To Tune or not to Tune? A Lightweight Physical Design Alerter

Nicolas Bruno
Microsoft Research
nicolasb@microsoft.com

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

ABSTRACT

In recent years there has been considerable research on automating the physical design in database systems. Current techniques provide good recommendations, but are resource intensive. This makes DBAs somewhat conservative when deciding to launch a resource-intensive tuning session. In this paper, we introduce an *alerter* that helps determining when a physical design tool should be invoked. The alerter is a lightweight mechanism that provides guaranteed lower (and upper bounds) on the improvement that a DBA could expect by invoking a comprehensive physical design tool. Moreover, it produces an accompanying recommendation that serves as a "proof" for the lower bound. We show experimentally that the alerter handles large workloads with little overhead, and help judiciously decide on launching subsequent tuning sessions.

1. INTRODUCTION

Database management systems (DBMSs) support varied and complex applications. As a consequence, physical design tuning has become more relevant than ever before. Database administrators (DBAs) presently spend considerable time either tuning a suboptimal installation for performance or maintaining a well-tuned installation over time. Most vendors nowadays offer automated tools to tune the physical design of a database as part of their products (e.g., [1, 8, 14]). Although each solution provides specific features and options, all the tools address the following common problem:

Physical Design Problem: Given a query workload W and a storage budget B, find the set of physical structures (or configuration), that fits in B and results in the lowest estimated execution cost of W.

Current tools recommend various physical structures such as indexes and materialized views, among others. As noted in previous work [2, 15], 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 strongly interact with each other. State-of-the-art techniques provide good recommendations, but are resource intensive. It is common for tuning sessions to run for a long time before returning a useful recommendation.

For this reason, DBAs usually face the following dilemma. On one hand, changes in workloads and data distributions *might* result in the current configuration becoming suboptimal. DBAs therefore would like to periodically run a tuning tool that recommends changes (if any) to the current configuration. On the other hand, unless the current configuration *is* suboptimal, no changes would be necessary and the significant load on the server due to the tuning session is wasted. Worse still, currently the only way to determine whether a tuning session would be worthwhile is to actually run it!

Therefore, it is attractive to investigate if we can determine whether the current configuration is suboptimal a-priori, i.e., *before* running an expensive tuning tool. In this paper we present a technique, which we henceforth call *alerter*, that answers this question. The alerter analyzes a workload and determines whether a comprehensive tuning session would result in a configuration significantly better than the current one. It has the following characteristics:

- *Low-overhead diagnostics:* The alerter might be called repeatedly, whenever the DBA suspects that changes might be necessary, or at fixed time intervals. For efficiency purposes, the alerter only works with the information that was gathered when the workload was originally optimized and does not rely on additional optimizer calls.
- *Reliable lower bound improvement:* When the alerter reports that certain improvement is possible, we can be certain (under broad assumptions) that the improvement achieved by a comprehensive tuning tool would indeed be at least as large[1]. This is crucial since false positives would defeat the alerting mechanism and are therefore unacceptable.
- *Upper bound improvement:* Depending on the amount of overhead that we are willing to tolerate during query optimization, the alerter produces different levels of tightness on upper bounds for improvement values. This effectively reduces the chances of false negatives, by bounding the best possible outcome of a comprehensive tuning tool.

[1]The alerter outputs a valid configuration as a proof of the lower bound. We can always implement this configuration if it is more attractive than the best one found by the comprehensive tuning tool, therefore guaranteeing the lower bound.

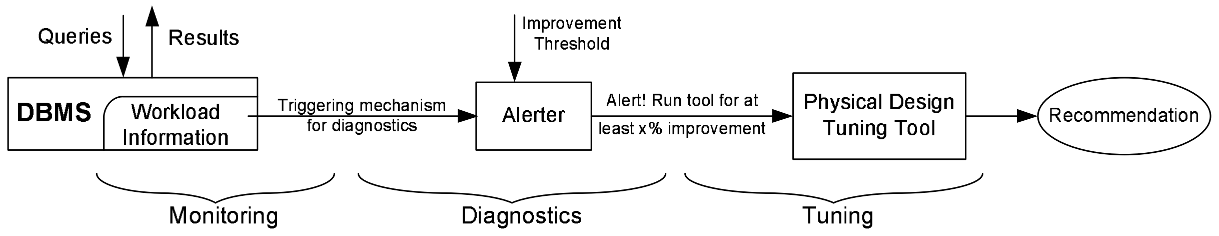


Figure 1: Monitor-Diagnose-Tune cycle for the physical design problem.

Figure 1 illustrates how the alerter fits into the *"monitor-diagnose-tune"* cycle for physical design tuning. As new queries are optimized and executed, the DBMS internally keeps information about the workload that would later be consumed by the alerter. After a triggering condition happens, the alerter is launched automatically and diagnoses the situation quickly. We do not take a position on the triggering mechanism (it could be a fixed amount of time, an excessive number of recompilations, or perhaps significant database updates) but we believe that such triggering events happen frequently enough that make it prohibitive to run a comprehensive tuning tool after each triggering condition. After the lightweight diagnostics, if the alerter determines that running a comprehensive tuning tool would result in an improvement beyond a certain pre-specified threshold, the DBA is alerted and urged to proceed. As part of the alert, the DBA gets lower and upper bounds for the improvement that would result from a comprehensive tuning session, as well as a valid configuration that serves as a proof of the lower bounds. Thus, we believe that this alerting mechanism fills a gap in the automatic physical design tuning cycle by ensuring that a subsequent physical tuning session is appropriate.

The rest of the paper is structured as follows. In Section 2 we explain a low-overhead mechanism to gather information during the normal operation of a database system. Sections 3 and 4 discuss how to exploit this information to obtain lower and upper bounds on the improvement in execution time of the workload being analyzed. In Section 5 we discuss extensions to the basic techniques. We report experimental results in Section 6 and discuss related work in Section 7.

2. INSTRUMENTING THE OPTIMIZER

An alerting mechanism must be extremely lightweight to be effective. Unlike comprehensive tools, we cannot issue optimizer calls when the alerter is launched due to the overhead that this would impose at runtime. Instead, our approach is to instrument the query optimizer itself so that it gathers additional information during normal optimization of queries[2]. Later, when the alerter is launched, we exploit all this precomputed information without calling the optimizer again. This model allows any workload model (such as a moving window, a subset of the most expensive queries, or just a sample) to be fed to the alerter without changes. We now briefly review how a query optimizer chooses access paths to implement logical sub-queries and then explain the information that we gather during optimization.

[2]This information can be maintained in memory and accessed programmatically [10], and also periodically persisted in a workload repository [8].

2.1 Review of Access Path Selection

In this work we assume that the optimizer has a unique entry point for access path selection (optimizers based on System-R [11] or Cascades [9] frameworks are usually structured in this way). Specifically, there is one component in the optimizer responsible for finding physical index strategies (including index scans, rid intersections and lookups) for logical sub-plans. During the optimization of a single query, the optimizer issues several *access path requests* (henceforth called *index requests*, or simply requests) for different sub-queries. For each request (see Figure 2), an access path generation module first identifies the columns that occur in *sargable* predicates, the columns that are part of a sort requirement, and the columns that are additionally referenced in complex predicates or upwards in the query tree. Then, it analyzes the available indexes and returns one or more candidate physical plans for the input sub-query.

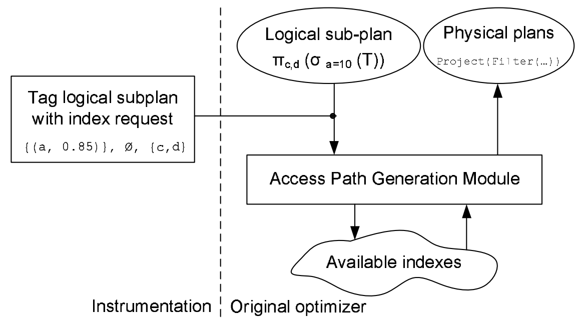


Figure 2: Access path selection and request tagging.

