Power Hints for Query Optimization

Nicolas Bruno, Surajit Chaudhuri, Ravi Ramamurthy

Microsoft Research, USA
{nicolasb,surajitc,ravirama}@microsoft.com

***Abstract*— Commercial database systems expose *query hints* to address situations in which the optimizer chooses a poor plan for a given query. However, current query hints are not flexible enough to deal with a variety of non-trivial scenarios. In this paper, we introduce a hinting framework that enables the specification of rich constraints to influence the optimizer to pick better plans. We show that while our framework unifies previous approaches, it goes considerably beyond existing hinting mechanisms, and can be implemented efficiently with moderate changes to current optimizers.**

I. Introduction

Relational database management systems (DBMSs) let users specify queries using high level declarative languages such as SQL. The query optimizer of the DBMS is then responsible for finding an efficient execution plan to evaluate the input query. 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 in 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 [1]) that currently deployed cost models can be inherently inaccurate due to several factors:

- Cardinality estimates, especially those on intermediate query expressions, are inferred based on statistical information on base tables making simplifying assumptions on correlation and uniformity.
- Calibration constants used for costing (e.g., the cost of a single disk I/O) might not be fully accurate or consistent with respect to the underlying hardware.
- Cost formulas cannot possibly capture every detail on the complex algorithms used in execution plans.
- Runtime parameters affect execution costs of queries (e.g., the amount of buffer pool memory affects the cost of an index-nested-loops join operator, which would find different numbers of tuples already cached in memory).
- Finally, even if we could determine the precise cost of a single query in isolation, concurrent query executions can have a drastic impact in the true cost of a query, due to factors like contention of indexes, excessive locking, and hotspots, among others.

As a consequence, query optimizers do not always produce optimal plans, and sometimes, for complex queries they might even return plans that are decisively poor. In such cases, database administrators need to "tweak" and correct the wrong choices that led the optimizer to pick a poor execution plan.

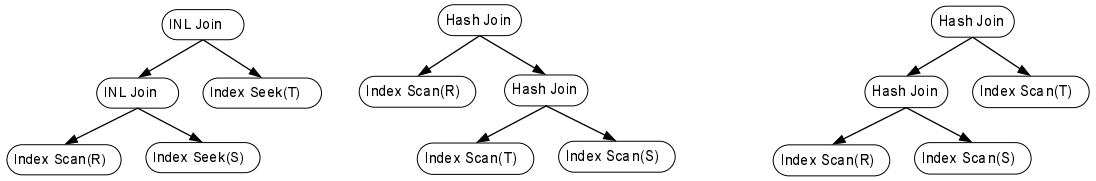
For that purpose, a common mechanism found in commercial database systems is called *query hinting*. Essentially, query hints instruct the optimizer to constrain its search space to a certain subset of execution plans (see Section VIII for an overview of existing query hints in commercial products). The following example clarifies the concept and usage of query hints. Consider the query below:

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

and suppose that the optimizer returns the query plan in Figure 1(a). Furthermore, suppose that the cardinality of $\sigma_{R.d<10}$ is severely underestimated, and thus the index-nested-loop join receives many more tuples than what was anticipated. In this case, the query might perform too many index seeks on S and result in a bad plan overall. In this situation, DBAs can optimize the query using a special hint `OPTION(HASH JOIN)` [1] that is simply concatenated to the query string. In this special mode, the optimizer is forced to choose only execution plans that use hash-based join alternatives. The result of this optimization might produce the execution plan of Figure 1(b). Note that this alternative plan uses a different join order, which might make a DBA wonder 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 finally chooses the most appropriate plan and uses the corresponding hint in a production environment. We make the following observations based on this example.

Hint granularity/expressivity: 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

[1] We use Microsoft SQL Server syntax for the sample hints, but note that the same concepts are also applicable to other DBMSs with minor variations.



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

Fig. 1. Using query hints to fix bad plans.

above). Hints are also not flexible enough to express certain constraints of the structure of the execution plan, such as when trying to force an early pre-aggregation below a join. In many cases, the granularity at which hints constrain the search space is either too coarse, which effectively prevents experienced DBAs from fine-tuning a poorly optimized query, or instead requires a fully specified execution plan [2], which might be too inflexible and tedious for all but the simplest queries. There is also no unifying principle behind available hints (which are a heterogeneous bag of guidelines that DBAs can apply).

Interactivity: The process of tuning a poorly performing query is usually interactive. DBAs might try some hint, observe the resulting plan picked by the optimizer, optionally execute it in scenarios that resemble the production environment, and keep modifying the hints to obtain a plan that is better than what the optimizer originally produced without hints. We believe that this trial and error process is inherent to the tuning of a poorly optimized query. Hints enable DBAs to enhance the search strategy of the query optimizer by using knowledge that is not captured by its cost model.

In this paper, we argue DBAs require a general and consistent framework for constraining the search space of current optimizers, which goes beyond what is currently offered in commercial DBMSs[2]. In this paper, we address hints that influence the plan choice of the optimizer. Our solution should satisfy the following requirements:

- Expressivity: Simple constraints should be easy to specify, but the framework should allow specifying non-trivial constraints to ensure DBAs have the necessary flexibility.
- Efficiency: Obtaining execution plans for given sets of query hints should be efficient (especially for interactive tuning sessions).
- Unification: The solution should not degrade to yet another collection of heterogenous tools, but instead should be based on some unifying methodology.

