Automatic Physical Database Tuning: A Relaxation-based Approach

Nicolas Bruno
Microsoft Research
nicolasb@microsoft.com

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

ABSTRACT

In recent years there has been considerable research on automated selection of physical design in database systems. In current solutions, candidate access paths are heuristically chosen based on the structure of each input query, and a subsequent bottom-up search is performed to identify the best overall configuration. To handle large workloads and multiple kinds of physical structures, recent techniques have become increasingly complex: they exhibit many special cases, shortcuts, and heuristics that make it very difficult to analyze and extract properties. In this paper we critically examine the architecture of current solutions. We then design a new framework for the physical design problem that significantly reduces the assumptions and heuristics used in previous approaches. While simplicity and uniformity are important contributions in themselves, we report extensive experimental results showing that our approach could result in comparable (and, in many cases, considerably better) recommendations than state-of-the-art commercial alternatives.

1. INTRODUCTION

Database systems (DBMSs) have been widely deployed in recent years and their applications have become increasingly complex and varied. Physical design tuning has therefore become more relevant than ever before, and presently database administrators spend considerable time tuning a sub-optimal installation for performance. As a consequence of this trend, automatic physical tuning became an important research problem. Most vendors nowadays offer automated tools to tune the physical design of a database as part of their products to reduce the DBMS's total cost of ownership (e.g., [1, 7, 11]). Although each vendor provides specific features and tuning options, all these tools address the following search problem.

Automatic Physical Design Problem: Given a workload W of representative queries and updates, and a space constraint B, find the set of physical structures, or *configuration*, that fits in B and results in the lowest estimated execution cost of the queries in W.

Current tools recommend physical structures such as indexes and materialized views, among others. As noted in previous work [2, 12], automating the physical design of a database is complex because (i) there is a combinatorial explosion of physical structures to consider, and (ii) these structures (e.g., indexes and materialized views) strongly interact with each other, making almost impossible to stage the problem into simpler, independent, sub-goals.

Reference [4] presented the first industrial-strength tool to address the automatic physical design problem when the structures to consider are single- and multi-column indexes. This reference introduced a number of techniques and assumptions that were incorporated in virtually all subsequent approaches. Specifically,

What-If API: Since it is not practical to materialize a candidate configuration to evaluate its impact on the input workload, the database server is extended to support *hypothetical* physical structures. These structures are not materialized, but instead are simulated inside the optimizer by adding meta-data and statistical information to the system catalogs, which is done very efficiently [5].

Dependence on the optimizer: Candidate physical structures are useful only if the optimizer exploits them (independently of how good we *know* they are). Therefore, it is not advisable to keep a separate cost model or set of assumptions while searching for the best configuration. Instead, the optimizer, along with its cost model, needs to be kept "in sync" with the tuning process.

Search Framework: Although new physical structures were later handled by evolving the first generation of tools, the search algorithm proposed in [4] stayed almost unchanged:

1. For each query in the workload, find a good set of candidate structures. This step guesses, from the query structure, which columns might be useful as index keys, or which subexpressions can make a difference if they are materialized as views. Some approaches extract this information from optimized execution plans (e.g., [4]) and others in a preprocessing step before optimizing the query (e.g., SAEFIS in [10]). All these algorithms, however, use heuristics to identify candidate structures, such as choosing columns that participate in equality or group-by predicates as index keys.
2. Augment the initial candidate set by "merging" two or more candidates together. The idea is to generate new candidate structures that, while not optimal for any given query, might simultaneously help multiple queries requiring less space [6]. Some techniques (e.g., [10]) limit merging to structures that are useful for a single query and others (e.g., [2, 12]) merge candidates across queries.
3. Search this augmented space. Some techniques use a greedy algorithm to incrementally augment valid configurations [4], and others use a variation of knapsack and subsequent random permutations [10]. All techniques are bottom-up, in the sense that they start with a valid *empty* configuration and incrementally add (or change) candidate structures until the space constraint is violated.

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA

In this paper we claim that while the *what-if API* and the *dependence on the optimizer* are good design choices, the overall search framework needs to be revised.

Consider the first step above (candidate selection). Since the search for candidates is done either before or after optimizing the query completely, we need to guess which structures are likely to be used by the optimizer. The problem is that for complex queries, the number of such structures can quickly grow to be very large. Consider a query that joins a fact table with 20 dimension tables. Each subset of dimension tables can result in a candidate view (there are over one million of them). To avoid generating such a large set of candidates, today's tools set bounds on the maximum number of structures to consider per query, and rank the candidates using heuristics. These heuristics could be off-sync with those of the optimizer (which also prunes the search space in a specific way), and therefore suboptimal choices are likely. Also, inferring candidate structures separately from the actual optimization might miss some alternatives that are visible while queries are being optimized.

Another problem with the search technique outlined above is that the merging and enumeration steps are separate. Therefore, to ensure that good solutions are not missed, we need to eagerly generate many alternatives during merging (which might clearly be useless had we performed a bit of enumeration beforehand). For reasonably sized workloads, the number of merged structures can also grow very large. Techniques make these steps scalable by imposing some constraints over which structures are merged. For instance, reference [2] restricts the merging step so that each structure in the initial set is merged at most once.

As illustrated above, attempts to add new physical structures in a scalable fashion –while maintaining the original algorithm design– introduced several special cases, shortcuts, and general complexity in the resulting algorithms. Current techniques additionally rely on the concept of frequent column- and table-subsets to rank candidate indexes and views, atomic configurations (with interaction patterns) to minimize optimization overhead, variations of greedy and knapsack search frameworks to enumerate configurations, and time-wise [1] or space-wise [11] staging to provide time-bounded solutions or minimize interaction between structures, to name a few. Our opinion is that we have reached a point in complexity that makes very difficult to analyze, evolve, and add new features to the algorithms without significant risks of regression.

In this paper we explore an alternative approach to the automatic physical design problem that addresses the difficulties described above. Our techniques are more integrated with the optimizer and exploit additional knowledge about its cost model. We base our solution on two orthogonal pillars:

- By *instrumenting* small portions of the optimizer, we eliminate the trial and error procedures currently used to identify candidates structures. Instead, we efficiently identify a small superset of physical structures that are *guaranteed* to result in an optimal configuration (usually taking too much space).
- We propose a different approach for searching the space of physical structures. Instead of starting with an empty configuration and progressively adding structures, we proceed in the opposite direction. We start with a large configuration that is time-wise optimal but too large to fit in the available storage, and progressively "shrink" it using transformations that aim to diminish the space consumed without significantly hurting the expected performance. We show that this approach has some advantages from *both* quality and performance points of view, and might also return valuable information to the database administrator about the distribution of more efficient (but larger) configurations.

We thus obtain a conceptually simple algorithm that results in recommendations that are either comparable or better than those of state-of-the-art commercial tools. Our approach significantly reduces the complexity of an important set of techniques that was becoming increasingly difficult to analyze and extend.

The rest of the paper is structured as follows. We next state the assumptions on which we rely in the rest of the paper. In Section 2 we show how to instrument a typical optimizer to obtain the *best* configuration (without constraints) for a given query. Section 3 discusses how we traverse the search space from this best (too large) configuration to a good configuration that fits in the available space. In Section 3.6 we discuss how to extend our techniques to handle workloads with update queries. Finally, Section 4 reports experimental results on a prototype implementation of our approach.

Assumptions

In this paper we focus on recommending indexes and materialized views to minimize the estimated cost of an input workload. In particular, indexes consist of a sequence of *key* columns optionally followed by a sequence of *suffix* columns[1]. Suffix columns are not present at internal nodes in the index and thus cannot be exploited for seeking (but can help queries that reference such columns in non-sargable predicates). The view language is restricted to `SPJG` queries (i.e., single block `SPJ` queries with `group-by` clauses). Predicates in the view definition are divided in conjuncts and assigned to one of three classes: *join* predicates, *range* predicates, and *other* predicates, as illustrated in the example below:

```
SELECT R.a, S.b, T.c FROM R, S, T
WHERE R.x=S.y AND S.y=T.z                 –join predicates
 AND R.a>5 AND R.a<50 AND R.b>5           –range predicates
 AND (R.a<R.b OR R.c<8) AND R.a*R.b=5     –other predicates
```

We assume that the optimizer has a unique entry point for single-relation access path selection (optimizers based on System-R [9] or Cascades [8] frameworks are usually structured in this way). In other words, there is one component responsible for finding physical index strategies (including index scans, id intersections and lookups) for single table logical sub-plans. Similarly, there is a view matching component that, once invoked with a `SPJG` subquery, returns zero or more equivalent rewritings of such query using an available view in the system.

