OptMark: A Toolkit for Benchmarking Query Optimizers

Zhan Li, Olga Papaemmanouil and Mitch Cherniack
Brandeis University, Waltham, MA, USA
{zhanli,olga, mfc}@cs.brandeis.edu

ABSTRACT

Query optimizers have long been considered as among the most complex components of a database engine, while the assessment of an optimizer's quality remains a challenging task. Indeed, existing performance benchmarks for database engines (like TPC benchmarks) produce a performance assessment of the query runtime system rather than its query optimizer. To address this challenge, this paper introduces *OptMark*, a toolkit for evaluating the quality of a query optimizer. OptMark is designed to offer a number of desirable properties. First, it decouples the quality of an optimizer from the quality of its underlying execution engine. Second it evaluates independently both the effectiveness of an optimizer (i.e., quality of the chosen plans) and its efficiency (i.e., optimization time). OptMark includes also a generic benchmarking toolkit that is minimum invasive to the DBMS that wishes to use it. Any DBMS can provide a system-specific implementation of a simple API that allows OptMark to run and generate benchmark scores for the specific system. This paper discusses the metrics we propose for evaluating an optimizer's quality, the benchmark's design and the toolkit's API and functionality. We have implemented OptMark on the open-source MySQL engine as well as two commercial database systems. Using these implementations we are able to assess the quality of the optimizers on these three systems based on the TPC-DS benchmark queries.

1. INTRODUCTION

Query optimizers have been recognized as among the most complex components of a DBMS. Among the myriad of optimizer design choices are whether they are top-down or bottom-up [8], how (or if) they constrain the search space of possible plans [9], or whether or not plans are modified dynamically [3]. Despite the wide variety in approaches to query optimization, the assessment of an optimizer's *quality* remains highly subjective. Indeed, DBMS performance benchmarks (e.g., the TPC benchmarks) conflate query optimization and query execution, producing a performance assessment that reflects upon the DBMS' query runtime system as much as (and arguably more than) its query optimizer.

Undoubtedly, a major reason that no query optimizer benchmark exists is because such a benchmark is extremely difficult to design and implement[20]. We have identified the following three key challenges in the design of an effective optimizer benchmark:

(1) Dual Assessment Measures: Optimizers should be evaluated for both their *effectiveness* and *efficiency* in generating plans for a given suite of queries. An optimizer benchmark must hence measure the effectiveness, i.e., the quality of plans generated for queries in a given query suite. But in comparing the optimizers of two different DBMSs, it is insufficient to compare the execution times of plans chosen by the optimizers for the same query, even if both DBMSs are implemented on the same platform. To illustrate, a plan generated by the MySQL optimizer [1] may fare poorly compared to that of commercial DBMS for a join-heavy query because joins in MySQL queries can only be executed as *nested-loop joins*. This does not necessarily reflect the effectiveness of the MySQL optimizer which might consistently generate the highest-performing plans possible that exclusively use nested-loop joins. Thus, optimizer effectiveness must be assessed evaluating generated plans relative to other plans the DBMS' query execution engine is capable of running.

An optimizer benchmark must also measure the efficiency, i.e., the resources (i.e., time and space) required by an optimizer to generate plans. A given optimizer could be very effective if it combines *exhaustive search* (thus considering all possible plans for a given query) with an extremely accurate cost model. But in practice, it is infeasible to exhaustively consider all possible plans, especially for the most complex and expensive queries (e.g., queries involving large numbers of tables) where optimization is needed most.[1] Thus, efficiency is a measure of how well a query optimizer can scale to process the most complex of queries.

(2) Benchmark Generality: An optimizer benchmark should be runnable over any DBMS regardless of the (hardware and OS) platforms over which it runs, and benchmark scores for different optimizers should be comparable even when run on differing platforms. This implies that (1) the benchmark code should be configurable to any DBMS (but ideally requiring minimal DBMS-specific code to do so), and (2) time-based metrics of effectiveness (runtime of generated plan for given query) and efficiency (time spent optimizing given query) should be avoided as they are incomparable for systems running on different platforms.

(3) Isolated Assessment: A DBMS optimizer's performance must be decoupled from that of the DBMS' query execution engine. Thus, end-to-end benchmarks such as the TPC-H and TPC-DS benchmarks [2] are not good query optimizer benchmarks because they report query execution times which depend not only on

[1] This is the reason that most optimizers have timeout settings that allow optimization to be curtailed prior to consideration of all plans.

