack to. Whenever we require to rank a set of configurations, 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.

4.2 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, richer 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,

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

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 objective configuration) with C_0. The semantics of $\mathcal{D}(C, F)$ are as follows:

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

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:

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

and propagate results through operators using rules, such as (i) $\uparrow + \uparrow = \uparrow$, (ii) $\uparrow + \downarrow = ?$, and (iii) $\max(\uparrow, \leftrightarrow) = \uparrow$. Consider the following example:

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

In this case, if `W[1]` is a `SELECT` query then $\mathcal{D}(C, F) =\uparrow$. In fact, $\mathcal{D}(C,$ `cost(W[1],C)`$)=\uparrow$, $\mathcal{D}(C,$ `cost(W,COrig)`$)=\leftrightarrow$, and $\uparrow / \leftrightarrow = \uparrow$. Constraints with generators and aggregations are handled similarly, but the inference mechanism is generally less accurate. For a constraint of the form `FOR x IN X ASSERT F(x) ≤ K` we need to check both $\mathcal{D}(C, F(x))$ for each x and also $\mathcal{D}(C, |X|)$. For instance, consider a generalization of the previous constraint:

```
FOR Q in W ASSERT cost(Q, C) / cost(W, COrig) ≤ 0.1
```

If all queries in the workload are `SELECT` queries, we would obtain, as above, that $\mathcal{D}(C, F(Q)) =\uparrow$ for each Q in W. Also, since transformations do not change the workload, we have that $\mathcal{D}(C, |W|) =\leftrightarrow$. Combining these facts we can infer that $\mathcal{D} =\uparrow$ overall (recall from Section 2.5 that in presence of generators we sum the values of each individual `ASSERT` clause).

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 again the constraint:

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

and suppose that during the search procedure the current configuration C satisfies $F(C) > 0.1$ (i.e., C violates the constraint). We can then 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). Note that for a simple storage constraint of the form `ASSERT size(C)` $\leq$ `B`, the pruning condition reduces to that of previous work (i.e., if the current configuration is smaller than the storage constraint B, prune it and backtrack to a previous configuration [7]).

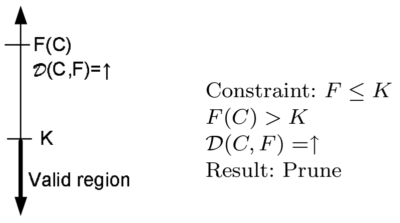


Fig. 4 Sample pruning condition.

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).

In the remainder of this section, we show how we can alter the default search procedure in our framework by modifying the way we deal with constraints, and thus obtain new functionality.

4.2.1 Additional Pruning Guidance

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

4.2.2 Heuristic Pruning

To allow for additional flexibility in defining the search strategy, we introduce 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 $\mathcal{D}(C,F) =?$. 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 additional flexibility into the search strategy, we introduce two annotations that alter 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.

4.2.3 Transformation Guidance

Suppose that we want an existing index *goodI* (or some of its merges/reductions) to appear in the final configuration. We can achieve this with:

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

This is a common situation so we provide an alternative, direct approach to achieve the same goal:

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

would ignore any transformation that matches the predicate. 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 discussed in this section, the usage of these alternatives should be guided by special knowledge about the search space and its impact on the input constraints.

4.2.4 Handling Constraint Priorities

By manipulating the pruning conditions, we can enable a prioritized way of dealing with constraints. In this special modality, constraints are sorted in the order in which they appear in the specification, and we must satisfy them in such order. For concreteness, let the ordered constraints be $\mathcal{X}_1, \ldots, \mathcal{X}_n$, and suppose that we transform C_{before} into C_{after}. Let $\mathcal{X}_i^{\text{before}}$ and $\mathcal{X}_i^{\text{after}}$ be the score of $\mathcal{X}_i$ under C_{before} and C_{after}, respectively. We can implement prioritized constraints by pruning C_{after} whenever the following condition holds:

$$\exists\, i \leq n : \mathcal{X}_i^{\text{after}} > \mathcal{X}_i^{\text{before}} \text{ and } \forall j < i : \mathcal{X}_j^{\text{before}} = 0$$

which is operationally described below:

```
Prioritized-Prune (CBefore, CAfter: Configurations)
01  i = 1
02  while (i ≤ n and $\mathcal{X}_i^{\text{before}}$ = 0)
03     if ($\mathcal{X}_i^{\text{after}}$ > 0)
04        return false
05     i = i + 1;
06  return (i ≤ n and $\mathcal{X}_i^{\text{after}}$ > $\mathcal{X}_i^{\text{before}}$)
```

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 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, the overall search process, and allow DBAs to analyze the benefits of a particular configuration.

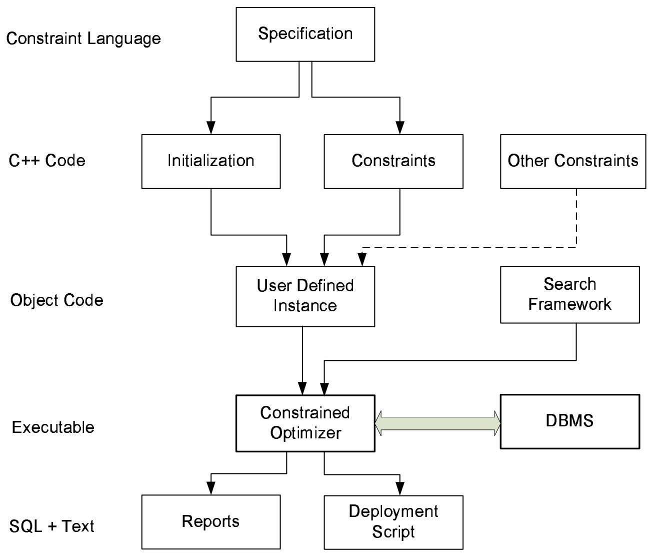


Fig. 5 From Problem Specification to Results.

