Automated SQL Tuning through Trial and (Sometimes) Error

Herodotos Herodotou
Duke University
hero@cs.duke.edu

Shivnath Babu
Duke University
shivnath@cs.duke.edu

ABSTRACT

SQL tuning—the attempt to improve a poorly-performing execution plan produced by the database query optimizer—is a critical aspect of database performance tuning. Ironically, as commercial databases strive to improve on the manageability front, SQL tuning is becoming more of a black art. It requires a high level of expertise in areas like (i) query optimization, run-time execution of query plan operators, configuration parameter settings, and other database internals; (ii) identification of missing indexes and other access structures; (iii) statistics maintained about the data; and (iv) characteristics of the underlying storage system. Since database systems, their workloads, and the data that they manage are not getting any simpler, database users and administrators often rely on trial and error for SQL tuning.

In this paper, we take the position that the trial-and-error (or, *experiment-driven*) process of SQL tuning can be automated by the database system in an efficient manner; freeing the user or administrator from this burden in most cases. A number of current approaches to SQL tuning indeed take an experiment-driven approach. We are prototyping a tool, called *zTuned*, that automates experiment-driven SQL tuning. This paper describes the design choices in zTuned to address three nontrivial issues: (i) how is the SQL tuning logic integrated with the regular query optimizer, (ii) how to plan the experiments to conduct so that a satisfactory (new) plan can be found quickly, and (iii) how to conduct experiments with minimal impact on the user-facing production workload. We conclude with a preliminary empirical evaluation and outline promising new directions in automated SQL tuning.

1. INTRODUCTION

SQL tuning—the attempt to improve a poorly-performing execution plan produced by the database query optimizer—is a critical aspect of database performance tuning. The rapid evolution of processors, storage systems, and data access patterns are causing estimates from traditional cost models to be increasingly inaccurate, leading to the selection of suboptimal execution plans by the optimizer. Even when the database system is well tuned, workloads and business needs change over time and the system has to be kept in step. New optimizer statistics, configuration parameter changes, software upgrades, and hardware changes are among a number of factors that can cause a query optimizer to change execution plans, perhaps with much worse performance than before [13].

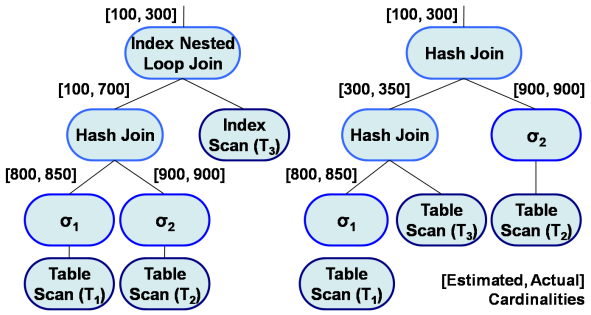

Figure 1: Execution Plans (a) Current (b) Final

The success of SQL tuning usually depends on the expertise of skilled database administrators (*DBA*s) or laborious trial-and-error steps. When called to tune a query, a DBA may use her experience, intuition, knowledge of the data being queried, tips from tuning manuals, or even guesses to complete the task. Initially, the DBA may collect some monitoring data on the production database in an attempt to diagnose the problem. However, data collection can increase the load on an already under-performing database, forcing the DBA to shift to a test database. The DBA's usual course of action would be:

1. Create a replica of the production environment on the test database to ensure that the query optimizer will select the same execution plan for the offending query.
2. Collect monitoring data to diagnose the problem. This step includes getting more insight into which plan was selected by the optimizer and why. The DBA will probably need to execute the query to collect data like the estimated and actual input cardinality values for operators in the plan. For example, suppose the current execution plan selected by the optimizer for a poorly-performing query is the one shown in Figure 1(a). The estimated and actual input cardinality values are shown for each operator. Based on this data, the DBA can see that the optimizer underestimated

DBTest'09, June 29, 2009, Providence, Rhode Island, USA.

the number of tuples the index nested loop join has to process—caused, perhaps, by a wrong assumption of independence between the filters σ_1 and σ_2.