The rest of the paper is structured as follows. First, in Section II we present a conceptual model to represent the search space of a query optimizer. Then, in Section III we introduce *Phints* (short for *Power Hints*), a small language used to specify rich constraints over the optimizer's search space of plans. Then, in Section IV we present efficient algorithms to obtain the plan that satisfies *Phints* constraints and has the least estimated cost. In Section V we present an architecture that enables interactive query tuning sessions using *Phints*. In Section VI we discuss algorithmic extensions and open problems for our techniques. Finally, in Section VII we report an experimental evaluation of our approach.

[2]We note that existing query hints can additionally affect certain key aspects of query execution (see Section VIII for some examples). Such hints, though important, are outside the scope of this work.

II. Search Space Model

In this paper we model hints as constraints over the search space of execution plans considered by the query optimizer. We next describe an abstract representation of such search space, which is implicitly used in one form or another by all optimizers. The idea is to compactly represent the very large set of possible execution plans by sharing as much structure as possible. Specifically, one can model the space of plans of a given query by using a directed acyclic graph, which we call `PlanSpace`. A `PlanSpace` graph contains two kinds of nodes, denoted `OP` and `OR` nodes. Each `OP` node in the `PlanSpace` corresponds to a specific operator in an execution plan (e.g., hash-join), and all of its children (if any) are `OR` nodes. In turn, each `OR` node in the `PlanSpace` corresponds to a set of equivalent alternatives that can be used interchangeably to form execution plans (e.g. $R \bowtie S \bowtie T$), and every one of its children (there is at least one) are `OP` nodes. We illustrate this representation with the query below:

```
SELECT * FROM R, S
WHERE R.x = S.y
ORDER BY R.x
```

Figure 2 shows a fragment of the `PlanSpace` corresponding to the query above. In the figure, `OR` nodes are represented by black circles, and `OP` nodes are represented with boxes that

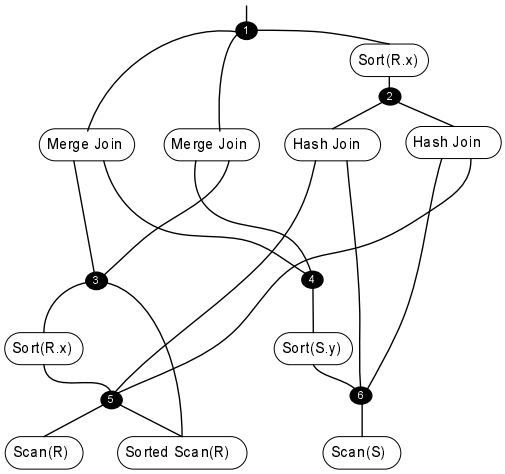


Fig. 2. `PlanSpace` graph representation of the search space.

contain the specific operators in the tree. At the bottom of the figure there are two OP nodes that return tuples from R. The first one, `Scan(R)`, traverses the primary index in the relation and returns tuples in a non-specific order. The second one, `Sorted Scan(R)`, traverses a covering index with clustering key column `x` and therefore returns tuples sorted by `x`. The OR node with label 5 in the figure represents all alternative ways to return tuples from R. In turn, the OR node with label 3 represents all alternative ways of returning tuples of R in `x` order (the rationale for considering such OR node is that the left-most merge join operator in the figure requires tuples sorted by the join columns, and thus cannot be directly connected with the OR node 5). Then, OR node 3 has two children: an explicit `Sort` operator whose child is OR node 5, and the original `Sorted Scan(R)` operator that already returns tuples in the right order and for which the additional sort would be redundant. The rest of the graph can be analyzed similarly up to the root, which is an OR node that compactly represents all possible execution plans for the input query.

Note that this graph model is general enough to be mapped to the search space of different optimizers that are commonly used in commercial systems. For instance, in the context of a transformation-based optimizer such as Cascades [3], OP nodes are directly mapped to physical group expressions in the MEMO, and OR nodes are mapped to subsets of group expressions in a MEMO group which satisfy the same set of required properties. In the context of a bottom-up optimizer like System-R [4], OP nodes are again mapped to implementation alternatives and OR nodes roughly group together all equivalence classes in the dynamic programming algorithm.

Given a `PlanSpace`, we can generate any execution plan by traversing the graph and picking one child for each OR node and all children for each OP node. The role of the optimizer is then to consider all execution plans induced by the `PlanSpace` and picking the one with the least execution cost.

Cost Model

Consider an OP node N. All of N's children (if any) are OR nodes whose children by definition return the same result. Therefore, the cost of any subplan rooted at N is the sum of costs of its children plus a *local processing cost*, which is independent of the choices below N. Consider the `Merge Join` node at the left of Figure 2. No matter what choice we make for the left input (OR node 3) or the right input (OR node 4), the `Merge Join` operator would locally do the same work.