Consider for instance an index request for the query fragment below (where τ specifies an order by clause):

$$\tau_b\big(\Pi_c(\sigma_{a=10}(T))\big)$$

In this case, the optimizer identifies column a in a sargable predicate, column b as a required order, and column c as an additional column that is either output or referenced upwards in the tree. This information allows the optimizer to identify the indexes that might be helpful to implement an efficient sub-plan for the sub-query. Suppose that an index on column $T.a$ is available. The optimizer then generates a plan that uses an index seek on such index to retrieve all tuples satisfying $T.a$=10, fetches columns $T.b$ and $T.c$ from a primary index, and finally sorts the resulting tuples by $T.b$. Depending on the cardinality of $T.a$=10, an index on columns $(T.b, T.a, T.c)$ might be more attractive. Scanning this index (in b order) and filtering on the fly the tuples that satisfy $T.a$=10 might be more efficient because it avoids sorting an intermediate result. A cost-based optimizer considers these alternative plans and returns the most efficient physical access plan with respect to the available indexes.

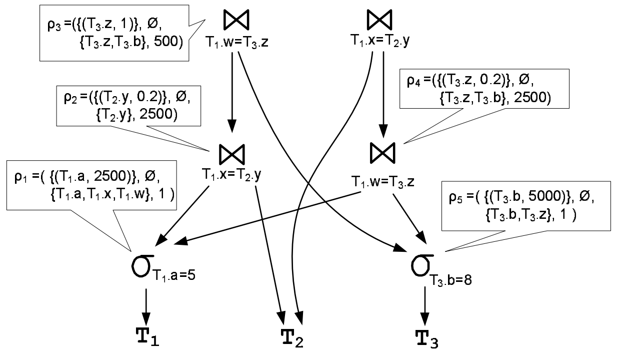
Note that this approach is also used to generate index-nested-loops plans (which implement joins between an arbitrary outer relation and a single-table inner relation that is repeatedly accessed using an index to obtain join matches). In this case, the access path generation module works with the inner table only, and the joined column in the table is considered as part of a sargable (equality) predicate. For instance, suppose that the logical sub-plan is $(\mathcal{Q} \bowtie_{\mathcal{Q}.x=T.y} T)$, where $\mathcal{Q}$ represents an arbitrary complex expression that returns the outer relation in the index-nested-loop join. Conceptually, the optimizer passes to the access path selection module the single-table expression $\sigma_{T.y=?}(T)$, and proceeds as usual, considering $T.y$ a column in a sargable predicate with an (unspecified) constant value.

2.2 Intercepting Index Requests

To gather information at optimization time, we instrument the optimizer by augmenting the technique in [3] (see Figure 2). During plan generation, each time the optimizer issues an index request we obtain the information that is relevant to such request and store it at the root of the originating logical plan. Specifically, for each index request we store the tuple (S, O, A, N), where S is the set of columns in sargable predicates and the predicate cardinalities, O is the sequence of columns for which an order has been requested, A is the set of additional columns used upwards in the execution plan, and N is the number of times the sub-plan would be executed (N is greater than one only if the sub-plan is the inner portion of an index-nested-loop join)[3]. Intuitively, each request encodes the properties of any index strategy that might implement the sub-tree rooted at the corresponding logical operator (or its right sub-tree in the case of requests originating from joins). This fact will later allow us to make inferences about changes in the physical design without issuing additional optimization calls.

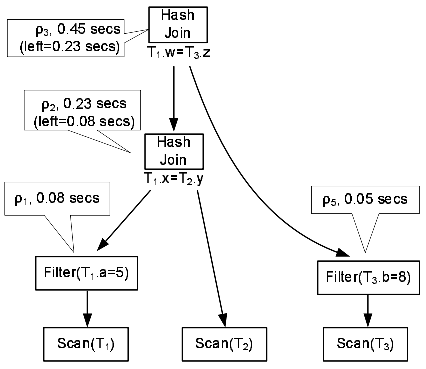
Figure 3(a) illustrates this procedure for a three-way join query. In the figure we can see a fragment of the search space that the optimizer considers for the query (only logical alternatives are shown in the figure). Along with some of the logical operators we show the index requests that were generated. As an example, request ρ_1 is attached to the selection condition on table T_1 and specifies that (i) there is one sargable column $T_1.a$ returning 2500 tuples, (ii) there is no order requested, (iii) the columns that are required are $T_1.a$, $T_1.w$ and $T_1.x$, and (iv) the sub-plan would be executed once at runtime. Similarly, request ρ_2 was intercepted when the optimizer tried to generate an index-nested-loop alternative with T_1 and T_2 as the outer and inner relations, respectively. Request ρ_2 specifies that $T_2.y$ is a sargable column which would be sought with 2500 bindings and produce 500 rows overall. Therefore, the average number of tuples matched and returned from T_2 is 0.2 per binding (hence the 0.2 cardinality value for ρ_2). There is no request for the logical join at the top-right node in Figure 3(a) because an index-nested-loop join requires the inner table (the right operand in the figure) to be a base table.

In contrast to the techniques in [3], we gather additional information on a subset of the above index requests that would later help us obtain improvement bounds. Specifically, after plan generation we traverse the resulting execution plan and associate each physical operator p with the index request (if any) that corresponds to the logical sub-tree that was implemented by p. We denote such requests the *winning* requests, because they are associated with the operators in the optimal plan found by the optimizer. Figure 3(b) illustrates this step, where the winning requests are $\{\rho_1, \rho_2, \rho_3, \rho_5\}$. For instance, ρ_1 is associated with the filter operator over table T_1 because ρ_1 was attached to the logical sub-tree $\sigma_{T_1.a=5}(T_1)$. We note that a request ρ_i is not necessarily associated with the physical operator implemented from ρ_i. For instance, ρ_2 is associated with the hash join operator, even though originally ρ_2 was generated in the context of an index-nested-loop alternative. Since at this stage optimization is finished, we augment the winning requests with the cost of the execution sub-plan rooted at the corresponding physical operator (see Figure 3(b)). If

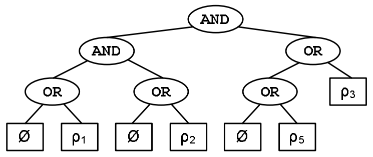


ρ3 =({(T3.z, 1)}, Ø, {T3.z,T3.b}, 500)
T1.w=T3.z
T1.x=T2.y
ρ4 =({(T3.z, 0.2)}, Ø, {T3.z,T3.b}, 2500)
ρ2 =({(T2.y, 0.2)}, Ø, {T2.y}, 2500)
ρ1 =({(T1.a, 2500)}, Ø, {T1.a,T1.x,T1.w}, 1)
T1.x=T2.y
T1.w=T3.z
ρ5 =({(T3.b, 5000)}, Ø, {T3.b,T3.z}, 1)
σ T1.a=5
σ T3.b=8
T1
T2
T3

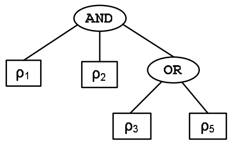
(a) Fragment of the search space.



(b) Final execution plan.



(c) Original AND/OR request tree.



(d) Normalized AND/OR request tree.

Figure 3: Gathering information during optimization.

[3]We additionally store the request table, the final cardinality of the request, and the type of sargable predicate for each element in S, but omit such details to simplify the presentation.

```
BuildAndOrTree($\mathcal{T}$:Execution Plan) =
   IF ($\mathcal{T}$.isLeaf) // Case 1
      RETURN $\mathcal{T}$.request
   ELSE IF ($\mathcal{T}$.request is null) // Case 2
      RETURN AND( BuildAndOrTree($\mathcal{T}$.child$_1$),
                  ...,
                  BuildAndOrTree($\mathcal{T}$.child$_n$) )
   ELSE IF ($\mathcal{T}$.isJoin) // Case 3
      RETURN AND( BuildAndOrTree($\mathcal{T}$.leftChild),
                  OR( $\mathcal{T}$.request,
                      BuildAndOrTree($\mathcal{T}$.rightChild)))
   ELSE // Case 4
      RETURN OR( $\mathcal{T}$.request,
                 BuildAndOrTree($\mathcal{T}$.child))
```

Figure 4: Generating the `AND/OR` request tree.

the request is associated with a join operator, such as ρ_2 in the figure, we additionally store the cost of its left sub-plan (in these situations, the left sub-plan would be the same for hash-join and index-nested-loop alternative plans, so we implicitly store the "remaining" cost of the whole sub-plan without counting the common left sub-plan).