3. Based on the observations, form hypotheses regarding potential fixes to the problem. In our example, one such hypothesis is that replacing the index nested loop join in Figure 1(a) with a hash join will fix the problem.

4. Do further runs of specific plans or subplans to refine or confirm the hypotheses. In our example, the DBA may give *hints* to the optimizer in order to force it to select and run a plan with no index nested loop join.

5. Finding a satisfactory fix may require much trial and error on the DBA's part. In our example, replacing the index nested loop join with a hash join may not solve the full problem. It may be necessary to use the plan shown in Figure 1(b) which has a different join order. Once a promising fix is found, a careful validation has to be done to ensure that the fix will work on the production system.

1.1 Automated SQL Tuning as an Experiment-Driven Workflow

Despite advances in individual steps, the above trial-and-error (or *experiment-driven*) process is very common, labor intensive, and requires an extensive knowledge of database internals. There is a lot to be gained from automating this process. The entire process can be represented by the *workflow* shown in Figure 2.

Intuitively, an experiment is an action that involves some cost but brings in useful information (in this case for SQL tuning). An experiment could be the execution of a particular query, execution of an alternative plan or a subplan for the query, costing of a plan using newly observed cardinality values, random sampling from a base table or a join, and so on. Section 2 gives more examples from commercial databases of the types of experiments used in SQL tuning.

Query optimizers select plans based on models of query execution. Even the best models leave parts of reality uncovered [1]. Thus, conducting experiments to probe reality is unavoidable in SQL tuning. Quoting researchers from Oracle [13]: "it is almost impossible to predict the impact of such changes on query performance before actually trying them". Here, "such changes" refer to actions like updating the statistics about the data, providing hints to the query optimizer, changes to the database software or hardware, and others.

The first and most critical step in the SQL-tuning workflow involves generating an *experiment design*, i.e., a sequence of experiments that will lead to a satisfactory execution plan. The total space of experiments is extremely large to cover in full. Thus, our goal is to choose a smart ordering of experiments to conduct in order to reach a satisfactory plan as fast as possible. This problem is very challenging. We will discuss how tools from three leading database vendors as well as zTuned address this problem.

Experiments may be run before or after the database goes into production use. Once the database is in production use, experiments can be run on: (i) the production database running the user-facing workload, (ii) the standby databases backing up the production database, or (iii) the test database used by DBAs and developers. Running experiments on the user-facing production database is a risky proposition unless the impact of experiments can be bounded or they do not make excessive use of database resources. Most DBAs will remain fearful of running experiments on the user-facing database. We have developed a comprehensive new framework (described in detail in [6, 2]) for running online experiments, by exploiting underutilized resources in the primary or standby database.

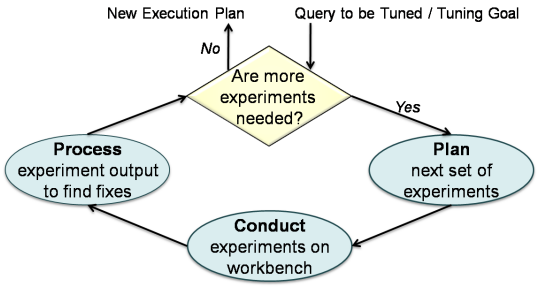

Figure 2: Experiment-driven Workflow that abstracts SQL Tuning

Finally, the output from conducted experiments is processed to decide the next step. Analysis of the output could lead either to a new and satisfactory query execution plan (thereby completing the tuning task) or to the design of the next set of experiments.