The cost model of an optimizer therefore approximates the local processing time for each possible operator in the graph. It essentially associates, with each OP node N, a real number *LC(N)*, that corresponds to the local execution time of the corresponding operator. Obtaining the cost of the best subplan rooted at each node N in the `PlanSpace`, denoted *BC(N)*, can be done as follows:

$$BC(N) = \begin{cases} LC(N) + \sum_{N_i} BC(N_i) & \text{if } N = \texttt{OP}(N_1, \ldots, N_k) \\ \min_{N_i} BC(N_i) & \text{if } N = \texttt{OR}(N_1, \ldots, N_k) \end{cases}$$

The definition of *BC* satisfies the well-known *principle of optimality* [5], which states that every subplan of an optimal plan is itself optimal. The best execution subplan rooted at node N is obtained as follows:

$$BP(N) = \begin{cases} OP_N(BP(N_1), \ldots, BP(N_k)) & \text{if } N = \texttt{OP}(N_1, \ldots, N_k) \\ BP(\text{minarg}_{N_i}\ BC(N_i)) & \text{if } N = \texttt{OR}(N_1, \ldots, N_k) \end{cases}$$

where OP_N is the operator associated with an OP node N, and *minarg* returns the N_i that minimizes value $BC(N_i)$.

In this conceptual model, current query hints can be seen as constraining the search space of plans (i.e., the `PlanSpace`) that is considered by the optimizer. For instance, the hint `OPTION (HASH JOIN)` would essentially remove from the original `PlanSpace` any OP node that is associated with a non hash-based join implementation, and would return the best execution plan out of that subgraph (if at least one such alternative exists). In the next section, we introduce *Phints*, a simple and powerful language to specify rich constraints over the plans considered by the optimizer.

III. The Language *Phints*

As discussed earlier, there is no uniform abstraction that can cover the set of hints in current systems. In this section we introduce *Phints*, a simple hinting language that can provide such an abstraction and at the same time is richer than today's query hints. *Phints* expressions are defined as tree patterns annotated with constraints. Conceptually, the evaluation of *Phints* expressions can be done by enumerating all possible execution plans in the `PlanSpace`, filtering out the alternatives that do not match *Phints* expressions, and finally returning the matching execution plan with the smallest estimated cost.

A subplan P in the `PlanSpace` *matches* a *Phints* tree pattern S if there is a mapping M between nodes in S and distinct OP nodes in P (see Figure 3) such that:

1) For each leaf node s in S, $M(s)$ is a leaf node in P (and those are the *only* leaf nodes in P).
2) For each internal node s in S with children $\{s_1, \ldots, s_k\}$, every $M(s_i)$ is a descendant of $M(s)$, and no $M(s_i)$ is a descendant of $M(s_j)$ for $i \neq j$.
3) For each node s in S, $M(s)$, and its descendants in P, must satisfy the constraints defined on s.

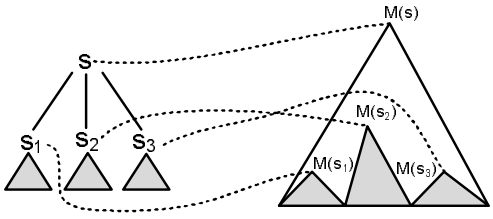


Fig. 3. Graphical representation of a pattern match.

Specifically, each node s in a tree pattern S can be optionally annotated with constraints of the form:

- *Self constraint*: a predicate that must be satisfied by $M(s)$ in the `PlanSpace`.
- *Subtree constraint*: a predicate that must be satisfied by each descendant of $M(s)$ in the `PlanSpace`.

- *Order constraint*: if an order constraint is present, an in-order traversal of the `PlanSpace` must return $M(s_1), \ldots, M(s_k)$ in that precise order.

Predicates in self- and subtree-constraints for node s are:

- *force(A)*, which specifies that the matching $M(s)$ must have A as the associated implementation algorithm (e.g., *force(Hash Join)*).
- *disable(A)*, which specifies that the matching $M(s)$ cannot have A as the implementation algorithm.
- *condForce(A_L, A_P)*, which specifies that the matching $M(s)$ must use A_P as the implementation algorithm but only if $M(s)$ is of type A_L (e.g., forcing a join operator to be hash-based is done by *condForce(Join, Hash Join)*).
- *forceTable(T)*, where T is a table, which specifies that the matching node $M(s)$ must be a leaf node that operates over T[3].

Concisely Expressing Phints *Expressions*

We next describe a concise syntax for specifying *Phints* expressions that is linearized and can be appended to a `SQL` query string in the same way that current query hints are. We use the following conventions:

- The tree pattern is written in infix order, using parenthesis to denote parent-child relationships, or square brackets if the order constraint is present in the parent node.
- If there are no self constraints, the node is denoted with the symbol $*$. Otherwise, the node is denoted with A for a *force(A)* self-constraint, $!A$ for a *disable(A)* self-constraint, T for a *forceTable(T)* constraint, and $L{=>}P$ for a *condForce(L,P)* constraint.
- Subtree constraints are written next to the node description, in curly braces.