2. INSTRUMENTING THE OPTIMIZER

During the optimization of a single query, the optimizer issues several access path requests for indexes and materialized views. For an index request over a single-table sub-plan (see Figure 2), an access path generation module first identifies the columns in sargable predicates, required sort columns, and the columns that are additionally referenced upwards in the query tree. Then, it analyzes the available indexes and returns one or more alternative physical plans that might be optimal for the input logical sub-query. In general, each generated plan is an instance of a template tree that (i) has one or more index seeks (or index scans) at the leaf nodes, (ii) combines the leaf nodes by binary intersections or unions, (iii) applies an optional rid lookup to retrieve missing columns, (iv) applies an optional filter operator for non-sargable predicates, and (v) applies an optional sort operator to enforce order. Consider an index request for the sub-query below (where τ specifies order):

$$\tau_D\left(\Pi_{D,E}\left(\sigma_{A<10 \wedge B<10 \wedge A+C=8}(R)\right)\right)$$

In this case, the optimizer identifies columns A and B in sargable predicates, column D as a required order, and columns E and C

[1]If the database system does not support suffix columns, we only consider *key* columns in indexes.

as additional columns that are referenced either by non-sargable predicates or upwards in the tree. Suppose that indexes on columns A and B are available. The optimizer can then generate the plan in Figure 1(a). However, depending on selectivity values, the alternative plan in Figure 1(b) (that avoids intersecting indexes but performs more rid lookups) can be more attractive. Also, if a covering index on columns D,A,B,C, and E is available, the alternative plan in Figure 1(c) might be preferable because it avoids sorting an intermediate result. A cost-based optimizer considers this space of alternative plans for given available indexes and returns the most efficient physical strategy. The same idea applies to view requests. In this case, the optimizer matches existing views against the input query and, if it is successful, returns an equivalent query that uses the view (subsequently, the optimizer will issue index requests on those materialized views, which are treated as base tables).

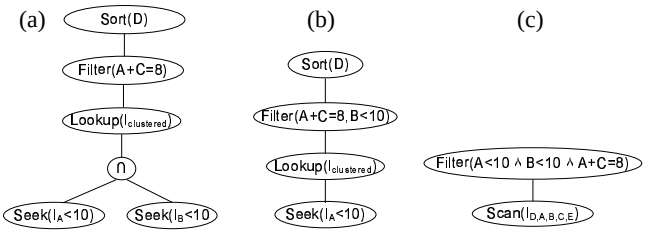


Figure 1: Alternative index strategies.

Now suppose that we instrument the optimizer as follows (see Figure 2 for the case of indexes). Each time the optimizer issues an index or view request, we suspend optimization and analyze the request[2]. That is, we consider all sargable and non-sargable predicates, order, and additional columns in the index requests, and the `SPJG` sub-queries in view requests. These requests (along with the implicit knowledge of how indexes and views are exploited) implicitly encode all possible physical structures that the optimizer might exploit[3]. After analyzing the request, we obtain the physical structures that result in the most efficient plan for such request (Section 2.1 shows how this step is achieved). We then *simulate* these hypothetical structures in the system catalogs and resume optimization. The optimizer will now consider the structures just created and obtain the "optimal" execution plan for each request.

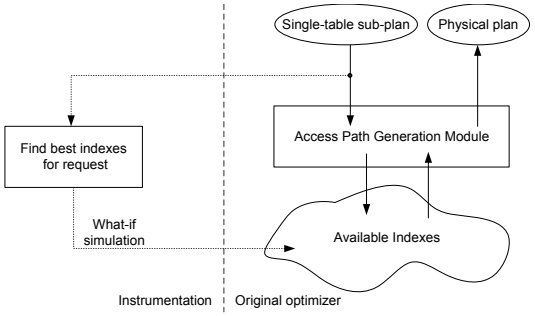


Figure 2: Instrumenting the Query Optimizer.

Since we repeat this procedure for each index or view request, the optimizer is always given the optimal set of physical structures to implement logical plans. For that reason, the execution plan returned by the optimizer would be the most efficient one over the space of all possible configurations[4]. The optimal configuration, thus, is obtained by gathering all the simulated physical structures that are generated during optimization. Table 1 shows the total number of requests for a typical 22-query TPC-H workload. We see that the number of requests (and thus the number of simulated structures) is rather small for this complex workload.

From an engineering point of view, this procedure is appealing since it is not very intrusive. In fact, the modifications required to instrument the optimizer as we describe above are restricted to two entry points within the optimizer (view matching and index strategy generation). From an algorithmic point of view, this technique does not rely on any guesswork to choose columns or subset of tables to consider in indexes and views. Since requests are intercepted *during* optimization, we do not miss candidates that might not be apparent by looking at the final execution plan (like in [4]), nor we propose many candidates that are syntactically valid but might not be exploitable during optimization (like in [10]). We next explain how to find the best physical structures for index or view requests.

2.1 Obtaining the Optimal Configuration

As explained above, the optimal configuration is obtained by gathering all the simulated physical structures generated during optimization, which essentially correspond to the union of optimal structures for each index or view request. Finding the optimal view for a view request is trivial. Since the input of a view request is an `SPJG` sub-query, the input sub-query itself is the most efficient view to satisfy the request (specifically, the best possible plan is a scan over any clustered index over such a view). For an index request, the situation is more complex. Consider an index request (S, N, O, A) where S are columns in sargable predicates, N contains subsets of columns in non-sargable predicates, O are columns in order requests, and A are other referenced columns. If no order is requested (i.e., $O = \emptyset$), the following lemma restricts the space of index sub-plans that we must consider.

LEMMA 1. *For any plan that intersects rids from two index seeks there is a more efficient plan that produces the same result by seeking one (larger) index.*

If, additionally, $|S|$=1 and N=$\emptyset$ we have the following lemma:

LEMMA 2. *For any plan that uses rid lookups over the result of an index seek, there is a more efficient plan that produces the same result by seeking one (larger) index.*

When both lemmas can be applied, we can guarantee that the optimal plan does not use index intersections nor rid lookups, and therefore it must seek a covering index with key columns S and suffix columns A. If several sargable predicates are present but N=$\emptyset$, we proceed as follows. Assuming independence between predicates, it can be easily shown that the optimal plan consists of a seek over a prefix of the columns in S sorted by selectivity, followed by an (optional) fetch. The best index can be efficiently identified by progressively including new columns from S to the index until no further benefit is obtained. In general, if the index request contains non-sargable predicates (i.e., $N \neq \emptyset$), the situation is more complex since there can be interaction between columns (i.e., a predicate `a+b>10` can be evaluated when we consider an index for other sargable predicates over columns `a` and `b`). While the main ideas remain the same (i.e., we obtain the index that results in the best plan using seeks followed by optional fetches) we omit the details for simplicity.

[2]This procedure is only active while in *tuning* mode, and is disabled during the normal execution of a production system.

[3]If no request can be answered using a candidate S, we can safely prune S from the list of candidates. Conversely, if S might be useful to answer some request, the optimizer would eventually consider a plan that uses S.

[4]This optimality claim assumes that no updates are present in the workload. Section 3.6 describes how to deal with updates.

TPC-H query #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	Total
Index Requests	4	20	11	4	14	2	13	6	8	5	3	4	4	3	3	6	8	4	6	17	8	5	158
View Requests	2	20	7	5	33	2	16	34	29	9	7	2	2	3	5	3	5	3	3	13	8	6	217

Table 1: Index and view requests for a typical TPC-H workload.

Consider now the general case of an index request (S, N, O, A) with $O \neq \emptyset$. If the optimal execution plan obtained earlier produces rows in the desired order, this is the best plan overall. Otherwise, we introduce a sort operator at the root of this plan and obtain the best plan that uses a sort. However, there might be an alternative plan that does not require sorting and is more efficient. To obtain this alternative plan, we create an index with O as its key columns. If $O \subseteq S$ we add to the index the remaining columns in S as key columns and columns in A as suffix columns. Otherwise, we add all columns in both A and S as suffix columns. Using similar arguments as before, we can show that this is the most efficient plan that does not use a sort operator. We finally compare the costs of the two alternatives (i.e., with and without a sort operator) and return the one with the minimal expected cost.

As explained earlier, if we gather the optimal set of physical structures for each request, we obtain a configuration that cannot be further improved for the given workload. The optimal configuration obtained in this way can be used in several ways. If the space taken by this configuration is below the maximum allowed and the workload contains no updates, we can return the configuration without further processing. Otherwise, we can use it to bound the benefit of the actual configuration. Consider Figure 3, which incrementally shows the best configuration found in the first 70 minutes of the execution of a commercial database tuning tool for a complex 30-query workload (the execution lasted over 3 hours, but the best configuration did not improve beyond what it is shown in the figure). If we had knowledge of the best possible configuration, we could have made the informed decision of stopping the tuning after 65 minutes, since the maximum additional improvement is small enough. Without this knowledge we have no choice but continue tuning until the end (or until we are satisfied in absolute terms with the current configuration).