CIKM'16 , October 24-28, 2016, Indianapolis, IN, USA

DOI: http://dx.doi.org/10.1145/2983323.2983658

the plan chosen by an optimizer but (even more so) on the capabilities of the DBMS query execution engine.

In this paper, we introduce *Opt-Mark*: a Query [Opt]imizer Bench[Mark] with the following key features:

1. *Effectiveness metrics* that assess the performance of optimizer-chosen plans relative to other plans that can be run by the same DBMS.
2. *Efficiency metrics* that are not based on optimization timing but instead on the size of the search space that an optimizer considers (thus measuring both space and time).
3. *A Toolkit* consisting of generic benchmark code together with a concise API that must be implemented for a benchmarked DBMS in a system-specific way.

This paper is structured as follows. We present the benchmark design, including effectiveness and efficiency measures in Section 2, and describe the toolkit code, including the API requiring DBMS-specific implementation in Section 3. We present benchmark results for 3 systems: MySQL and two well-known commercial DBMSs (which we refer to as Systems X and Y respectively) in Section 4. We describe related work in Section 5 and conclude with our final remarks in Section 6.

2. BENCHMARK DESIGN

In this section we present the design of OptMark, its *effectiveness* and *efficiency* measures and techniques used for determining them.

2.1 Optimizer Effectiveness

The *effectiveness* of a DBMS' optimizer reflects the quality of the plans it generates. The main challenge here is isolating the effectiveness of the optimizer from the underlying DBMS's query execution engine. We argue that effectiveness should be evaluated relatively to the capabilities of the underlying query (runtime) engine so as to decouple the effectiveness assessment of this component. In other words, an optimizer should not be penalized for not considering query operations (i.e., join algorithms, access methods, etc) that are not supported by the runtime engine of the DBMS.

The necessity of decoupling the optimizer from the query engine is illustrated in the case of MySQL [1] whose query engine supports *nested-loop joins* (NLJ) as the sole means of evaluating joins. Especially for join-heavy queries, MySQL will frequently be outperformed by DBMSs that also support other join operations such as *sort-merge* and *hash* joins. However, the MySQL optimizer should be considered effective if it consistently identifies the best NLJ plan for a given query, even though the MySQL query engine is less effective than those that can perform other types of joins.

Driven from the above discussion, we introduce the concept of a *relative optimal plan* of a query in a given DBMS. The relative optimal plan refers to the best plan the DBMS can run for that query. This plan might be different across different DBMSs.

Effectiveness Metrics OptMark measures optimizer effectiveness using two metrics that both compare the plans that an optimizer chooses for a given query suite Q with the plans it could have chosen for the same query engine:

1. *Performance Factor:* For any query $q \in Q$ and optimizer O_D for DBMS D, the *Performance Factor* of O_D relative to q, $\mathbf{PF}(O_D, q)$, measures the proportion of plans in the search space that are worse than the optimizer-chosen plan, which is defined as

$$\mathbf{PF}(O_D, q) = \frac{|\{p | p \in P_D(q), \mathtt{r}(D, p) \geq \mathtt{r}(D, O_D(q))\}|}{|P_D(q)|} \tag{1}$$

such that $O_D(q)$ is the plan O_D generates for q, $P_D(q)$ is the set of all plans that could be executed by D to evaluate q, $\mathtt{r}(D, p)$ is the measured runtime of plan p over D and $\mathtt{r}(D, O_D(q))$ is the runtime of the plan O_D generates for q over D. We note here that the timing of $\mathtt{r}(D, p)$ is subject to numerous environment conditions (e.g., empty DB and OS buffers, no contention) and one should control these factors (e.g., by executing the queries in isolation and using a cold cache every time). Thus, the best possible score for this metric is 1 (indicating that an optimizer chooses the optimal plan for the query q) and the closer the score is to 0, the poorer is the optimizer-chosen plan.
2. *Optimality Frequency:* Optimizer O_D finds a relative optimal plan for query $q \in Q$ if $\mathbf{PF}(O_D, q) = 1$. Thus, the optimality frequency of O_D, $\mathbf{OF}(O_D, Q)$, is the percentage of queries in the query set Q for which O_D chooses the relative optimal plan.