1.2 Contributions

We make the following contributions in this short paper:

- We formulate the problem of automating SQL tuning using an experiment-driven approach. Section 2 shows how current approaches for SQL tuning fit this framework.
- We introduce *Explain_Plan*, a new command in PostgreSQL that enables physical execution plans for queries to be costed and executed (if needed). Explain_Plan is discussed in Section 3.
- We develop automated algorithms for planning experiments aimed at improving a poorly-performing query plan. Section 4 discusses the planning process.
- Section 5 describes *zTuned*, a system that formalizes and automates SQL tuning using an experiment-driven approach. A preliminary empirical evaluation of zTuned is provided in Section 6.

2. CURRENT APPROACHES

Query execution feedback is a technique used in [4, 5, 12] to improve the quality of plans by correcting cardinality estimation mistakes made by the query optimizer. LEO's approach [12] consists of recording the number of rows produced by each operator during the execution of a particular query, and then relaying the new information back to the query optimizer. The *pay-as-you-go* framework [4] proposed more proactive monitoring mechanisms and plan-modification techniques for gathering additional cardinality information from a given execution plan. This information might then lead to the selection of better execution plans for future queries.

Experiments in [4] and [12] comprise: (i) the recording of cardinalities for various expressions while query plans are running, and (ii) running the new plans picked by the optimizer for queries once the recorded cardinalities are incorporated into query optimization. All experiments are conducted on the production database system; a limitation of

these approaches because of the need to maintain low overhead on the production system.

A major source of concern for DBAs when using approaches like [4] and [12] is the possible performance regression of plans for user-facing queries. As shown in [9], incorporating some actual cardinalities alongside estimated cardinalities can bias conventional query optimizers towards picking plans whose performance is highly unpredictable. This phenomenon is called "fleeing from knowledge to ignorance" in [9]. Furthermore, the cardinality of expressions that do not correspond to a subplan of the current (or slightly modified) plan cannot be easily obtained through monitoring. In contrast, well-designed experiments in zTuned can target the collection of any desired statistics, and thus explore different parts of the execution plan space.

Reference [7] describes some new tuning tools introduced with Oracle Database 11g. These tools enable the database to monitor and diagnose itself on an ongoing basis, and alert the DBA when it finds any problems. The *Automatic Tuning Optimizer* is a new mode of the optimizer that is specifically used during designated maintenance sessions. When run in this mode, the optimizer generates additional information through sampling (which forms the experiments in this case) that can be used during plan selection in the future. Like zTuned [6, 2], Oracle 11g provides some novel mechanisms to limit the impact of experiments on the user-facing workload.

The tuning optimizer's experiment planning process targets statistics that are considered critical to the selection of good plans. First, random samples are collected from the base tables to validate the cardinalities of filter predicates; since cardinality errors early in the execution tree might propagate up in a very negative way. Given more time, the tuning optimizer performs random sampling to validate join cardinalities, starting with two-way joins and proceeding step-by-step to n-way joins. In addition, Oracle acknowledges the need to verify the effect of the new information collected, and performs experimental validations of plan performance. zTuned's overall goals are similar, but the approach used to explore the space of plans based on experiments is very different (see Section 4).

3. INTERFACING WITH THE QUERY OPTIMIZER

In an attempt to correct the mistakes of the query optimizer, DBAs often need to manually experiment with different execution plans. First, DBAs need to find out the plan selected by the optimizer along with the cardinality estimates that led to the selection. For example, PostgreSQL has a very useful command, called *Explain*, that can be used to display the selected execution plan. The output includes all operators present in the plan along with other useful information per operator like: (a) the estimated startup time[1] before the first row can be returned, (b) the estimated total time to return all rows, and (c) the estimated number of rows returned. The Explain command offers an additional option, called *analyze*, that causes the statement to be executed, not only planned. The output then also contains the actual running times and cardinalities alongside the estimated ones—like in Figure 1—which makes it easy to spot estimation errors.