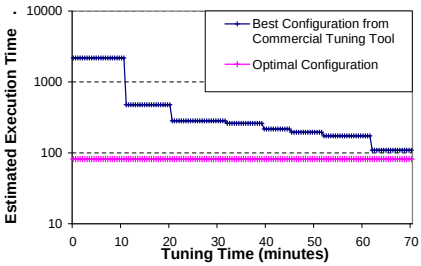


Figure 3: Bounding the improvement of the final configuration.

As we discuss in the next section, however, the main benefit of identifying the "best" configuration is that it allows us to rethink the search strategy, and in particular, move towards a *relaxation-based* approach, which, as we will see, has some additional advantages.

3. RELAXATION-BASED SEARCH

As stated earlier, all previous approaches for the automatic physical design problem tune a database and a workload by (i) identifying a set of candidate structures that are likely to speed up each workload query in isolation, (ii) extending this set by "merging" structures, and (iii) searching this extended set for a subset of structures that satisfies the space constraint and results in the largest improvement in execution cost for the queries in the workload. In all cases, steps (ii) and (iii) above are performed separately, and (iii) follows a bottom-up strategy that starts with an empty configuration and progressively adds candidate structures to the current configuration until the space constraint is no longer satisfied.

Our ability to identify the optimal configuration as described in the previous section suggests a completely different approach to search the space of configurations. Specifically, we start with an optimal configuration that might be too large to fit in the available space and progressively transform it into new configurations that consume less space (but are less efficient) than the previous one. We continue in this way until some configuration satisfies the space constraint[5]. Possible transformations to the current configuration are not restricted to just removing structures but also incorporate the notion of merging (or more generally, *relaxing*) a subset of structures. Conceptually, this approach has the following advantages over the basic search strategy:

- The analogous of *merging* and *enumeration* steps are interleaved. It is not required to obtain all merged structures before starting enumeration, but instead these can be generated lazily, on demand, when relaxing a specific configuration.
- Since a configuration is relaxed by replacing some physical structures by smaller but less efficient ones, re-optimizing a relaxed configuration to evaluate its cost is more efficient. Consider configuration C=$\{c_1, \ldots, c_n\}$ and suppose that we relax C into C' by replacing c_1 and c_2 by c_3 (e.g., an index on (a,b) and an index on (a,d) by an index on (a,b,d)). Since C' is composed of less efficient structures than C, we know that any query that did not use indexes c_1 or c_2 in configuration C would remain unchanged in C'. In other words, we only need to re-optimize queries that used some of the relaxed structures in C. In contrast, in a bottom-up strategy, adding a new index to an existing configuration requires that we re-optimize all queries that reference the index table (or resort to heuristic approximations, such as using atomic configurations [4], which introduces additional inaccuracies).
- A relaxation-based approach provides more useful information to the database administrator. Since we iteratively relax good configurations so that they use less space while performing slightly worse, at the end of the tuning process we have many alternative configurations that are more efficient than the final recommendation (using more resources). This might provide hints about the distribution of more efficient configurations to the database administrator and help taking decisions (e.g., increasing the disk storage in the current database installation). Figure 4 shows a sample tuning output of the algorithms described in this paper for a TPC-H workload tuned for indexes. Using the initial configuration (requiring 1.25GB) the workload is estimated to execute in 2,469 time-units. The optimal configuration can bring the execution cost down to 540 time-units but requires over 6GB of space. The best configuration under 1.75GB (the input constraint) is estimated to result in 1,388 time-units (a 43% improvement). The figure shows that adding up to 250MB of additional disk space can result in an additional 10% improvement (a reasonable trade-off). It also shows that having

[5]See Section 3.6 for extensions that handle updates.

more than 4GB only improves the situation by a marginal 3% and therefore is not advisable (see the steep slope in Figure 4 for configurations larger than 4GB). While this analysis can also result from running existing tools repeatedly with varying storage constraints, our approach produces the distribution of more efficient configurations as a by-product of the normal execution.

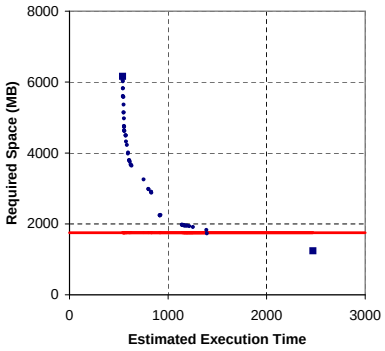


Figure 4: Relaxation-based search for a TPC-H database.

In the next section we describe the space of relaxations for a given configuration and then introduce our search algorithm.

3.1 Relaxing Configurations

As described earlier, relaxing a configuration is done by replacing a subset of its indexes or views by another so that the resulting configuration is smaller at the cost of being generally less efficient. In this way, by progressively relaxing configurations we eventually obtain one that fits in the available space and is hopefully just slightly less efficient than the initial, optimal configuration. In this section we describe the set of relaxations that we can apply to a given configuration. We designed the set of transformations by exploiting knowledge about the optimizer (such as, for instance, how indexes and views are used in execution plans). Since the transformations are transitive, we focus on those that replace one or at most two structures (in general we can apply a given transformation multiple times). Some of the transformations (e.g., index and view merging) are similar in spirit to previous work [6, 2] while others are specifically designed for our problem.

3.1.1 Index Transformations

In this section we denote an index I with a sequence of key columns K and a set of suffix columns S as $I = (K; S)$. Also, we assume that if S_1 and S_2 are sequences, the expression $S_1 \cap S_2$ (similarly $S_1 - S_2$) returns the sequence that has elements in the intersection (similarly, difference) of S_1 and S_2 in the same order that they appear in S_1. We next introduce five transformations that apply to indexes:

Merging: The concept of index merging has been proposed before [6]. In this work we define the (ordered) merging of two indexes $I_1 = (K_1; S_1)$ and $I_2 = (K_2; S_2)$ as the best index that can answer all requests that either I_1 and I_2 do, and can be efficiently sought in all cases that I_1 can (some requests that can be answered by seeking I_2 might need to scan the merged index, though). Specifically, we define the merging of I_1 and I_2 as $I_{1,2} = (K_1; (S_1 \cup K_2 \cup S_2) - K_1)$. As a minor improvement, if K_1 is a prefix of K_2, we define $I_{1,2} = (K_2; (S_1 \cup S_2) - K_2)$. For instance, merging $I_1 = ([a, b, c]; \{d, e, f\})$ and $I_2 = ([c, d, g]; \{e\})$ results in $I_{1,2} = ([a, b, c], \{d, e, f, g\})$. A configuration C that is relaxed by merging I_1 and I_2 results in the new configuration $C' = C - \{I_1, I_2\} \cup \{I_{1,2}\}$.

Splitting: This transformation aims to introduce suboptimal index intersection plans by rearranging overlapping columns of existing (wider) indexes. Consider I_1=$(K_1; S_1)$ and I_2=$(K_2; S_2)$. Splitting I_1 and I_2 produces a common index I_C and at most two additional residual indexes I_{R1} and I_{R2}. The idea is that we could replace usages of index I_1 (respectively, I_2) by a less efficient index intersection between I_C and I_{R1} (respectively, I_{R2}), or rid lookups over I_C's result if I_{R1} (respectively, I_{R2}) does not exist. Specifically, we define $I_C = (K_C$=$K_1 \cap K_2; S_C$=$S_1 \cap S_2)$ provided that K_C is non-empty (index splits are undefined if K_1 and K_2 have no common columns). In turn, if K_1 and K_C are different, $I_{R1} = (K_1 - K_C, I_1 - I_C)$, and if K_2 and K_C are different $I_{R2} = (K_2 - K_C, I_2 - I_C)$. Consider $I_1 = ([a, b, c]; \{d, e, f\})$, $I_2 = ([c, a]; \{e\})$, and $I_3 = ([a, b]; \{d, g\})$. Splitting I_1 and I_2 results in $I_C = ([a, c]; \{e\})$, $I_{R1} = ([b]; \{d, f\})$ and $I_{R2} = ([d])$. Splitting I_1 and I_3 results in $I_C = ([a, b]; \{d\})$ and $I_{R1}([c]; \{e, f\})$. A configuration C that is relaxed by splitting I_1 and I_2 results in the new configuration $C' = C - \{I_1, I_2\} \cup \{I_C, I_{R1}, I_{R2}\}$.

