Online AutoAdmin (Physical Design Tuning)

Nicolas Bruno
Microsoft Research
nicolasb@microsoft.com

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

ABSTRACT

Existing solutions for the automated physical design problem require explicit invocations of tuning tools and critically depend on DBAs gathering representative workloads manually. In this demonstration, we show an alternative approach to the physical design problem. Specifically, we demonstrate a novel monitoring/tuning DBMS component that we prototyped in Microsoft SQL Server 2005 as a server-side extension. This component is always-on and continuously modifies the current physical design reacting to varying workload or data characteristics. Our solution imposes low overhead and takes into account storage constraints, update statements, and the cost to create physical structures.

Categories and Subject Descriptors

H.2.2 [**Physical Design**]: Access Methods

General Terms

Algorithms, Design

Keywords

Continuous tuning, Physical Design, Online algorithms

1. INTRODUCTION

Database applications have become increasingly complex and varied. Presently, most database vendors (e.g., [1, 4, 5]) 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, but still leave several significant decisions to DBAs. Specifically, DBAs need to continuously monitor and diagnose when to re-tune physical designs. Furthermore, to tune the database using these tools, DBAs need to explicitly gather representative workloads. Finally, DBAs need to decide when to deploy recommendations.

The above tasks are difficult and in fact are becoming more problematic. Consider, as an increasingly common example, large installations that support multiple, intermittent database applications (e.g., some ISPs provide such backend service already). It is common that these hosted applications come and go, and usually exhibit unexpected spikes in their loads. In such cases, the hosting installation is best served if it can use its resources to accommodate the spikes. In terms of physical design, this entails perhaps building redundant structures for one application and subsequently redirecting resources to another application as the load pattern changes dynamically. 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. The most popular usage of such a database is querying/browsing (`select` load), but the usage pattern completely changes on specific days when the test team finds and inserts large numbers of bugs (`update` load), e.g., on a bug-bash day. If we gather a representative workload over, say, a month, chances are that no index is globally useful for the bug database, as the gains in query processing are outweighed by the update costs during bug-bash periods. It is very difficult to explicitly and statically model the workload in these scenarios, and equally difficult to decide when to tune the database and deploy the resulting recommendations.

Since DBMSs support building *online indexes* while allowing query processing to continue, it is important to seek an even more automated solution to the physical design problem. There are, however, new and significant challenges to address. First, fully automated solutions need to be *always-on*, continuously monitoring changes in both the workload and the database state, and refining the physical design as needed. Note that this requires such solutions to have low overhead. Additionally, in contrast to current approaches, fully automated solutions must also balance the cost of transitioning between index configurations and the potential benefits of such indexes for the future workload. In particular, while 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 react fast to significant workload changes.

The online nature of our problem implies that we would lag behind optimal offline solutions "that know the future". However, by carefully reacting to changes, we ensure that we do not suffer disproportionately although we do not know the future. Thus, we can bound the amount of loss that we incur. Our work was done as part of the AutoAdmin project at Microsoft Research (see details at `http://research.microsoft.com/dmx/autoadmin/`) and the technical details of our approach are explained in [3].

In Section 2 we present an architectural overview of our solution, which we prototyped in Microsoft SQL Server 2005. In Section 3 we provide additional details on the internal algorithms. Finally, in Section 4 we give some examples that would be showcased as part of the demonstration.

SIGMOD'07, June 11–14, 2007, Beijing, China.
ACM 978-1-59593-686-8/07/0006.

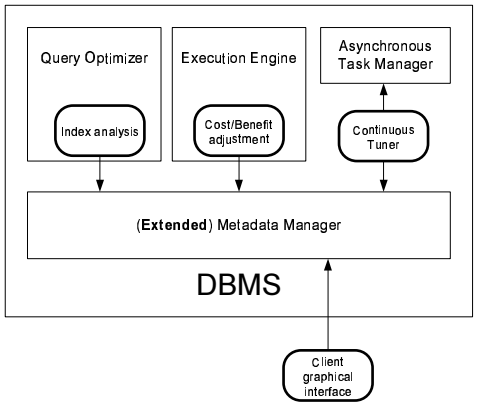

Figure 1: Continuous Tuning Architecture in a DBMS.

