Interactive Plan Hints for Query Optimization

Nicolas Bruno
Microsoft Research
nicolasb@microsoft.com

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Ravi Ramamurthy
Microsoft Research
ravirama@microsoft.com

ABSTRACT

Commercial database systems expose *query hints* to fix poor plans produced by the query optimizer. However, current query hints are not flexible enough to deal with a variety of non-trivial scenarios, and can be at times cumbersome for DBAs to interact with. In this demonstration we present a framework that enables visual specification of hints to influence the optimizer to pick better plans. Our framework goes considerably beyond existing hinting mechanisms and significantly improves the usability of such functionality.

Categories and Subject Descriptors

H.2.7 [**Database Administration**]

General Terms

Algorithms, Design

Keywords

Query Hinting, Query Optimization

1. INTRODUCTION

Relational query optimizers are responsible for finding efficient execution plans to evaluate input SQL queries. For that purpose, cost-based optimizers search a large space of alternative execution plans, and choose the one that is expected to be evaluated in the least amount of time. In doing so, query optimizers rely on a cost model that estimates the resources that are needed for each alternative plan under consideration.

Optimizer cost models are usually very complex, and depend on cardinality estimates, plan properties, and specialized cost formulas for each operator in an execution plan. It is well known (e.g., see [4]) that currently deployed cost models are inherently inaccurate due to several factors. First, cardinality estimates, especially those on intermediate query expressions, are inferred based on statistical information on base tables and assume correlation and uniformity. Also, cost calibration constants (e.g., the cost of a single disk I/O) might not be fully accurate or consistent with respect to the underlying hardware (in general, cost formulas cannot possibly capture every detail on the execution plan operators). Finally, several runtime parameters affect execution costs of queries, but are not modeled in current optimizers.

As a natural consequence, query optimizers do not always produce optimal plans, and sometimes might even return poor plans. In such cases, database administrators need to correct the bad plan picked by the optimizer. For that purpose, a common mechanism found in commercial database systems is called *query hinting* [1, 3, 6]. Essentially, query hints instruct the optimizer to constrain its search space to a certain subset of feasible execution plans[1]. Consider the query below:

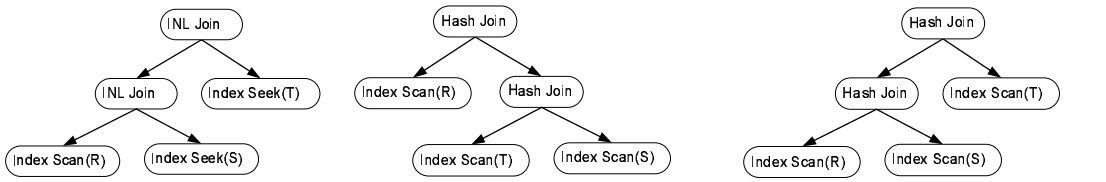
```
SELECT R.a, S.b
FROM R, S, T
WHERE R.x = S.y AND S.y = T.z
      AND R.a = 10 AND S.b = 15 AND T.c = 20
```

Suppose that the optimizer returns the query plan in Figure 1(a). If the cardinality of $\sigma_{R.a=10}$ is underestimated, and thus the index-nested-loop join processes many more tuples than what was estimated by the optimizer. In this case, the query might perform too many index seeks on S thus resulting in a bad plan. If such a situation can be detected, then DBAs can optimize the query using a special hint `OPTION(HASH JOIN)` that is concatenated to the query string. In this special mode, the optimizer is forced to consider only execution plans that use hash-based join alternatives, and chooses the execution plan of Figure 1(b). However, this alternative plan uses a different join order, which might make a DBA question whether the original join order was indeed more efficient. To answer this question, the DBA would want to execute the plan in Figure 1(b) and also an alternative plan that shares the same join order with the plan in Figure 1(a) (but does not use an index-nested-loop join between R and S). This new plan, shown in Figure 1(c) can be obtained by using the expanded hint `OPTION(HASH JOIN, FORCE ORDER)`, which only considers plans whose join order is compatible with the order of tables in the `SQL` query string. After actually executing all these alternatives, the DBA chooses the most appropriate plan and uses the corresponding hint in a production environment.