The two metrics described above can be leveraged to provide insight on (a) the quality of the optimizer-chosen plan, (b) the quality of the cost model and (c) the quality of the plan enumeration process of a given optimizer. First, the performance factor reflects the quality of the optimizer-chosen plan compared to other plans the DBMS is capable of running relative to a given query. A performance factor of 1 indicates that the optimizer-chosen plan is better than *all* plans while the lower the performance factor the more plans are better than the chosen one. Second, the quality of the cost model can be measured by determining how many of the plans that did better than the optimizer-chosen plan were considered by the optimizer. Third, the quality of the plan enumeration can be measured by the number of plans that did better than the optimizer-chosen plan and were not considered by the optimizer. For the plans that were considered by the optimizer, the optimizer did not choose them because of an inaccurate cost model. For the plans that were not considered by the optimizer, the optimizer did not include them in its search space due to poor plan enumeration.

2.1.1 Plan Space Generation

Evaluating the effectiveness requires the enumeration and execution of *all* possible plans the optimizer could execute for a given query. This process is unfeasible, especially for complex queries due to the exponential number of queries to be executed. Alternatively, one could collect those plans that optimizer considered (i.e., costed) in the process of choosing a plan. We can then compare these plans with the optimizer-chosen plan and calculate the above effectiveness metrics. However, this approach ignores flaws that might exist in the optimizer's enumeration strategy that might have resulted in an optimizer not considering a plan that it should have. For example an optimizer that only considers poor plans but costs them correctly would be considered effective (i.e., with performance factor of 1) despite choosing poor plans.

OptMark takes an alternative approach and generates a set of comparable plans for a given query by generating a random sample set of candidate execution plans that may or may not have been considered by the optimizer. Our approach assesses the quality of the optimizer-chosen plan relative to the plans that the runtime engine is capable of executing. The challenge here is to estimate the proportion of plans that perform worse than the optimizer's chosen plan (i.e., the performance factor) without having any prior knowledge of the performance distribution of the candidate plans. In the following section we describe how to identify the size of the sample set required to estimate the performance factor with a given confidence and precision. We then proceed to describe how we generate these sample plans.

2.1.2 Sample Size

To calculate the sample size required to estimate (with a specified level of confidence and precision) the performance factor of O_D relative to a query q, $\mathbf{PF}(O_D, q)$, we use the formula from [7]:

$$n = \frac{Z^2 p(1-p)}{e^2} \qquad (2)$$

such that, e is the desired level of precision (aka sampling error), Z is the value from the standard normal distribution that corresponds to the desired confidence level (e.g., $Z = 1.96$ for a a confidence interval of 95%) and p is an estimate of the proportion of plans that will be worse than the optimizer's plan or 0.5 when this estimate is unknown. Equation 2 assumes that our candidate plan set size (i.e., population size) is large compared with the sample size while for smaller populations a modified formula can be used that reduces slightly the required sample size.

It follows that if the user desires to estimate the performance factor of a query with a 95% confidence level and 5% precision, it needs to generate at least 385 random sample plans. Hence, if we randomly generate 385 plans and found that 80% of the plans perform worse the optimizer's chosen plan, we can conclude with 95% confidence that the optimizer's plan is better than 75% - 85% possible plans, i.e., the performance factor is 0.75 - 0.85.

2.1.3 Random Plan Generation

Given a number of random plans to generate, OptMark generates these plans by exploring the three main features that characterize query plans: the *join ordering*, the *physical join algorithm* used for each join and *table access methods*. Our process first produces a random join ordering for a given query and then expands this plan with randomly selected physical join operations and access methods. Next we describe this process starting with the random join ordering generation for a given query.

Random Join Orderings Traditionally, join orderings are represented as binary trees (aka join trees) where internal nodes represent join operations and the leaf nodes represent tables. Our join ordering generation process generates an unbiased random binary join tree and then generates a random sequence of tables to populate the leaves of the join tree. Every join ordering of a query joining n tables can be encoded as a pair (s, p) such that:

1. s is a bit sequence that represents the preorder traversal of the ordering's binary tree such that each successive bit denotes the next node of the tree visited in the traversal and is a '1' if that node is an internal node (join) and a '0' if that node is a leaf node (table). Hence, the bit sequence has length $(2n-1)$ consisting of $(n-1)$ '1's and n '0's.
2. p represents the sequence of tables to populate the leaf nodes in the binary tree. Specifically, p is some permutation of the tables in the query. The permutation sequence then specifies the leaves of the join tree from left to right.