2. ARCHITECTURE

Figure 1 highlights the components in the database server that were modified or added to support online physical design tuning. Specifically, we use an extended metadata manager that additionally supports "candidate hypothetical indexes" which are not materialized (and therefore do not help during normal query processing), but are placeholders for tracking the benefit of alternatives. Additionally, we added to the internal representation of indexes (for both real and candidate hypothetical indexes) a small set of counters that are used to track the benefit of each alternative. Initially, when a new query is optimized, we piggyback on the optimization call and quickly identify a relevant set of candidate indexes that could improve performance (*index analysis* in the figure). For that purpose, we use *AND/OR request trees* and local transformations (see Section 3) and produce a compact set of updates to the index counters, to be performed every time the query is evaluated. Subsequently, during query execution, we leverage the preprocessing done during optimization and efficiently track the potential benefits that we lose by not having the candidate indexes materialized in addition to measuring the utility of existing indexes (*cost/benefit adjustment* in the figure). After each query (or after a variable number of query executions, if we need to throttle down the tuning process) we analyze the cost/benefit ratio and determine whether creating or dropping indexes would be beneficial (*continuous tuner* in Figure 1). If a design change is beneficial, we send the index creation or deletion request to an asynchronous task manager in Microsoft SQL Server, which would process the `DDL` at a later time based on system policies. Advanced DBAs can monitor the internal state of the continuous tuner by using a client application.

3. ALGORITHMIC DETAILS

We now briefly provide additional information on the internal algorithms of our solutions (due to space constraints our presentation is very brief; see [3] for more details).

We gather information during optimization (*index analysis* in Figure 1) by instrumenting the optimizer and intercepting optimization rules that generate *access-path-requests*. Such requests encode the logical properties of any physical plan that is able to implement the sub-tree rooted at the corresponding operator. Figure 2 shows an example tagged execution plan and the corresponding `AND/OR` tree for the following query:

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

The requests generated during optimization allow us to make inferences about execution plans for varying physical designs, and do so without issuing additional optimization calls. The idea is as

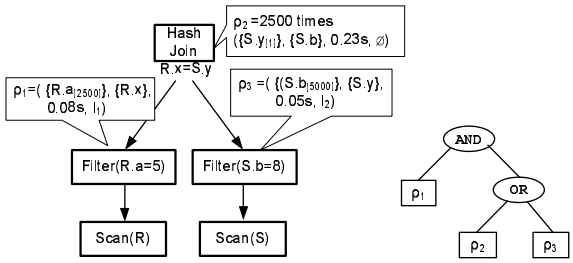

Figure 2: Execution plan and `AND/OR` request tree.

follows: 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 logically equivalent to the original one. We know the cost of the original sub-plan (it was obtained during regular optimization) and can calculate the cost of the newly generated alternative using p. Therefore, we can infer how much would the original execution plan improve or degrade if we substituted the given sub-tree with the logically equivalent physical plan using p (exploiting local replacement). Thus, after optimization we identify a set of candidate indexes for the given query, and track the benefit that we lose by not having these materialized, as well as the utility of the current existing indexes (we use Δ to denote these benefit/utility values).

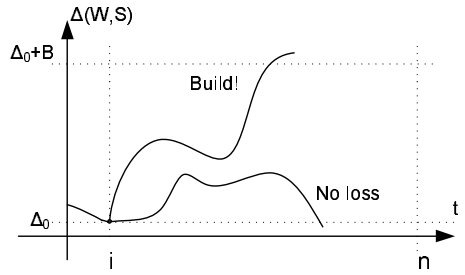

Figure 3: Tracking benefit/utility values for indexes.