Prefixing: Consider $I = (K; S)$. If we take any prefix K' of K (including $K' = K$ if S is not empty) we obtain an index $I_P = (K', \emptyset)$ that can answer arbitrary requests that I does by optionally performing rid lookups to get the remaining columns $(K$-$K') \cup S$. A configuration C relaxed by prefixing index I with I_P results in the new configuration $C' = C - \{I\} \cup \{I_P\}$.

Promotion to clustered: Any index I over table T in configuration C can be promoted to a clustered index provided that C does not already have another clustered index over table T.

Removal: Finally, any index I in a configuration C can be removed to obtain a new configuration $C' = C - \{I\}$.

3.1.2 View Transformations

We denote a view V as a 6-tuple $V = (S, F, J, R, O, G)$, where S is a set of base-table or aggregate columns, F is a set of tables, J is a set of equi-join predicates, R is a set of range predicates, O is a conjunction of predicates that are not in J or R, and G is a set of base-table columns (all components except for S and F can be empty)[6]. The SQL equivalent for V is:

```
SELECT S
FROM F
WHERE J AND R AND O
GROUP BY G
```

Consider an SPJG query Q, and suppose that we want to try matching Q and view $V = (S_V, F_V, J_V, R_V, O_V, G_V)$. We first rewrite Q as a 6-tuple $Q = (S_Q, F_Q, J_Q, R_Q, O_Q, G_Q)$ and then apply a subsumption test to each pair of components. If all subsumption tests are successful, we can rewrite Q using V. Subsumption tests vary among specific systems balancing efficiency and completeness. In this paper we assume that for Q and V to match, it must be the case that $F_Q = F_V$ (the rationale is that if $F_V \subseteq F_Q$, then V would have already matched a sub-query of Q during optimization) and O_V's conjunctions are included in O_Q's (conjunct equality is structurally tested without complex rewritings, so we simply check that the predicate trees are the same modulo column equivalence). The remaining components are checked using simple inclusion tests modulo column equivalence. We next introduce the two transformations that apply to materialized views:

Merging: Similarly to indexes, merging views V_1 and V_2 is expected to result in the most specific view V_M from which all information for both V_1 and V_2 can be extracted. Specifically, we

[6]Classifying the query predicates in sets J, R, and O is done to simplify the view matching procedure.

require that V_M be matched whenever either V_1 or V_2 are. With that property in mind, we define view merging as follows. Consider V_1=$(S_1, F_1, J_1, R_1, O_1, G_1)$ and $V_2 = (S_2, F_2, J_2, R_2, O_2, G_2)$. Due to the specific view matching procedure described earlier, we require that $F_1 = F_2$ as a prerequisite for merging. We then define the merging of V_1 and V_2 as V_M=$(S_M, F_M, J_M, R_M, O_M, G_M)$, where F_M=F_1=F_2, J_M=$J_1 \cap J_2$, R_M=R_1"merge" R_2 (i.e., R_M combines same-column range predicates in R_1 and R_2), $O_M = O_1 \cap O_2$ (where the intersection uses structural equality as in the view matching algorithm), G_M=$G_1 \cup G_2$ if both G_1 and G_2 are non-empty (if either G_1 or G_2 are empty, $G_M = \emptyset$), and $S_M = S_1 \cup S_2$ if $G_M \neq \emptyset$ (if $G_M = \emptyset$, $S_M = S_1 \cup S_2 - S_A \cup S'_A$ where S_A is the set of aggregated columns in either S_1 or S_2 and S'_A is the set of base-table columns in S_A). As a minor improvement, if some range predicate in R_M becomes unbounded (e.g., after merging $R.a < 10$ and $R.a > 5$) we eliminate it altogether from R_M (if $G_M \neq \emptyset$ we add the corresponding column to both G_M and S_M so that range predicates can still be evaluated with V_M). The following example illustrates the merging procedure. Consider views V_1 and V_2 defined below:

```
V1=                          V2=
SELECT R.a, R.b              SELECT R.a, sum(R.c)
FROM R,S                     FROM R,S
WHERE R.x=S.y                WHERE R.x=S.y AND R.w=S.z
 AND 10 ≤ R.a ≤ 20            AND 15 ≤ R.a ≤ 25
 AND R.b*R.b < 10             AND 10 ≤ R.b
                              GROUP BY R.a
```

Merging V_1 and V_2 results in the following view:

```
VM=  SELECT R.a, R.b, R.c
     FROM R,S
     WHERE R.x=S.y
      AND 10 ≤ R.a ≤ 25
      AND 10 ≤ R.b
```

After views V_1 and V_2 are merged into V_M, all indexes over V_1 and V_2 are promoted to V_M. In other words, for each index $I(K;S)$ over V_1 (respectively, V_2) we create and index $I_M(K';S')$ where K' and S' consist of all columns in K and S mapped from V_1 (respectively, V_2) to V_M[7]. A configuration C that is relaxed by merging view V_1 and V_2 into V_M results in the new configuration C'=$C - \{V_1, V_2\} - I_{V_1} - I_{V_2} \cup \{V_M\} \cup I_{V_M}$, where I_{V_1}, I_{V_2} and I_{V_M} are the indexes associated, respectively, with V_1, V_2 and V_M. Note that we do not include a "split" transformation for views. The reason is that the analogous of index intersections in this case is to join simpler views to obtain the original one, but this is already handled by our model. In fact, if a view V' that is simpler than the original view V (say, V' contains fewer joined tables) could be used to answer a query, then V' should have been already requested during the initial optimization.

Removal: Any view V in a configuration C can be removed to obtain a new configuration $C' = C - \{V, I_1, \ldots, I_n\}$, where $I_1, \ldots, I_n$ are all indexes defined over V.

3.2 Search Algorithm

Having defined the set of transformations to relax a given configuration into a new configuration that is smaller but generally less efficient than the original one, we now design a generic search strategy as follows (see Figure 5). While optimizing each query q in the input workload W we intercept all index and view requests and obtain an optimal initial configuration c_{best} following the ideas in Section 2 (lines 1-2 in the figure). We then create a pool of configurations (CP) that initially consists of c_{best} (line 3) and initiate the proper search until we run out of time (lines 4 to 9). In the main search loop, we select some configuration c from the configuration pool CP (line 5) and apply some transformation to relax c into c_{new} (line 6). We add the new configuration c_{new} to the pool CP and, if c_{new} fits in the available space B and it is more efficient than the current best configuration c_{best} we keep c_{new} as the best configuration so far. When time is exceeded line 10 returns c_{best}.

```
Search_Strategy (W:workload, B:space constraint)
01 Get optimal configurations for each q ∈ W // Section 2
02 c_best = ∪_{q∈W} optimal configuration for q
03 CP = { c_best }; c_best=NULL;   // cost(NULL)=∞
04 while (time is not exceeded)
05    Pick c ∈ CP that can be relaxed   // template
06    Relax c into c_new                // template
07    CP = CP ∪ { c_new }
08    if ( size(c_new) ≤ B ∧ cost(c_new)<cost(c_best) )
09       c_best=c_new
10 return c_best
```

Figure 5: Generic physical-design search algorithm.

Figure 5 is a template algorithm because lines 5 and 6 are not fully specified. When we instantiate specific procedures to choose the next configuration (line 5) and transformation (line 6) to apply to it, we obtain a concrete search procedure. Since we keep relaxing configurations, we implicitly prune the search space of configurations. In fact, this space is not the power set of all possible physical structures, but a much more reduced one that is traversed only by transforming (e.g., merging) structures that are useful in some other configuration. Figure 6, however, illustrates that in general the search space is still extremely large. In the figure we instantiated line 5 by selecting the last relaxed configuration, and line 6 by picking an arbitrary new transformation for such configuration. The figure shows the total number of transformations that can be chosen in lines 5-6 at each iteration of the search algorithm for a 22-query TPC-H workload. We observe that each iteration introduces hundreds of new transformations (which in turn result in hundreds of new configurations). Clearly, an exhaustive search algorithm is not feasible even for small to medium workloads.

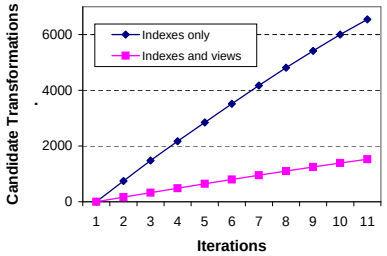


Figure 6: Candidate transformations for a TPC-H workload.

For that reason, it is crucial to develop good heuristics to guide the search strategy so that good configurations are quickly identified. In the next section we show how to estimate some useful properties of candidate transformations, and then we introduce heuristics that rely on those properties to instantiate lines 5 and 6 in the algorithm of Figure 5.