Query hinting is a mechanism used in most database systems to enable DBAs to influence the choice of query execution plans. However, this support comes in different forms. PostgreSQL has a coarse hinting mechanism [11] that limits experienced DBAs from fine-tuning a poorly-performing query. All hints are in terms of enabling or disabling particular operators. For example, setting the parameter *enable_hashjoin* to false would disable the use of hash joins in a particular query. Hence, for a plan with two hash joins, there is no way to force one join to be a merge join and the other join to remain a hash join.

The hinting mechanism in SQL Server is more flexible since it includes support for specifying an access path for a table (e.g., to force an index scan on the table) and also the ability to force a join order for all tables that appear in the query [10]. However, it has the same problem with join specification as PostgreSQL. It is worth noting some very recent work on query hinting, called *Power Hints* [3], that targets a generic hinting framework. Oracle hints [7] can specify the first join to be used in the execution plan as well as the join operator to be used for a particular pair of tables. Even though Oracle hints can constrain particular joins, they do not support any finer scopes over an arbitrary set of tables. IBM DB2 offers a different approach to query hinting through the creation of a particular table called PLAN_TABLE [8]. Hints are given to the query optimizer in the form of SQL queries over the PLAN_TABLE, and can be used at the scope of a subquery. Hints in Sybase Adaptive Server Enterprise [1] are specified through Abstract Plans that implement a physical-level relational algebra.

The Explain command and query hints allow DBAs to reason at the level of queries. However, SQL tuning also requires DBAs to reason at the level of plans. Based on the insight they acquire, DBAs need the ability to execute alternative plans in an easy way. The current hinting mechanism in PostgreSQL is inadequate for this purpose. In order to alleviate this problem, we developed a new command called *Explain_Plan*. This command is similar to the Explain command, but the input also includes a string representation of a query execution plan. The command can be used for:

- Costing any valid plan for a particular query using the database cost models and estimated cardinality values.
- Costing a plan while providing some or all cardinality values as input. For example, if the DBA knows based on execution history that a join will produce 42 tuples, then she can specify it in the input. This data will be used instead of the optimizer's estimates during costing.
- Executing a plan to compare estimated and actual running times and cardinalities for each plan operator.

This command is extremely useful in many situations and is discussed further in Section 7. In addition, it is used by our automated SQL tuning tool for conducting experiments as described in Section 5.

4. PLAN SPACE EXPLORATION

The Explain_Plan command can be used to fully specify an execution plan for a particular query, and then cost it or execute it. The next step is to find a way to generate alternative plans in a structured and systematic way. Our goals here are twofold. First, when we generate plans that

[1]Time is measured in terms of disk page fetches

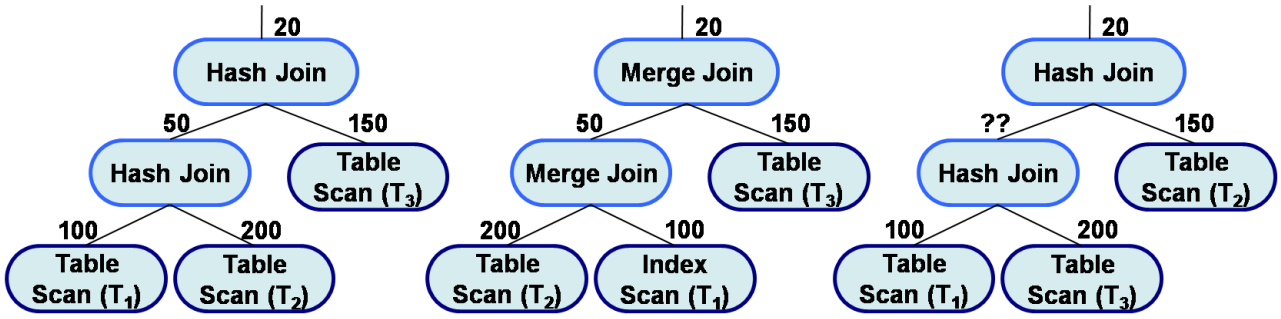

Figure 3: Execution Plans (a) Original (b) Same Neighborhood (c) Different Neighborhood

are similar to each other and cost them, we want to maximize the use of all available information. Second, we want the ability to generate plans that are very different from the current plan, in order to explore different parts of the plan space.