OptMark generates random join orderings by generating random encodings of join trees (s, p). The random generation of the bit sequence s involves flipping a biased coin for each bit in the sequence (going from left-to-right) to decide if the "next" bit should be a '0' or an '1' [4]. Because the last bit in the bit sequence must be a '0', given that the last visited node in a preorder traversal must be a leaf node, we use the algorithm in [4] to generate $2(n-1)$ bits at random and add a '0' at the end. To ensure that every tree is enumerated with equal probability, one must use a biased coin when deciding between a '1' and a '0' and the degree of bias depends on what has been generated thus far. For each bit in the sequence s, we determine the probability that the bit should be filled with a '0', which is expressed as $P(r, k)$ such that r is the number of '1's in the bit sequence thus far minus the number of '0's in the bit sequence thus far and k is the number of bits that have yet to be assigned. The formula to calculate $P(r, k)$ as written in [4] is:

$$P(r, k) = \frac{r(k + r + 2)}{2k(r + 1)} \qquad (3)$$

Given a query that joins n tables, OptMark uses the above formula to estimate the probability of each bit in a $2(n-1)$ bit sequence and create a random sequence s. It then generates a random permutation p of the tables and outputs a random join ordering (s, p).

Given the join ordering, we then replace all join operators with randomly selected physical join operators to generate a physical plan to include in the sample. Specifically, for each join node in the join tree, if its inputs have no corresponding join predicate in the query, we force a cross join. Otherwise, we randomly select one of the physical join operators supported by the execution engine. We then add random access methods for each input table by randomly selecting an applicable index or, if one does not exist, a sequential scan as the table access method. The above process is repeated until we generate as many plans as specified by the sample size determined as described in Section 2.1.2.

2.2 Optimizer Efficiency

The *efficiency* of a DBMS' optimizer reflects the resource requirements (i.e., time and memory) necessary for the optimizer to choose a query plan. In theory, a DBMS could consider all possible candidate plans for a query (*exhaustive enumeration*) regardless of the time and space that this requires[2], and provided it was armed with an accurate cardinality and cost model, would always choose an optimal plan. In practice, exhaustive enumeration is infeasible for complex, join-heavy queries and most optimizers "time-out" prior to consideration of all plans for such queries.

One possible approach to measure optimizer efficiency is to calculate the average time that an optimizer spends optimizing queries in the benchmark query suite. However, this metric has two notable deficiencies: 1) optimizer times recorded for DBMSs running on different platforms are incomparable, and 2) this metric only measures time and not memory. The OptMark benchmark instead measures efficiency using four metrics that specify the size of the search space processed during the optimization of a query:

1. **#LP**: the number of logical plans enumerated,
2. **#JO**: the number of join orderings enumerated,
3. **#PP**: the number of physical plans costed, and
4. **#PJ**: the number of physical join plans costed.

It is clear that each of these metrics is a measure of the size of the search space explored for a given query. But as we show in Table 1, these metrics are also strongly correlated with optimization time. This table shows the degree of correlation between each metric and the time spent by four different DBMSs optimizing averaged over the 93 join queries from the TPC-DS benchmark. Correlation is demonstrated with r^2 values that show the goodness-of-fit of the linear regression, and that fall between 0.0 and 1.0 with higher values indicating higher correlation. Note that the most highly correlated metric (shown for each DBMS in boldface) varies from system to system, demonstrating that there is no single "best" metric for all systems. However, all metrics have very strong correlations (over 0.7 in all cases) with the optimization time and therefore can be used as predictors of an optimizer's optimization time. An interesting observation is that the r^2 value of **#PP** and **#PJ**, and that of **#LP** and **#JO** are very close to, or even identical to each other on all the DBMSs. The reason is that non-join plans (e.g., table

[2]The time and space required by an optimizer will always impact ad hoc queries however.

System	#LP	#JO	#PP	#PJ
MySQL	0.92	0.93	**0.94**	**0.94**
PostgreSQL	0.72	0.72	**0.97**	**0.97**
System X	**0.81**	**0.81**	0.71	0.72
System Y	0.77	0.75	**0.85**	**0.85**

Table 1: Correlation of efficiency metrics and optimization times

scan plans and index scan plans) are typically much fewer than join plans, and thus take much less optimization time[13].

To benchmark a given DBMS one must be able to extract at *least one* and as many of the four metrics as possible, so that efficiency results can be compared to as many other DBMSs as possible.

Linear Regression Results To collect the results of Table 1, we processed an optimization structure exposed by System X, added instrumentation to the open source code of PostgreSQL and parsed trace files of MySQL and System Y. These techniques allowed us to count the physical plans that were costed by each DBMS during optimization, and from these results we were able to determine the values of the other metrics: determining physical join plans by removing all non-join physical plans (e.g., group-by plans and index plans) from the physical plans, determining logical plans by converting each physical operator in the physical plan to its logical equivalent and ignoring duplicates, and determining join orderings on the basis of physical join structures.