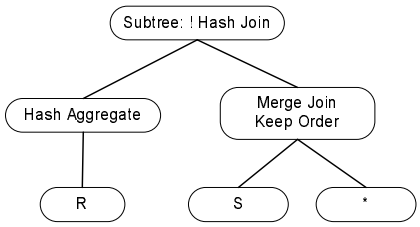


Fig. 4. Graphical representation of a *Phints* expression.

Figure 4 shows a sample tree expression that illustrates the features of the language, and would match any execution subplan which (i) is a three-way join between tables R, S, and some other table in the query, (ii) no join is hash-based, (iii) the inner join is merge-based between S and the remaining table *in that order*, and (iv) there is an early pre-aggregation on table R. Using the above notation, this expression can be written as follows: `*{!HashJoin}(HashAggregate(R), MergeJoin[S,*])`

Although outside the scope of this paper, we are aware that alternative linearized representations can be generated, such as the following XML-based representation for the constraint above:

```
<Op SubTree="!HashJoin">
  <Op Self="HashAggregate">
    <Op Table="R"/>
  </Op>
  <Op Self="Merge Join" Ordered="True">
    <Op Table="S"/>
    <Op Table="*"/>
  </Op>
</Op>
```

Relationship with XML Query Processing

Our query model is similar to XML pattern matching. We might be tempted to equate the `PlanSpace` with a specialized XML document, and *Phints* expressions with XPath or XQuery query fragments [6]. While in principle these two problems are similar (both deal with pattern matching over trees) there are substantial differences between the approaches:

- The representation of the optimizer search space as a `PlanSpace` graph is qualitatively different from traditional XML documents, which are mostly trees (with a small number of `ID/IDREF` values). The graph-based representation of the `PlanSpace` would enable us to use different techniques to efficiently obtain results.
- For an execution plan P to match a *Phints* tree pattern S, P must not contain any additional leaf node (see condition 1 earlier in this section). The *Phints* expression `*(R,S)` would not match the execution subplan `(R` $\bowtie$ `T)` $\bowtie$ `S`, which would be considered a match in the analogous evaluation of the XML query `*[.//R]//S`.
- The notion of subtree constraints, which evaluate predicates over every node in subtrees of the matched query plan is a construct that, as far as we know, has not been considered in the context of XML query processing.
- Rather than obtaining all execution plans that match the pattern (as XML queries do), we are interested in the one with the minimum optimizer estimated cost.

A. Additional Examples

We next introduce additional examples that further illustrate the capabilities of *Phints* expressions. We first show how some of the common hints supported by the current systems fit naturally in our framework and then we outline some more complex scenarios that can be easily specified in *Phints*, but are not be supported by most current systems.

1) Consider the `OPTION(HASH JOIN)` hint used in Section I to force all joins to be hash based. This can easily be specified as `*{Join=>Hash Join}(R,S,T)`.
2) Consider a join of four tables R, S, T and U and suppose that we require the tables to be joined in that particular order. Some commercial systems support the ability to specify a join order in the final plan based on the syntactic order of the tables in the input SQL statement (e.g., `OPTION(FORCE ORDER)` in Section I). We can specify any arbitrary join tree easily using *Phints* expressions, such as for instance: `*[R,*[*[S,T],U]]`.

[3]We assume that there is a single node in the `PlanSpace` to represent access-path related execution subplans, but lift this restriction in Section VI.

Note, however, that there are two interesting generalizations of this pattern. First, we can force certain join implementations for some nodes, disable others, and leave a third group unspecified (so that the optimizer finds the best alternative), as illustrated by the *Phints* expression `HashJoin[R,*[!NLJoin[S,T],U]]`. Second, by specifying the expression `*[R,S,T,U]` we can match any execution plan P that joins only these four tables, and for which an in-order traversal of the nodes returns the tables in the right order, is valid (e.g., `(R,(S,T),U)`, `(R,(S,(T,U)))`, `(((R,S),T),U)`).

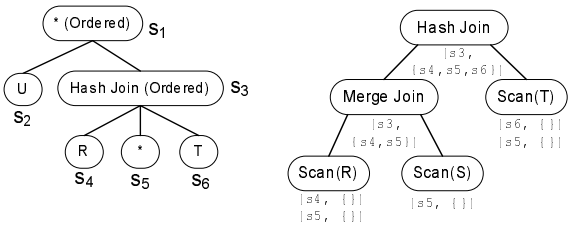
3) Some systems support the specification of a table to be the first one in the join order (e.g., `LEADING` hint in Oracle [7]). For instance, if we want the optimizer to pick a plan that uses R as the initial joined table, we can use the following *Phints* expression: `*[R,*,*,*]`. Again, note that we can specify more complex interactions with *Phints* expressions, such as `*[*(R,S),*,*]`, which instructs the optimizer to pick a join order that *"begins"* with tables R and S (in some order), and the joins the result with the remaining tables.
4) Finally, suppose that in the previous example R is a fact table and the remaining ones are dimension tables. Suppose that we want to force a plan that joins the dimension tables first (possibly using cartesian products) and then uses an index to access the fact table. This plan is interesting since it avoids scanning the large fact table R. At the same time, however, we do not want to explicitly specify the ordering of the dimension tables. This can be specified using the following *Phints* expression: `IndexNestedLoops[*(*,*,*),R]`.