Consider the plan shown in Figure 3(a). The numbers above each operator are the number of tuples produced by the operator, i.e., the cardinality for that operator. Having these values allows us to use the database cost models to accurately estimate the cost of each operator. For example, in order to calculate the cost of a hash join, we would use the cardinalities of the two child operators. If the hash join is changed to a merge join to generate a new plan, then we can cost this new plan accurately as well since we know all the cardinalities required.

We call the set of cardinality values that are needed for costing a particular plan the *Cardinality Set* of that plan. We then define the *Plan Neighborhood* as the set of all plans that are associated with the same cardinality set. In more practical terms, two plans belong in the same neighborhood if when we know the cardinalities for all operators in the first plan, we can induce the cardinalities for all operators in the second plan. *This allows us to maximize the use of the cardinality values we have obtained from running a single plan P, as we can use these cardinalities to cost all plans in P's neighborhood.*

Consider the plans shown in Figure 3 and assume they are all valid for the same query. The first two plans belong to the same neighborhood since if we had the cardinalities from the plan in Figure 3(a), we would know exactly what the cardinalities are for the plan in Figure 3(b). Even though the plans use different operators to join tables T_1 and T_2, the number of output tuples produced will be the same.

Comparing the two plans from Figures 3(a) and 3(c), we see that the only change is in the join order of tables T_2 and T_3. However, this change causes the plan in Figure 3(c) to belong to a different neighborhood. Given the cardinality estimates from the plan in Figure 3(a), we can induce the cardinalities for all the scans and the upper join, but we cannot draw any conclusions about the cardinality of the join over tables T_1 and T_3.

Given a particular execution plan P, we need a mechanism to generate a set of different plans that belong to the same neighborhood as P. For this purpose, we created a set of transformation rules which we call *intra-transformations*. These are operator transformations that can be applied to a single node in the operator tree, in order to generate a different execution plan within the same neighborhood.

Three general categories of intra-transformations have been identified: (a) the *Scan Operator Transformations* are used to transform a particular scan operator to a different one (e.g., transform a sequential scan into an index scan); (b) the *Join Operator Transformations* are used to transform a particular join operator to a different one (e.g., transform a hash join into a merge join); and, (c) the *Single Join Order Transformations* swap the order of the outer and inner subplans of a particular join (e.g., transform $A \bowtie B$ into $B \bowtie A$, where A and B represent subplans).

Intra-transformations are used to generate plans within a particular neighborhood. However, in order to explore the full plan space, we also need to generate plans that belong to different neighborhoods. For this purpose, we defined a new set of transformation rules which we call *inter-transformations*. These are transformations that can be applied across multiple operators, and produce execution plans that belong to different neighborhoods.

Inter-transformations affect the structure of the execution plan and thus can produce logically-equivalent plans that are very different from the original plan. In particular, we consider join order changes across multiple joins. For example, a join sequence $((A \bowtie B) \bowtie C)$ can be transformed into $(A \bowtie (B \bowtie C))$ to produce a new execution plan that belongs to a different neighborhood.

5. ZTUNED: AUTOMATED SQL TUNING

zTuned is a system that formalizes and automates the process of SQL tuning using an experiment-driven approach. It has the ability to generate different yet equivalent execution plans using a combination of intra- and inter-transformations, and then find a better execution plan through a smart exploration of the plan space.

Initially, zTuned collects the necessary statistics from the execution history that affect the poorly-performing query we are asked to tune. These statistics include past execution times as well as actual cardinalities for each operator. If this information is not available, zTuned will first execute the offending query.