MySQL shows a quite strong correlation of the logical plans: almost as high as the correlation of the physical plans. MySQL resolves all joins to nested-loop joins and when mapping logical plans to physical ones it only needs to convert selection operators to table access methods (e.g., table scan, index scan). Hence, in this engine we are able to reconstruct almost the majority of the logical plans, as very few logical ones are pruned before converting to physical ones. On the other hand, **#LP** and **#JO** seems to be a stronger predictors of optimization time than the physical plan metrics **#PP** and **#PJ** for System X. For this engine we were able to collect *all* logical plans and join orderings considered by the optimizer and hence the correlation is high. However, the data structure offered by this specific engine only reports the physical plans that the optimizer considered to be "promising" plans. A high percentage of the pruned plans are not reported although they do add an overhead to the optimization process (since the optimizer examined them). Hence our experiments used a reduced set of physical plans and the regression results thus show a lower correlation of **#PP** compared with **#LP**.

3. THE OPTMARK TOOLKIT

We next introduce OptMark, a query optimizer benchmark toolkit that assists developers measure optimizer effectiveness and efficiency. Any DBMS for which benchmark results are desired can use our toolkit by implementing a small, simple set of API functions. These functions extract all the optimization metadata and statistics we need to evaluate the effectiveness metrics and efficiency predictors we discussed in the Sections 2.1-2.2. OptMark is minimum invasive to the underlying database engine. It is executed as a stand-alone tool and runs against a given target database using a standard JDBC interface. It could be extended to any schema and query set, but we expect it to be used on queries that stress the performance of an optimizer. Typically, these are queries with non-trivia number of joins.

Benchmark Requirements For a given DBMS to run the benchmark, it must satisfy the following requirements. First, it must support JDBC. The benchmark compares the runtime of the optimizer-chosen plan with the sample set of plans it generates, so OptMark must be able to connect to DBMS through JDBC to execute specific query plans. Second, it must support query hints (or directives) to force the optimizer to consider only plans that adhere to a specific join ordering and/or use specific join operations and access methods. This allows us to to generate a sample set of physical plans with which to compare optimizer generated plan and use it to evaluate the optimizer's effectiveness. Finally, it must expose at least one of the four predictors we describe in Section 2.2. These allow us to quantify the time and space used by optimizer in generating a plan for a query and determine the optimizer's efficiency. Note that not every optimizer exposes each of these indicators, but any two optimizers can be compared if they expose a *common* indicator.

3.1 Effectiveness Assessment & API

In Section 2.1 we discussed our technique for evaluating the effectiveness of an optimizer. A building block of our approach is enumerating a sample set of physical join plans for a given join query and processing them to collect their execution time. This indicates two requirements for our toolkit. First, it must be aware of the physical join algorithms supported by the underlying DBMS it is executed on (e.g., MySQL supports only nested-loop join, while DBMSs X and Y support also hash join and merge-sort join). Second, OptMark must be able to enforce the execution of a query using a given physical plan. This plan will specify the join ordering, the physical join operators to be used as well as table access methods. The latter could be either an index-based scan, if an index is available, or a sequential scan.

Our approach to execute a given physical plan relies on a standard feature in modern databases: query hints that affect the plan choice by the query optimizer. As the syntax of query hints varies for different systems, OptMark users need to implement a set of API functions which return the exact syntax of query hints on the system OptMark is running against. This API allows our toolkit to be independent of the underlying DBMS.

Effectiveness API Our effectiveness API consists of three main functions which return (a) the supported physical join operators, (b) the syntax for hinting the use of a given index and (c) the syntax for hinting a specific join method. Next we provide the formal signatures of these API functions.

1. `Set{String} joinTypes()`
   This method returns a list of physical join methods the system supports in the syntax of join hints.
2. `String indexHint(String t, String ind)`
   Given the table name, `t`, and an applicable index on it, `ind`, the method returns the hint syntax for forcing an index scan on `t` using the index `ind`.
3. `String joinHint(String t1, String al1, String index1, String t2, String al2, String index2, String join, String clause)`
   Given the name, alias and the indexes of two joining tables, the method returns the hint syntax to force a two-way join using the specific indexes on each table. The alias and index parameters are optional. The above API function can also support nested queries. Specifically, the table parameters `t1` and `t2` could be a base table, or a sub-query. If the table parameter is a subquery then the alias parameter is used as a reference of that query. The index parameters `ind1` and `ind2` are the output of the method `indexHint()`, i.e., they are the hint syntax for using an index-based table scan. The parameter `join` is the output of the method `joinTypes()`, i.e., it is the syntax for forcing the optimizer to use a specific physical join operator for executing

the query. The parameter `clause` is a string representation of the join conditions to be used.