It is important to note that some winning requests might conflict with each other. For instance, requests ρ_3 and ρ_5 in Figure 3(b) are mutually exclusive. In other words, if a plan implements ρ_3 (that is, contains an index-nested-loop join with T_3 as the inner table), it could not simultaneously implement ρ_5. As another example, request ρ_5 in Figure 3(b) would conflict with a request $\rho_6=(\emptyset, \emptyset, \{T_3.b, T_3.z\}, 1)$ rooted at the `Scan(T3)` operator (we do not show ρ_6 in Figure 3 to keep the example simple). The reason is that any execution plan uses one access path for each table in the query. Therefore we can implement either ρ_6 (by scanning some index on T_3 and filtering $T_3.b = 8$ on the fly) or ρ_5 (by directly seeking the valid tuples in T_3), but not both.

To explicitly represent these relationships, we encode the winning requests in an `AND/OR` tree, where internal nodes indicate whether the respective sub-trees can be satisfied simultaneously (`AND`) or are mutually exclusive (`OR`)[4]. The `AND/OR` tree is built by traversing the execution plan in post-order. A recursive functional specification of this procedure for an input execution plan $\mathcal{T}$ is given in Figure 4. Intuitively, if $\mathcal{T}$ is a single node, we return a simple `AND/OR` tree with the request (if any) of such node (Case 1 in Figure 4). Otherwise, if $\mathcal{T}$'s root node has no requests, we `AND` together the trees generated for each of $\mathcal{T}$'s execution sub-plans, since these are orthogonal (Case 2). Otherwise, if the root of $\mathcal{T}$ has a request, the answer depends on the type of node. If it is a join, its request ρ corresponds to an attempted index-nested-loop alternative. We know that ρ and the requests on $\mathcal{T}$'s right sub-plan are mutually exclusive (see ρ_3 and ρ_5 in Figure 3(b)). However, these requests are orthogonal to the requests in $\mathcal{T}$'s left sub-plan, and thus we return the `AND/OR` tree of Case 3. Finally, if the root $\mathcal{T}$ is not a join node, the request ρ conflicts with any request in a sub-plan of $\mathcal{T}$ (we cannot implement both alternatives) and therefore we return the `AND/OR` tree of Case 4.

Figure 3(c) shows the resulting `AND/OR` tree for the winning requests of Figure 3(b). As a final step, we normalize the `AND/OR` tree so that it contains no empty requests or unary intermediate nodes, and strictly interleaves `AND` and `OR` nodes (by possibly introducing *n-ary* internal nodes). In our example (see Figure 3(d)), the normalized tree consists of an `AND` root node whose children are either base requests, or simple `OR` nodes. Due to the specific nature of execution plans and requests, this is true in general, as we prove next.

[4]We use `AND/OR` trees based on the common interpretation in the context of memo structures and query optimization, but not in the strict logical sense.

PROPERTY 1. *The normalized* `AND/OR` *request tree for an input query is either (i) a single request, (ii) a simple* `OR` *root whose children are requests, or (iii) an* `AND` *root whose children are either requests or simple* `OR` *nodes.*

Proof (Sketch): By structural induction on the specification of *BuildAndOrTree*. For a single-node execution sub-plan (Case 1), *BuildAndOrTree* returns a single request, which satisfies (i). Suppose that input sub-plan is a join tree with root $\mathcal{T}$ and sub-trees $\mathcal{T}_L$ and $\mathcal{T}_R$, and $\mathcal{T}$ has a request (Case 3). Then, it must be the case that $\mathcal{T}_R$ is either a base table or a selection on a base table (this is a condition for the index-nested-loop join alternative that generated the request on $\mathcal{T}$). Therefore, the recursive call for $\mathcal{T}_R$ would use Cases 1 or 4, returning either a single request (i) or a simple `OR` node (ii) (i.e., no `AND` nodes). By inductive hypothesis, the recursive call for $\mathcal{T}_L$ satisfies (i), (ii) or (iii). We can then verify case by case that the resulting trees can again be normalized to either (i), (ii) or (iii). A similar procedure can be used to verify Cases 2 and 4. ∎

Since requests for different queries are orthogonal and can be satisfied simultaneously, we combine the `AND/OR` request trees of a given workload by using an `AND` root node. Normalizing this combined tree, we obtain, for an arbitrary input workload, an `AND/OR` request tree that satisfies Property 1. This normalized tree would be used by the alerter to make inferences about the workload in the presence of physical design changes *without issuing additional optimization calls* (which is crucial for performance).

3. LOWER BOUNDS FOR IMPROVEMENT

We now describe how to obtain a lower bound on the improvement that would be obtained by running a comprehensive tuning tool. Of course, if we just ran the comprehensive tuning tool, we would directly obtain the best possible configuration, and therefore the actual improvement. But this is precisely what we are trying to avoid (i.e., running an expensive tool without some assurance that it would be beneficial). Our goal is to very efficiently obtain a (relatively tight) lower bound on the workload improvement. The improvement of a configuration is defined as $100\% \cdot (1 - cost_{after}/cost_{current})$, where $cost_{current}$ and $cost_{after}$ are the estimated costs of the workload for the original and recommended configurations, respectively. The larger the improvement value, the more attractive is the recommended configuration. A lower bound on the improvement is equivalent to an upper bound on $cost_{after}$ (because $cost_{current}$ is a constant value). We then need to provide (*without* calling the optimizer) an upper bound on the cost of each query in the workload over all permissible physical design configurations.

3.1 Locally Transforming Plans

In this section we explain how we can use the `AND/OR` tree generated when the workload was originally optimized (see Section 2) to make inferences about query execution plans for varying physical designs, and do so without making additional optimizer calls.

As explained in the previous section, each request encodes the requirements of any index strategy that might implement the sub-tree rooted at the corresponding operator (or its right sub-tree for the case of join operators). Additionally, each request reports the cost of the best execution plan found by the optimizer to implement the logical sub-query associated with the request. As an example, consider ρ_1=$(\{(T_1.a, 2500)\}, \emptyset, \{T_1.a, T_1.x, T_1.w\}, 1)$ and a cost of 0.08 time units in Figure 3(b). This information implies that any index-based sub-plan rooted at the corresponding `Filter` operator would need to seek 2500 tuples from table T_1 using an equality predicate on column $T_1.a$, and return a projection of these tuples (in no particular order) on attributes $T_1.a$, $T_1.x$ and $T_1.w$. It also specifies that the best execution plan found by the optimizer using the original configuration is estimated to execute in 0.08 time units. Similarly, ρ_2 specifies that the corresponding sub-tree needs to join 2500 tuples on column $T_2.y$ from its left sub-tree with 0.2 tuples (on average) on table T_2. The cost of the best execution plan found by the optimizer for this sub-tree (without the left sub-tree cost) is 0.23-0.08=0.15 time units.

The examples above suggest that if we produce any physical sub-plan p that implements a given request ρ, we can *locally* replace with p the original physical sub-plan associated with ρ, and the resulting plan would be valid and equivalent to the original one. A sub-plan implements ρ=(S, O, A, N) if it returns the columns in A sorted by O and filtered by the predicates over columns in S for as many bindings as N specifies (if applicable). If we calculate the cost of the new sub-plan p, we can obtain the difference in cost between the original and new index strategies (i.e., how much would the original execution plan improve or degrade if we replaced the given sub-tree with a equivalent one).

Of course, if we keep the configuration fixed, no alternative would result in a more efficient plan since, by definition, the optimizer returns the most efficient overall execution plan. Suppose, however, that we create a new index in the database and would like to predict how this index would affect the execution plans of queries in the workload. We could re-optimize the queries under the new configuration and obtain a precise answer, but that would be too expensive. Instead, we can check whether the newly added index can implement some request ρ more efficiently that what the optimizer found originally, and in this way decrease the overall cost of the corresponding query (Section 3.2.1 explains in detail how this procedure works).