IV. Evaluating *Phints* expressions

In this section we present our main algorithm to obtain the most efficient execution plan in a `PlanSpace` subject to *Phints* constraints. In Section IV-A we introduce the notion of a candidate match, which keeps track of partial matches between nodes in *Phints* expressions and those in the `PlanSpace`. Then, in Section IV-B we show how we can compute candidate matches bottom-up in a `PlanSpace`. In Section IV-C we review the original dynamic programming algorithm to obtain the best execution plan under no constraints, and discuss why the principle of optimality, which is valid in the original formulation, does not hold under constraints. Finally, in Section IV-D we introduce the notion of *interesting matches* to address this problem and present the main algorithm to evaluate *Phints* expressions.

A. Candidate Matches

Consider a *Phints* expression `*[R,S,T]` and an execution subplan P=$R \bowtie S$. Although the subplan does not match the full expression, it does so "partially," since further joining P with table T would result in a match as described in the previous section (note that $R \bowtie T$ is not a partial match). We therefore need a mechanism to refer to and reason with such partial matches. Consider a tree pattern S and a subplan



(a) *Phints* expression S. (b) Execution subplan P.

Fig. 5. Candidate Matches in *Phints*.

P. We define a *candidate match* between P and S as (s, c), where s is a node in S and $c = \{s_1, \ldots, s_k\}$ is an *ordered* subset of children of s, such that:

1) There exist k disjoint subtrees of P, $\{p_1, \ldots, p_k\}$, such that each p_i matches s_i as in the definition of Section III.
2) If s has an order constraint, then an in-order traversal of P returns $p_1, \ldots, p_k$ in this order, and $\{s_1, \ldots, s_k\}$ is a *subsequence* of *children(s)* (there are no "gaps" in s_i).
3) If $\{s_1, \ldots, s_k\}$ cover all of s' children, then P matches s as in the definition of Section III.

Consider the sub-expression `HashJoin[R,*,T]`, graphically shown in Figure 5(a), and the execution subplan of Figure 5(b). Below each node in the execution plan, we show the candidate matches of the operator. For instance, the candidate matches for `Scan(R)` are both $(s_4, \{\})$ and $(s_5, \{\})$. In general, the candidate matches for a leaf node L in an execution plan are all $(s_l, \{\})$ for leaf nodes s_l in the *Phints* expression for which (i) L satisfies s_l's self-constraints and (ii) the subtree-constraints of all s_l's ancestors.

Also in Figure 5, the `Merge Join` operator matches $(s_3, \{s_4, s_5\})$ because (i) subtrees `Scan(R)` and `Scan(S)` match s_4 and s_5, respectively, (ii) the order of children is preserved without gaps, and (iii) since s_6 is not a match, we do not need to additionally satisfy the `HashJoin` self-constraint on s_3. By a similar reasoning, we can also show that the `Hash Join` operator matches both s_3 and all its children $\{s_4, s_5, s_6\}$. We next formally explain how to derive candidate matches for a non-leaf node, given the candidate matches for its children.

B. Computing Candidate Matches

We next describe how to compute all candidate matches in a `PlanSpace` bottom up. We first show how we can derive a candidate match for a node given matches for each of its children, and then show how to compute all candidate matches.

Deriving A Candidate Match

We next show our algorithm to derive candidate matches of a given execution subplan (for a node `n` in a tree, we use `parent(n)`, `children(n)`, and `isRoot(n)` to denote, respectively, the parent of `n`, the children of `n`, and whether `n` is the root of the tree). To further simplify the notation, the algorithm uses $m.s$ and $m.c$ to denote the node and ordered set of children of a given candidate match m. Consider an `OP` node `A` with arity k in a given execution plan, and suppose that each

```
deriveMatch (A: OpNode of arity k, {m1, ..., mk}: matches for A's children)
returns candidate match for A
01 for each mi=(si, ci)
02   if ci = children(si)
03     mi = (parent(si), {si})
04 if (∃ mi, mj such that mi.s ≠ mj.s)
     or (∃ mi, mj, i ≠ j, such that mi.c ∩ mj.c ≠ ∅)
05   return NULL;
06 rm = (m1.s, m1.c ⊕ m2.c ⊕ ... ⊕ mk.c) // ⊕ is ordered union
07 if  (A does not satisfy rm.s subtree constraints)
     or (rm.s has order constraint and ∃i>0 1+rm.c[i-1] ≠ rm.c[i] )
     or (rm.c = children(rm.s) and A does not satisfy rm.s self constraints)
08   return NULL
09 return rm
```

Fig. 6. Deriving candidate matches bottom-up in an execution plan.

m_i ($i \leq k$) is a candidate match of the i-th child of A. Then, deriveMatch(A, $\{m_1, \ldots, m_k\}$) returns the corresponding match for A (if any) based on matches of A's children. Lines 1-3 in Figure 6 promote each of the fully matched children into an equivalent form in terms of the parent node. For Figure 5(a), this step would convert $(s_3, \{s_4, s_5, s_6\})$ into $(s_1, \{s_3\})$, but would keep $(s_3, \{s_5, s_6\})$ unchanged. Then, line 4 checks whether all transformed m_i partially match the same tree pattern m_i.s (otherwise, we cannot extend the match for A). Additionally, line 4 checks whether the same candidate match is returned by multiple children (which would result in a non-unique mapping). If no match is possible, we return NULL in line 5. Otherwise, we assemble the result rm by concatenating the candidate matches of each child m_i in line 6, and then perform additional checks on rm in line 7. First, we check whether the input A node satisfies the subtree constraints of rm's tree pattern rm.s. If rm.s has an order constraint, we check whether order is satisfied by A and its subtrees. Finally, if rm is a full match (i.e., rm.c=*children*(rm.s)), we check if the self-constraints of rm.s are satisfied. If so, we return the valid match for A in line 9.