As illustrated by this example, the process of tuning a poorly performing query is usually exploratory and interactive. DBAs might try some hints, observe the resulting plan picked by the optimizer, optionally execute it, and keep modifying plans using hints until they obtain one that is better than what the optimizer originally produced. We believe that this trial and error process is inherent to the tuning of a poorly optimized query. Hints enable DBAs to enhance the optimizer's search strategy by using knowledge beyond its cost model.

[1]Existing query hints can additionally affect some aspects of query execution. Such hints, though important, are outside the scope of this work.

SIGMOD'09, June 29–July 2, 2009, Providence, USA.



(a) Original Plan. (b) Using `OPTION(HASH JOIN)`. (c) Using `OPTION(HASH JOIN, FORCE ORDER)`.

Figure 1: Using query hints to fix bad plans.

Current hints are usually not flexible enough for many scenarios. In most systems, forcing the first join in Figure 1(a) to be hash-based would also result in either a fixed join order (e.g., by using local join hints), or the hash-join algorithm being used for all joins in the query block (e.g., by using the `HASH JOIN` hint discussed above). Hints are also not flexible enough to express certain constraints of the structure of the execution plan, such as trying to force an early pre-aggregation below a join.

In this demonstration, we present *Phints*, a visual framework for constraining the search space of current optimizers, which goes beyond what is currently offered in commercial DBMSs. Our framework, which is implemented on top of an instrumented version of Microsoft SQL Server, provides a visual interface that allows DBAs to graphically see and compare alternative execution plans, and specify desired characteristics of resulting query plans. We believe that this framework considerably simplifies the interactive process of debugging poorly performing queries.

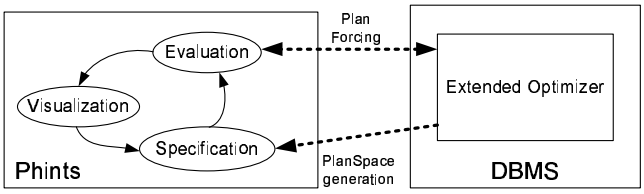


Figure 2: The *Phints* Framework.

2. USING *PHINTS* AS A DBA TOOL

A session in the *Phints* framework is typically initiated when a DBA decides to tune the execution plan of a problematic query. At this point, the client calls the query optimizer and generates a representation of the search space (denoted `PlanSpace`) for the given query. Then, the DBA interacts in the *Phints* framework as follows (see Figure 2):

Specification: The DBA, based on knowledge about the application and the current execution plan, decides to modify the search space of the optimizer by specifying constraints using a domain-specific language (see Section 3 for some examples). A full description of the language can be found in [2]. Additionally, the visual framework further extends the language in [2] to incorporate certain commonly used operations (e.g., cardinality hints).

Evaluation: Once a specification is in place, we produce the best execution plan that satisfies the constraints. Note that in the context of interactive sessions, we would be interested in real-time response to constraint evaluation. To enable such functionality, we cache the `PlanSpace` at the client, and thus we do not require a round-trip to the server at each iteration [2]. The resulting execution plan (or sub execution plans thereof) can optionally be sent to the server and executed to obtain feedback on cost and cardinality, which can be used to further refine the constraints.

Visualization: The resulting plan (with optional feedback from actual execution) is displayed graphically. Multiple execution trees resulting from previous *Phints* expressions can be compared, and the execution trees themselves can be used to specify additional constraints easily.

This process continues until the DBA is satisfied with an execution plan for the problematic query. At that point, the framework generates a description of such plan to be passed to the optimizer in a production system using a suitable *plan forcing* interface [5].