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, 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);
...
}
```

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 4.2. 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))
```

The generated function would then 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 (i.e., `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) ) ≤ 200MB
```

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

$$\sum_{I \in \text{toConf}} size(I) =$$

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

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. We could then 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 work.

6.1 Experimental Setting

Our experiments were conducted using a client prototype that connects to an customized version of Microsoft SQL Server 2005. The server code-base was extended to support the techniques in [7,11] 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 `QGen`[5].

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

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

framework and the relaxation approach of [7] augmented with the techniques of [11] 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 [7] 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 [7,11], 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 [7] to work with arbitrary constraints (i.e., many components are hardwired in [7]). Considering that our framework is substantially more general and there are many opportunities for performance improvement, we believe that our approach is very competitive.

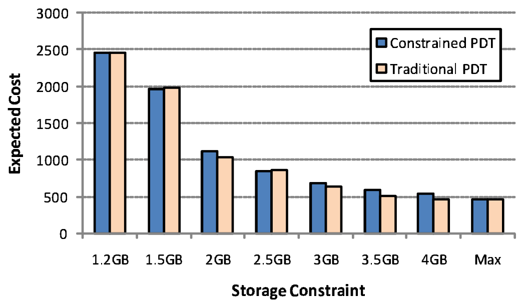


Fig. 6 Quality of recommendations for storage constraint.

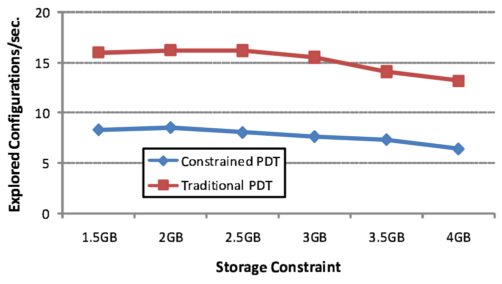


Fig. 7 Efficiency of different alternatives.

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.

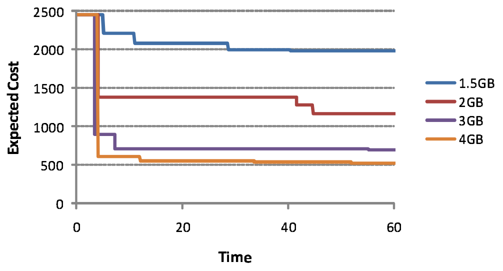


Fig. 8 Quality of recommendations over time.

Figure 9 illustrates the six initial iterations with 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.

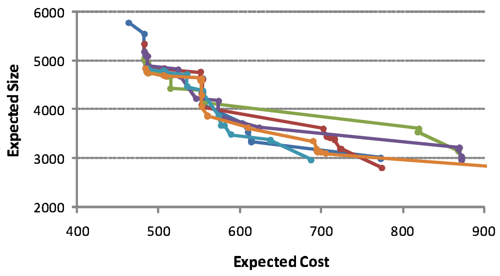


Fig. 9 Backtracking to an earlier configuration.

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.

6.3 Single-Objective Optimization

In Section 2.5 we argued against combining different optimizing functions into a single quantity and used

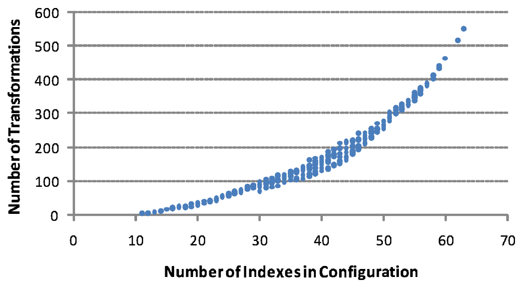


Fig. 10 Number of candidate transformations.

instead Pareto-optimality concepts to rank configurations. The reason is that combining different optimizing functions is not a robust approach as there is no easy way to balance the different original optimizing objectives. To better illustrate this issue, we tuned the 22-query workload of the previous section with a storage bound of 2GB. However, rather than using Pareto-Optimality to rank the alternative configurations, we used the following scoring function:

$$\mathit{score}_W(C) = \mathit{cost}(W, C) + w \cdot \max(0, \mathit{size}(C) - 2000)$$

where *size(C)* returns the size of C in megabytes and w is a real number that balances the relative importance of the two conflicting constraints (i.e., minimizing cost of the workload versus getting the size of the configuration below 2GB). We used three different values for w: 1/1000, 1, and 1000, which correspond to trading one unit of execution with one gigabyte, megabyte, or kilobyte of space, respectively. Given enough time, the search strategy would visit every possible configuration, so the difference in scoring functions make sense in scenarios for which an exhaustive search is unfeasible.

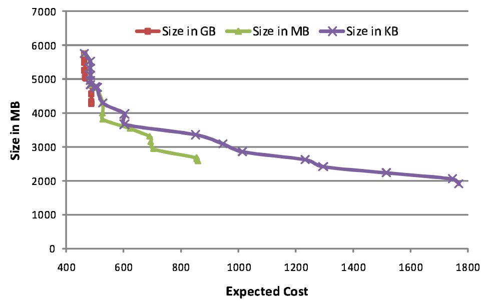


Fig. 11 Initial search space with different ranking functions.

Figure 11 shows the first 25 iterations of the search framework when using the three versions of the scoring function (each point in the figure corresponds to a different configuration). We can see that when using $w = 1/1000$ (i.e., balancing execution units and gigabytes) the search is very conservative about moving toward configurations that increase execution cost. The reason is that each execution unit of overhead is the same as 1 gigabyte (out of just six) of reduction in space. Analogously, when using $w = 1000$ the search strategy is much more liberal in reducing storage and reaches the goal of 2GB quicker than the other alternatives (by also performing worse in terms of expected cost). Using $w = 1$ results in a middle-ground strategy between the previous two extremes. Even in this simple example with two constraints, the choice of w heavily influences the search strategy, and it is not obvious how to set a good value of w for arbitrary constraints.

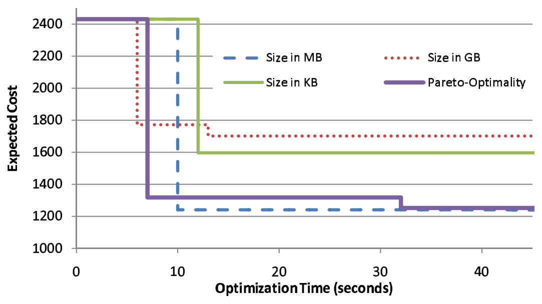


Fig. 12 Best configuration for different ranking functions.

To further illustrate the issue above, Figure 12 shows the expected cost of the best configuration found over time for each of the ranking functions, and also for the strategy that uses Pareto-optimality to rank alternatives. We can see that in this case the strategy that uses $w = 1$ (coincidentally) performs better than either $w = 1000$ or $w = 1/1000$. The reason is that workload costs vary in the range of 1000-2500 execution units, and configuration sizes vary in the range of 2000-6000 MB, which are very similar in magnitude. Of course, this is easy to see on hindsight, but very difficult to predict beforehand, specially for multiple complex constraints. It is interesting to note, also, that our approach, which uses Pareto-optimality to rank alternatives, is initially worse than the approach that uses $w = 1$, but recovers and results in a configuration that is comparable to that of $w = 1$. In contrast, the strategies using $w = 1/1000$ and $w = 1000$ do not find a better configuration for a long time (roughly 5 times the interval shown in the figure).

6.4 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 13 show the number of indexes per table in the resulting configuration. We can see that many tables have 6 or 7 indexes. Suppose 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
```