As an example, consider matches $(s_3, \{s_4, s_5\})$ and $(s_6, \{\})$ for the Merge Join and Scan(T) operators in Figure 5(b). The procedure deriveMatch(A), where A is the Hash Join node in Figure 5(b) would first promote $(s_6, \{\})$ into $(s_3, \{s_6\})$ but would leave $(s_3, \{s_4, s_5\})$ unchanged. Both matches now share the same tree node (s_3) and have disjoint children subsets. The new match is assembled as $(s_3, \{s_4, s_5, s_6\})$, and since it satisfies all self- and subtree-constraints, is therefore returned as a match for A itself.

Computing All Candidate Matches

Each operator in an execution plan might have multiple candidate matches (e.g., see the Scan(R) node in Figure 5(b)). The algorithm in Figure 7 can be used to propagate candidate matches bottom-up in an execution plan. For a given operator A of arity k (note that k is rarely larger than two), and sets of matches $M_1, \ldots, M_k$ for each of A's children, the algorithm first identifies whether some M_k is a full candidate match in lines 2-4, and if so, it adds it to the result set[4]. Then, for each combination of input candidate matches, lines 5-7 try to extend the partial matches with respect to A using the algorithm deriveMatch described in Figure 6. We address a special case for unary OP nodes in lines 8-10. Specifically, note that even if an unary operator in the execution plan can be extended by deriveMatch, the same operator can still share its child matches. (This is not the case for non-unary operators, since the resulting execution plan would contain additional leaf nodes). Lines 8-10 thus check whether the candidate matches in the single child of an unary operator are still valid for A, and if so, these are added to the result. Finally, all matches accumulated in lines 4, 7 and 10 are returned.

```
calculateMatches ( A:OpNode of arity k,
   {M1, ..., Mk}: match sets for A's children)
returns candidate matches for A
01 result = {}
02 for each mj in each Mi
03   if (isRoot(mj.s) and mj.c = children(mj.s))
04     result += mj
05 for each (m1, ..., mk) in (M1 ×...× Mk)
06   rm = deriveMatch(A, m1, ..., mk)
07   if (rm ≠ NULL) result += rm
08 if (k=1) for each mi in M1
09   if (A satisfies mi.s subtree constraints)
       and (mi.c = children(mi.s) ⇒
           A satisfies rm.s self constraints)
10   result += mi
11 return result
```

Fig. 7. Calculating all candidate matches.

C. Principle of Optimality Revisited

In Section II we presented a function *BP(N)* to obtain the execution plan in a PlanSpace with the minimum cost (where no constraints are present). We now show how we can implement this recursive definition. We begin by extending the nodes in the PlanSpace as follows. Each OR node N is augmented with a field N.bestPlan which would eventually contain the best subplan rooted at N. Similarly, each OP node N

[4]Our matching semantics require that the full expression be satisfied for *some subtree* of an execution plan, and thus we need to propagate full matches upwards in the tree.

```
getBestPlan(OrNode O) (returns OpNode)
01 if (O.bestPlan = NULL) // not yet memoized
02   for each (OpNode A_i in children(O))
03     if (A_i.bestCost = ∞) // not yet memoized
04       A_i.bestCost = localCost(A_i) // See Section II
05       for each (OrNode O_j in children(A_i))
06         bestSubPlan = getBestPlan(O_j) // recursive call
07         A_i.bestSubplans[j] = bestSubPlan
08         A_i.bestCost += bestSubPlan.bestCost
09     if (O.bestPlan = NULL or A_i.bestCost < O.bestCost)
10       O.bestPlan = A_i; O.bestCost = A_i.bestCost
11 return O.bestPlan
```

Fig. 8. Obtain the best execution plan from a PlanSpace.

with k children is augmented with (i) an array bestSubplans of size k which would contain the best subplans for the operator rooted at N, and (ii) a real value N.bestCost which would contain the cost of the best execution plan rooted at N.

Algorithm getBestPlan, shown in Figure 8, is called with the root OR node in the PlanSpace, and returns the root OP node of the best execution plan. This execution plan can be found by recursively traversing bestSubplans arrays. Initially, N.bestPlan=NULL for each OR node N, and N.bestCost=∞ for each OP node N in the PlanSpace.

Line 1 implements memoization and guarantees that each OR node is computed once. Processing an OR node iterates over each of the child alternatives (line 2). Line 3 implements a second layer of memoization, which guarantees that each OP node is processed once. Processing an OP node starts by computing its local cost (line 4) and then (i) recursively calls getBestPlan for each subplan (line 6), (ii) caches the best result in the array bestSubplans (line 7), and (iii) adds the total cost for the node in line 8. Line 9 keeps the OP node with minimum cost for node O, which is memoized and returned.

