An Online Approach to Physical Design Tuning

Nicolas Bruno Surajit Chaudhuri
Microsoft Research
{nicolasb,surajitc}@microsoft.com

Abstract

There has been considerable work on automated physical design tuning for database systems. Existing solutions require offline invocations of the tuning tool and depend on DBAs identifying representative workloads manually. In this work, we propose an alternative approach to the physical design problem. Specifically, we design algorithms that are always-on and continuously modify the current physical design reacting to changes in the query workload. Our techniques have low overhead and take into account storage constraints, update statements, and the cost to create temporary physical structures.

1 Introduction

Database applications have become increasingly complex and varied. Physical design tuning has therefore emerged as more relevant than ever before. Presently, most database vendors (e.g., [3, 8, 11]) offer automated tools to tune the physical design of a database, with the objective of reducing the DBMS' total cost of ownership. These automated tools are very sophisticated and useful. However, they take an offline approach to the physical design problem and leave several significant decisions to DBAs. Specifically, DBAs need to explicitly identify representative workloads and feed them to the tuning tool. DBAs are also required to *guess* when a tuning session is needed, and when to deploy recommendations. Naturally, this is not a one-time process, but instead DBAs continuously monitor, diagnose, and re-tune database installations. In that sense, we still require some effort to "get physical tuning right".

Unfortunately, these manual tasks become even more problematic in current complex applications. Consider, as an increasingly common example, large installations that support multiple database applications (for instance, ISPs provide this back-end service to advanced users). Such hosted applications may come and go, and usually exhibit unexpected spikes in their loads. At the same time, the hosting installation might have some additional amount of resources to globally tune the physical design, and all applications would compete for these valuable resources. As another example, some applications exhibit periodic, sometimes unexpected changes in the `select/update` mix in the workload. Consider, for instance, a bug-tracking system. Most days the system is queried and browsed (`select` load), but a few days –sometimes called "bug-bash" days– are used to primarily identify and insert large numbers of bugs (`update` load). If we gather a representative workload over, say, a month, chances are that no index is globally useful, as the gains in query processing are outweighed by the update costs during bug-bash periods. It is very difficult to explicitly model the workload in these scenarios, and equally difficult to decide *when* to tune the database and deploy the resulting recommendations (in fact, tuning too frequently results in wasted resources, but tuning too sporadically misses critical opportunities to improve performance).

With increasingly common DBMS features like *online indexes*[1], it is appealing to explore more automatic solutions to the physical design problem that advance the state of the art. There are, however, new and significant challenges to address. Such fully automated solutions will be *always-on*, continuously monitoring changes in both the workload and the database state, and refining the physical design as needed. It is therefore critical that such solutions have very low overhead and do not interfere with the normal functioning of the DBMS. Additionally, in contrast to current offline approaches, fully automated solutions must also balance the cost of transitioning between physical design configurations and the potential benefits of such design changes for the future workload. Although we would like to react quickly to changes in the workload, reacting too quickly can result in unwanted oscillations, in which the same indexes are continuously created and dropped. A fully automatic solution must "do no harm" for stable workloads, but also react in a timely manner to significant workload changes.

In this paper we introduce our online algorithm to tune indexes in a DBMS, developed in the context of the AutoAdmin project at Microsoft. Its main characteristics are:

- As queries are optimized, we identify a relevant set of candidate indexes that would improve performance.

[1] This feature allows query processing to continue in parallel with indexes that are built in the background.

- At execution time, we track the potential benefits that we lose by not having such candidate indexes and also the utility of existing indexes in the presence of queries, updates, and space constraints.
- After we gather enough "evidence" that a physical design change is beneficial, we automatically trigger index creations or deletions.
- The online nature of our problem implies that we will generally lag behind optimal solutions that know the future. However, by carefully measuring evidence, we ensure that we do not suffer from "late" decisions significantly, thus bounding the amount of incurred loss.

The rest of the paper is structured as follows. In Section 2 we show how to track index benefits and penalties by building on top of technology previously developed for offline solutions. In Section 3 we formally define our problem, present a three-competitive online solution for a restricted scenario, and generalize it to address the general problem. Section 4 reports an experimental evaluation of our algorithm, and Section 5 reviews related work.

2 Preliminaries

We now briefly review techniques introduced in [4, 6] that capture crucial information during query optimization and allow us to efficiently infer cost and plan properties for varying physical designs. Due to space constraints, the presentation is brief and high-level (see [6] for more details).

2.1 Access-Path/Index Requests

We gather information during optimization by instrumenting the optimizer and intercepting the optimization rules that generate execution sub-plans using index strategies. We then tag operators in the final execution plan with annotations (we call them *access-path-* or *index-requests*). Such requests encode the logical properties of any physical plan that might implement the sub-tree rooted at the corresponding operator (or its "right" sub-tree in the case of requests tagging joins). Consider the execution plan in Figure 1 for the following query:

```
SELECT S.b FROM R,S
WHERE R.x=S.y AND R.a=5 AND S.y=8
```

In the figure, request ρ_1 is associated with filter $R.a$=5 and specifies that (i) there is one sargable column $R.a$ returning 2500 tuples, (ii) $R.x$ is required upwards in the tree, and (iii) the plan found by the optimizer costs 0.08s and uses index I_1. Similarly, request ρ_2 was obtained when the optimizer tried an index-nested-loop join with R and S as the outer and inner relations, respectively. Request ρ_2 specifies that $S.y$ is a sargable column which would be sought with 2500 bindings and would produce 1 row for each binding (ρ_2 was not implemented in the final plan, which uses a hash join).

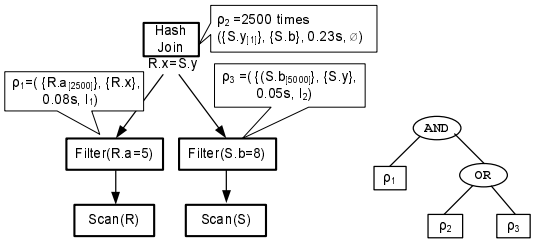


Figure 1. Execution plan and AND/OR request tree.

As shown in [6], some requests might conflict with each other. For instance, ρ_2 and ρ_3 in the figure are mutually exclusive (if a plan implements ρ_2 with an index-nested-loop join using S as the inner table, it cannot simultaneously implement ρ_3). To represent these relationships, we encode the requests in an AND/OR tree (see Figure 1), where internal nodes indicate whether their sub-trees can be satisfied simultaneously (AND) or are mutually exclusive (OR).