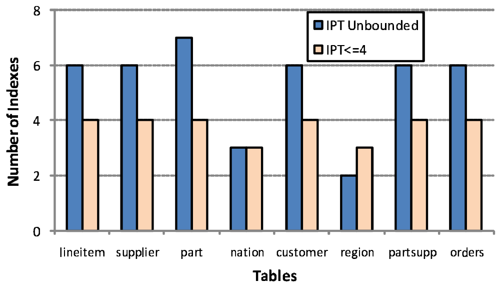


Fig. 13 Number of indexes per table in two configurations.

Since we specified a single soft-constraint, there is a single optimal configuration. Figure 14 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 constraint violation.

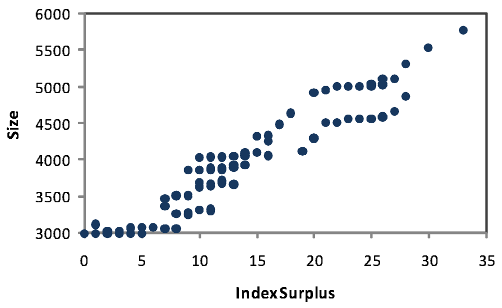


Fig. 14 Non-dominated set of configurations for `IPT` $\leq$ 4.

The chosen configuration at the bottom-left of Figure 14 satisfies the new `IPT` constraint, as shown with the lighter bars in Figure 13. 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 15, 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.

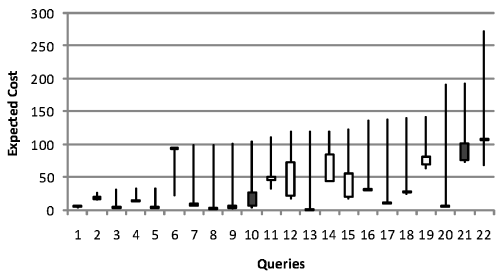


Fig. 15 Expected query costs for `IPT` $\leq$ 4.