Our approach then is to track the aggregated Δ values as queries are executed for every candidate index under consideration (the set of "candidate indexes" itself varies over time depending on the workload). Figure 3 shows two examples of aggregated Δ values over time for a given index. In the figure, we denote by B the cost of creating the corresponding index. Intuitively, if the aggregated benefit of having a candidate index exceeds its creation cost, we trigger an online creation of such index, since we gathered enough evidence that the index is useful. In contrast, if the benefit of having an index oscillates between some value Δ_0 and $\Delta_0 + B$, we can confidently avoid creating the index, since the benefits are not significant enough (in this way, we can also bound how much we lag behind an optimal solution). By using this line of reasoning, we can obtain a strategy that is 3-competitive (i.e., no more than 3 times worse than the optimal strategy) for restricted scenarios [3]. However, we extend this core technique with heuristics that tackle the following additional challenges:

Interactions. Consider indexes I_1=(a, b, c) and I_2=(a, b, c, d). If we do not consider the inherent 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. Also, `OR` trees in the

AND/OR request tree must be handled carefully, since any execution plan can only take advantage of one of the sub-trees. Naive strategies would overestimate the benefit of indexes below OR nodes.

Storage constraints. Additionally, if there is a storage constraint, we might not be able to create all the required indexes. In those 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 it is somewhat useful to free up space for better alternatives, and (iii) consider index merging [2] to obtain additional indexes that might better trade off space and efficiency.

Reference [3] explains in detail the properties, challenges, and solutions mentioned above, which we omit due to space constraints.

4. THE DEMONSTRATION

In this demonstration, we showcase the integrated continuous tuning feature on a prototype built on top of Microsoft SQL Server 2005. We present scenarios that highlight the different challenges (discussed in earlier sections) and how we address those in our solution. We will exploit the GUI client tool for this demonstration.

The following examples are representative of tuning sessions that will be shown during the demonstration. Table 1 shows, for a few simple workloads, the online configuration schedules generated by our solution, and its total execution time (for illustration purposes, we also manually calculated the optimal schedule and evaluated its cost). We use the following notations: (i) $kE(q)_{[c]}$ represents k executions of query q, each one with 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 contain 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`

and 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=$R(a, b, c, d, e, id)$

Workload	Online Configuration Schedule	C_{online}
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 [Opt=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 [Opt=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 [Opt=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 [Opt=56.86]

Table 1: Configuration schedules for simple workloads.

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. Subsequent executions of q_1 cost only 0.29, but q_1 can still be improved by index I_2. After such 38 executions, 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 the table 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 storage is 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 demonstration will also present complex workloads (such as Figure 4 that shows the cost of a typical schedule over a complex TPC-H workload with select and update queries). Thus, the demonstration will highlight the key aspects of the online AutoAdmin physical design tuning component showing how it adjusts to changing workload patterns with low overhead while paying close attention to important systems issues such oscillations, index interactions, and storage bounds.

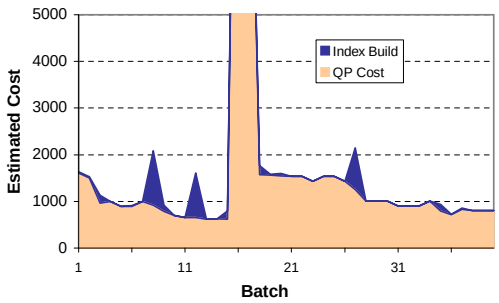

Figure 4: Online schedule for a complex TPC-H workload.

5. REFERENCES

[1] S. Agrawal et al. Database Tuning Advisor for Microsoft SQL Server 2005.

[2] N. Bruno and S. Chaudhuri. Physical design refinement: The "Merge-Reduce" approach. In *In Proceedings of EDBT*, 2006.

[3] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *In Proceedings of ICDE*, 2007.

[4] B. Dageville et al. Automatic SQL Tuning in Oracle 10g. In *In Proceedings of VLDB*, 2004.

[5] D. Zilio et al. DB2 design advisor: Integrated automatic physical database design. In *In Proceedings of VLDB*, 2004.