OptMark also needs to determine which indexes exist on which tables and over which attributes, this information can be got by a JDBC function `getIndexInfo()`.

Benchmarking Effectiveness Next we describe OptMark's approach for assessing the effectiveness of an optimizer. Given a set of queries to execute the benchmark on and an implementation of the effectiveness API for the DBMS OptMark is running on, our toolkit generates a sample set of physical plans against which the optimizer chosen plan is compared with based on the execution time. As discussed in Section 2.1.2 the size of the sample set is determined by the user-defined confidence level and margin of error.

For a given query, the random plans with which the optimizer-chosen plan is compared with are generated as follows:

1. First it generates random join orderings as discussed in Section 2.1.3.
2. For each of joins in a given join ordering, if there is no join predicate between two tables it forces a cross join. Otherwise, it randomly selects one physical join operation from all physical join operations returned by `joinType()`.
3. For every table in the query we identify if there are any indexes on a predicate attribute (all index information is returned by `getIndexInfo()`) and we randomly select to use an index or sequential scan to access each table.
4. Steps 1 - 3 generate a single random plan. We repeat this process until we collect the desired sample size.

For each plan in our sample set we create a SQL query that enforces the specified join ordering, join algorithm and access method. The join orderings of step 2 and the choice of join operations in step 3 are enforced according to directives specified in `joinHint()` while the access method is enforced by adding the SQL directive specified by `indexHint()`. The output hint-based SQL query is executed and OptMark collects its runtime and evaluates the query's performance factor. The process is repeated for all queries in a given benchmarking query suite and the optimality frequency of the query set is reported.

3.2 Efficiency Assessment & API

In Section 2.2 we introduce our four efficiency predictors used by OptMark, namely the number of logical and physical plans as well as the number of join orderings and physical join plans. Given a set of benchmark queries, OptMark will report the predictors the DBMS exposes for each of the queries in this query set.

Efficiency API The API that must be implemented in a DBMS specific way to support efficiency assessment of its optimizer should include a function that returns for any given query at *least one* of our four efficiency indicators: (1) number of logical plans, (2) number of logical join plans, (3) number of physical plans and (4) number of physical join plans. All numbers refer to the plans the optimizer considers for the given query.

4. BENCHMARK RESULTS

We have implemented OptMark's effectiveness and efficiency APIs over three DB systems: MySQL, a commercial system with a top-down optimizer that we refer to as System X, and a commercial system with a bottom-up optimizer that we refer to as System Y. Our implementations are illustrated in [15].

We run our toolkit on a server equipped with a 3.06 GHz Octa CPU and 32 GB of memory. We use the TPC-DS benchmark [2] for generating our benchmarking dataset and query suite. The workload consists of 24 queries with a minimum of 5 tables in each query. The benchmark toolkit runs on a dataset of 100GB.

System	#LP	#JO	#PP	#PJ
MySQL	N/A	N/A	48	40
PostgreSQL	N/A	N/A	810	778
System X	146	123	N/A	N/A
System Y	N/A	N/A	540	514

Table 2: Values of efficiency predictors

4.1 Efficiency Results

We use four metrics, **#LP**, **#JO**, **#PP**, **#PJ**, to measure the efficiency of an optimizer (Section 2.2). Since all metrics have high correlation with optimization time, any one of them can be used as a metric for efficiency evaluation. The metrics can be used to compare the efficiency of different versions of an optimizer. For example, if one introduces a new enumeration strategy, we can use the metrics to check if the new strategy makes the optimizer more efficient. However, one needs to be careful using them to compare different optimizers. For example, the efficiency metrics of MySQL should be less than those of optimizers which supports more physical join operators than just nested loop join which is the only physical join operation MySQL supports. But the comparison between them does not make much sense because they support different join operations.