3. THE DEMONSTRATION

In this demonstration, we showcase the visual framework for query hinting. Our prototype consists of a C++ client application that talks to a modified version of Microsoft SQL Server. In the demonstration we will present scenarios that highlight the salient features of our solution. The following examples are representative of tuning sessions that will be shown during the demonstration, and are based on the following `TPC-H` query:

```
SELECT TOP 10 L_ORDERKEY, O_ORDERDATE,O_SHIPPRIORITY,
       SUM(l_extendedprice*(1-l_discount)) AS REVENUE
FROM  CUSTOMER, ORDERS, LINEITEM
WHERE C_MKTSEGMENT = 'MACHINERY'
  AND C_CUSTKEY = O_CUSTKEY
  AND L_ORDERKEY = O_ORDERKEY
  AND O_ORDERDATE < '1995-03-26'
  AND L_SHIPDATE > '1995-03-26'
GROUP BY L_ORDERKEY, O_ORDERDATE, O_SHIPPRIORITY
ORDER BY REVENUE, O_ORDERDATE
```

When launching a session in our framework with the query above, we are presented with the user interface of Figure 3. To the left of the figure there is a visual representation of the plan picked by the optimizer. To the right there are different options that allow

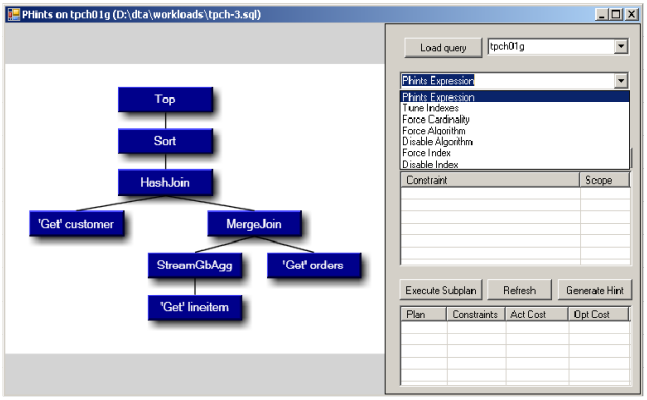


Figure 3: Original plan for the input query.

DBAs to specify constraints. For instance, the highlighted option in the figure enables the direct specification of *Phints* expressions. The framework also exposes some commonly used tuning tasks as macros (e.g., forcing the choice of an algorithm).

Runtime Feedback and Partial Execution. DBAs can choose to execute either the full plan or any execution sub-plan by simply clicking at the appropriate root node. Figure 4 illustrates the result of executing the entire plan of Figure 3. Each node in the plan is annotated with cardinality information (both actual and estimated). Additionally, we color-code the plan to help DBA identify nodes with small (green) and large (red) cardinality errors. In the figure, the `MergeJoin` operator has a large error in cardinality. We can fix this problem by providing a cardinality hint for this specific operator, and forcing the right number of tuples to be estimated at such node (see Figure 4). Once the cardinality hint has been incorporated, the resulting plan is shown in Figure 5. We can see that the cardinality hint resulted in the outer-most join being reversed (and a different join algorithm picked).

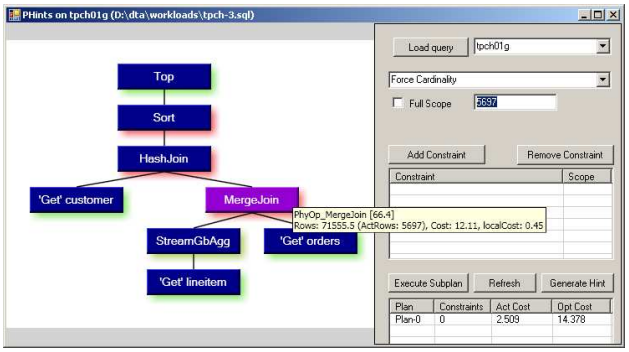


Figure 4: Partial execution and cardinality feedback.

Constraints on Join Ordering. Suppose that we want to specify that both tables `Orders` and `Customer` in Figure 5 join together (without giving a specific order or join implementation). We can do that by visually selecting such nodes (shown in Figure 5), which automatically generates an appropriate *Phints* expression. Additionally, DBAs can directly edit the generated *Phints* expressions. Although the example in the figure is very simple, *Phints* expressions can model rich constraints. For instance, the expression `*[Customer, Orders, Lineitem]` enforces a specific order of tables in the final plan (e.g., `Customer` ⋈ `(Orders` ⋈ `Lineitem)` and `(Customer` ⋈ `Orders)` ⋈ `Lineitem` satisfy the constraint). The expression `Join(StreamAgg(lineitem), orders))` forces the resulting plan to have a pre-aggregate on table `lineitem`. Finally, the expression `*(lineitem, ?, ?)` specifies a "prefix" constraint, in which table `lineitem` is the first table referenced in the plan (without specifying the order of the remaining tables). For more details and examples, refer to [2]. After specifying that both `lineitem` and `orders` be joined together, the system returns the plan in Figure 5 (note that all previously specified constraints are still satisfied).