3.3 Estimating Configuration Properties

In the physical design problem, the two critical properties of any given configuration C are (i) the space C consumes, and (ii) the expected execution cost of the workload when C is available.

[7] A small number of additional columns is sometimes added to S' to allow V_M efficiently answer requests for V_1 and V_2 without performing rid lookups. We omit those details for simplicity.

Clearly, any heuristic used to choose a transformation to relax the current configuration (line 6 in Figure 5) might greatly benefit from knowledge about space consumption and expected execution cost of the resulting configuration. In fact, if we knew how much would the space consumption decrease or the expected execution cost increase after applying a given transformation, we could make a more informed decision in choosing the best alternative.

Unfortunately, there is no efficient way to precisely calculate such values in general. In fact, to calculate the space consumed by an arbitrary configuration (especially if the configuration contains materialized views) we first need to materialize all of its physical structures in the database. In turn, to obtain the expected execution cost of a given workload we need to re-optimize all queries in the workload after materializing (or simulating) the physical structures in the database. These procedures are obviously not scalable to help decide which transformation to apply for relaxing the current configuration. In the example of Figure 4, we would need to re-optimize the workload hundreds of times per iteration to obtain the expected execution cost of each relaxed configuration.

Since it is unfeasible to obtain exact quantities, in this section we show how to obtain approximate values for space consumption and expected cost. These approximations are not exact, but adequate to guide the search. Specifically, in the next section we review how to estimate the size of a given configuration without materializing it, and then we propose a technique to obtain a (tight) upper bound of the expected execution cost of a relaxed configuration without calling the optimizer.

3.3.1 Space Consumption

The space consumed by a configuration is the sum of sizes of all its physical structures. In this section we briefly describe how we estimate the space consumed by indexes and materialized views.

Consider first index $I = (K; S)$ over table T. To estimate its size we first calculate the width of an entry in any of I's leaf nodes as $W_L = \sum_{c \in K \cup S} \mathit{width}(c)$, where $\mathit{width}(c)$ is a system-dependant constant if c is a fixed-length column (e.g., integers are four bytes long), or is the average length of values of c in the database if c is a variable-length column (we approximate c's average length using sampling). Similarly, we calculate the width of an entry in an internal node of the B-Tree as $W_I = \sum_{c \in K} \mathit{width}(c)$. Using W_L and W_I we then calculate the number of entries per page in leaf (P_L) and internal (P_I) nodes of the B-Tree. Finally, we calculate the total number of pages used by I as the sum of pages per level in the B-Tree. Specifically, leaf nodes in the B-Tree fit in $S_0 = \lceil |T|/P_L \rceil$ pages and level i ($i \geq 1$) nodes in the B-Tree fit in $S_i = \lceil S_{i-1}/P_I \rceil$ pages[8].

Since materialized views are defined as regular views for which a clustered index has been implemented, obtaining the size of a materialized view is almost equivalent to the case described above. That is, the space consumed by a materialized view V is estimated as the sum of sizes of each index (including the clustered index) defined over V. We can apply the same procedure as before, with the only caveat that we do not know the value $|V|$ (while cardinality values for base tables are typically stored in the database catalogs, the cardinality of arbitrary views is not known in advance). To approximate $|V|$, we use the cardinality module of the optimizer itself to estimate the number of tuples returned by the view definition. We note that more accurate procedures can be used (e.g., using sampling for single-table views, or views that use foreign-key joins), but those procedures are not general (e.g., see [3]).

[8]The analysis is slightly more complex due to factors such as index fill factors, hidden rid columns in secondary indexes, and page overhead due to fixed- and variable-length columns, but we omit those details for simplicity.

3.3.2 Expected Execution Cost

Consider a configuration C that is relaxed to C', and suppose we want to estimate the increase in expected execution cost for the input workload when using C' instead of C. An expensive alternative consists of re-optimizing all queries in the workload using C' and calculate the difference with complete information. Due to the specific transformations that we consider (see Section 3.1), there is a more effective method that we can use. As we explained in Section 3, if some query q in the workload does not use any of the replaced structures from C, the execution plan for q under configuration C' would not change. For that reason, we only need to re-optimize the subset of queries in the workload that originally used some of the physical structures that belong to C but do not appear in C'. This technique is expected to be much more efficient, since in general there are just a few queries that use each physical structure in a configuration, and therefore the fraction of re-optimized queries is rather small.

Unfortunately, even when considering the optimization described above, this approach remains too costly. As shown in Figure 4, for a small 22-query workload there are hundreds of candidate transformations per iteration. If we were to estimate the increase in execution cost of each resulting configuration, and assuming that there is only one query per configuration to re-optimize, we would require hundreds of optimizer's calls per iteration, which becomes prohibitively expensive. In this section we take a different approach and develop techniques that allow us to (tightly) upper-bound the increase in cost for a candidate relaxed configuration without making a call to the optimizer[9]. We will then use these upper bounds as a measure of how costly a relaxed configuration might become.

Consider the execution plan P at the left of Figure 7. Index $I = ([a]; \{b, c\})$ 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 *prefixing* transformation that replaces I in the current configuration C with the alternative $I' = ([a]; \{b\})$ in the relaxed configuration C'. In general, the optimal execution plan under C' might be arbitrarily different from the original execution plan in C, and the only way to find the new plan would be to re-optimize the query. However, because of the way we defined transformations, we know that I' can answer any request that the original I did, albeit less efficiently (e.g., by introducing an additional rid lookup or sort operator, or by having to scan the whole index instead of performing a few index seeks). We can then replace the small portion of the execution plan that uses I with a small compensating plan that uses I'. This plan would be valid and therefore as least as efficient as the best plan found by the optimizer. Specifically, the alternative plan at the right of Figure 7 uses I' and additionally performs rid lookups to obtain the remaining required c column.

The example above illustrates the principle that we follow to obtain an upper-bound on the cost of executing a given query under a relaxed configuration. In short, we isolate the usage of each physical structure that is removed from the original configuration and estimate (without re-optimizing) how expensive would be to evaluate those sub-expressions using the physical structures available in the relaxed configuration. Clearly, there is a spectrum of alternatives to obtain these costs, ranging from simple estimators to complex procedures that almost mimic the optimizer itself. In this work we use a simple non-intrusive approach that can be implemented by simply analyzing information exposed by current optimizers.

[9]Note that we do not completely eliminate optimizer's calls. After a transformation is chosen and the current configuration is relaxed, we re-optimize the queries in the workload that use some of the deleted physical structures to obtain the actual configuration cost in line 8 of Figure 5.

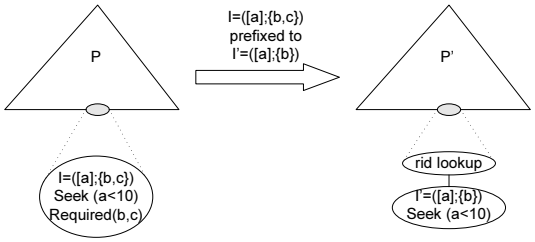


Figure 7: Estimating execution-cost upper-bounds.

Specifically, the source of information we use to obtain execution cost upper-bounds is the optimized execution plans of each query in the workload under the original configuration. We assume that we can extract from a query's execution plan the following information for each used index over a base table or materialized view: estimated I/O and CPU cost, estimated number of rows returned, type of usage (i.e., whether the index is used to seek a fraction of the rows or to scan all rows), the optional required order that is enforced on the returned rows, the optional set of columns sought or used for ordering, and the set of additional columns that are required upwards in the tree. We note that current optimizers expose this minimal information via special "explain" interfaces.

Index Transformations

Consider a transformation that removes an index $I = (K; S)$ that is used to evaluate some query under the original configuration. Suppose that we want to bound the increase in execution cost of such a plan when we replace I with a suitable use of an available index $I_R = (K_R; S_R)$ (e.g., I_R can be the merged index in the case of a merge transformation). We now describe such procedure.

First suppose that index I was completely scanned in the original execution plan. We then estimate the cost of scanning the alternative I_R as $cost(I) \cdot size(I_R)/size(I)$ (i.e., we linearly scale the cost based on the number of pages accessed in each scan). Then, we check whether all columns provided by I in the original execution plan can be provided by I_R. If that is not the case, we add the cost of performing $rows(I)$ rid lookups, where $rows(I)$ is the estimated number of rows returned by I. Finally, if the result produced by I was required to be sorted and the order columns are not compatible with those in I_R (i.e., key columns K and K_R do not share the same prefix) we add the cost of an intermediate sort operator. In that way, we obtain the cost of an alternative plan that uses I_R to return the same result that I did under the original configuration.