We are not able to implement OptMark on PostgreSQL because it doesn't support query hints. But as PostgreSQL exposes ways to extract physical plans and physical join plans, we can still do efficiency evaluation for PostgreSQL. Table 2 shows the values of the four efficiency predictors for the four DBMSs we used. For each predictor, we take the average number over all queries in the workload. Some efficiency predictors are not applicable for some DBMSs because they are not exposed by the DBMSs. From the result we can observe MySQL has the least number of physical plans and physical join plans, which makes sense because MySQL resolves all joins to nested-loop join, its search space is far smaller than the other systems.

4.2 Effectiveness Results

We used OptMark to collect effectiveness results on System X. Similar results for System Y and MySQL can be found in [15]. For each of our 24 TPC-DS queries we generated a random sample set of 385 plans, allowing us to estimate the performance factor of the optimizer with 95% confidence and 5% margin of error.

Effectiveness metrics Table 3 shows the performance factor of each query for System X. The optimality frequency of System X is 0.5, and hence one can conclude that the optimizer of System X chose the relative optimal plan in no more than 50% of the queries. The average performance factor for the queries that do not find the relative optimal plan is 0.927. Furthermore, we can say with 95% confidence that System X finds a plan that is better than 80% of the generated sample plans (PF$\pm$ 5% > 80%) for 96% of the queries.

Quality of optimizer-chosen plan For System X at most 50% of the queries used the best plan (these are the queries in Table 3 with a performance factor of 1). For the remaining of the queries one should interpret the results along with 95% confidence and 5% margin of error we used to generate our sample set of plans. For example, the performance factor of query 13 is 0.88, hence, with 95% confidence we can say that the optimizer-chosen plan is better than 88% $\pm$ 5% plans in the search space.

An interesting observation is that while the optimality frequency (0.5) of System X shows that in half of the queries the chosen plan was not the best, in 96% of the queries the chosen plan was better than 80% of the plans in the sample set. Our results indicate that while the chosen plan by the optimizer of System X might not be the best in half of the cases, it is one of the top plans, which is a good enough plan for an optimizer with limited resources.

query	q6	q7	q13	q17	q18	q19	q24	q25	q26	q27	q29	q40	q46	q48	q50	q54	q61	q62	q66	q68	q73	q79	q84	q85
PF	1	1	0.88	0.917	0.854	1	1	1	0.899	0.989	0.825	0.979	0.94	1	0.969	1	1	0.977	0.982	1	1	1	1	0.912

Table 3: Performance Factor (PF) for System X

Quality of Cost Model and Plan Enumeration For the queries with performance factor less than 1 one can examine the plans that perform better than the optimizer-chosen plan and get some insight of the quality of the cost model and plan enumeration approach. We examined query 29 which has the lowest performance factor and we discovered that there are 71 plans in the sample that did better than the optimizer-chosen plan. Among them there are 54 plans that were considered by the optimizer, while 17 plans were not considered by the optimizer. As discussed in Section 2.2, we were not able to extract all physical plans the optimizer considered for System X, the number of plans considered here serves as a lower bound. This indicates the cost model fails to estimate accurately the cost of at least 54/71 (76%) of these better plans. The enumeration quality was however high as the optimizer did not consider only at most 17/71 (24%) of the better plans.

For query 18 which also has a low performance factor, we discovered 56 plans that did better than optimizer-chosen plan. Out of these, 24 plans were considered by the optimizer, while 32 plans were not. So the cost model fails to accurately estimate the cost of at least 43% of these plans while the enumeration approach did not even consider 57% of these plans.

Finally, to evaluate the efficiency of the optimizer for these queries we collected the number of logical plans. For query 29, the optimizer considers 93 logical plans, while for query 18, the optimizer considers 74 logical plans, indicating the optimizer was more efficient in coming up with a plan for query 18.

5. RELATED WORK

Despite of the importance of optimizer benchmarks to date no end-to-end optimizer benchmark is available. [19] offers a set of tools that support the design and generation of custom testbeds for optimizers but does not provide any measures to evaluate the quality of the produced optimizers. In [11] they provide a high-level overview of the unique challenges in testing a query optimizer. Our work covers two of the metrics they discussed: optimization time (i.e., efficiency), and query performance (i.e., effectiveness). A number of papers present tools to assist optimizer benchmarking. [21] presents algorithms to generate either a whole space of alternative plans, or a uniform random sample. But their research was specific to one system while the way we generate our sample set of plans is applicable to any DBMS.