Note that the best we can do by following this approach is to obtain a *locally-optimal* execution plan. That is, we replace the physical sub-plans associated to each winning request in the original plan with alternatives that are as efficient as possible. We would not be able to, say, obtain a plan with different join orders, or other complex transformation rules that optimizers apply during plan generation. In that sense, we are giving up some opportunities to obtain the globally optimal execution plan but avoid expensive optimization calls and are able to ensure low overhead. The cost of the plan that we obtain by local changes is therefore an upper bound of the global optimal plan that the optimizer would find under the new configuration.

3.2 The Alerter Main Algorithm

The core idea of our lower-bound technique is to iterate over different configurations, generating valid alternative execution plans for the input workload by means of local changes as described above. For any explored configuration, we calculate an upper bound on the cost of the workload under such configuration, which in turn results in a lower bound on the improvement that could be obtained by a comprehensive tuning tool.

3.2.1 Impact of a Hypothetical Configuration

A crucial component in our techniques is the ability to calculate the difference in cost of the workload when we make a local change in a query execution plan by implementing a given request differently from what was originally optimized. Consider a request ρ=(S, O, A, N). Suppose that we want to calculate the cost of an alternative sub-plan that uses an index I over columns $(c_1, \ldots, c_n)$ to implement ρ. Let I_ρ be the longest prefix $(c_1, \ldots, c_k)$ that appears in S with an equality predicate, optionally followed by c_{k+1} if c_{k+1} appears in S with an inequality predicate. We then implement ρ by (i) seeking I with the predicates associated with columns in I_ρ, (ii) adding a filter operator with the remaining predicates in S that can be answered with all columns in I, (iii) adding a primary index lookup to retrieve the missing columns if $S \cup O \cup A \not\subseteq \{c_1, \ldots, c_n\}$, (iv) adding a second filter operator with the remaining predicates in S, and (v) adding an optional `sort` operator if O is not satisfied by the index strategy. As a simple example, consider ρ_1 in Figure 3(b) and $I_1 = (T_1.a, T_1.x)$. The execution sub-plan obtained as explained above is an index seek over I_1 returning 2500 rows, followed by 2500 primary index lookups to retrieve the missing column $T_1.w$. If instead we consider the same request and index $I_2 = (T_1.x, T_1.w, T_1.a)$ the resulting plan consists of an index scan over I_2 (retrieving all tuples) followed by a filter for column $T_1.a$ that results in 2500 rows.

Note that for costing purposes, we only need a *skeleton plan* with physical operators and cardinality values at each node. In fact, we do not need to execute the resulting plan, so for our purposes the exact predicates associated with the requests are not needed (e.g., $T_1.a = 5$ for request ρ_1). The `AND/OR` request tree therefore provides the minimal information required to obtain cost differences. We can use the optimizer's cost model effectively over the skeleton plan to estimate its execution cost (let us denote this cost C^ρ_I). If the original cost of the sub-plan associated with the request ρ was C^ρ_{orig}, we define $\Delta^\rho_I = C^\rho_{orig} - C^\rho_I$. Then, Δ^ρ_I is the local difference in cost if we implement ρ with an index strategy based on I rather than the one used originally by the optimizer. Note that Δ^ρ_I need not be positive; a bad choice of I can result in a sub-plan that is more expensive than the one originally obtained by the optimizer.

Multiple Indexes, Multiple Requests

A configuration generally contains multiple indexes defined over the table of a given request. In principle, we could use more than one index to obtain a physical sub-plan that implements a request (e.g., by using index intersections). As a design choice, we rule out those alternatives since they would increase the processing time of the alerter with modest gains in quality. We then calculate the difference in cost by implementing a request ρ with the best index strategy from a configuration $\mathcal{C}$ as $\Delta^\rho_\mathcal{C} = \min_{I \in \mathcal{C}} \Delta^\rho_I$ (if I and ρ are defined over different tables, we define Δ^ρ_I=∞).

In general, the workload is encoded as a `AND/OR` request tree and `OR` nodes rule out multiple simultaneous requests in a query plan. The difference in cost for an AND/OR request tree and a configuration $\mathcal{C}$ is defined inductively as[5]:

$$\Delta_{\mathcal{C}}^{\mathcal{T}} = \begin{cases} \Delta_{\mathcal{C}}^{request(\mathcal{T})} & \text{if } \mathcal{T} \text{ is a leaf node} \\ \sum_i \Delta_{\mathcal{C}}^{\mathcal{T}.child_i} & \text{if } \mathcal{T} \text{ is an AND node} \\ \min_i \Delta_{\mathcal{C}}^{\mathcal{T}.child_i} & \text{if } \mathcal{T} \text{ is an OR node} \end{cases}$$

The value $\Delta_{\mathcal{C}}^{\mathcal{T}}$ is therefore the difference in the workload execution cost between $\mathcal{C}$ and the original configuration. We note again that $\Delta_{\mathcal{C}}^{\mathcal{T}}$ values are lower bounds on such difference, since we obtain feasible (perhaps suboptimal) plans that the optimizer would find for $\mathcal{C}$.

3.2.2 Initializing the Search Strategy

The alerter efficiently searches a space of configurations for one (or some) that fits in the available space and is as efficient as possible. Similarly to the work in [3, 4], we perform a relaxation-based approach that starts with the best locally optimal configuration and progressively relaxes it into smaller and less efficient ones. The initial configuration is obtained as the union of the indexes that implement the best strategy for each request in the AND/OR request tree.

Consider a request ρ=(S, O, A, N) where each element in S contains a column, a predicate type (i.e., equality or inequality), and the cardinality of the predicate. We obtain the index that leads to the most efficient implementation of ρ (i.e., the *best index* for ρ) as follows:

1. Obtain the best "seek-index" I_{seek} consisting of (i) all columns in S with equality predicates, (ii) the remaining columns in S in descending cardinality order, and (iii) the columns in $(O \cup A) - S$. (Note that if the DBMS supports suffix columns [3], only columns in (i) and the first column in (ii) are key columns and the rest are suffix columns.)
2. Obtain the best "sort-index" I_{sort} with (i) all columns in S with single equality predicates (they would not change the overall sort order), (ii) the columns in O, and (iii) the remaining columns in $S \cup A$.
3. Return $minarg_{I \in \{I_{seek}, I_{sort}\}} \Delta_I^{\mathcal{T}}$.

As an example, consider request ρ_3 in Figure 3. This request has no sort columns, so the best overall index is in this case the "seek-index" $(T_3.z, T_3.b)$ obtained as described above. We repeat this procedure for each index request and combine the resulting indexes into the initial configuration $\mathcal{C}_0$=$\{(T_1.a, T_1.x, T_1.w), (T_2.y), (T_3.z, T_3.b), (T_3.b, T_3.z)\}$. We can guarantee that this configuration results in the most efficient locally optimal execution plans among all possible configurations because each request is implemented as efficiently as possible. At the same time, this configuration is usually very large. The reason is that each request is associated with a very specific index. It is likely that many indexes in $\mathcal{C}_0$ are unique (i.e., they implement optimally a single request) and therefore the size of the locally optimal configuration $\mathcal{C}_0$ tends to be rather large.

3.2.3 Relaxing Configurations

Once we obtain the initial, locally optimal configuration, we gradually relax it to obtain alternative ones that might be more attractive from a cost-benefit point of view. Specifically, we transform each configuration into another one that is smaller but less efficient. Since the alerter needs to be very fast, we made the following design choices for this exploratory step:

1. Use *index-deletion* and *index-merging* [3, 4, 7] as the only transformations (we do not consider index *reductions* [4] since they significantly increase the search space while marginally decreasing execution costs)[6].
2. Perform a greedy search, in which we move from one configuration to the next one using the *locally* most promising transformation.

The concept of index merging has been proposed before as a way to eliminate redundancy in a configuration without losing significant efficiency during query processing [4, 7]. We define the (ordered) merging of two indexes I_1 and I_2 as 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 (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 a new index that contains all the columns of I_1 followed by those in I_2 that are not in I_1. For example, merging I_1=(a, b, c) and I_2=(a, d, c) is $I_{1,2}$=(a, b, c, d). We note that index merging is an asymmetric operation (i.e., in general merge$(I_1,I_2) \neq$ merge(I_2,I_1)), so we need to consider both cases.