Principle of Optimality and Candidate Matches

In the presence of *Phints* expressions, we need to generalize the algorithm in Figure 8 so that it only considers execution plans that match the given constraints. An initial idea might be to simply generalize the notion of bestSubplan in line 9 of Figure 8 so that it gives preference to execution plans that match larger portions of the *Phints* expression (although they might be suboptimal in the unconstrained case). For instance, we could change the condition in line 9 of Figure 8 to:

```
O.bestPlan = NULL
or "A_i.matches >_M O.matches"
or (A_i.matches = O.matches and
     A_i.bestCost < O.bestCost)
```

We would then prefer a subplan P_1 that matches "more" than another subplan P_2, although P_1.bestCost > P_2.bestCost. We now show that, no matter how we rank candidate OP nodes (i.e., the semantics of $>_M$), such extensions to getBestPlan would produce wrong results. As an example, consider the following query:

```
SELECT R.x, S.a FROM R, S WHERE R.x = S.y
UNION
SELECT T.x, U.b FROM T, U WHERE T.x = U.y
```

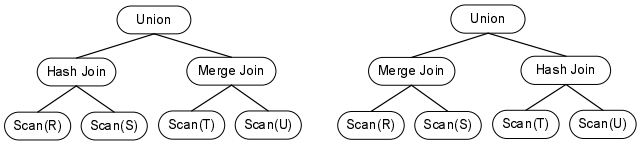


Fig. 9. *Phints* expressions might violate the principle of optimality.

and the *Phints* expression HashJoin(*,*). Furthermore, assume that the smallest cost alternative for either $R \bowtie S$ and $T \bowtie U$ is a merge-join implementation. In this situation, however, the hash-based alternative matches "more" than the merge-based one for both the OR nodes which model $R \bowtie S$ and $T \bowtie U$, and therefore would be picked by this suggested modification. The net effect is that the chosen plan would use hash-based joins for each input to the UNION operator. Note that both plans in Figure 9, which are more efficient and also satisfy the *Phints* constraint, would not be considered.

This problem is analogous to what happens in a System-R join reordering algorithm based on dynamic programming. There, the principle of optimality does not hold whenever a suboptimal implementation of a given sub-expression returns tuples in a sort order that is different from that of the optimal implementation. The reason is that ancestors of such expression might exploit the output sort orders and compensate for its local sub-optimality. System-R addresses this issue by introducing the concept of *interesting orders*, and essentially consider the best implementation for *every tuple order* for a given query sub-expression ([4], [8])[5]. We similarly generalize the algorithm in Figure 8 to return the optimal execution plan that satisfies arbitrary *Phints* constraints.

D. Putting it All Together

The previous section illustrates why we need to handle situations in which different OP children nodes of a given OR node result in different, non-comparable candidate matches. In the example of Figure 9, both Merge Join(R,S) and Hash Join(R,S) should be considered. While the hash-based alternative matches a larger "pattern" than the merge-based alternative, the merge-based alternative is cheaper and can still be part of a valid execution plan.

[5]We adopt this technique in the PlanSpace by explicitly taking in consideration the order of output tuples (e.g., OR nodes 3 and 5 in Figure 2).

Interesting Candidate Matches

In general, we need to keep around a set of equivalent execution plans that are non-comparable with respect to their respective matches. Specifically, we need an operator which, given a set of execution plans associated with a given OR node, returns the subset of "non-comparable" alternatives. This is the *skyline* operator [9], which, for a set P and a partial order $\leq$, is defined as $\{p \in P$ such that $\forall p_j \in P : p_j \leq p\}$. We use the following partial order for comparing plans P_1 and P_2:

```
P1 ≤ P2 if   P1.matches ≤ P2.matches OR
             ( P1.matches = P2.matches AND
               P1.cost ≤ P2.cost )
```

which in turn depends on a partial order for candidate matches. Consider sets of matches M_1 and M_2. We say that $M_1 \leq M_2$ if for each match $m_1 \in M_1$ there exists a match $m_2 \in M_2$ such that $m_1 \leq m_2$. In turn, this definition is based on a third partial order between individual matches. Given two matches m_1=(s_1, c_1) and m_2=(s_2, c_2) we say that $m_1 \leq m_2$ if (i) s_1 is an ancestor of s_2, or (ii) $s_1 = s_2$ and $c_1 \subseteq c_2$.

The partial order described above requires us to keep track of multiple execution plans for a given root OP node. For that purpose, we define the concept of an *extended* OP node, which is a special kind of OP node that (i) is additionally associated with a set of candidate matches, and (ii) its children are *extended* OP nodes themselves. The same OP node A in the original PlanSpace might now be associated with multiple extended OP nodes, each one binding to different execution plans (with the same root A) and different sets of candidate matches. Also, we replace bestPlan in OR nodes (which contains the best OP node) with a set bestEPlans of extended OP nodes (which contains the set of non-comparable subplans).