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 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)
```

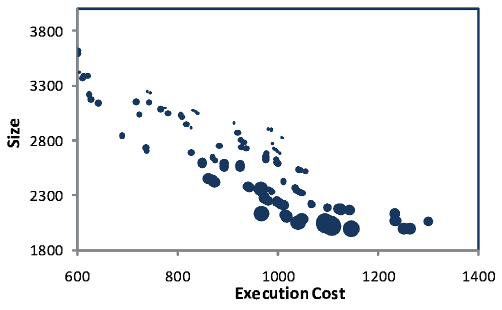


Fig. 16 Non-dominated configurations for *S70*.

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 16 (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 modify the storage constraint as follows:

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

thus resulting in 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 17. 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 such configuration requires 2.4GB).

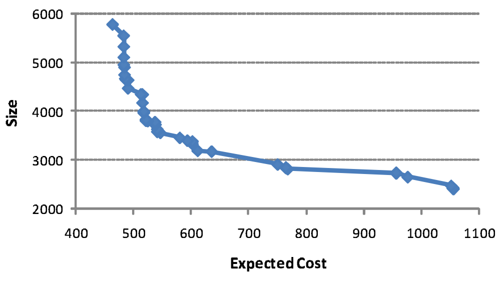


Fig. 17 Non-dominated configurations for relaxed *S70*.

Suppose that we pick this *smallest* configuration in Figure 17 (after all, our initial hard constraint limited the storage to 2GB). Figure 18 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 two points: the expected cost of the query under the configuration obtained with just a storage constraint (*No S70* in the figure) , and the configuration obtained by additionally enforcing *S70* (*S70* in the figure). 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. The *S70* constraint is worse than the storage-only constraint overall (905 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.

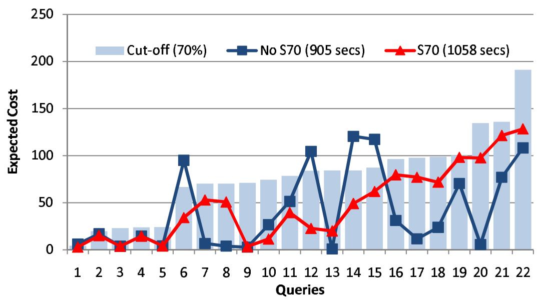


Fig. 18 Expected query costs for *S70*.

6.5 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 19 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[6]. It is important to note that the approach in [7] without the optimizations in [11] analyzes 1.09 configurations per second for a single storage constraint.

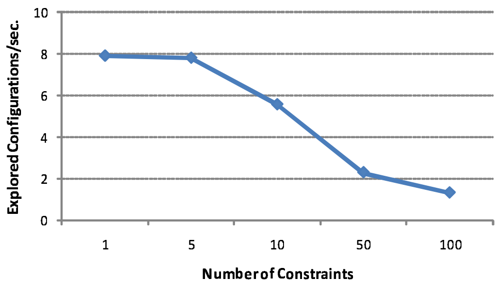


Fig. 19 Scalability with respect to number of constraints.

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 20 shows the number of configurations evaluated per second when varying the values of parameters α and β for the dummy

[6] A fraction of the overhead is due to using suboptimal code to maintain non-dominated configurations.

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 still evaluate one configuration per second, which is similar to the performance in [7]. 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 work result in sub-millisecond α and β values. 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 [11] to evaluate configurations. Our experiments showed average values of α=9.2 ms and β=0.008 ms for this constraint.

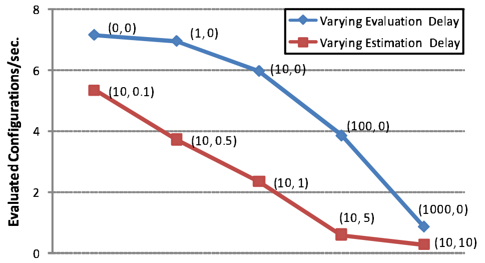


Fig. 20 Scalability with respect to constraint complexity.

7 Interactive Physical Design Tuning

The previous sections introduced a language to enable specification of constraints for physical design tuning. We showed that this mechanism results in significant flexibility and allows much more control during the physical design tuning process. At the same time, however, it also accentuates the need for a different way of conducting the physical design tuning process. Current physical design tools are monolithic, expose tuning options that are set at the beginning, and generate, without further user input, a final configuration to deploy into a production system.

We believe that a change of paradigm is required to take physical design tuning to the next level. Specifically, we claim that tuning sessions should be highly interactive. Current monolithic architectures in physical design tools force users to specify the whole problem upfront and prevent users from making changes a posteriori or in general interacting with the system. We hinted at a more interactive approach in the examples of Section 6.4, where the output of a session was analyzed, tweaked and re-submitted with different (relaxed) constraints. In the remaining of this section we present a new architecture that is more suitable for interactive sessions (Section 7.1), review Windows PowerShell [19] as one possible infrastructure that can support our architecture (Section 7.2), and present a working prototype that illustrates our vision (Section 7.3).

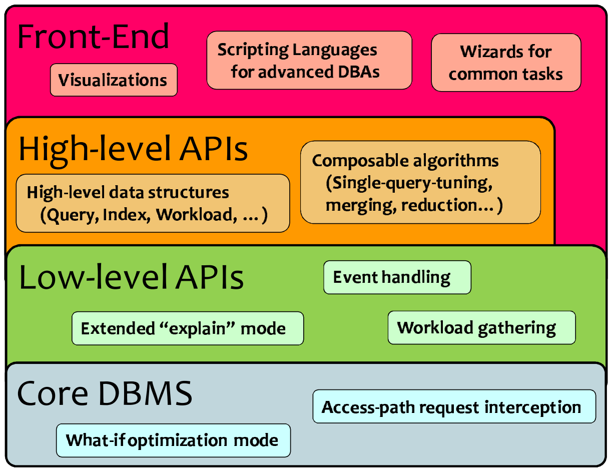

Front-End
Visualizations
Scripting Languages for advanced DBAs
Wizards for common tasks
High-level APIs
High-level data structures (Query, Index, Workload, ...)
Composable algorithms (Single-query-tuning, merging, reduction...)
Low-level APIs
Event handling
Extended "explain" mode
Workload gathering
Core DBMS
Access-path request interception
What-if optimization mode

Fig. 21 Architectural layers for next-generation physical design tuning tools.

7.1 A Layered Architecture for Physical Design Tuning

Figure 21 shows a layered architecture for next generation physical design tools that can result in better and richer interaction with DBAs. While the two lowest layers are already implemented in commercial systems, the remaining ones differ considerably from current implementations. We next describe the architecture in more detail.

- Core DBMS: The lowest layer resides within the database system and provides native support for operations such as what-if optimization [13] (with the additional fast variant of [11]) and access-path request interception functionality, as described in [7].
- Low-level APIs: The low-level APIs expose, in formats that are simple to consume (e.g., XML), the functionality of the Core DBMS layer (and also the DBMS itself). As an example, they expose primitives to manipulate what-if mode, and also richer explain modes after optimizing queries, which surface optimization information required at higher levels. These APIs also encapsulate existing DBMS functionality, such as the ability to monitor and gather workloads.

- High-level APIs: The previous two layers are, in some form or another, already present in current commercial DBMS. Physical design tools are typically built on top of the low-level APIs and only expose a rigid functionality (e.g., point to a workload, set the storage constraint, and optimize). Instead, we suggest a high-level API layer that exposes the internal representation and mechanisms used by current physical design tools in a modular way. Basic concepts such as queries, indexes, and access-path requests are exposed as to higher layers to be used in different ways. In addition to these data structures, the high-level API layer exposes composable and simple algorithms that current tuning tools rely on. For instance, this layer exposes mechanisms to merge two indexes, or obtaining the best set of indexes for a single query. These primitive data structures and algorithms are not necessarily meant to be consumed by DBAs, but instead provide a foundational abstraction for applications to be built on top.
- Front-ends: Front-ends are based on both the low- and high-level APIs and deliver functionality to end-users. A very powerful interaction model, which we describe in detail in the rest of this section, is a scripting platform to interact with physical database designs. The scripting language understands the data structures and algorithms exposed by the underlying layers and allows users to write small interactive scripts to tune the physical design of a database. Common tasks, such as minimizing cost for a single storage constraint (or other functionality provided by current physical design tools), can be seen as just pre-existing scripts that can be accessed using graphical user interfaces by relatively inexperienced DBAs.

We next review Windows PowerShell, a scripting language that can be used as a front-end in our architecture, and then describe a prototype implementation of our architecture using Windows PowerShell.

7.2 An Extensible Shell and Scripting Language

Windows PowerShell is an interactive, scripting language that integrates with the Microsoft .NET Framework. It provides an environment to perform administrative tasks by execution of cmdlets (which are basic operations), scripts (which are composition of cmdlets), stand-alone applications, or by directly instantiating regular .NET classes [19,20]. There are several server products that already leverage Windows PowerShell for management support, such as Windows Server, SQL Server, Exchange Server, and IBM WebSphere MQ, among others. The main features of Windows PowerShell are:

- Tight integration with .NET: Windows PowerShell leverages the .NET framework to represent data, and understands .NET classes natively, as illustrated below:

```
> # integer assignment and expression evaluation
> $a = 14
> $a + 10
24

> # create a new .NET random number generator
> $r = New-Object System.Random
> # invoke a method on $r
> $r.Next()
198831340
```

  This also means that new classes written in the .NET framework are easily available as first-class citizens in Windows PowerShell.
- Strict naming conventions: All cmdlets in Windows PowerShell follow strict naming conventions, namely `verb-noun`, and parameters are passed in a unified manner. Some examples of such built-in cmdlets are `Start-Service`, which starts an OS service in the current machine, `Get-Process`, which returns a list of processes currently executing, `Clear-Host`, which clears the screen, and `Get-ChildItem` which, if located in a file system directory, returns all its subdirectories or files. There are also aliases for the common cmdlets, so we can write `dir` or `ls` rather than `Get-ChildItem`.
- Object pipelines: Similar to Unix shells, cmdlets can be pipelined using the `|` operator. However, unlike Unix shells, which typically pipeline strings, Windows PowerShell pipelines .NET objects. This is a very powerful mechanism. For instance, the script:

```
> Get-Process | Sort-Object -Property Handles -Desc |
    Select-Object -first 5 | Stop-Process
```

  obtains the list of all running processes, pipes the result (which is a list of `System.Diagnostics.Process` .NET objects) to the `Sort-Object` cmdlet, which understands the semantics of the objects and sorts them by the property `Handles` in descending order. In turn, the result of this cmdlet (i.e., an ordered list of processes) is passed to the `Select-Object` cmdlet, which takes the first five and passes them to the next cmdlet in the pipeline, `Stop-Process`, which terminates them (something that might not be advisable to do in a production system). The following script returns the number of lines that contains the word *constraint* in any LaTeX file in the current directory that is below 100,000 bytes long:

```
> Get-ChildItem -Path *.tex |
   Where-Object -FilterScript { $_.Length -lt 100000 } |
   Foreach-Object -Process {
        Get-Content $_ | Select-String constraint
    } |
   Measure-Object
Count : 404
```

which gets all files in the current path that have a `tex` extension and keeps only those that are smaller than 100,000 bytes. Then, each file is processed by first getting its content (which returns a list of string .NET classes), selecting only those that contain the work *constraints*. The combined result of this subscript (which is a list of strings) is measured and the count is returned. This might seem a little verbose, but there are several mechanisms to shorten PowerShell scripts, such as aliases (e.g., `Get-ChildItem` becomes `dir`, `Where-Object` becomes `?`, `Foreach-Object` becomes `%`), and positional cmdlet parameters, so for instance we do not need to explicitly write `-Path` after `dir`. An equivalent script is shown below:

```
> dir *.tex | ? { $_.Length -lt 100000 } |
    % { gc $_ | Select-String constraint } | measure
Count : 404
```

- Data Providers: PowerShell has the ability to expose hierarchical data models by means of *providers*, which are then accessed and manipulated using a common set of cmdlets. As an example, the file system is one such provider. When situated in some node inside the file system provider, we can use `Get-ChildItem` to obtain the subdirectories or files in the current location, access contents of elements using `Get-Content`, and navigate the provider using `Set-Location` (aliased as `cd`). However, Windows PowerShell natively exposes the registry and the environment variables as providers. There also are third-party providers that give a unified surface to access, query, and modify Active Directory, SharePoint and SQL Server, among others.

In the next section, we describe how we take advantage of the different features of Windows PowerShell to provide an interactive experience for physical design tuning.

7.3 Towards Interactive Physical Design Tuning

In the remainder of this section we introduce a prototype implementation that enables interactive physical design tuning sessions. We first discuss the architecture of our solution and then provide examples that showcase the functionality and flexibility of our approach.

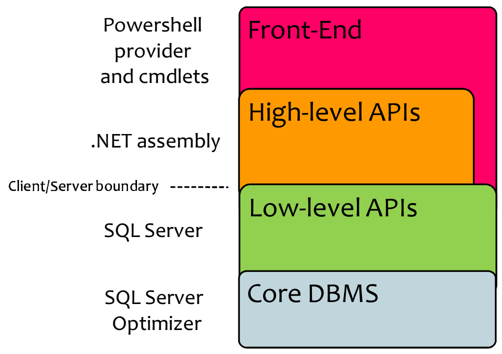


Fig. 22 Prototype enabling interactive physical design tuning.

7.3.1 Architecture

Figure 22 shows how we map the different layers of Figure 21 into our prototype implementation. We next describe each layer in more detail.

Low-level APIs

We implemented both the Core DBMS and Low-level APIs layers by instrumenting Microsoft SQL Server. Some components (e.g., what-if optimization) are already part of the shipping product, while others (e.g., access-path request interception) were added. Essentially, we re-used the same instrumented DBMS described in Section 6.

High-level APIs

We implemented the high-level API layer by introducing a new .NET assembly that encapsulates and exposes classes and algorithms relevant to physical design tuning. Among the classes that the assembly exposes are `Database`, `Table`, `Index`, `Column`, `Query`, `Configuration` and `Request`. We made these classes rich in functionality, so for instance the class `Index` has methods that return merged and reduced indexes, and methods that create hypothetical versions of the index in the database. The class `Query` has methods that evaluate (optimize) it under a given configuration, and methods that return its set of access-path requests.

Additionally, as part of the .NET assembly, we built a sophisticated caching mechanism that avoids optimizing the same query multiple times in the database server. Instead, each query remembers previous optimizations and, if asked again to optimize itself with a previously seen configuration, it returns the cached values without doing the expensive work again.

Using these classes, we can rather easily build functionality similar to existing physical design tools and package it in yet another monolithic tool. However, the more interesting alternative is that, since all classes are exposed in an assembly, we can load all definitions directly into Windows PowerShell and start exploring, in a rudimentary but already interactive form, the physical design of a database, as illustrated below:

```
> # create a new Database object from namespace PDTCore
> # pointing to server nicolasb02 and database tpch01g
> $db = New-Object PDTCore.Database("nicolasb02", "tpch01g")

Connecting to server nicolasb02
Populating tables: LINEITEM CUSTOMER SUPPLIER NATION REGION
PARTSUPP PART ORDERS (8 read)

> # examine the Database object just created
> $db

Name           : tpch01g
Connection     : nicolasb02
Tables         : LINEITEM, CUSTOMER, SUPPLIER, NATION...
Configurations : base, initial
Queries        : {}

> # obtain information on tables larger than 500 pages
> # and sort the result by size
> $db.Tables.Values | ? { $_.Pages -gt 500 } | sort Pages |
   select Name, Pages, Columns

Name     Pages Columns
----     ----- -------
PARTSUPP  1573 {PS_PARTKEY, PS_SUPPKEY...}
ORDERS    2288 {O_ORDERKEY, O_CUSTKEY...}
LINEITEM 11486 {L_ORDERKEY, L_PARTKEY...}

> # create a new query
> $q = [PDTCore.Query]::Create($db,
          "SELECT * FROM LINEITEM WHERE L_ORDERKEY=15")

> # evaluate the query under the base configuration
> $db.Configurations["base"].Eval($q)

Name    Config Query Source Cost      SelectInfos Update
----    ------ ----- ------ ----      ----------- -------
base_Q1 base   Q1    Server 0.0032831 1           False
```

Front-end

Although the examples above are compelling, they are still not user friendly (we need to call the .NET methods directly). Also, they require considerable effort before actually tuning a database design. Using the capabilities of Windows PowerShell, we add the following functionality to our prototype:

- Provider. We implemented a PowerShell provider that exposes all the information about a tuning session in a hierarchical and intuitive object model. Figure 23 shows an XML representation of a portion of a simplified version of the hierarchy exposed by the provider. By using this provider, we can navigate and manipulate the state of a tuning session easily, as illustrated below:

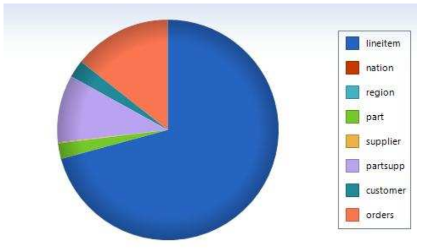
```
- <Databases>
  - <Tables>
      <Columns />
    - <Indexes>
        <KeyColumns />
        <SuffixColumns />
      </Indexes>
    </Tables>
  - <Queries>
      <Requests />
    - <QueryInfos>
        <SelectInfos />
      </QueryInfos>
    </Queries>
  - <Configurations>
      <Indexes />
    - <Transformations>
        <AddedIndexes />
        <RemovedIndexes />
      </Transformations>
      <QueryInfos />
    </Configurations>
  </Databases>
```

Fig. 23 Fragment of the Provider object model.

```
> # create a new provider on server nicolasb02
> # and default database tpch1g
> New-PDTDrive -Name P -Server nicolasb02 -Database tpch1g

> # go into the tables of tpch1g
> cd p:/tpch1g/tables

> # return all tables that start with part
> dir part* | sort Rows

Name         Database     Rows    Pages   Cols Indexes
----         --------     ----    -----   ---- -------
part         tpch1g       200000  3618    9    2
partsupp     tpch1g       800000  15628   5    2

> # get all indexes in the base configuration that have
> # more than two key columns
> dir P:/tpch01g/configurations/base/indexes |
    ? { $_.NumKeys -gt 1 }

Name                     Table     Keys Includes
----                     ------    ---- --------
PK__LINEITEM__07F6335A   LINEITEM   2 14
PK__PARTSUPP__0519C6AF   PARTSUPP   2 3
```

- Visualizations: An interesting side effect of using a composable, interactive script language is that we can very easily include third-party add-ins that offer specific functionality. One such example is PowerGadgets[7], which provides simple cmdlets to display data graphically. One such cmdlet is `Out-Chart`, which displays a chart of the data that is pipelined in with many different visualization options. As a very simple example, we can graphically display the relative sizes of all tables in a database by using:

```
> dir p:/tpch1g/tables |
    Out-Chart -gallery pie -label Name -values Pages
```



[7] Available at `http://www.powergadgets.com`.

- Cmdlets: In addition to the provider, we augmented the bare .NET classes and methods with composable cmdlets. The following examples illustrate a fraction of such cmdlets:

```
> # load queries stored in a file. Note that results
> # can later be accessed via P:/tpch1g/queries
> Get-Query -Path D:/workloads/tpch-first-3.sql

Reading queries from D:/workloads/tpch-first-3.sql...
Name  Database  Type    Rows      Requests SQL
----  --------  ----    ----      -------- ---
Q0    tpch1g    Select  5.74262   6        SELEC...
Q1    tpch1g    Select  100       63       SELEC...
Q2    tpch1g    Select  10        26       SELEC...