When transforming a configuration $\mathcal{C}$ we have many alternatives. We can delete each index in $\mathcal{C}$, or we can merge any pair of indexes defined over the same table. To rank the transformations, we use the *penalty* of transforming a configuration $\mathcal{C}$ into $\mathcal{C}'$ by an index deletion or index merge (see [3]). Penalty values measure the increase in execution cost per unit of storage that we save in $\mathcal{C}'$ compared to $\mathcal{C}$. For a AND/OR request tree $\mathcal{T}$,

$$penalty(\mathcal{C}, \mathcal{C}') = \frac{\Delta_{\mathcal{C}}^{\mathcal{T}} - \Delta_{\mathcal{C}'}^{\mathcal{T}}}{size(C) - size(C')}$$

3.2.4 Putting all Together

Figure 5 shows a pseudo-code of the main algorithm for the Alerter. Recall from Figure 1 that during normal operation, the DBMS gathers relevant information about the execution plans that are processed. This information is consolidated in the form of an AND/OR request tree. When a prespecified triggering event happens, the alerter is launched. The inputs to the alerter are the AND/OR request tree, space bounds B_{min} and B_{max} that are acceptable for a new configuration, and the minimum percentage improvement P that we consider important enough to be alerted. We first obtain the locally optimal configuration $\mathcal{C}_0$ in line 2 (see Section 3.2.2). We then progressively transform the current configuration until the resulting size is below the minimum storage constraint or the expected improvement is below the minimum we deem necessary for an alert (lines 3-7). At each step, we choose the transformation TR (index merging or deletion) with the smallest penalty value and create a new configuration (lines 5-6). After we exit the main loop,

[5] Note that we overload the definition of Δ depending on the input parameters (e.g., requests or AND/OR request trees), but the usage is clear from the context.

[6] These transformations tend to generate wide indexes, which are commonly used in decision support systems. In other scenarios (e.g., OLTP systems), these indexes might be too expensive to maintain, and thus we should also consider index reductions [4] to generate narrow indexes.

```
Alerter ($T$:AND/OR request tree,
         $B_{min}, B_{max}$:storage constraints,
         $P$:minimum percentage improvement)
1  $\mathcal{R} = \emptyset$; i=0
2  Obtain locally optimal configuration $\mathcal{C}_0$
3  while (size($\mathcal{C}_i$)>$B_{min}$ and
          $100\% \cdot \Delta^{\mathcal{T}}_{\mathcal{C}_i} / cost_{current} > P$)
4     if (size($\mathcal{C}_i$)<$B_{max}$)
         $\mathcal{R} = \mathcal{R} \cup \mathcal{C}_i$
5     Pick transformation TR that minimizes
         penalty($\mathcal{C}_i$, TR($\mathcal{C}_i$))
6     $\mathcal{C}_{i+1}$ = TR($\mathcal{C}_i$)
7     i=i+1
8  if ($\mathcal{R} \neq \emptyset$)
      ALERT($\mathcal{R}$)
```

Figure 5: Pseudo-code for the Alerter.

in line 8 we check whether some configuration satisfies all the constraints, and in such a case, we issue an alert. The alert contains the list of all configurations (C) that fit in the available space (i.e., $B_{min} \leq size(\mathcal{C}) \leq B_{max}$) and are estimated to have at least P improvement. The DBA can then analyze the alert and proceed as appropriate.

4. UPPER BOUNDS FOR IMPROVEMENT

In the previous section we explained how the alerter obtains a lower bound on the improvement that a comprehensive tuning tool would recommend. Although the alerter avoids false positives which would waste valuable resources, we still face the problem of false negatives. In other words, there could be cases in which our techniques do not alert the DBA because the expected improvement is not significant. This can be due to a lower bound that is not sufficiently tight, and therefore we might miss a good tuning opportunity. To mitigate these situations, in this section we explain how the alerter can also generate upper bounds on the potential improvement. This additional information can help DBAs refine policies to trigger tuning sessions (e.g., alert me if the minimum improvement is 25% or there is potential for 75% improvement). We next present a fast mechanism that produces upper bounds for improvements and requires almost no changes to the query optimizer. Then, we discuss an alternative approach that produces tighter bounds but incurs more overhead during optimization.

4.1 Fast Upper Bounds

In Section 2 we discussed how the optimizer issues multiple index requests during the processing of each query. These requests in turn are used to generate index strategies which are weighted and combined in the final execution plan. While we do not know what index strategy will be used under the best possible configuration, we are certain that, for each table in the query, *some* request would be implemented in the final execution plan[7]. We now extend the approach of Section 2 as follows. In addition to returning the *winning* requests (i.e., those requests that are associated with the final execution plan), we generate a list of the remaining candidate requests considered during query optimization. We return this information grouped by the table over which each request is defined. Then, for each table in the query, we compute the cost of the best index strategy for each request as explained in Section 3.2.2. Finally, we keep the most efficient alternative for each table and add all the estimated costs together. Note that this is *necessary* work that *any* execution plan would have to perform for the given query and therefore to a lower bound of the query itself.

Consider again the example in Figure 3(a) (recall that it represents only a portion of the search space, but the general ideas still apply). The requests grouped by table are $T_1 \rightarrow \{\rho_1\}$, $T_2 \rightarrow \{\rho_2\}$, and $T_3 \rightarrow \{\rho_3, \rho_4, \rho_5\}$. We do not know whether the execution plan for the best possible configuration implements ρ_3 or ρ_4 for table T_3 (performing a nested-loop-join with T_1 or $T_1 \bowtie T_2$, respectively, as the outer relation) or even ρ_5 (seeking tuples for which $T_3.b = 8$). We know, however, that *some* of these alternatives would be necessary. We then calculate a lower bound on the cost of the query as the sum of costs of the sub-plan that implements ρ_1, the sub-plan that implements ρ_2, and the most efficient alternative among the sub-plans that implement ρ_3, ρ_4 and ρ_5. We thus obtain a lower bound on the cost of any execution plan for the input query, and therefore an upper bound on any recommended improvement.

This procedure is very efficient since it simply outputs additional information used during optimization with minimal additional computation. However, by definition, it returns a loose upper bound. One reason is that we only consider necessary work for the leaf nodes in the execution plans, but we do not assign any cost to intermediate nodes such as joins or aggregates (we chose not to exploit these additional sources of information to keep the overhead low). Additionally, we do not pay attention to storage constraints, which often restrict the space of feasible configurations and therefore diminish the best possible improvement.

4.2 Tighter Upper Bounds

We now describe a technique that requires additional overhead during optimization but produces tighter upper bounds by extending the interception mechanism used in [3]. The idea is to intercept all index requests as we do in Figure 2, but instead of just tagging the logical operators with the requests, we (i) suspend optimization and analyze the request, (ii) calculate the index that result in the most efficient plan for such request, (iii) simulate the index in the system catalogs (see [6]), and (iv) resume optimization so that the newly simulated index is picked. We repeat this procedure with each index request so the optimizer gets the best indexes to implement each logical plan.

As explained in [3], this procedure returns the optimal execution plan for each input query over the space of all possible configurations. In other words, assuming that no storage constraint is given, we obtain the tightest possible upper bound for the improvement of each optimized query, which is precisely our goal. Unfortunately, the plan obtained as described above is not necessarily executable, since it might refer to hypothetical indexes. For that reason, we would require a second optimization of the query, this time using only the existing indexes. In this way, after two optimization calls we obtain *both* the best hypothetical plan when all possible indexes are present *and* the best "executable" plan that only uses available indexes. Since optimizing each query twice during normal operation is expensive, we next show how to interleave the two optimization calls and simultaneously obtain both execution plans with less overhead (see Section 6 for experimental results).

[7]This step assumes that the query is normalized before optimization and redundant tables are eliminated.