After obtaining the execution plan selected by the query optimizer, zTuned will explore the plan's neighborhood. As described in Section 4, all plans in the same neighborhood can utilize the same cardinality estimates for all their operators. Hence, zTuned uses the costing engine of the database (through the Explain_Plan command from Section 3) to estimate the execution cost of each plan it generates, and compares them in order to select the best execution plan in the neighborhood.

Even though a plan neighborhood consists of plans with similar structure, its size could be very large. Hence, it is

Query	Run Time of Optimal Plan (sec)	Run Time of Optimizer Plan (sec)	Run Time of zTuned Plan (sec)	Percent Improvement over Optimizer	Tuning Time (sec)	Number of Plans Explored
7	0.90	6.65	0.90	86.51	67.03	630
8	-	31.59	30.24	4.28	329.88	4000
9	-	224.88	95.28	57.63	884.92	7800
10	37.61	42.86	38.23	10.81	131.73	190
11	2.28	3.69	2.30	37.71	13.10	48
21	32.14	43.36	34.38	20.71	81.25	188

Table 1: Tuning Results for TPC-H Queries

also important to explore the plans in a single neighborhood in a systematic way. We order the operators in the current plan P in decreasing order based on the difference between the estimated and actual cost of each operator. A ranked list of plans within the same neighborhood as P is then generated by applying intra-transformations subject to this operator order. Operators with higher difference in cost are more likely to be the cause of the problem since they are an indication that the query optimizer has made a significant costing mistake. Thus, higher ranked plans in the list are more likely to be better than P than lower ranked plans.

If the above approach does not lead to a better execution plan, then zTuned will use inter-transformations to generate plans that belong to different neighborhoods. Again, we must prioritize the exploration of the other neighborhoods. We order the neighborhoods based on the estimated cost of the plans that were generated from the optimizer's plan.

However, we are no longer able to induce all the cardinalities for the new plans, since they belong in different neighborhoods. Hence, zTuned must execute one or more of these further-away plans in order to collect the additional cardinalities needed. zTuned executes the "best" plan from each neighborhood and uses the actual running times to compare them. Currently, "best" is defined as lowest estimated cost based on all the cardinality information available so far, but we are exploring other possibilities.

The plan generation occurs independently from the query optimizer allowing for a larger and different exploration of the plan space. In addition, since zTuned works outside the optimizer, it can potentially be used with any database that uses a cost-based optimizer and supports the specification of full execution plans. It is important to note that this process is similar to what a DBA would perform in order to diagnose and fix the issue manually. We automated this process in a way that provides confidence to the DBA that she can trust the system to perform SQL tuning automatically.

6. EMPIRICAL EVALUATION

The purpose of our preliminary empirical evaluation of zTuned is twofold. First, we evaluate the effectiveness of zTuned in finding better execution plans for queries that perform poorly. Second, we provide an interesting study of the effects of skewed data on the performance of both the regular query optimizer and zTuned.

Our experiments were run on a VMware Virtual Machine running Ubuntu Linux 8.10, with an Intel Core Duo 2.53GHz CPU and 1GB of RAM memory. The database server used was PostgreSQL 8.3.4. We used the TPC-H Benchmark[2] with a scale factor of 1. We used an index advisor to produce indexes for the TPC-H workload and used default values for all PostgreSQL configuration parameters. We ensured that statistics about the data were up to date.

The Explain_Plan command was implemented as an internal command in PostgreSQL and was written in C. As a result, some core PostgreSQL code was modified. zTuned is currently implemented as a stand-alone Java application separate from the database.

6.1 Evaluation of Effectiveness

The first set of experiments targets the ability and efficiency of zTuned to find better execution plans for poorly-performing queries. For this purpose, we selected some TPC-H queries and executed all possible execution plans for these queries in PostgreSQL. The true optimal plan is the one that executes in the least amount of time. The results are shown in Table 1. We note that in all cases, zTuned was able to find a plan that was either the true optimal plan (which was found through exhaustive search) or was much closer to the true optimal plan compared to the plan picked originally by the query optimizer.