If, instead, I was sought in the execution plan under the original configuration, we proceed as follows. We first identify the columns in K that were used to seek I (usually just one column) and the selectivity s_I of the predicates that were used in the seek. Then, we identify the longest column prefix in K_R that has the same columns as the corresponding prefix in K and obtain the selectivity s_{I_R} of the corresponding sargable predicates (if no columns are shared, the selectivity is one and I_R has to be scanned completely). Using a similar argument as in the previous case, we first estimate the cost of using I_R as $cost(I) \cdot (s_{I_R} \cdot size(I_R))/(s_I \cdot size(I))$ and then perform the tests for rid lookups and sort operators to obtain the total cost of using I_R in the relaxed configuration.

In the case of split transformations, we independently apply the previous procedure to the common and residual indexes and then include the cost of intersecting rids. We omit the details of this case for brevity, but the main ideas remain the same.

View Transformations

When we merge views V_1 and V_2 into V_M, we additionally promote all indexes over V_1 and V_2 to V_M (see Section 3.1). Also, the view merging transformation has the property that the optimizer would match V_M whenever it matches either V_1 or V_2 (optionally using additional filters or group-by operators). We therefore bound the increase in cost due to a view-merge transformation in two steps. First we calculate, using the techniques in the previous section, the increase in cost for each index on V_1 or V_2 that is used to answer some query under the original configuration. Then, we estimate the cost of any compensating operation (e.g., group-by operators) that need to be inserted on top of V_M to obtain the results that either V_1 or V_2 produce. Finally, we obtain an upper-bound of the cost of the query under the new configuration as the sum of all these partial quantities. While the main concepts are clear, there are a number of subtleties that need to be taken into account. We illustrate some of these below, but we omit the full details for simplicity. Consider the following views:

```
V1= SELECT R.a, R.b      V2= SELECT R.a, sum(R.c)
    FROM R                   FROM R
    WHERE R.a<10             WHERE R.a<20 AND R.b<20
                             GROUP BY R.a

                              SELECT R.a, R.b, R.c
 which are merged into VM =   FROM R
                              WHERE R.a<20
```

and suppose that query Q seeks an index I_1=($[b, a]$) on V_1 for some range predicate on column b. The corresponding index on V_M additionally contains tuples that satisfy $10 \leq R.a < 20$. When bounding the cost of evaluating the same sub-plan with the promoted index on V_M, the expected fraction of tuples retrieved from the index does not change since we assume independence, but we need to add the cost of a compensating filter for predicate $R.a < 10$. On the other hand, if the index sought is originally I_2=($[a, b]$) on V_1, the total number of tuples touched in the corresponding index on V_M stays the same (and therefore the fraction of tuples changes) since the leading key-column in the index is precisely $R.a$. Finally, queries that use some index on V_2 need to add the cost of a final group-by operator after the index sub-plan because the merged view V_M removed the grouping clause on column $R.a$. We note that we reuse components in the optimizer (e.g., costing or cardinality estimation modules) to implement these steps, so we do not keep the optimizer out of the loop by creating our own parallel estimators.

For transformations that delete views, the situation is more complex. The problem is that we do not know how to replace a sub-plan that uses the removed view without calling the optimizer. An inexpensive approach to address this problem is as follows. Each time we consider a new view V, we optimize V with respect to the base configuration (i.e., the configuration that contains only indexes that enforce constraints and must be present in any configuration) and obtain CB_V, the cost to obtain V in the base configuration. To estimate an upper bound of the cost of each query that used V in the original configuration we first calculate, for each index on V used in the query, the increase in cost when the replacement index is a heap. Then, we add CB_V to this partial cost and obtain the final upper bound. In other words, the implied plan is one that first obtains V (in no particular order) and then replaces each index usage by a scan over V. We can use a more accurate procedure than estimating simply CB_V values. Every time we obtain a new configuration C, we estimate the cost to obtain each view V that is used in C with respect to the smaller configuration $C - \{V\}$. We apply the same

ideas discussed earlier to avoid unnecessary optimization calls: if V was optimized in the configuration that was relaxed to obtain C and V's plan contains all indexes that are still present in C, we can assume that the optimizer would find the same plan for V and avoid re-optimizing it. Since the number of views per configuration is not very large, we obtain more accurate estimates than CB_V with a small overhead.

3.4 Heuristics for Guiding the Search

In this section we propose specific implementations of lines 5 and 6 of Figure 5 and therefore fully specify our algorithm to automatically tune the physical design of a database system.

Consider first line 6, which chooses some transformation to apply to the current configuration C. Using the techniques in the previous sections, we can efficiently estimate, for each valid transformation tr that relaxes C into C_{tr}, the expected decrease in storage space (denoted ΔS_{tr} = *Space(C)-Space(C_{tr})*) and the maximum increase in cost (denoted ΔT_{tr} = *CostBound(C_{tr}) - Cost(C)*). The value *penalty*$_{tr}$=$\Delta T_{tr}/\Delta S_{tr}$ estimates the increase in execution cost per unit of storage that each transformation is expected to return. Increasing *penalty*$_{tr}$ values seems a reasonable heuristic to rank possible transformations, since we are interested in relaxed configurations that are significantly smaller in space without increasing the expected cost too much (this heuristic is also used in greedy approximations to the knapsack problem). We introduce a small variation on the definition of *penalty*$_{tr}$ values as follows. Suppose that the space constraint is B (i.e., we are interested in the best configuration that fits in B). Any decrease in space beyond B is not strictly useful but we might artificially decrease the *penalty*$_{tr}$ value of transformations that significantly decrease the space below B. For that reason, we refine the penalty function as follows:

$$penalty_{tr} = \frac{\Delta T_{tr}}{\min(\textit{Space}(C) - B, \Delta S_{tr})}$$

Line 6 in Figure 5 becomes:

```
06   Relax c into c_new using the transformation tr
         that minimizes penalty_tr
```

We note that in this step we need to evaluate the penalty of each transformation in the current configuration. While at first this might seem expensive, we do not need to re-optimize queries to evaluate penalty values, and we can also cache results from one iteration to the next, so the amortized number of transformations that we evaluate per iteration is rather small and can be done very efficiently.

The only missing piece in the algorithm is a procedure to choose which configuration to relax at each iteration (line 5 in Figure 5). A reasonable alternative is to choose, at each iteration, the configuration with the minimal estimated cost. This way, we always work on the current most efficient configuration. While this alternative is interesting, it is also impractical. Usually, the most efficient configurations are the ones that require the largest amount of storage, and therefore the time to converge to a configuration that is under the required space constraint is too long. Instead, we select the next configuration to relax as follows:

1. If the last relaxed configuration does not fit in the available space, we choose and further relax it. In this way, we keep relaxing the same configuration until we reach one that is under the space constraint. Using the greedy approach, we usually find a good-quality configuration quickly, but we might miss better alternatives. If there is more time available after we reach a valid configuration, we use the next heuristic.
2. We obtain the chain of relaxed configurations from the last one (that fits in the available space) to the initial (optimal) configuration. We then pick the configuration that resulted in the actual largest penalty when relaxed (with the aim of "correcting" what went wrong in the previous iteration).
3. If there is no candidate in the current chain of configurations with at least one valid transformation, we choose the configuration with the minimum expected cost that additionally has at least one available transformation.

As we show experimentally in Section 4, applying these heuristics to the physical design problem results in high-quality recommendations in relatively short amounts of time.

3.5 Variations and Optimizations

In this section we briefly mention some minor optimizations and variations to the main algorithm described earlier.

Shortcut evaluation: When evaluating the cost of a relaxed configuration C_{tr}, we might reach a point in which the cost of a subset of queries in C_{tr} is larger than the total cost of the current best configuration C_{best}. In this case, we know that neither C_{tr} nor any configuration that is further relaxed from C_{tr} would be more efficient than C_{best}. Therefore, we can (i) stop evaluating C_{tr} (thus saving optimization calls), and (ii) remove C_{tr} from the pool of candidate transformations `CP` (thus pruning the search space).

Multiple transformations per iteration: In our current algorithm we apply a single transformation to relax the current configuration. In general, we might apply more than one transformation. We need to be careful that we do not select conflicting transformations (such as merging I_1 and I_2 after removing I_1). This alternative might reduce the overall time to arrive to a valid transformation, but introduces additional inaccuracies because often transformations strongly interact with each other.

Shrinking configurations: Another variation consists of removing, at each iteration, all indexes and views from the current configuration that are not used to evaluate any query in the workload. While this approach would reduce the search space because fewer transformations are available, it might also decrease the quality of the final recommendation, since some structures that are not used in the current configuration might become useful after applying some transformation.