2.2 Local Plan Transformations

The query requests obtained during optimization and stored in the resulting plan allow us to make inferences about characteristics and costs of execution plans for varying physical designs (i.e., index creations or drops), and do so without issuing additional optimization calls. 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 (overall) plan would be valid and equivalent to the original one. We know the cost of the original sub-plan and can calculate the cost of the newly generated alternative. Thus, we can infer how much would the original execution plan improve or degrade if we replaced the given sub-tree with the alternative one.

As an example, consider ρ_1 in Figure 1 and suppose we would like to infer what would happen if a new index I_3=(a, x) is added to the current physical design. In that case, we can implement ρ_1 by using an index seek on column $I_3.a$ returning 2500 (a, x) tuples, and passing the projection on x upwards in the tree. If the calculated cost of this alternative is, say, 0.03s, we know that the overall execution plan would be $0.08s-0.03s$=$0.05s$ more efficient than when the original index I_1 was used. Furthermore, by just analyzing the request, we can infer what is the index that would result in the cheapest plan *without* enumerating all possibilities. In our example, I_3=(a, x) is the best index for ρ_1 (see [6] for more details).

In this way, we can obtain a *locally-optimum* 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. Instead, we are able to explore a limited set

of alternate plans without having to re-optimize the query. The cost of the resulting plan is therefore an approximation (tight upper bound) of the global optimal plan that the optimizer would find under the new physical design.

To summarize, the following functions (adapted from [4, 6]) form the basis of the online algorithms in this paper:

- `getRequests(q:query)`: gets the `AND/OR` request tree for q encoding the requirements of each index strategy that can be implemented via local transformations.
- `getBestIndex(`ρ`:request)`: gets the index that results in the cheapest alternative implementing ρ.
- `getCost(`ρ`:request,` $\{I_j\}$`:indexes)`: approximates the cost of the best locally transformed plan implementing ρ when $\{I_j\}$ are available.

3 Online Physical Tuning

We now define the online physical tuning problem and present algorithms to solve it. In Section 3.1 we focus on the case of a single index and obtain a three-competitive algorithm. In Section 3.2 we extend this approach to the general case (but cannot guarantee three-competitiveness anymore).

A *physical configuration* is the set of indexes available at some point in time. For a configuration s, we denote the cost of creating an index I as B_I^s (note that B_I^s depends on the indexes in s). Let a workload W=$(q_1, q_2, \ldots, q_n)$ be a sequence of queries and updates. We define $cost(q_i, s)$, or simply c_i^s if q_i is clear from the context, as the estimated cost of q_i when optimized under configuration s.

A *configuration schedule* S is a sequence of configurations S=$(s_0, s_1, \ldots, s_n)$, such that q_i is executed when the DBMS is in configuration s_i. The cost of W under S is:

$$cost(W, S) = \sum_{i=1}^{n} \left(c_i^{s_i} + transition(s_{i-1}, s_i) \right)$$

where $transition(s_0, s_1)$=$\sum_{I \in (s_1 - s_0)} B_I^{s_0}$. Therefore, $cost(W, S)$ is the sum of each query cost in W under the corresponding configuration, plus the total cost to transition between configurations in S. The optimal configuration schedule S^* is the one with minimum cost, so S^*= $minarg_S(cost(W, S))$. An online algorithm that solves this problem must progressively determine S=$(s_0, \ldots, s_n)$ *without seeing* the complete workload W=$(q_1, \ldots, q_n)$. Specifically, to determine each physical configuration s_i, we only have knowledge about $(q_1, \ldots, q_i)$.

3.1 Single-Index Scenario

To simplify the presentation, in this case a physical configuration s is either 1 (when the given index I is present) or 0 (when it is absent). We can only create I from s=0, so we denote I's creation cost simply as B_I. The transition cost between configurations is given by:

$$\text{transition}(s_0, s_1) = \begin{cases} B_I & \text{if } s_0 = 0 \text{ and } s_1 = 1 \\ 0 & \text{otherwise} \end{cases}$$

3.1.1 Optimal Strategy Opt-SI

We now explain how to obtain the optimal schedule S^* for a given workload W[2].

Definition 1 *For a workload W and integers i_0, i_1, we define $\Delta(W, i_0, i_1)$=$\sum_{i=i_0}^{i_1}(c_i^0 - c_i^1)$ where c_i^0 (c_i^1) is the cost of q_i when index I is present in (absent from) configuration s_i. If W is clear from the context, we simply write Δ_{i_0,i_1}.*

Intuitively, Δ_{i_0,i_1} measures, for a sub-sequence of the workload, the cumulative difference in cost between the configuration that does not contain index I (s=0) and the one that contains it (s=1). If Δ_{i_0,i_1}=C, executing queries $(q_{i_0}, \ldots, q_{i_1})$ without the index (s=0) is C units more expensive than doing it with the index (s=1). Therefore, we can see Δ values as the aggregated benefit (or penalty, for negative Δ values) of having the index in the configuration for a given workload sub-sequence.

Opt-SI (W=$(q_1, \ldots, q_n)$`:workload,` s_0`:configuration`)

```
01 i=0
02 while (i < n)
03   if (s_i=0) // see Cases A1, A2, A3 in Figure 3
04     if (Case A1) s_k=0 for i+1≤k≤j; i=j
05     else if (Case A2) s_k=1 for i+1≤k≤j; i=j
06     else (Case A3) s_k=0 for i+1≤k≤n; i=n
07   else // s_i=1, see Cases B1, B2, B3 in Figure 3
08     if (Case B1) s_k=1 for i+1≤k≤j; i=j
09     else if (Case B2) s_k=0 for i+1≤k≤j; i=j
10     else (Case B3) s_k=0 for i+1≤k≤n; i=n
```

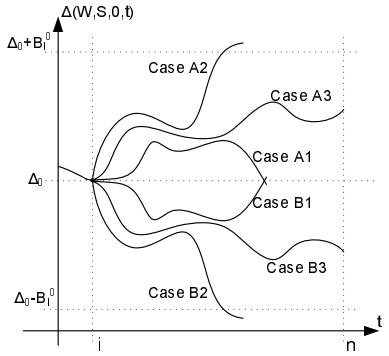
Figure 2. Optimal algorithm for single-index case.

Figure 2 introduces the optimal algorithm `Opt-SI` using Δ values. The idea is to progressively determine the optimal schedule S^* for longer workload prefixes by using a case-by-case analysis on the future behavior of Δ. Each new sub-schedule is appended to the optimal prefix after determining whether a physical change would be beneficial. Consider Case $A2$ in Figure 3. If the benefit of the index at a point in the future is larger than its creation cost (and it is never negative), it makes sense to create the index for such period of time. We next prove that this intuition is correct.