> # create two indexes and merge them
> $i1 = New-Index -Table lineitem -Keys l_orderkey
> $i2 = New-Index -Table lineitem -Keys l_partkey
                  -Includes l_tax, l_orderkey
> $i3 = $i1.merge($i2)
> $i3.keys

Name          Table      Width
----          -----      -----
l_orderkey    lineitem   4
l_partkey     lineitem   4

> # create a new configuration
> $c = New-Configuration -Indexes $i1, $i3

> # evaluate the three most expensive queries
> # under the new hypothetical configuration
> dir p:/tpch1g/queries | sort -desc Cost |
    select -first 3 | Eval-Query -Configuration $c

Name   Config   Query    Cost
----   ------   -----    ----
C2_Q2  C2       Q2       143.075
C2_Q1  C2       Q1       6.05483
C2_Q0  C2       Q0       120.384
```

- Scripts: The power of our implementation comes from scripts (either user defined or several of the ones that we already built). For instance, the script below is a simplified implementation of a common operation called *refinement*, in which a given input configuration is repeatedly "relaxed" via merging an reduction operations until it fits in the available storage, so that the expected cost of the resulting configuration is as good as possible[8]. At each iteration, we calculate all possible transformations (using a cmdlet) and obtain the one that is expected to result in the smaller execution cost, repeating this process until a valid configuration is reached:

```
function Refine-Configuration() {
  Param ([PDTCore.Query[]] $Workload,
         [PDTCore.Configuration] $Configuration,
         [double] $Size)
  $act = $Configuration
  while ($act.Size -gt $Size) {
    $tr = Get-Transformations $Workload -Config $act |
            sort Cost | select -first 1
    $act = $tr.Apply()
  }
  return $act
}
```

In addition to `Refine-Configuration`, we implemented other common algorithms, such as the relaxation-based tuning approach in [7] and also a version that handles the constraint language of Section 2. This script is called `TuneConstrained-Workload` and takes as inputs a workload, a timeout, and a set of constraints as defined in Section 2. It heavily uses the .NET classes exported by the high-level APIs and is implemented as a PowerShell script in fewer than 100 lines of code. It is interesting to note that this PowerShell script corresponds to a rather non-trivial algorithm but is easily implemented reusing primitives exposed by lower layers.

[8] See [8] for additional details on this operator.

7.3.2 A Sample Interactive Tuning Session

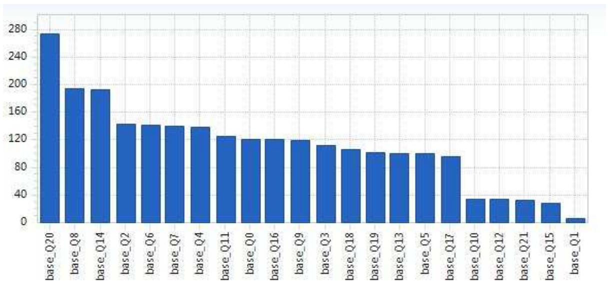
We conclude this section with an annotated trace of a session that we conducted using our prototype. Figure 24 illustrates how an interactive approach can benefit DBAs by providing flexibility and control during physical design tuning. The example uses the provider, cmdlets and scripts described earlier, and some new others (specially concerning visualization). We expect that advanced DBAs create their own scripts to further customize the physical design tuning experience. Moreover, native PowerShell features, such as remoting (which allows to execute commands in other machines) or eventing and automation can surely complement tuning scripts providing additional flexibility.

8 Related Work

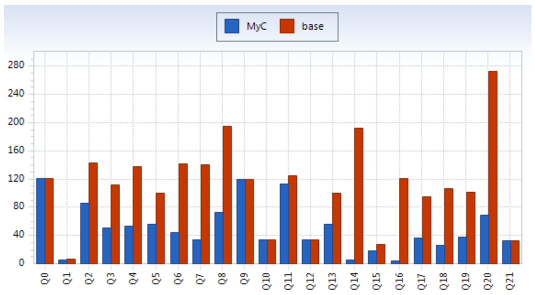
With the aim of decreasing the total cost of ownership of database installations, physical design tuning has become an important and active area of research. Several pieces of work (e.g., [2,12,14,21,24,26]) present solutions that consider different physical structures, and some of these ideas found their way into commercial products (e.g., [1,2,12–14,16,24–26]). In contrast with this work, most of previous research has focused on a single storage constraint.

References [7–9,11] introduce some of the building blocks of our search strategy. Specifically, [7] introduces the concept of a transformational engine and the notion of a `CSelectBest` configuration. Reference [9] exploits the techniques in [7] in the context of local optimizations, by transforming a final execution plan into another that uses different physical structures. Reference [8] considers a unified approach of primitive operations over indexes that can form the basis of physical design tools. Finally, reference [11] 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, [11] is able to generate a compact representation of the optimization space that can

```
> # create a new provider
> New-PDTDrive -Name P -Server nicolasb02 -Database tpch1g

> # set the current location at the root of the provider
> cd P:

> # load TPC-H workload
> $w = Get-Query -Path D:/workloads/tpch-all.sql

Reading queries from D:/workloads/tpch-all.sql...