3.6 Handling Updates

So far we have exclusively focused on workloads that only query the database without updating it (i.e., we assume that no `UPDATE`, `INSERT`, or `DELETE` queries are present in the workload). In reality, most workloads are composed of a mixture of "select" and "update" queries, and any physical design tool must take into consideration both classes to be useful. The main impact of an update query is that some (of all) indexes defined over the query's updated table must also be updated as a side effect. Therefore, it is not true anymore that adding more indexes would always reduce the expected cost of a workload. In the rest of this section we explain how the different components in our approach change when updates are present in the workload. (While the core concepts stay the same, we chose to stage the presentation in this way for simplicity.)

Evaluating Configurations

An important goal in our techniques is to minimize the number of optimization calls, which are the most expensive component of our algorithms. The main approach we use for that purpose is based

on the optimality principle of the optimizer. Since we always relax a configuration C into a less efficient C', a query under C that uses indexes which are not removed in C' does not need to be re-optimized under C'. This approach, while still correct, can become inefficient when update queries are present. The reason is that each update query implicitly references all indexes (or some of them, in the case of `update` queries) over its referenced table. Therefore, it is more likely that any transformation affect some index used in an update query, and many more re-optimizations are likely to occur. To mitigate this effect, we separate each update query into two components: a pure select query, and a small update shell. For instance, the following query:

```
UPDATE R SET a=b+1, c=c*c+5 WHERE a<10 AND d<20
```

is separated into (i) a pure select query and (ii) an update shell:

```
(i)  SELECT b+1, c*c+5 FROM R WHERE a<10 and d<20
(ii) UPDATE TOP(k) R SET a=0, c=0
```

where k is the estimated cardinality of the corresponding select query. We now can optimize each component separately. Specifically, we calculate, for each index I and update-query q in the workload, the cost of updating I for the update shell of q. Later, when evaluating a configuration, we use the select portion of update queries (therefore avoiding many additional optimization calls) and then add to the resulting partial cost the update cost of all indexes in the configuration over the table referenced by the update query.

Optimal Configuration

In section 2.1 we showed how to obtain a configuration that cannot be improved, which was then the starting point of our relaxation-based approach. Such configuration gives the database user more information about an ongoing tuning session, since it bounds how efficient a configuration can get. When updates are present, the configuration obtained in this way is not optimal anymore because indexes also need to be updated, raising the overall cost of the workload. However, the resulting configuration is still optimal for the select component of each update query, and we can use this fact to obtain a lower bound. Specifically, the execution cost for the select portion of the queries in the workload (see above) is added to the cost of all the update shells under the base configuration (which contains all the indexes that must be present in any configuration). We then obtain a cost that cannot be improved by any configuration. The main difference is that this bound is not tight (i.e., there might be no configuration that meets the lower bound) but can anyway offer valuable additional information to the user while tuning a complex workload. We implicitly consider update costs as another variable in the optimization problem that moves in the opposite direction of select costs (like configuration sizes).

Choosing configurations

In Section 3.4 we showed how we pick the next configuration to relax. Specifically, we keep relaxing the current configuration until it fits in the available space. If the workload contains updates, it is advisable to continue relaxing the current configuration even beyond that point, because removing indexes that result in expensive updates might further decrease the cost of the relaxed configuration. We therefore change the first heuristic of Section 3.4 as follows:

1. If the last relaxed configuration does not fit in the available space *or its cost is smaller than the configuration it was relaxed from*, we choose and further relax it.

Transformation Penalty

For workloads with update queries, the cost upper-bound of a relaxed configuration can be negative (sometimes the cost of removing some index can be outweighed by the benefit of not having to update it). In those cases the penalty function correctly chooses transformations with negative over positive cost upper-bounds, but sometimes makes poor decisions comparing two transformations with negative costs. If $\Delta T_{t1} = -10$, $\Delta S_{t1} = 10$, $\Delta T_{t2} = -20$, and $\Delta S_{t2} = 30$, the penalty for t_1 (i.e., -1), would be smaller than the penalty for t_2 (i.e., -2/3). However, the configuration relaxed using t_2 is clearly better than the one relaxed from t_1 both in terms of space and cost (we say that t_2 dominates t_1). To remedy those situations, we first obtain the skyline of transformations (i.e., we consider only transformations that are not dominated by any other transformation) and then use the original *penalty* definition over this restricted subset.

Additionally, the denominator in the definition of *penalty* is given by $\min(\textit{Space}(C) - B, \Delta S)$. Since now we can relax a configuration that requires less than B storage (see above), the denominator might become negative, which is undesirable. However, in those situations space is not relevant since the configuration already fits in B. Therefore, when the current configuration requires less than B storage, we simply use ΔT_{tr} as the penalty associated with tr.

Name	Size	Tables	Average cols/table	# Workloads; min/max size
TPC-H (Synthetic)	1.25 GB	8	7.6	26 (1/24)
DR1 (Real)	2.9 GB	116	6.9	30 (1/30)
DR2 (Real)	13.4 GB	34	8.5	11 (1/10)
DS1 (Synthetic)	700 MB	26	58.2	57 (1/96)
Bench (Synthetic)	530 MB	6	21.0	163 (1/144)
DS2 (Synthetic)	170 MB	16	19.1	259 (1/160)

Table 2: Databases and workloads used in the experiments.

4. EXPERIMENTAL EVALUATION

In this section we report an extensive experimental evaluation of our proposed technique over both synthetic and real databases, with respect to hundreds of different workloads (see Table 2 for a summary of the experimental setting). We implemented the client component of our prototype in C++ and modified Microsoft SQL Server to support our extensions of Section 2. We note that the complete implementation of our prototype was finished in less than two months by a single developer, and the resulting prototype was robust enough to handle virtually all input workloads. The aim of this section is to show that our approach, which is based on sound principles and relies on few assumptions, results in high-quality recommendations and often even outperforms state-of-the-art tools available in commercial database systems.

In the rest of this section we use PTT (Prototype Tuning Tool) to denote our prototype implementation, and CTT (Commercial Tuning Tool) to denote the commercial state-of-the-art alternative in Microsoft SQL Server we compare PTT against. The metric to evaluate a physical recommendation is *improvement*, defined as:

$$improvement(C_I, C_R, W) = 100\% \cdot \left(1 - \frac{cost(W, C_R)}{cost(W, C_I)}\right)$$

where C_I is the initial configuration, C_R is the recommended configuration, and $cost(W, C)$ is the expected cost of evaluating all queries in the input workload W under configuration C. Improvement values can be negative (when the recommended configuration is less efficient than the initial one due to stricter space constraints), but always are smaller than 100%.

4.1 No Constraints

We first validate the approach of Section 2 (i.e., the ability to intercept index and view requests during optimization and generate

optimal configurations for workloads without updates). For that purpose, we obtained physical recommendations for each available SELECT-only workload using PTT and CTT (we did not impose any time or space constraint on CTT to obtain the best alternative). We report experimental results in Figure 8. Specifically, each bar in the figures represents one tuned workload and its magnitude measures $\Delta Improvement = Improvement_{PTT} - Improvement_{CTT}$. (Values of $\Delta Improvement$ greater than zero represent cases in which PTT obtained better recommendations than CTT.) We observe that:

1. There is a large number of cases (around 64%) for which both PTT and CTT return the same recommendation (or recommendations within 1% of quality, which is attributed to small inaccuracies in the various cost models used by both the server and the tools, or small statistical fluctuations when creating histograms using sampling).
2. There is a important number of cases (around 34%) for which PTT returns better-quality recommendations than CTT (up to 60% additional improvement when recommending indexes and even above 95% when also recommending views). It is interesting to note that a small fraction of these cases helped uncover design and implementation problems in CTT that were not found during regular testing.
3. There is a very small number of cases (less than 2%) for which PTT returns a configuration that is worse than that of CTT. We examined these workloads in detail, and found that they belong in two classes. On one hand, there were several instances for which the optimizer made suboptimal choices (for instance, returning a worse plan when additional indexes or views were added to a configuration) and therefore were violations of our assumptions. On the other hand, there were a small number of cases (around 5 globally) that required special handling by our techniques. This is expected as the complexity of data sets and workloads is very high, but we chose not to implement any special case in our prototype to evaluate a clean and minimalist approach.
4. In general, the largest differences appear when recommending views in addition to indexes. The reason is, in addition to view recommendation being inherently more complex, that CTT has more shortcuts and special cases to handle views.