Theorem 1 *Algorithm* `Opt-SI` *determines the optimal configuration schedule for an input workload W.*

Proof: From Figure 3 it follows that any instance of Δ satisfies one and only one among $\{A1, A2, A3\}$, or one and only one among $\{B1, B2, B3\}$. Algorithm `Opt-SI` therefore always advances i at each iteration, and in doing so determines longer prefixes of the optimal schedule. Eventually, it reaches i=n and terminates. We need to show that each determination in lines 4-6 and 8-10 leads to an optimal schedule. For instance, consider Case $A2$ and suppose that s_i=0. Algorithm `Opt-SI` appends the sub-schedule

[2]Note that there are simpler ways to obtain the optimal schedule (e.g., see [2]), but our version is more easily adapted to an online algorithm.



Case A1:	$\exists j{>}i$ s.t.$\Delta_{i,j}{\leq}0$ and $\forall j'{<}j$, $0{<}\Delta_{i,j'}{<}B_I$
Case A2:	$\exists j{>}i$ s.t.$\Delta_{i,j}{\geq}B_I$ and $\forall j'{<}j$, $0{<}\Delta_{i,j'}{<}B_I$
Case A3:	$\forall i{<}j{\leq}n, 0{<}\Delta_{i,j}{<}B_I$
Case B1:	$\exists j{>}i$ s.t.$\Delta_{i,j}{\geq}0$ and $\forall j'{<}j$, $-B_I{<}\Delta_{i,j'}{<}0$
Case B2:	$\exists j{>}i$ s.t.$\Delta_{i,j}{\leq}-B_I$ and $\forall j'{<}j$, $-B_I^-{<}\Delta_{i,j'}{<}0$
Case B3:	$\forall i{<}j{\leq}n, -B_I{<}\Delta_{i,j}{<}0$

Figure 3. Possible behavior of $\Delta_{i,n}$.

S_O=$(1, 1, \ldots, 1)$ from positions i+1 to j. Suppose there is an alternative schedule S_A that contains at least one index deletion. S_A starts with a block of zero or more configurations with no index (s=0), continues with a strict alternation between blocks of configurations with the index (s=1) and without it (s=0), and optionally ends with a block of configurations with the index (s=1). Let us obtain the difference in cost between sub-schedules S_A and S_O :

S_A	$C_1^0 B_I$	C_2^1	$C_3^0 B_I$	C_4^1	...	C_n^0	$[B_I C_{n+1}^1]$
S_O	$B_I C_1^1$	C_2^1	C_3^1	C_4^1	...	C_n^1	$[\ C_{n+1}^1]$
S_A-S_O	δ_1	0	B_I+δ_3	0	...	δ_n	$[B_I\ \ ]$

where C_i^0 and C_i^1 denote the partial cost of the blocks with configuration s=0 and s=1, respectively. Before switching from s=0 to s=1 in S_A, we add B_I, the cost of creating I. Also, the final costs between brackets represent the optional block with s=1. Consider now δ_1. According to Definition 1, δ_1=$\Delta_{i,i'}$ for some $i < i' \leq j$. By definition of Case A_2, we know that $\delta_1 > 0$. Similarly, it can be shown that δ_n=$\Delta_{j',j} > 0$ and for each $1 < k < n$, $|\delta_k| \leq B_I$ (see Figure 3 for visual intuition). Putting it all together, $cost(W, S_A, i{+}1, j) > cost(W, S_O, i{+}1, j)$. (Note that if S_A contains no index creations nor deletions, $S_A{-}S_O$=$C_n^0{-}C_n^1$=$\Delta_n > 0$ as well.) The remaining cases are proved analogously, but we omit the details due to space constraints. ∎

3.1.2 Online Strategy Online-SI

A careful look at algorithm `Opt-SI` reveals the following property. Suppose that at some iteration we added a configuration block of s_i=0. Algorithm `Opt-SI` will then transition to s=1 if $\Delta_{i,j}{>}B_I$ for the smallest $j{>}i$, and $\Delta_{i,j'}$ does not goes below zero for $i < j' < j$. Another way of simulating this behavior is to maintain the minimum value of $\Delta_{0,i}$ since `Opt-SI` lastly transitioned to s=0 (let us call it Δ_{min}), and transition to s=1 if there is $j{>}i$ such that $\Delta_{0,j}{>}\Delta_{min}{+}B_I$ and no $j' < j$ satisfies $\Delta_{0,j'} < \Delta_{min}$. Similarly, if s=1, we maintain the maximum value of $\Delta_{0,i}$ since `Opt-SI` lastly transitioned to s=1 (let us call it Δ_{max}), and transition to s=0 if there is a $j > i$ such that $\Delta_{0,j} < \Delta_{max}{-}B_I$ and no $j'{<}j$ satisfies $\Delta_{0,j'}{>}\Delta_{max}$.

This alternative formulation of `Opt-SI` suggests an online algorithm. We maintain Δ_{min} and Δ_{max} as explained above, but instead of looking into the (unknown) future, we transition configurations *after* gathering the information that proves that the optimal strategy would have done so at a past point in time. Algorithm `Online-SI` is shown in Figure 4. We note that in line 1 we need to obtain the expected cost of the input query under the "opposite" physical configuration. We do that without issuing an additional optimization call by using the `getCost` function from Section 2 over the request that used (or could have used) index I. Note that we store a constant amount of information per index (i.e., Δ, Δ_{min}, and Δ_{max}). Also, every time we execute a query we only update Δ values, whose cost is negligible compared to that of executing the actual queries.

Online-SI (q_i :query, s:current configuration)

```
// Initially, Δ=Δmin=Δmax = 0
1  δ = c_i^0 − c_i^1
2  Δ = Δ + δ
3  Δmin = MIN(Δmin, Δ)
4  Δmax = MAX(Δmax, Δ)
5  if (s=0 and Δ−Δmin≥BI)
6     Δmax=Δ; s=1  // create index
7  if (s=1 and Δmax−Δ≥BI)
8     Δmin=Δ; s=0  // drop index
```

Figure 4. Online algorithm for single index Case.

3.1.3 Competitive Analysis

Conceptually, algorithm `Online-SI` lags behind `Opt-SI` and transitions physical designs only after the evidence that `Opt-SI` would have gathered "from the future" has already passed. We now bound the sub-optimality of `Online-SI` by presenting a worst-case scenario for which `Online-SI` keeps creating and dropping index I as often as possible without ever exploiting it.