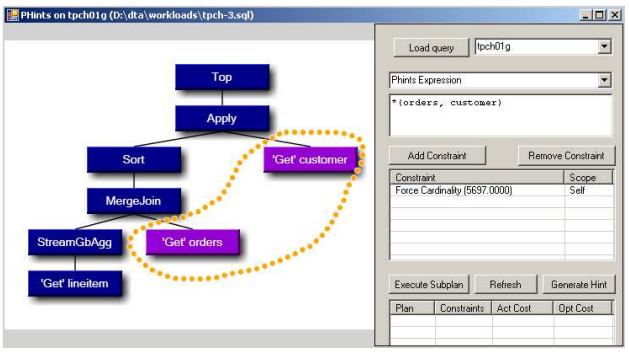


Figure 5: Specifying *Phints* expressions.

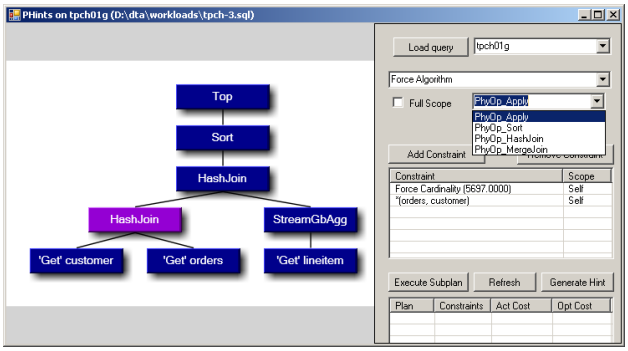


Figure 6: Forcing the choice of an algorithm.

Constraints on Implementation Algorithms. The resulting plan in Figure 6 uses two hash joins. Suppose that we want to try an alternative plan that uses an index-nested loop join between `customer` and `orders`. We can do that by using the "macro" *force algorithm* shown in Figure 6. The possible alternative implementation algorithms for each operator are automatically displayed in the menu.

4. CONCLUSIONS

Optimizer hinting mechanisms for forcing plans are a valuable tool for experienced DBAs to influence the optimizer's choice of execution plans. Unfortunately, existing mechanisms available in commercial DBMS are an ad-hoc collection of hinting options. Additionally, there is no support for visually specifying query hints, which imposes a higher bar on the usability of hints. In this demonstration, we proposed *Phints*, a visual and unifying framework to express an important class of optimizer hints. Additionally, our framework offers DBAs the ability to express a much richer class of constraints beyond what traditional hinting mechanisms offer.

5. REFERENCES

[1] Giving optimization hints to DB2, IBM, 2003. Available at `http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/-index.jsp?topic=/com.ibm.db2.doc.admin/p9li375.htm`.

[2] N. Bruno, S. Chaudhuri, and R. Ramamurthy. Power hints for query optimization. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2009.

[3] I. Chan. Oracle(R) Database Performance Tuning Guide 10g Release 2 (10.2), Oracle, 2008. Available at `http://download.oracle.com/docs/cd/B19306_01-/server.102/b14211/hintsref.htm`.

[4] S. Christodoulakis. Implications of certain assumptions in database performance evauation. *ACM TODS*, 9(2), 1984.

[5] B. A. Patel. Forcing query plans, Microsoft Corp., 2005. Available at `http://www.microsoft.com/technet/prodtechnol/-sql/2005/frcqupln.mspx`.

[6] SQLServer Books Online. Query hint (transact-sql), Microsoft Corp., 2007. Available at `http://technet.microsoft.com/en-us/library/ms181714.aspx`.