Table 3 shows the top-10 workloads that required the most time to be tuned with CTT along with the time it took PTT to obtain the optimal configuration and the improvements of the respective recommendations (we included at most two workloads per database). Clearly, when no space constraints are present our techniques are very efficient, since the starting point is already the desired goal. In contrast, CTT spends considerable time in the merge and enumeration phases. In the next section, we show that a relaxation-based approach is also a better alternative in presence of updates or when space constraints are reasonably loose.

Workload	$Time_{CTT}$	$Time_{PTT}$	$Impr_{CTT}$	$Impr_{PTT}$
Bench-1-IV	774'39"	43'02"	99.12%	99.49%
DR2-1-IV	552'25"	14'12"	79.82%	98.44%
DR2-2-I	73'13"	5'15"	79.96%	80.06%
DR1-1-IV	67'19"	1'21"	42.83%	54.36%
DR1-2-IV	66'49"	1'13"	67.64%	81.99%
TPCH-1-IV	57'43"	9'27"	84.46%	97.04%
Bench-2-IV	34'03"	2'03"	79.46%	80.00%
TPCH-1-I	15'30"	2'14"	79.10%	79.27%
DS2-1-IV	13'24"	1'43"	93.77%	94.28%
DS1-2-IV	7'48"	2'11"	98.49%	99.87%

Table 3: Tuning time for the most expensive workloads.

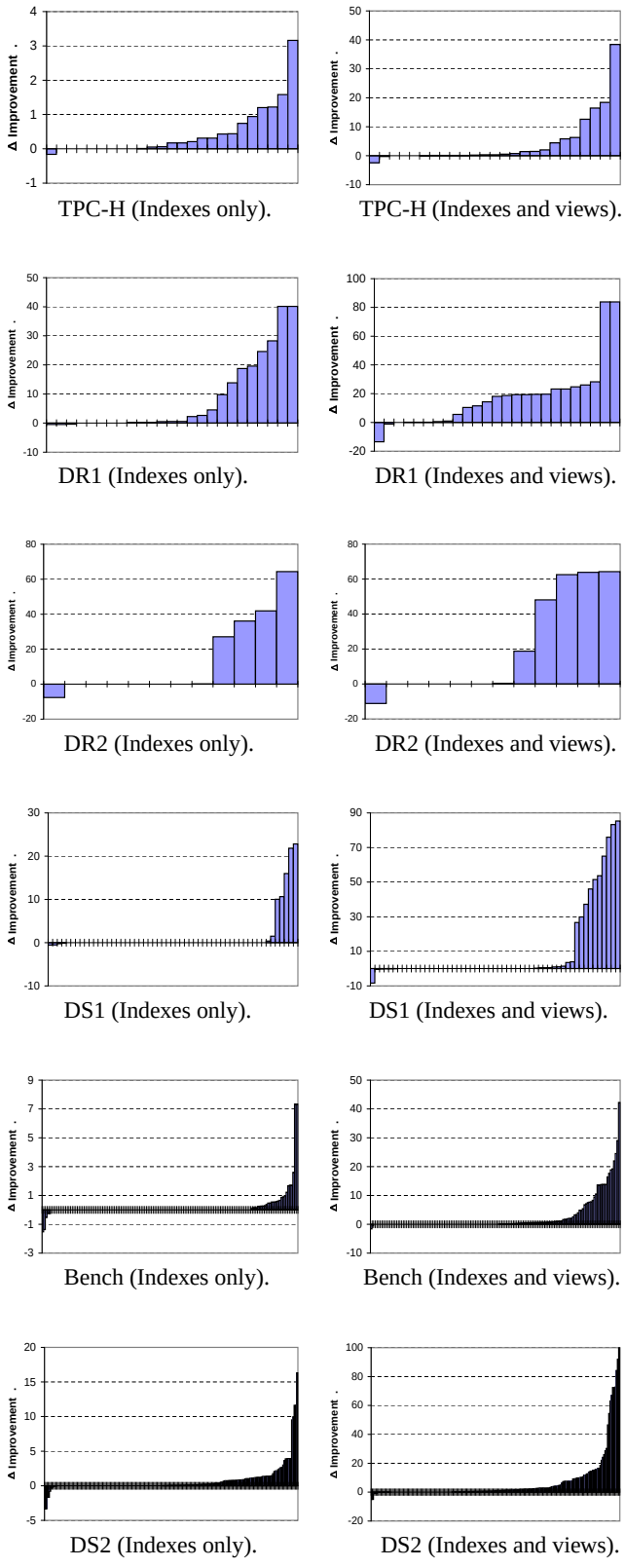
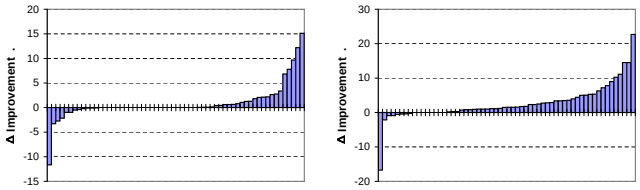


Figure 8: Quality of recommendations when using PTT and CTT for varying databases and workloads.

4.2 Space and Update Constraints

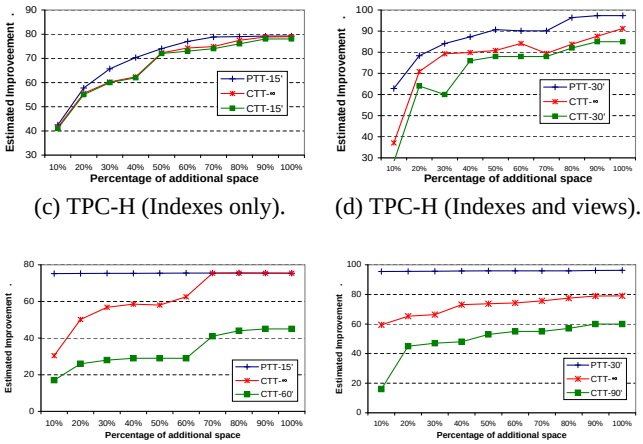
In this section we report experimental results for constrained versions of the problem. Specifically, we consider input workloads with `UPDATE` queries (which impose overheads to each recommended index or view), and tuning sessions with storage constraints. Figures 9(a-b) show Δ*Improvement* values for all these workloads (we used both real workloads with updates and synthetically generated ones, such as those obtained with `dbgen` for TPC-H databases). While we imposed no time-bounds for CTT, we gave PTT 15 and 30 minutes, respectively, for index-only and indexes-and-views recommendations. As we can see (specially when recommending indexes and views), a large percentage of workloads (83%) resulted in equal or better recommendations when using PTT. In the remaining 17% of the cases, either the optimizer violated our assumptions (see previous section) or the search strategy in PTT failed to obtain the better recommendation in the allotted time. We note, however, that with only one exception, PTT resulted in at most 5% degradation with respect to CTT for these workloads.



(a) Indexes only. (b) Indexes and views.

Figure 9: Quality of recommendations for `UPDATE` workloads.

Figure 10 shows the quality of recommendations when varying the storage constraints. For these experiments, we first used PTT to obtain the optimal, largest, configuration (in our experiments, the sizes of the optimal configurations were between 1.8 and 6 times the sizes of the databases themselves). We then defined the space taken by such configuration as 100% and the space taken by the minimum configuration (even smaller than the existing configuration) as 0%. Finally, we varied the space constraint between these two extreme values and tuned each workload using PTT and CTT. We can see that even when we do not impose a time bound to CTT, the recommendations obtained by PTT are of better quality. Also, our search strategy guarantees that the more space is available, the better the quality of the recommendations. Due to multiple heuristics and greedy approximation, CTT might recommend worse configurations when slightly more space is available.



(c) TPC-H (Indexes only). (d) TPC-H (Indexes and views).

(d) DR2 (Indexes only). (f) DR2 (Indexes and views).

Figure 10: Quality of recommendations with space constraints.

5. SUMMARY

Motivated by the increasing complexity of current physical design tools, we proposed a new architecture for the physical design problem that is based on sound principles and geared towards avoiding guesswork. Our technique is conceptually simpler than the current alternatives and our preliminary experiments indicate its potential to improve the quality of recommendations and the time needed to arrive to such recommendations compared to state-of-the-art techniques.

6. 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 30th International Conference on Very Large Databases*, 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*, 2000.

[3] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *Proceedings of the ACM International Conference on Management of Data*, 1999.

[4] S. Chaudhuri and V. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *Proceedings of the 23rd International Conference on Very Large Databases*, 1997.

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

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

[7] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic SQL Tuning in Oracle 10g. In *Proceedings of the 30th International Conference on Very Large Databases*, 2004.

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

[9] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM International Conference on Management of Data*, 1979.

[10] 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*, 2000.

[11] 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 30th International Conference on Very Large Databases*, 2004.

[12] 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.