Theorem 2 *Algorithm* `Online-SI` *is three-competitive (i.e., it is no worse than 3 times the optimal algorithm* `Opt-SI`*).*

Proof: Consider workload W=$(q_1, q_2, q_1, q_2, \ldots)$, where $cost(q_1, 0)$=$\epsilon + B_I$, $cost(q_1, 1)$=ϵ, $cost(q_2, 0)$=ϵ and $cost(q_2, 1) = \epsilon + B_I$. The optimal schedule S^* for W is $(s_0{=}0, 1, 0, 1, 0, \ldots)$. In other words, index I is built before each instance of q_1 and dropped before each instance of q_2. The cost of such an schedule is $(B_I + 2\epsilon)$ for every pair (q_1, q_2) in W. The schedule produced by

`Online-SI` is S_{online}=(s_0=0, 0, 1, 0, 1, 0, ...). The cost of such an schedule is $(\epsilon + B_I) + B_I + (\epsilon + B_I)$, or $(3B_I + 2\epsilon)$ for every pair (q_1, q_2) in W. Then, the ratio $cost(W, S_{online})/cost(W, S^*)$ is $\frac{3B_I+2\epsilon}{B_I+2\epsilon}$<3 since ϵ>0. ∎

3.2 Multiple-Index Scenario

We now extend the ideas of the previous section to multiple given indexes. For that purpose, we first revise the definition of Δ values to reflect this scenario.

Definition 2 *For a workload W, a configuration s, an index I, and integers i_0, i_1, we define* $\Delta(W, s, I, i_0, i_1)$=$\sum_{i=i_0}^{i_1} c_i^{s-\{I\}} - c_i^{s\cup\{I\}}$ *where c_i^s is the cost of q_i under configuration s. If W, s, and I are clear from the context, we simply write Δ_{i_0,i_1}.*

Using Δ values, we can generalize `Online-SI`. For each query q_i that is executed, we execute `Online-SI` in parallel for each index $I \in \{$`getBestIndex(`ρ`):` $\rho \in$ `getRequests(`q_i`)`$\}$. This generalization, however, introduces two challenges that we address below.

Consider indexes I_1=(a, b, c) and I_2=(a, b, c, d). If we do not consider the inherent **index interaction** between I_1 and I_2, we risk (i) underestimating Δ values for I_2 by ignoring sub-optimal –but better than existing– plans that use I_2 for requests served optimally by I_1, (ii) overestimating Δ values for I_1 after creating I_2 because I_2 can be a better alternative than the original one if I_1 is not present, and similarly (iii) underestimating Δ values for I_2 if I_1 is removed from the current configuration. Consequently, physical design changes might be unexpectedly affected.

Additionally, if there is a **storage constraint**, we might not be able to create all indexes I for which $\Delta - \Delta_{min} \geq B_I^s$. In these situations, we need to (i) decide which indexes to create in case of competing alternatives, (ii) decide whether to drop an index I from the current configuration s even though $\Delta_{max} - \Delta < B_I^s$ to make space for better alternatives, and (iii) consider index merging [5] to obtain additional indexes that might better trade off space and efficiency.

3.2.1 Index Interactions

We noted the importance of taking into account index interactions when maintaining Δ values. It is equally important to address this issue with low bookkeeping overhead, or the tuning requirements would be too large. We now explain our approach to address index interactions, which requires a constant amount of information per index. While ours is not a definitive solution, it reasonably balances the accuracy of the resulting approximations and the required overhead.

Recall that, for each index I that we consider in configuration s, we need to accumulate the value $\Delta_{i_0,i_{now}}$=$\sum_{i=i_0}^{i_{now}} c_i^{s-\{I\}} - c_i^{s\cup\{I\}}$. To simplify the notation, we use O_i instead of $c_i^{s-\{I\}}$ and N_i instead of $c_i^{s\cup\{I\}}$. For each incoming query q_i, we obtain O_i (original cost for q_i when I is not present) and N_i (new cost of q_i when I is present) by using function `getCost` as described in Section 2. Instead of maintaining Δ=$\sum(O_i - N_i)$, we exploit the equality $\sum_i(O_i - N_i)$=$(\sum_i O_i)-(\sum_i N_i)$ and maintain these two aggregates separately. Additionally, we decompose each aggregate into four terms, $\sum_i O_i$=O^0+O^1+O^2+O^U, and $\sum_i N_i$=N^0+N^1+N^2+N^U, and modify these values depending on *how* index I is used for each request coming from the workload:

- If I's columns are required in no particular order, we add O_i to O^0 and N_i to N^0.
- If I's key column is required to be there (e.g., for an index seek), we add O_i to O^1 and N_i to N^1.
- If more than one key column in I is required to be there (e.g., for a multi-column index seek or sort request), we add O_i to O^2 and N_i to N^2.
- If I is updated by the query, we add O_i and N_i from the update shell to O^U and N^U, respectively.

We store these eight values along with each index, and obtain back Δ as O^0+O^1+O^2+O^U−N^0−N^1−N^2−N^U. Since we now have more granular information about each index usage, we can handle index interactions more accurately (although still in an approximate sense). For that purpose, we use the following definition.

Definition 3 *Let I_1 and I_2 be indexes. The* usefulness level *of I_1 with respect to I_2 is given by the following table:*

Level	*Condition*
-1	*I_1 columns do not include I_2 columns.*
0	*I_1 columns include I_2 columns.*
1	*Additionally, I_2's leading column agrees with I_1's.*
2	*Additionally, I_2 is a prefix of I_1.*

Informally, if the usefulness level of I_1 with respect to I_2 is $l \geq 0$, then I_1 can (sub-optimally) implement requests whose costs were stored in O^m and N^m components of Δ for I_2 (for $m \leq l$). Note that this is an approximation, and some indexes can help implement additional requests, but we only consider these cases to keep the overhead low. As an example, consider I_1=(a, b, c) and I_2=(a, c). The usefulness level of I_1 with respect to I_2 is 1, and the usefulness level of I_2 with respect to I_1 is -1. This means that all the requests whose costs were stored in (O^0, N^0) or (O^1, N^1) for I_2 can also take advantage of I_1.

Adjusting Δ values after index creation. Suppose that we create index I in the current configuration. We then need to update the Δ values for the remaining indexes that we consider to reflect the fact that the current configuration contains I. We then proceed as follows for each index I_j:

1. Find l_j, the usefulness level of I with respect to I_j.
2. For each level $l \leq l_j$, set O^l to $\min(O^l, \alpha_j \cdot N^l)$ where α_j=$size(I_j)/size(I)$. The rationale is that if I is created, the original cost O^l in I_j for all $l \leq l_j$ might be

reduced due to I. We thus refine O^l for I_j as the minimum between the original value and a factor α_j of N^l (we linearly extrapolate the cost of index usages as a function of the index sizes). Since I_j was optimal for the requests it served, N^l values remain unchanged. The net effect is that we potentially reduce the value of Δ for index I_j as a result of creating I.