For that purpose, we modify the optimizer by adding a new sub-plan property, which we call *feasibility.* A sub-plan is *feasible* if it does not refer to hypothetical indexes in any of its physical operators. Additionally, we extend the access path generation module so that *after* generating all the traditional index strategies, it produces a new candidate with the best hypothetical index. This last index strategy (by definition) is the most efficient alternative, so in a normal situation we would immediately discard the remaining index strategies as being suboptimal. Instead, we exploit the *feasible* property analogously to interesting orders in a traditional System-R optimizer or the notion of required properties in Cascades-based optimizers. The net effect is that we maintain suboptimal, feasible plans. As a consequence of applying this strategy, at the end of query optimization we obtain both the best *feasible* and *overall* plans, which correspond, respectively, to the best execution plans when no hypothetical indexes are present, and when all possible hypothetical indexes are available. The best feasible plan is the same that would have been obtained by a traditional optimizer, so we use it for query execution. The best overall plan correspond to the optimal execution plan over all possible configuration, and we use it to obtain an upper bound on the improvement of a comprehensive tool.

In summary, we introduced two techniques to obtain upper bounds on the improvement of a comprehensive tool. These approaches trade off efficiency during with tightness of the bound. In Section 6 we contrast these alternatives.

5. EXTENSIONS TO THE BASIC MODEL

We now discuss some extensions to the techniques described earlier that take into account important factors such as query updates and materialized views.

5.1 Update Queries

So far we implicitly focused on discussing workloads composed entirely of `SELECT` queries. In reality, most workloads consist of a mixture of select and update queries, and 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 updated table must also be updated as a side effect.

Similarly to [3], we *conceptually* separate each update query into two components: a pure select query (which we process as before), and a small update shell (which we process separately). For instance, the following query:

```
UPDATE T SET a=b+1, c=c*2 WHERE a<10 AND d<20
```

is seen as (i) a pure query and (ii) an update shell:

```
(i)  SELECT b+1, c*2 FROM T WHERE a<10 and d<20
(ii) UPDATE TOP(k) T SET a=a, c=c
```

where k is the estimated cardinality of the corresponding select query. We next explain how the different components in the alerter are extended to handle updates.

Instrumenting the Query Optimizer

In addition to the `AND/OR` tree discussed in Section 2 for select queries, we also gather information for the update shells during optimization. Specifically, for each update query we store (i) the updated table, (ii) the number of added/changed/removed rows, and (iii) the query type (i.e., insert, delete or update). We note that this is the only information required to calculate the update overhead imposed by a new arbitrary index. By using the optimizer cost model, we can accurately calculate the cost of any update shell for a given index. The difference in cost of an `AND/OR` request tree for a given configuration $\mathcal{C}$ is extended for the case of updates as $\Delta_{\mathcal{C}}^{\mathcal{T}} + \sum_{I \in \mathcal{C}} \sum_{u \in \text{update shells}} \mathit{updateCost}(I, u)$.

The Alerter Main Algorithm

The presence of update queries requires two changes to the algorithm of Figure 5. First, we need to relax the predicate on line 3 by removing the condition on the minimum improvement P. The reason is that when updates are present, we can transform a configuration into another that is both smaller and more efficient. This happens when the indexes that we are removing or merging have large update overhead and relatively smaller benefits for query processing. For that reason, we cannot stop the loop in lines 3-7 after the first configuration with an improvement below P because a later configuration might again put the improvement above P.

For the same reason, some configurations in $\mathcal{R}$ might dominate others (i.e., a configuration in $\mathcal{R}$ can be both smaller and more efficient than another in $\mathcal{R}$). This cannot happen unless updates are present because each transformation decreases both the size of the configuration and its performance. As a postprocessing step, we eliminate the dominated configurations from $\mathcal{R}$ so that the alert does not contain redundant information and is easily analyzable.

Upper Bounds for Improvement

When updates are present, we refine the upper bounds of Section 4 by adding the necessary work by any update shell (i.e., the cost of each update shell for all the indexes that must be present in any configuration). We note that this extension makes the upper bound discussed in Section 4.2 loose, since there might be no configuration that meets the upper bound even without storage constraints.

5.2 Materialized Views

Physical design tools not only recommend indexes but also materialized views. One of the main challenges of extending our techniques to handle materialized views is the increased overhead at virtually every step in our algorithms. We next discuss extensions to our techniques that address materialized views and their associated overhead issues.

Similar to the access path selection module illustrated in Figure 2, query optimizers rely on a view matching component that, once invoked with a sub-query, returns zero or more equivalent rewritings of such query using an available view in the system. We can instrument the optimizer to analyze such requests and tag the root of every sub-query that is passed to the view matching mechanism as in the case of index requests. These *view requests* are more complex than index requests, since we have to encode the view expression itself (which might contain joins, grouping clauses and computed columns). However, the idea is still the same, and at the end of query optimization we return an `AND/OR` tree that contains both index and view requests[8], along with the cost of the best execution sub-plan found by the optimizer (which might or might not use materialized views). As a simple example extending Figure 3, an additional view request ρ_V with definition $\Pi_{T_1.w}\big(\sigma_{T_1.a=5}(T_1) \bowtie_{x=y} T_2\big)$ is attached to

[8]The resulting request tree is not necessarily simple anymore as in Property 1 due to the different places that the optimizer might invoke the view matching component.

the join node that currently contains ρ_2. The cost associated with ρ_V would be 0.23 time units, since that is the cost for the best sub-plan found by the optimizer. When creating the `AND/OR` tree, we extend the algorithm in Figure 4 by `OR`ing each node that contains a view request with the original `AND/OR` tree that would result if no view requests are present. The reason is that we can either implement the index requests in the sub-trees or the view request, but not both. In the example, the normalized request tree is `AND( OR( AND(`ρ_1`, `ρ_2`), `ρ_V`), OR(`ρ_3`, `ρ_5`))`, which is not simple anymore as in Property 1.

When view matching succeeds, the optimizer rewrites the corresponding sub-query with the view, and subsequently issues index requests to obtain physical sub-plans with indexes over materialized views. View requests are inherently less precise than index requests, since we usually have no information on what index strategies would be requested over the corresponding views if these are not matched during optimization. However, as explained in Section 3, our lower bound techniques only require that we generate valid sub-plans rooted at requests. In those situations, we can simply generate the naive plan that sequentially scans the primary index of the materialized view and filters all the relevant tuples. This would be a loose bound in general, because specialized indexes over the materialized views can evaluate the same sub-query more efficiently. However, in many situation this technique provides reasonable approximations (specially for aggregate views that return a few tuples after performing complex computation).

Full processing of materialized views might be too expensive for an alerting mechanism as described in this work, because the new search space becomes larger and more complex. However, the simple extensions described in this section might be a good compromise between the quality of improvement bounds and the overhead required to obtain them. We believe that an interesting piece of future work is to evaluate the techniques of this paper for the case of materialized views and extend the ideas to other physical design features (e.g., partitioning).

6. EXPERIMENTAL EVALUATION

In this section we report an experimental evaluation of our proposed techniques over both synthetic and real databases (see Table 1 for a summary of the experimental setting). We implemented the client component of the alerter in C++ and modified Microsoft SQL Server 2005 to support our extensions of Section 2 and 4. When experiments require a comprehensive tuning tool, we use Microsoft SQL Server Database Tuning Advisor [1]. The aim of this section is to show that the alerter results in very efficient executions with good quality upper and lower improvement bounds. The metric to evaluate a recommendation is *improvement*, defined as:

$$improvement(\mathcal{C}_I, \mathcal{C}_R, W) = 100\% \cdot \left(1 - \frac{cost(W, \mathcal{C}_R)}{cost(W, \mathcal{C}_I)}\right)$$

where $\mathcal{C}_I$ is the initial configuration, $\mathcal{C}_R$ is the recommended configuration, and $cost(W, \mathcal{C})$ is the expected cost of evaluating all queries in the workload W for configuration $\mathcal{C}$.

6.1 Single-Query Workloads

We first evaluated the simplest scenario for our techniques: returning lower and upper improvement bounds for single-query workloads and no storage constraints. Specifically, we ran the main alerter algorithm with each of the 22 queries of the `TPC-H` benchmark [12]. Figure 6 shows the results, where each bar corresponds to a different query and reports the lower bound, fast upper bound and tight upper bound improvements obtained using the techniques of Section 3 and 4. Since each workload consists of a single query and we do not specify any storage constraint, the tight upper bounds agree with the optimal improvement that a comprehensive tuning tool would recommend.