Furthermore, we generated a large set of TPC-H queries and tuned them using zTuned. All queries were run three times and we report average times. Table 1 provides detailed results for 6 queries. The percentage improvement that zTuned can offer varies greatly depending on the query. In general, larger queries see more improvement than smaller queries since there is a higher chance that the query optimizer makes mistakes. On average, zTuned is able to generate a plan with a 30% performance improvement over the plan selected by the query optimizer, and in some cases even up to 86% improvement.

The total number of plans explored in each tuning session also varies greatly. It is directly related to the number of tables accessed by the query. For instance, TPC-H queries 8 and 9 involve joins over 6 tables, and we see that zTuned explores 4000 and 7800 plans respectively. On the other hand, TPC-H query 12 consists of a single join over two tables, and the number of possible plans is just 12. Tuning time is also directly related to the number of plans explored, as well as to the running time of the plans (experiments).

6.2 Case Study with Skewed Data

By default, the TPC-H benchmark populates the tables with uniform data. However, real-life datasets routinely contain data skew. Therefore, we used a popular TPC-H data generation tool that produces skewed data following a Zipfian distribution. We created two databases: one with uniform data and one with skewed data, and explored the impact of data skew.

Most databases keep track of single table statistics with the use of histograms, which are usually relatively coarse.

[2]TPC-H is a decision support benchmark composed of queries that simulate business-intelligence workloads

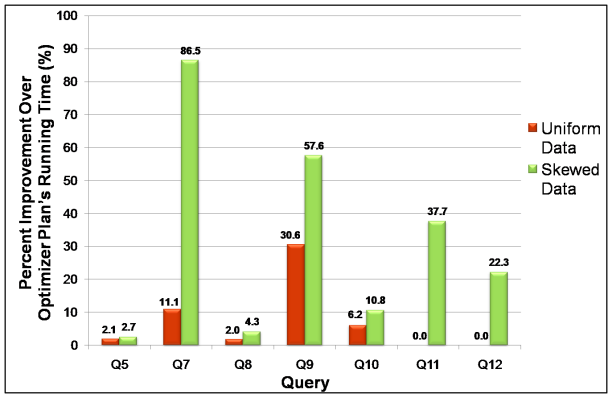

Figure 4: Tuning Improvement over Uniform and Skewed Data

Hence, when the query optimizer needs to calculate cardinality estimates over predicates, it usually resorts to uniformity assumptions. With uniform data, those assumptions are correct and thus, the query optimizer makes right decisions.

The results from our case study can be seen in Figure 4. The percentage improvement for queries over skewed data was larger than that over uniform data; which is to be expected. Over uniform data, the cardinality estimates of the query optimizer were relatively close to the actual cardinalities, and hence it made better plan choices. In some cases, zTuned was in agreement with the query optimizer as to which plan is better, and in other cases the plan it found was marginally better. However, over skewed data, the query optimizer was making more mistakes on cardinality estimations, which led to many suboptimal plans. In these cases, zTuned was able to find much better plans.

7. DISCUSSION

The experiment-driven approach and the mechanisms described in this paper can be generalized and used in other contexts. Specifically, the ability of Explain_Plan to specify the expected number of tuples produced from each operator is useful in many situations and allows for a much deeper analysis of database internals during SQL tuning. For instance, suppose the DBA has several plans in mind to experiment with. Without the cardinality feature, she would have to use the *analyze* option (Section 3) to run each plan, which might be very time consuming for large queries. Instead, she can now use Explain_Plan to compare their estimated costs based on actual cardinalities per operator. She can quickly prune the space of plans she was considering initially, and use the *analyze* option to run only the most promising plan.