3. Adjust Δ_{min} and Δ_{max} as appropriate.

Adjusting Δ values after index deletion. Similarly, if we drop index I in the current configuration, we update Δ values of each remaining indexes I_j as follows:

1. Find l_j, the usefulness level of I with respect to I_j.
2. For each level $l \leq l_j$, multiply O^l by β^l, where β^l=O^l/N^l *from index* I. The rationale is that if I is dropped, the original cost O^l in I_j for all $l \leq l_j$ might be increased if I was originally used in the corresponding requests. We then multiply the original O^l values by β^l, the average increase in cost for level l when I is not present in the configuration. Since I_j was optimal for the requests it served, the values of N^l remain unchanged. The net effect is that we potentially increase the value of Δ for index I_j as a result of dropping I.
3. Adjust Δ_{min} and Δ_{max} as appropriate.

Obtaining Δ values from sub-optimal plans. When we update Δ values for index I, it is because I can optimally serve some request in the workload. Sub-optimal usages are not recorded explicitly, but can be approximated from the available information (so we get more accurate Δ values for indexes under consideration, or infer Δ values for newly considered indexes such as those resulting from index merging). To approximate Δ for an index I taking into account suboptimal usages we proceed as follows:

1. For each index I_j under consideration, find l_j, the usefulness level of I with respect to I_j. For each level $l \leq l_j$, add to Δ for I the value $O^l - \alpha_j \cdot N^l$ from I_j, where α_j=$size(I)/size(I_j)$.
2. If I is a newly considered index, find I', the most similar index to I among the considered ones using the distance function $|I \cap I'|/|I \cup I'|$. Then, subtract from I's Δ the value $(O^U - N^U)$ from I' (i.e., we approximate the update cost for I' from the most similar index).

Addressing `OR` nodes. Note that an additional kind of index interaction results from `OR` nodes in the `AND/OR` request tree. In fact, only one of the multiple requests with an `OR` parent node can be implemented in an execution plan. For this reason, every time we create an index, the Δ values of the remaining indexes that were optimal for requests that shared an `OR` parent node need to be updated. To address this issue, we maintain an additional value per index that captures the fraction of $(\sum_i N_i)$ that was generated from "shared-`OR` nodes" and update Δ values appropriately. The details, however, are omitted due to space constraints.

3.2.2 Storage Constraints

After executing an input query, there might be indexes I that should be created (i.e., indexes for which $\Delta - \Delta_{min} > B_I^s$) but no available space and no existing indexes to drop (i.e., indexes $I' \in s$ for which $\Delta_{max} - \Delta > B_{I'}^s$). We now explain how we handle this common scenario in our approach.

We define the *residual cost* of an index I under configuration s as $residual(I,s)$=$B_I^s - (\Delta_{max} - \Delta)$. If $residual(I,s) < 0$, I should be dropped from s. Otherwise, $residual(I,s)$ indicates how much slack I has before being deemed a "dropping candidate". Also, we define the *benefit* for an index $I \notin s$ as $benefit(I,s)$=$(\Delta - \Delta_{min}) - B_I^s$. Thus, if $benefit(I,s) > 0$, index I should be added. Also, positive values of $benefit(I,s)$ indicate the "excess in confidence" for adding I to s (see Figure 5 for an illustration).

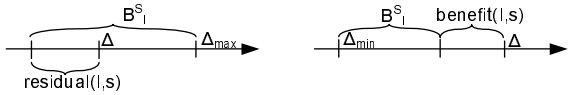


Figure 5. $Residual(I,s)$ **and** $benefit(I,s)$ **values.**

Suppose that $benefit(I,s) > 0$ for some $I \notin s$, but no space is available for creating I and, for all indexes $I' \in s, residual(I',s) > 0$ (i.e., we cannot drop any existing index). Now, if we find a subset of indexes $s' \subseteq s$ such that $\sum_{I' \in s'} residual(I',s) < benefit(I,s)$, we know that the benefit of creating I exceeds the combined slack of indexes in s'. We can then update s=$s - s' \cup \{I\}$. There might be many choices for I and s' at any time. We explain how we choose from these alternatives in the next section.

Addressing the oscillation problem. Suppose that we identify a set of indexes that are useful but do not fit in the available space. We know that, by definition, $residual(I,s)$ is upper-bounded by B_I^s for indexes $I \in s$. At the same time, $benefit(I,s)$ keeps growing for $I \notin s$ as new queries arrive. Therefore, eventually indexes that are not in s would replace indexes in s. But now, the indexes we just dropped would start increasing their $benefit$ values while the ones we just created would have a bounded $residual$ value. We are caught in an endless oscillation although the relative benefit of all indexes is similar. To address this oscillation problem, we proceed as follows. Suppose that we are updating the Δ value of some index $I \in s$ with an additional δ, but $residual(I,s)$=B_I^s. After updating Δ to Δ+δ, $\Delta_{\max}$ would also be updated appropriately and $residual(I,s)$ would stay unchanged at B_I^s. To make I's benefit explicit, in these situations we proportionally decrease Δ values of all indexes $I' \notin s$ so that the new value of $benefit(I',s)$ becomes $\max(0, benefit(I',s) - \delta)$. In other words, as current indexes $I \in s$ keep being helpful, we adjust down the benefit of the remaining indexes $I' \notin s$, thus avoiding the oscillations described above.

3.3 Putting It All Together

Figure 6 shows a pseudo-code of our online algorithm for physical design tuning. Each time a query is optimized, we generate its AND/OR request tree T and obtain the best index to implement each request. When a query is executed, we retrieve its AND/OR request tree T and update Δ values for the indexes that are not in s but optimally implement some request in T (lines 3-4). (We maintain in H the set of candidate indexes that were optimal for some request in the workload.) We also update Δ values for the indexes in s that were used to implement some request in T (lines 5-6). If the input query was an update we refine Δ values in lines 7-8. Lines 1-8 are very efficient because they only manipulate in-memory scalars. In line 9 we drop all indexes $I \in s$ for which $\Delta_{max} - \Delta > B_I^s$. In lines 10-18 we analyze the current candidate indexes and determine if we can create (and optionally drop) indexes in s. For that purpose, we initialize ITC=H and process each index in ITC. We first obtain accurate Δ values (line 13) and optionally find a subset of elements from s that, if dropped, would make enough space for I to be created. For efficiency, we periodically sort the existing indexes by $residual(I,s)/size(I)$ so that indexes that are either large or are almost dropping candidates are chosen first. In lines 15-17 we adjust the benefit of I by subtracting the combined $residual$ values from s'. If the resulting benefit is the largest seen so far, we keep I as the best candidate. Finally, we lazily generate merged indexes and include them in ITC for later analysis (line 18). After all indexes in ITC are processed, we implement the best design change (if any) in lines 19-21.