Database	Size	#Tables	#Queries
TPC-H (Synthetic)	1.2 GB	8	24
Bench (Synthetic)	0.5 GB	6	144
DR1 (Real)	2.9 GB	116	30
DR2 (Real)	13.4 GB	34	10

Table 1: Databases and workloads evaluated.

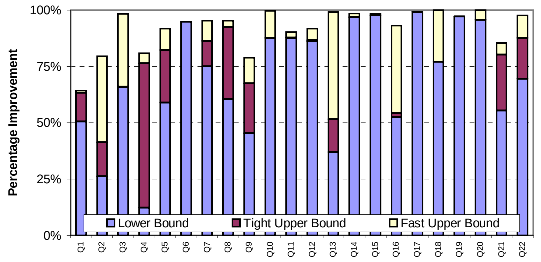
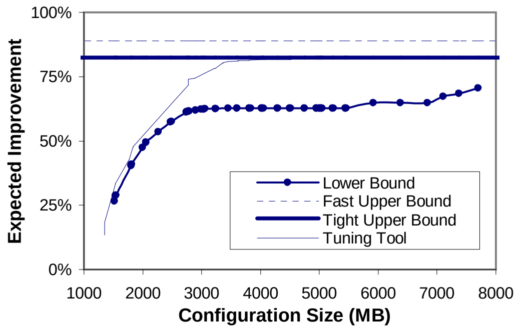


Figure 6: Lower and upper bounds for `TPC-H`.

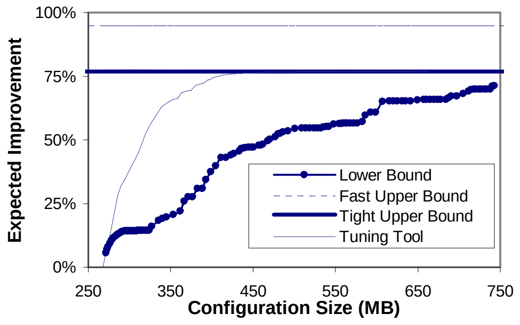
We see that in virtually all cases (except for Q_4) the lower bound improvement obtained by our techniques is relatively tight (less than 20% worse than the tight upper bound, or global optimum improvement). This shows that the lower bound technique is adequate to obtain good configurations by only analyzing local plan changes according to winning requests. The difference between the lower and tight upper bounds represents the difference in cost between the *locally* optimal execution plan obtained by implementing each winning request as efficiently as possible and the *globally* optimal execution plan obtained by fully optimizing the query with all possible indexes. We see in the figure that about half of the queries agree between locally and globally optimal plans. In such cases, our lower bound technique is as tight as possible. Finally, the difference between the tight and fast upper bounds represents the loss in quality that we incur when obtaining upper bounds very efficiently (without changing the query optimizer). In most of the cases the difference is modest (from 0% to 10%), but there are cases with a 30% and even 40% gap. These situations correspond to query plans that contain expensive intermediate operators. Our fast upper bounds do not account for such operators and therefore underestimate the best cost of the queries.

6.2 Multi-Query Workloads

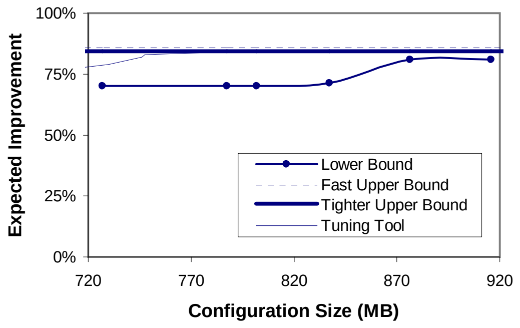
We next evaluate the main alerter algorithm. For that purpose, we used the synthetic and real databases of Table 1. The original physical design for each database consisted of all the primary indexes and, for the real databases, additional secondary indexes (databases `DR1` and `DR2` had an average of 2.1 and 4.2 indexes per table, respectively). For each experiment, we executed the respective query workloads and then passed the information gathered during query optimization to the client alerter application. Figure 7 reports the results of this experiment, showing the lower, fast



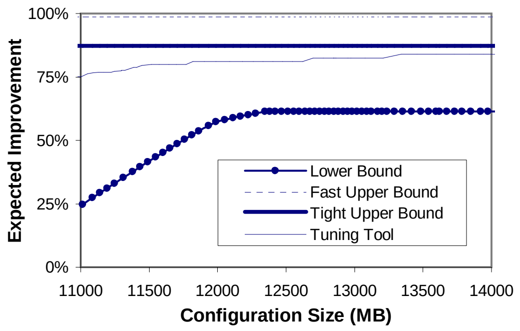
(a) TPC-H data set.



(b) Bench data set.



(c) DR-1 data set.



(d) DR-2 data set.

Figure 7: Complex workloads and storage constraints.

upper, and tight upper bounds returned by our techniques, and also the improvement obtained by running a comprehensive tuning tool. In all cases, the alerter returned the skyline of lower and upper bounds in less than a second.

We did not specify any storage constraints, so the alerter returned all configurations with positive improvement regardless of their sizes. Specifically, we range the configuration sizes from the minimum possible (i.e., when only the primary indexes are present) to the maximum possible (when the best indexes for all the requests in the AND/OR tree are available). We see that the alerter returns useful lower bounds. In the figure, when the size of the configuration is two to three times that of the smallest possible one (a reasonable storage constraint) , the lower bounds returned by the alerter are just 10% to 20% below the best configurations found by a comprehensive tuning tool.

In Figure 7(c) the difference between lower and upper bounds is smaller than for the other workloads since in this scenario index merging is very effective and therefore we can reduce the storage significantly without losing efficiency. In general, upper bounds present the following behavior: when no storage is given, the difference between the lower and tighter upper bound comes from locally versus globally optimal plans as discussed in Section 3.1. Since we do not perform any search for upper bounds, these values are independent of the storage constraint. Therefore, the smaller the space constraint, the larger the difference between upper and lower bounds.

Varying the Initial Physical Design

In this section we vary the initial physical design for a given fixed workload, and analyze the behavior of the alerter when the database is already (at least partially) tuned. For that purpose, we took the original TPC-H database with only primary indexes and triggered the alerter with the input workload (see Figure 7(a), repeated in Figure 8 with the C_0 label). We then implemented the recommended configuration for 1.5GB of storage as C_1 in the database, re-executed the workload, and triggered the alerter once again. We continued in this way, obtaining C_2 as the recommended configuration for 2GB when starting from C_1, C_3 as the recommended configuration for 2.5GB when starting from C_2, and so on. Figure 8 shows the results of the alerter for the same workload and different initial configurations. We can see that when using a better initial configuration than C_0, the overall gains returned by the alerter are smaller, because there are fewer opportunities for optimization. Also, if we keep the storage constraint fixed, we can see that the expected improvement of an already tuned workload is close to zero, as expected. For instance, there is virtually no expected improvement for C_1 and 1.5GB, because C_1 was the configuration that the alerter recommended from C_0 for 1.5GB.

Figure 8 shows that, for a given storage constraint and minimum improvement, the alerter would trigger an alarm only for certain configurations. For instance, if the original configuration is C_5 and the minimum improvement is 30%, the alerter would not trigger any alarm independent of the storage constraint, since C_5 is already a good configuration. For a 20% improvement and 3GB of maximum storage, the alerter would only trigger alarms for C_0, C_1, and C_2. Contrast this behavior with that of Figure 7, in which the alerter would (correctly) trigger an alarm for all the workloads for a storage constraint twice as large as the size of the (untuned) databases and a minimum improvement of 30%.

Figure 8 also indirectly shows the effect of locally optimal plans discussed in Section 3.1. We can see that the improvement for C_4 around 4.5GB-5GB is larger than that of

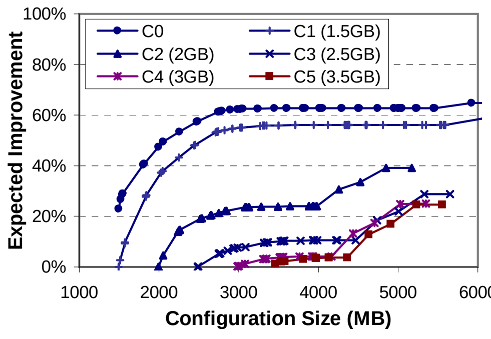


Figure 8: Varying the initial configuration.