With the cardinality feature, we can additionally test a query optimizer. We can now ask questions like: if the query optimizer had perfect information about the flow of tuples in a logical plan, would it still select the same physical plan? Such questions would allow us to validate the accuracy of the cost models used by the database. Currently, most approaches to SQL tuning assume that the cost models are accurate and that the main source of mistakes is inaccurate cardinality estimation. However, the intelligence of new storage systems are causing cost models to be increasingly inaccurate and not representative of the underlying hardware. Hence, it is crucial to devise new techniques for testing cost models over different storage systems.

The use of experiments for SQL tuning (or any other tuning task) pose a tradeoff between cost and benefit. In order to get more benefit, we would perhaps need to execute more experiments. Emerging mechanisms, like *cloud computing*, enable the use of experiments on a larger scale. For example, Amazon's Elastic Compute Cloud (EC2) provides cheap resources that can be leveraged for experiments. At the same time, it creates challenging research questions, like how to get the data into the cloud, and how to leverage parallelism in this context.

The problem of SQL tuning can now be formulated in a different way: given a budget (say U.S. $100), how many queries and how quickly can they be tuned? From Table 1, we see that tuning 6 queries took about 25 minutes. Assuming the data is stored on Amazon's Elastic Block Store (EBS), the total cost for acquiring an EC2 machine, as well as transferring and storing data in EBS would be less than U.S. $1, given current prices. With just a dollar in cost, zTuned offers an average of 30% improvement for those queries! In the future, we plan to investigate the promising potential of cloud computing in SQL tuning, as well as new ways to perform database tuning in a truly automated way.

8. REFERENCES

[1] M. Andrei and P. Valduriez. User-Optimizer Communication using Abstract Plans in Sybase ASE. In *Proc. of the 27th Intl. Conf. on Very Large Data Bases*. ACM, 2001.

[2] S. Babu, N. Borisov, S. Duan, H. Herodotou, and V. Thummala. Automated Experiment Driven Management of (Database) Systems. In *12th Workshop on Hot Topics in Operating Systems (HotOS-XII)*, Monte Verita, Switzerland, 2009.

[3] N. Bruno, S. Chaudhuri, and R. Ramamurthy. Power Hints for Query Optimization. In *25th International Conference on Data Engineering*, Shanghai, China, 2009.

[4] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. A Pay-As-You-Go Framework for Query Execution Feedback. In *Proc. of the 34th Intl. Conf. on Very Large Data Bases*, pages 1141–1152. VLDB Endowment, 2008.

[5] C. M. Chen and N. Roussopoulos. Adaptive Selectivity Estimation using Query Feedback. *SIGMOD Record*, 23(2):161–172, 1994.

[6] S. Duan, V. Thummala, and S. Babu. Tuning Database Configuration Parameters with iTuned. In *Proc. of the 35th Intl. Conf. on Very Large Data Bases*, Lyon, France, 2009.

[7] R. G. Freeman and A. Nanda. *Oracle Database 11g New Features*. McGraw-Hill Osborne Media, 2007.

[8] IBM DB2. *Giving optimization hints to DB2*, 2003. http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/index.jsp?topic=/com.ibm.db2.admin/ p91i375.htm.

[9] V. Markl, P. J. Haas, M. Kutsch, N. Megiddo, U. Srivastava, and T. M. Tran. Consistent Selectivity Estimation via Maximum Entropy. *VLDB Journal*, 16(1):55–76, 2007.

[10] Microsoft Corporation. *SQLServer Books Online: Query hint (transact-sql)*, 2007. http://technet.microsoft.com/en-us/library/ms181714.apsx.

[11] PostgreSQL. *Tuning Your PostgreSQL Server*. http://wiki.postgresql.org/wiki/Tuning_Your _PostgreSQL_Server.

[12] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's Learning Optimizer. In *Proc. of the 27th Intl. Conf. on Very Large Data Bases*, pages 19–28, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[13] K. Yagoub, P. Belknap, B. Dageville, K. Dias, S. Joshi, and H. Yu. Oracle's SQL Performance Analyzer. *DEB*, 31(1), 2008.