In the rest of this section, we discuss some refinements and technical details of algorithm OnlinePT.

Throttling. Although OnlinePT is efficient, it might still impose overhead to the normal DBMS execution. To mitigate this overhead, we propose to throttle OnlinePT down as follows. We always execute lines 1-8 for each query, which imposes minimal overhead and keeps the required information up-to-date. If the load on the database server increases, we execute lines 9-21 once every certain period of time, and therefore slightly delay changes in the physical design. To further decrease the server overhead, we can only consider index merges (line 18) a fraction of the executions.

Impact of online index creation. There is a period of time between the asynchronous online index creation (line 21) and the time the index is ready to be used. During this time, queries cannot use the index, but OnlinePT must understand that the index is being created and not consider it again for creation. We achieve this by removing the index from H as soon as the creation begins, so it is not considered again in ITC at the next iteration. However, we keep updating its Δ value as new queries arrive. If the $benefit$ value of the index being created drops more than B_I^s due to updates, we abort the index creation and thus save time.

OnlinePT (q_i:query, s:current configuration)

```
// Initially, H = ∅ (no candidate indexes to create)
01 AOT = getRequests(q_i)
// Update Δ values
02 for each request ρ in AOT
03   I = getBestIndex(ρ); if I ∉ s
04     H=H ∪ {I}; update Δ for I // Section 3.2.1
05   I_used = index in s used to implement ρ
06   update Δ for I_used // Sections 3.2.1 and 3.2.2
07 if q_i is INSERT/UPDATE/DELETE on table T
08   add O^U, N^U to Δ for each index over T
// Remove bad indexes
09 drop I ∈ s if residual(I,s)≤0
// Analyze candidate indexes to create
10 ITC = {I ∈ H, benefit(I,s)>0} // candidates
11 bestI = NULL; bestB = 0; bestS'=∅
12 for each index I in ITC
13   b_I = Δ − B_I^s  // see Section 3.2.1
14   get prefix s' of s in residual(I',s)/size(I')
       order such that size(s−s' ∪ {I}) fits in
       the available storage
15   b_I = b_I − Σ_{I'∈s'} residual(I',s)
16   if (b_I ≥ bestB)
17     bestI = I; bestB = b_I; bestS'= s'
18   ITC= ITC ∪{merge(I,I') : I' ∈ s ∪ ITC}
// Create indexes (optionally removing others)
19 if (bestI is not NULL)
20   drop I' ∈ s' from s
21   create bestI in s; H= H − {bestI}
```

Figure 6. Online physical tuning algorithm.

Index suspend/restart. Some DBMSs allow indexes to be "suspended" (a suspended index is not updated and cannot help query processing) and later "restarted" (which is done by propagating changes from the log to the index, which is generally faster than a full rebuild). In this scenario, every time we drop an index due to high update costs (line 9 in Figure 6) we only suspend it. The value I_B^s needs to change so that it reflects the alternative procedure to bring the index back to operational mode.

Manual intervention. DBAs might still want to manually create and drop indexes. Our algorithms enable this functionality by reusing the techniques of Section 3.2.1. For each index that is created or dropped manually, we adjust Δ values of the remaining indexes in the same way as if the physical change was done automatically.

Supporting statistics. The approximations of cost in our techniques might benefit from statistics. However, we cannot greedily trigger statistics computation due to the additional overhead. As a middle ground, we propose to trigger asynchronous statistics creation tasks on an index key

columns whenever $\Delta - \Delta_{min}$ is larger than a fraction (e.g., 0.8) of B_I^S. Thus, after we gather enough evidence about the usefulness of a given index, we create supporting statistics to have more accurate information in the near future.

Online versus traditional physical tuning. The online algorithms in this work are useful when DBAs are uncertain about the future behavior of the workload, or have no possibility of doing a comprehensive analysis or modelling. If a DBA has full information about the workload characteristics, then a static analysis and deployment by existing tools (e.g., [2, 3]) would be a better alternative.

4 Experimental Evaluation

We now report an experimental evaluation of our techniques, implemented on Microsoft SQL Server 2005. To focus on the characteristics of our algorithm, we enforce that each physical change is done before evaluating new queries (i.e., index creation and drops are synchronous).

4.1 Simple Workloads

Table 1 shows, for simple workloads, the online configuration schedules generated by `OnlinePT`, and the total execution time of both `OnlinePT` and a manually constructed optimal schedule `Opt`. We use the following notation: (i) $kE(q)_{[c]}$ represents k executions of query q, each one with expected cost c, (ii) $C(I)_{[c]}$ represents the creation of index I with cost c, and (iii) $D(I)$ represents the deletion of index I. The workloads are composed of the following queries:

q_1 = `SELECT a,b,c,id FROM R WHERE a<100`
q_2 = `SELECT a,d,e,id FROM R WHERE a<100`
q_3 = `INSERT INTO R SELECT * FROM S`

The schedules start with only primary indexes and consider the following candidate indexes:

I_1=$R(id, a, b, c)$ I_2=$R(a, b, c, id)$ I_3=$R(id, a, d, e)$
I_4=$R(a, d, e, id)$ I_5=$R(a, b, c, d, e, id)$

Table 1 starts with workload W_1=$250q_1; 250q_2$ (i.e., 250 instances of q_1 followed by 250 instances of q_2). The total space for the database is 135 MB, which is just enough for a single 4-column index. We start executing q_1 five times at cost 0.57. For this query, both I_1 and I_2 are useful (I_1 as a vertical partition for a scan request, and I_2 as a better overall alternative for a seek request). Note also that the cost to create I_1 (1.33) is significantly smaller than that of I_2 (8.96) because I_1 shares the same key columns with the primary index and therefore no intermediate sort is necessary. After five executions of q_1, the benefit of creating I_1 is larger than its creation cost of 1.33, so we create I_1. But q_1 can still be improved by index I_2. In fact, after 38 executions at 0.29 cost, the benefit of I_2 is larger than its creation cost plus the residual benefit for the existing I_1, so we drop I_1 and create I_2. The remaining 207 executions of q_1 cost only 0.09. Right after that, query q_4 starts executing, and after 24 executions with cost 0.57, the benefit of index I_4 (over table S) is larger than its creation cost plus the residual cost of I_2, so we swap I_4 and I_2. The remaining 226 instances of q_4 are executed at cost 0.09.