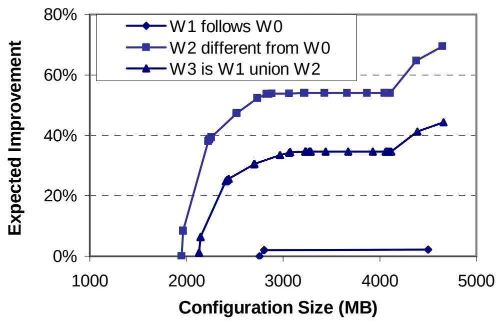


Figure 9: Varying workloads.

C_3. The reason is that the improvement for C_3 is measured with respect to the best locally optimal plan. Of course, when considering C_4, the execution plans for the queries in the workload might be even better than the best locally optimal ones with respect to C_3, and therefore might new optimization opportunities. These opportunities are only captured when optimizing the workload under C_4.

Varying the Workload

In this section we analyze our techniques when we vary the query workload. For that purpose, we generated a workload W_0 consisting of random instances of the first 11 query templates of the `TPC-H` workload, and tuned the database using a comprehensive tuning tool (the resulting configuration was around 2.5GB). Then, we generated three workloads: W_1 (which consists of additional random instances of the first 11 queries of the `TPCH-H` workload and therefore follows W_0), W_2 (which consists of random instances of the last 11 queries of the `TPC-H` workload and therefore is different from W_0), and $W_3 = W_1 \cup W_2$. Figure 9 shows the results of the alerter when triggered for each of W_1, W_2 and W_3. We can see that if the workload characteristics do not change (W_1) the tuned configuration is already optimal and the alerter would not issue an alarm. On the other hand, workload W_2 results in the alerter giving over a 60% improvement when no storage is given. Note that for W_2 the alerter gives positive benefit even below the size of the original configuration (i.e., 2.5GB) but it does not give any improvement below 2GB. The reason is that some of the indexes of the original configuration do help the queries in W_1, and below 2GB we cannot find anything better than a subset of what was already present initially. Finally, W_3 results in an expected intermediate behavior.

6.3 Client/Server Overhead

In this section we report the overhead of our techniques for different databases and workloads. Table 2 shows the execution time of the alerter (without counting the workload gathering step). We see that even for hundreds of input queries the alerter executes in the order of seconds (in many cases, in less than one second). We note that the number of queries in Table 2 refers to the number of *distinct* queries in the workload. In fact, if the same query is executed multiple times, we scale up the costs of the `AND/OR` request tree but do not augment the tree with additional information. The execution cost of the alerting client is therefore proportional to the number of *distinct* queries in the workload. For completeness, we also tuned the same workloads with a commercial physical design tool (we only tuned for indexes using the commercial tool, to avoid comparing different feature sets). In these situations, the alerting mechanism was several orders of magnitude more efficient than comprehensive tools, and thus a valuable addition to the DBA's toolkit.

Database	Queries	Requests	Alerter
TPC-H	22	113	0.21 secs.
	100	662	0.33 secs.
	500	3344	1.25 secs.
	1000	6680	4.25 secs.
Bench	60	215	0.37 secs.
DR1	11	114	0.12 secs.
DR2	11	215	0.36 secs.

Table 2: Client overhead for the alerter.

Figure 10 shows the server overhead when gathering workload information. Specifically, we took each of the 22 `TPC-H` queries, and measured the increase in optimization time when we additionally gathered the information required for obtaining both lower bounds and either fast or tight upper bounds. We see that if we use the fast upper bounds the overhead over traditional optimization is very low (below 1% for all but one query which results in 3% overhead). When using the tighter upper bound technique, the overhead significantly increases (reaching as high as 40% overhead for some complex queries). We analyzed these cases and a significant portion of the overhead comes from redundant work that our prototype executes during query optimization. While we believe that a more careful implementation of the techniques of Section 4.2 would significantly reduce the overhead in Figure 10, it would still be considerably higher than that of the fast upper bounds. We believe that both mechanisms can be useful since they balance the overhead at optimization time and the resulting quality of the upper bounds. Upper bounds are an important addition that reduces the chances of false negatives, but the crucial aspect of the alerter is the ability to give robust lower bounds. If we eliminate any upper bound mechanism in the server, the overhead drops below 1% in all cases.

7. RELATED WORK

In recent years there has been considerable research on automating the physical design in database systems. Several pieces of work (e.g., [2, 5, 7, 13, 15]) present solutions that consider different physical structures, and some of these ideas started appearing in commercial products (e.g., [1, 8, 14]). This line of work, while successful, fails to address the scenarios discussed in the introduction. Specifically, DBAs

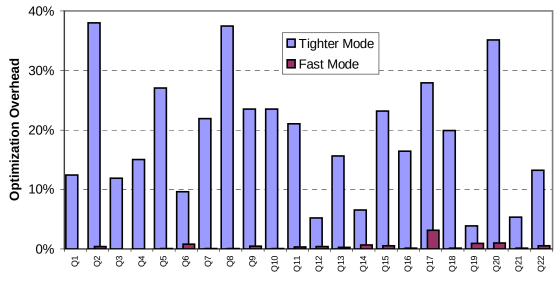


Figure 10: Server overhead for the alerter.

do not know when tuning a workload would result in a sensible improvement short of running expensive tuning tools. In this paper we complement such techniques by providing a lightweight alerting mechanism that identifies when an expensive tuning session would be useful.

Previous work in the literature presents transformations that can be exploited for physical database design. Reference [7] introduces the concept of index merging. Similarly, reference [2] exploits a few transformations to combine the information in materialized views. Reference [4] considers a unified approach of primitive operations over indexes and materialized views that can form the basis of physical design tools. In this work we leverage these ideas and select a subset of transformations that balance the overhead at runtime and the quality of the resulting configurations.

Some of the ideas in this work are inspired by [3]. We share with this reference the goal of transforming a given configuration into another one that fits in the available storage, using for that purpose an instrumented optimizer. However, our focus is significantly different from that of [3]. Specifically, our goal is not to produce an optimal physical design. Rather, the focus of the current work is to produce lower and upper bounds very efficiently. This trade-off resulted in a significantly different technique presented in the current paper. Specifically, we generate (suboptimal) configurations that are relatively close to the ones obtained by the techniques in [3], and we produce such solutions orders of magnitude more efficiently that the techniques in [3]. This is achieved by a novel interception mechanism to extract additional information from the optimizer and thus avoid optimization calls completely when executing the alerter (in contrast, the relaxation-based approach in [3] repeatedly calls the optimizer during its search phase). The techniques in [3] are indeed an example of a comprehensive tuning tool that would be invoked after receiving an alert based on the techniques of this work.

8. CONCLUSIONS

Motivated by the complexity of today's automated physical design tools, in this paper we propose a new architecture for the automatic physical design cycle in a DBMS. Specifically, we developed a crucial component, the *alerter*, which is a low-overhead procedure that runs periodically during normal operation of a DBMS and alerts DBAs if the current configuration is suboptimal. The alerter reports lower and upper bounds on the improvement that could be obtained if a comprehensive tuning tool is launched, and justifies the lower bounds by generating feasible configurations.

9. REFERENCES

[1] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proceedings of the 30th 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] N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2005.

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

[5] 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 (VLDB)*, 1997.

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

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

[8] 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 (VLDB)*, 2004.

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

[10] Microsoft Corporation. Dynamic management views and functions. Accessible at `http://msdn2.microsoft.com/en-us/library/ms188754.aspx`.

[11] P. G. Selinger et al. Access path selection in a relational database management system. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1979.

[12] TPC Benchmark H. Decision support. Available at `http://www.tpc.org`.

[13] 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.

[14] D. Zilio et al. DB2 design advisor: Integrated automatic physical database design. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*, 2004.

[15] D. Zilio et al. Recommending materialized views and indexes with IBM DB2 design advisor. In *International Conference on Autonomic Computing*, 2004.