> # get the cost of all queries in the base configuration in decreasing order of cost
> $c = Get-Configuration base
> $w | Eval-Query -Configuration $c | sort -desc cost | out-chart -values Cost -label Name
```



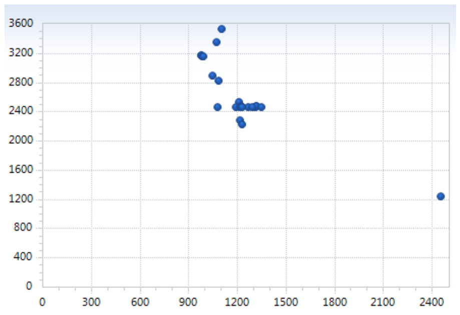
```
> # show the top-3 most expensive queries
> $expW = $w | Eval-Query -Configuration $c | sort cost -desc | select -first 3 | % {$_.query}
> $expW

Name     Database     Type       Rows       Requests
----     --------     ----       ----       --------
Q20      tpch1g       Select     100        45
Q8       tpch1g       Select     172.421    48
Q14      tpch1g       Select     999.809    8

> # for each expensive query, obtain the access-path requests and infer the best indexes
> $bestIdx = $expW | % {$_.Requests} | % { $_.BestIndex }
> $bestIdx

Name        Table    Keys Includes
----        -----    ---- --------
__PDT_I17   orders   1    1
__PDT_I18   orders   1    1
__PDT_I19   lineitem 1    3
...

> # create a new configuration with all these best indexes
> $bestC = New-Configuration -Indexes $bestIdx -Name "MyC"

> # compare this configuration with the base one for all queries in the workload
> Compare-Configurations -Workload $w -Config1 $bestC -Config2 (Get-Configuration base)
```



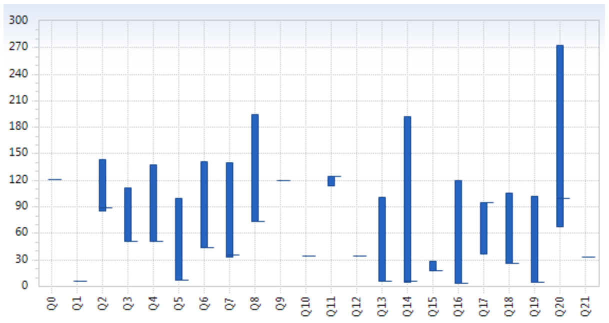
```
> # bestC surely is better, but what is its size compared to that of base?
> $bestC.size, (get-configuration base).size
3535.4453125
1234.5234375
```

Fig. 24 Interactive Physical Design Tuning Example

```
> # bestC is 2.9 times larger than base, refine it down to 2.5GB
> $refC = Refine-Configuration -Configuration $bestC -Size 2500 -Workload $w -Timeout 20
> $refC
Name     Database       Size             Cost             Indexes
----     --------       ----             ----             -------
C11      tpch1g         2454.9765625     1080.27443       27

> # show all configurations evaluated by Refine-Configuration graphically
> dir P:/tpch1g/configurations | out-chart -values size -xvalues cost -gallery scatter
```



```
> # looks like refC gives a good time/space tradeoff; compare queries in refC against all evaluated configurations
> # displaying, for each query, min/max cost and current cost under refC
> Contrast-Configuration -Workload $w -Configs (dir P:/tpch1g/configurations) -Config $refC
```



```
> # identified Q17 and Q20 as the only two that could be improved given more space
> # of these, Q17 is as bad as it gets (around 90 units)

> # Use Constraints to tune again so that no query is worse than 1.2x the cost under refC, but additionally
> # Q17 is expected to execute in fewer than 60 units. For that, try to get as close as possible to 2000MB
> $ct1 = "FOR Q in W ASSERT cost(Q,C) <= cost(Q,refC)*1.2"
> $ct2 = "ASSERT cost(W['Q17'], C) <= 60"
> $ct3 = "SOFT ASSERT size(C) = 2000"
> TuneConstrained-Workload -Workload $w -Timeout 600 -Constraints $ct1, $ct2, $ct3
> ...
```

Fig. 24 (cont.) Interactive Physical Design Tuning Example

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 [15]. 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 [22] proves that the general physical design problem is NP-Hard). Therefore, usually heuristic search methods (or *metaheuristic* algorithms) have been studied. Examples of such techniques are simulated annealing, tabu search, or evolutionary algorithms (e.g., [18, 23]).

9 Conclusion

In this work we introduced the constrained physical design problem and proposed a language that enables the specification of rich constraints. 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 work, 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. When using constraints during physical design tuning, many new opportunities are possible. While this is clearly an advantage for advanced DBAs, it is not clear how feasible it is to follow a model in which a long script with constraints is written, executed, and analyzed for deployment. In this work we claim that a paradigm shift in the way DBAs interact with physical design tools is required, where interactivity is crucial. We believe that the prototype discussed in Section 7 represents a first step in that direction, and opens up new and exciting opportunities for both research and practice.

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. Agrawal, E. Chu, and V. Narasayya. Automatic physical design tuning: workload as a sequence. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2006.
4. S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.
5. S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2001.
6. G. Brassard and P. Bratley. *Fundamental of Algorithmics.* Prentice Hall, 1996.
7. 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.
8. N. Bruno and S. Chaudhuri. Physical design refinement: The “Merge-Reduce” approach. In *International Conference on Extending Database Technology (EDBT)*, 2006.
9. 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.
10. N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2007.
11. 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.
12. 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.
13. S. Chaudhuri and V. Narasayya. Autoadmin 'What-if' index analysis utility. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1998.
14. S. Chaudhuri and V. Narasayya. Index merging. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 1999.
15. 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.
16. 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.
17. B. Duncan. Deadlock Troubleshooting (Part 3), 2006. Accessible at `http://blogs.msdn.com/bartd/archive/2006/09/-25/ deadlock-troubleshooting-part-3.aspx`.
18. 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.
19. Microsoft. Windows Powershell, 2006. Accessible at `http://www.microsoft.com/windowsserver2003/-technologies/management/powershell/default.mspx`.
20. Microsoft. How Windows Powershell Works, 2008.
21. S. Papadomanolakis and A. Ailamaki. An integer linear programming approach to database design. In *Workshop on Self-Managing Database Systems*, 2007.
22. G. P. Shapiro. The optimal selection of secondary indices is NP-Complete. In *SIGMOD Record 13(2)*, 1983.
23. 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.
24. 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.
25. 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.
26. 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.