The next three schedules in Table 1 correspond to workload W_2=$250[q_1; q_2]$ (i.e., 250 interleaved executions of q_1 and q_2). While 135MB only allow one 4-column index to be created (i.e., I_5 is too large), 138MB allows any index (but only one) to be created, and 150MB allow multiple indexes to be created. For 135MB, we start executing $(q_1; q_2)$ until we create I_1 which helps q_1. Index I_2 starts increasing its benefit with respect to I_1 and at some point replaces I_1. From this point on, the schedule executes q_1 at only 0.09, and q_2 at the original cost 0.57 (the relative benefits of indexes for q_2 are roughly the same to the corresponding ones for I_1, so the schedule does not change further and avoids oscillations). The overall cost is then 180.1. In contrast, if 138MB are available, the schedule starts similarly, but instead of changing I_1 by I_2, index merging produces I_5 which serves both queries simultaneously. The remaining 237 executions of (q_1, q_2) cost 0.11 for each query, and the overall cost is reduced to just 79.71. Finally, when $150MB$ are available, both I_1 and I_3 are created initially, and after creating the merged index I_5 we are able to additionally create optimal indexes for both q_1 and q_2 (at small cost, since I_5 avoids intermediate sorts to create I_2 and I_4). The remaining executions of (q_1, q_2) cost 0.09 for each query, and the overall cost is still smaller at 75.36.

The last schedule in the figure shows a workloads with updates. Specifically, W_3=$100q_1; 100q_3$. In this case, the schedule starts similarly to previous cases and creates indexes I_1 and I_2 to help q_1. When q_3 starts executing at cost 3.17, the benefits of the existing indexes decrease, and after 4 executions we drop I_1. The cost of q_1 decreases to 2.32, and after 22 additional executions we drop I_2 (it takes longer to drop I_2 because its creation time is larger). The remaining 74 executions of q_3 cost only 1.47, since only the primary index is updated, and the overall cost is 199.92.

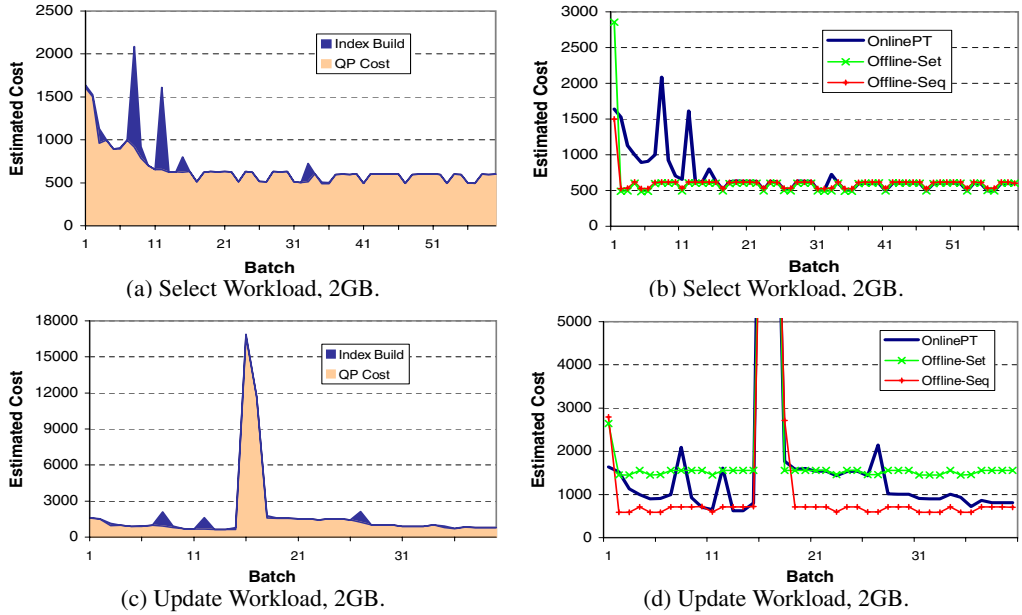
4.2 Complex Workloads

We generated random `TPC-H` workloads by appending multiple *batches* together. Each *batch* consists of a permutation of a random instance of all the `TPC-H` queries. We then evaluated the workloads using (i) our algorithm `OnlinePT`, (ii) an offline, set-based tool `Offline-Set` [3], and (iii) the offline "sequence-based" `Offline-Seq` of [2].

Figure 7(a) shows the cost of executing a 60-batch workload with `OnlinePT` on a $1GB$ database with an additional $1GB$ of storage for indexes. As batches are executed, the query processing cost decreases as better indexes are built. Between batches 1 and 15 there is more activity due to physical design, and several indexes are created and sometimes later dropped. The physical design stabilizes over time, and after batch 15 virtually no further changes are neces-

Workload	Online Configuration Schedule	$C_{onlinePT}$[$Cost_{opt}$]
W_1, 135MB	$5E(q_1)_{[0.57]}; C(I_1)_{[1.33]}; 31E(q_1)_{[0.29]}; C(I_2)_{[8.69]}; 214E(q_1)_{[0.09]}; 24E(q_2)_{[0.57]}; D(I_2);$ $C(I_4)_{[8.96]}; 226E(q_2)_{[0.09]}$	85.77 [62.92]
W_2, 135MB	$4E(q_1;q_2)_{[0.57;0.57]}; C(I_1)_{[1.33]}; 14E(q_1;q_2)_{[0.29;0.57]}; D(I_1); C(I_2)_{[8.96]}; 232E(q_1;q_2)_{[0.09;0.57]}$	180.01 [173.96]
W_2, 138MB	$4E(q_1;q_2)_{[0.57;0.57]}; C(I_1)_{[1.33]}; 9E(q_1;q_2)_{[0.57;0.29]}; D(I_1); C(I_5)_{[9.2]}; 237E(q_1;q_2)_{[0.12;0.12]}$	79.71 [69.21]
W_2, 150MB	$4E(q_1;q_2)_{[0.57;0.57]}; C(I_1)_{[1.33]}; C(I_3)_{[1.33]}; 30E(q_1;q_2)_{[0.29;0.29]}; C(I_5)_{[9.2]}; E(q_1)_{[0.12]};$ $C(I_2)_{[1.2]}; E(q_2)_{[0.12]}; C(I_4)_{[1.2]}; 215E(q_1;q_2)_{[0.09;0.09]}$	75.16 [56.86]
W_3, 150MB	$5E(q_1)_{[0.57]}; C(I_1)_{[1.33]}; 30E(q_1)_{[0.29]}; C(I_2)_{[8.69]}; 65E(q_1)_{[0.09]}; 4E(q_3)_{[3.17]}; D(I_1);$ $22E(q_3)_{[2.32]}; D(I_2); 74E(q_3)_{[1.47]}$	199.92 [164.69]

Table 1. Configuration schedules for simple workloads.