There has also been work on testing different components of the query optimizer. [18] introduces a toolkit to visualize the plan space to facilitate the analysis of the cost model and behavior of a plan. In [22] and [12] they focus on the accuracy of cost model. The impact of I/O cost estimation on quality of query optimizers has been studied in [17]. In [14] they present ways to quantify the contributions of cardinality estimation, the cost model and the plan enumeration algorithm and provide guidelines for the complete design of a query optimizer.A number of papers addressed the problem of testing cardinality estimation models. [10] describes the replacement and validation of a new cardinality estimation model in Microsoft SQL Server. [16] defines a metric to measure deviations of size estimations from actual sizes. [6] presents a set of techniques that make exact cardinality query optimization a viable option. The effectiveness of transformation rules are studied in [5]. All of the above work focuses only on testing specific components of the query optimizer, while OptMark provides an end-to-end optimizer benchmark, aiming to reveal overall deficiencies and strengths of the benchmarked optimizer.

6. CONCLUSIONS

This paper introduces OptMark, a toolkit for evaluating the quality of database optimizers. OptMark offers a set of desirable features to support the assessment of optimizer quality. First, it provides methods for assessing both effectiveness (i.e., quality of the optimizer's chosen plan) *and* efficiency (i.e., optimization time) of an optimizer. Second OptMark decouples the evaluation of the optimizer performance from the performance of its underlying DBMSs execution engine, which distinguishs it from existing DBMS benchmarks (like TPC). Finally, it is minimum invasive to the underlying engine in that any DBMS for which benchmark results are desired can use our toolkit by implementing a simple set of API functions.

7. ACKNOWLEDGMENT

This research was funded by NSF IIS 1217952.

8. REFERENCES

[1] MySQL: An open source relational database, http://www.mysql.com/.

[2] TPC-DS Benchmark, http://www.tpc.org/tpcds/.

[3] Adaptive query optimization. In *Encyclopedia of Database Systems*, page 50. 2009.

[4] M. Atkinson and J. Sack. Generating binary tree at random. *Information Processing Letters*, 1992.

[5] S. Chaudhuri, L. Giakoumakis, V. R. Narasayya, and R. Ramamurthy. Rule profiling for query optimizers and their implications. In *ICDE*, 2010.

[6] S. Chaudhuri, V. R. Narasayya, and R. Ramamurthy. Exact cardinality query optimization for optimizer testing. *VLDB*, 2009.

[7] W. G. Cochran. *Sampling Techniques, 2nd Ed.* New York: John Wiley and Sons, Inc., 1963.

[8] D. DeHaan and F. W. Tompa. Optimal top-down join enumeration. In *SIGMOD*, 2007.

[9] P. Fender, G. Moerkotte, T. Neumann, and V. Leis. Effective and robust pruning for top-down join enumeration algorithms. In *ICDE*, 2012.

[10] C. Fraser, L. Giakoumakis, V. Hamine, and K. F. Moore-Smith. Testing cardinality estimation models in SQL server. In *DBTest*, 2012.

[11] L. Giakoumakis and C. A. Galindo-Legaria. Testing SQL server's query optimizer: Challenges, techniques and experiences. *IEEE Data Eng. Bull.*, 2008.

[12] Z. Gu, M. A. Soliman, and F. M. Waas. Testing the accuracy of query optimizers. In *DBTest*, 2012.

[13] I. F. Ilyas, J. Rao, G. M. Lohman, D. Gao, and E. T. Lin. Estimating compilation time of a query optimizer. In *SIGMOD*, 2003.

[14] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? In *VLDB*, 2016.

[15] Z. Li, O. Papaemmanouil, and M. Cherniack. Optmark: A toolkit for benchmarking query optimizers. *arXiv:1608.02611 [cs.DB]*.

[16] G. Moerkotte, T. Neumann, and G. Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *VLDB*, 2009.

[17] Y. Qin, K. Salem, and A. K. Goel. Towards adaptive costing of database access methods. In *ICDE*, 2007.

[18] N. Reddy and J. R. Haritsa. Analyzing plan diagrams of database query optimizers. In *VLDB*, 2005.

[19] M. Stillger and J. C. Freytag. Testing the quality of a query optimizer. *IEEE Data Eng. Bull.*, 18(3):41–48, 1995.

[20] F. Waas. benchmarking query optimizers, http://database-research.com/2011/04/19/benchmarking-query-optimizers/.

[21] F. Waas and C. A. Galindo-Legaria. Counting, enumerating, and sampling of execution plans in a cost-based query optimizer. In *SIGMOD*, 2000.

[22] F. M. Waas, L. Giakoumakis, and S. Zhang. Plan space analysis: an early warning system to detect plan regressions in cost-based optimizers. In *DBTest*, 2011.