(a) Select Workload, 2GB. (b) Select Workload, 2GB.

(c) Update Workload, 2GB. (d) Update Workload, 2GB.

Figure 7. Online and Offline schedules for complex `TPC-H` workloads.

sary. Figure 7(b) contrasts the overall cost of `OnlinePT` with that of the offline alternatives. `Offline-set` incurs a large overhead at the beginning to create all the necessary indexes, but then serves the workload without further tuning. `Offline-seq` is slightly more efficient than `Offline-set` because it treats the input as a sequence. Note that when the physical design for `OnlinePT` stabilizes, the cost per batch is almost indistinguishable from that of offline techniques (i.e., `OnlinePT` *does no harm* compared to offline techniques that have knowledge of the whole workload in advance).

Figures 7(c) and 7(d) repeat the experiment when we add a disruptive sequence of updates after the first 14 batches in the original workload. This scenario is particularly attractive for sequence-based approaches (i.e., `OnlinePT` and `Offline-Seq`). Since the disruptive updates mostly touch the `lineitem` table, `Offline-Set` does not recommend any index on `lineitem` because the (aggregated) benefit of such indexes is outweighed by the cost of updating them. In contrast, `OnlinePT` and `Offline-Seq` generate schedules that begin similar to the ones in Figure 7(a). At batch 14, the algorithms drop the indexes on `lineitem` and efficiently process the updates (`OnlinePT` lags behind the offline `Offline-Seq`). When the normal sequence of batches is resumed, indexes on `lineitem` are re-created and exploited again. In this case, `OnlinePT` is more efficient than `Offline-Set` (even when counting the index creation time).

Figure 8 summarizes the overall cost of all techniques for the different workloads. As expected, `OnlinePT` is slightly worse than `Offline-Seq`, which knows the future. However, we see that `OnlinePT` sometimes results in better performance than the offline, set-based technique `Offline-Set`.

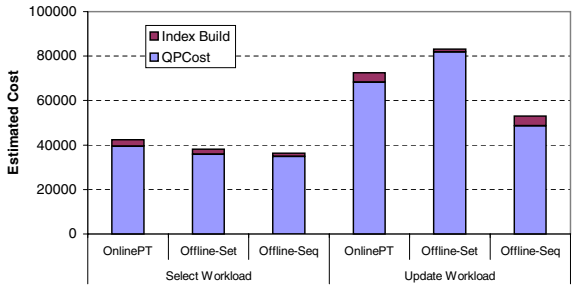


Figure 8. Expected cost of tuning alternatives.

4.3 Server Overhead

We now report the overhead of our prototype implementation of algorithm `OnlinePT`. Figure 9 shows the average elapsed time of `OnlinePT` when evaluating the different workloads in the previous sections. We also show the average overhead fraction over the minimum between optimization and execution time of the queries in the workload (to use the worst case scenario for `OnlinePT`). The full algorithm requires an additional 1.7% (0.41% for the simpler workload) on top of query optimization/execution, which is rather acceptable. In fact, the critical section of the algorithm (lines 1-8) imposes less that 0.2% overhead over query processing (as we discuss in Section 3.3, this is the only module that must be executed for each query – the others can be throttled down). Our preliminary evaluation demonstrates the promise of our online tuning approach.

Module	TPC-H ($\|W\| = 640$)	Simple ($\|W\| = 500$)
Total	19.8msecs (1.7%)	0.12msecs (0.82%)
Line 1	0.77msecs (0.05%)	0.1msecs (0.65%)
Lines 2-8	0.16msecs (0.01%)	0.02msecs (0.13%)
Lines 9-18	18.9msecs (1.6%)	0.012msecs (0.08%)
Line 18	15.5msecs (1.3%)	0.002msecs (0%)

Figure 9. Server overhead of `OnlinePT`.

5 Related Work

There has been considerable research on automating the physical design in DBMSs (e.g., [1, 7, 3, 8, 11]). In contrast to our work, they focus exclusively in the offline scenario.

References [4, 6] introduce the underlying functionality that we review in Section 2 and use in our approach. In contrast to our work, these references exploit such technology in the context of an offline tuning tool [4] and an alerting mechanism that provides lower bounds for the improvement of a comprehensive tuning tool [6].

Recently, reference [2] presents an offline technique that finds the optimal physical schedule considering the workload as a sequence. While it shares the same problem formulation with our paper, it presents an offline algorithm to help application developers writing code that takes into account physical design changes.

Finally, references [9, 10] present brief descriptions of prototypes that address some aspects of online physical tuning. These references share the same overall goals with our work, but differ in significant ways. Specifically, in contrast to our approach, reference [10] requires multiple additional calls to the optimizer to obtain information about candidate indexes, and [9] is not fully integrated with the query optimizer. In addition, both references ignore the issue of index interactions (Section 3.2.1) and do not explicitly provide solutions for the oscillation problem.

6 Conclusions

We introduced an online algorithm designed to be always-on and able to continuously monitor changes in the workload and data to refine the physical design. Such architecture is especially useful when the DBMS needs to adapt to unexpected changes in workload characteristics. We presented proposals that address difficult issues such as index interactions and unwanted physical design oscillations. Another key challenge we tackled was to ensure low overhead on query execution, with promising results. We plan to build an industrial strength implementation for a broader understanding of the scope and limitations of our approach and refine the architecture appropriately.

References

[1] 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.

[2] S. Agrawal, E. Chu, and V. Narasayya. Automatic physical design tuning: workload as a sequence. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2006.

[3] S. Agrawal et al. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*, 2004.

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

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

[6] N. Bruno and S. Chaudhuri. To tune or not to tune? A Lightweight Physical Design Alerter. In *Proceedings of the 32th International Conference on Very Large Databases*, 2006.

[7] 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.

[8] B. Dageville et al. Automatic SQL Tuning in Oracle 10g. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*, 2004.

[9] K.-U. Sattler, I. Geist, and E. Schallehn. Quiet: Continuous query-driven index tuning. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2003.

[10] K. Schnaitter et al. Colt: continuous on-line tuning. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2006.

[11] 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.