The Main Algorithm

Algorithm getBestMatches, shown in Figure 10, takes as input an OR node in the PlanSpace and returns a set of extended OP nodes, each one of these incomparable with respect of the set of candidate matches they satisfy. Lines 1-3 are analogous to those in the original algorithm of Figure 8. Processing an OP node starts by recursively calling getBestMatches for each of the OP child nodes in lines 4-5. The result of each recursive call, which is itself a set of extended OP nodes, is stored in $\texttt{alts}_j$. Lines 6-11 assemble a new extended OP candidate for each combination of child extended OP nodes in $\texttt{alts}_j$. Each such extended OP node is created from the current OP node $\texttt{A}_i$ being processed in line 7. The children of this extended OP node are set in line 8, its cost is computed in line 9, and its matches are computed in line 10 by calling calculateMatches (described in Figure 7). Each one of these alternative extended OP nodes is analyzed and the non-dominated ones are kept. Finally, we aggregate all non-dominated alternative extended OP nodes from all children of the input OR node in lines 12-13, and return such set. To obtain the best execution plan in a PlanSpace for given *Phints* expressions, we first call getBestMatches passing the root of the PlanSpace. Then, we inspect each resulting extended OP node and return the alternative that satisfies all *Phints* constraints and has the smallest estimated cost.

V. The *Phints* Framework

The previous sections introduced an algorithm that evaluates a PlanSpace and returns the most efficient execution plan that satisfies *Phints* expressions. We now discuss how this evaluation module fits in our proposed *Phints* framework. Figure 11 illustrates a block diagram of our proposed solution. We do not take a position on the inner-workings of the DBMS (shown at the right of the figure). However, we do require that the query optimizer supports the following two interfaces:

1) PlanSpace generation: This interface takes a query and returns its corresponding PlanSpace (rather than its best execution plan). Each optimization framework might require specific extensions to expose this interface (e.g., transformation-based top-down optimizers based in the Cascades Framework [3], [10] can use the techniques in [11] to return the PlanSpace).
2) Plan forcing: This interface takes a fully specified execution plan and evaluates it in the server. Most DBMS already support this interface (e.g., see [2]).

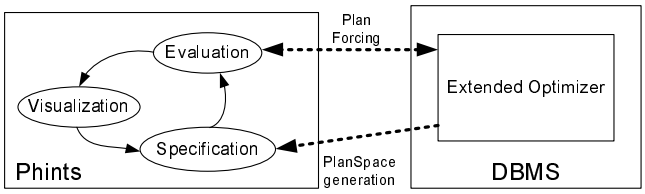


Fig. 11. The *Phints* Framework.

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, a *Phints* client calls the optimizer *a single time* to generate the PlanSpace corresponding to the query. Then, the DBA interacts in the *Phints* framework as follows:

Specification: The DBA, based on knowledge about the application, the runtime system, and the current execution plan, decides to constrain the input query by creating *Phints* expressions or modifying previously submitted ones. DBAs can use the concise representation of *Phints* expressions (discussed in Section III) or some higher-level visual interface.

Evaluation: Once a *Phints* specification is in place, we use the algorithms in Section IV to produce the best execution plan in the client. This step does not require a round-trip to the server, but it is fully in sync with the DBMS since we work off the PlanSpace produced earlier and knowledge about the costing logic in the server. The resulting execution plan can optionally be sent to the server and executed to obtain feedback on cost, cardinality, and runtime characteristics.

Visualization: The resulting plan (with optional feedback from actual execution) is displayed graphically. Multiple execution trees resulting from previous *Phints* expressions can be compared side-by-side to help

```
getBestMatches (OrNode O)
returns set of ExtendedOpNode
01 if (O.bestPlans = {}) // not yet memoized
02    for each (OpNode A_i in children(O))
03       if (A_i.bestEPlans = {}) // not yet memoized
04          for each (OrNode O_j in children(A_i)) // assume A_i has arity k
05             alts_j = getBestMatches(O_j) // recursive call, alt_j is set of extended OP nodes
06          for each (alt_1, ..., alt_k) in alts_1 ×...× alts_k
07             EA_i = createExtendedOpNode(A_i)
08             EA_i.bestSubplans[j] = alt_j (j=1..k)
09             EA_i.cost = localCost(EA_i) + Σ_i alt_i.cost
10             EA_i.matches = calculateMatches(A_i, {alt_1, ..., alt_k})
11             A_i.bestEPlans = skyline (A_i.bestEPlans ∪ {EA_i})
12       for each EA in A_i.bestEPlans
13          O.bestPlans = skyline (O.bestPlans ∪ {EA})
14 return O.bestPlans
```

Fig. 10. Obtain the best execution plan from a `PlanSpace` that satisfies *Phints* constraints.

determine whether a good execution plan has been identified or otherwise more iterations are needed.

When the DBA is satisfied with an execution plan for the problematic query, *Phints* generates a description of such plan to be passed to the optimizer in a production system using the *plan forcing* interface described above. Figure 12 illustrates a simple prototype that implements some of the ideas discussed in this Section. The prototype in the figure uses a visual paradigm to create constraints, which are then translated into *Phints* expressions. For instance, the highlighted execution subplan in the figure corresponds to the *Phints* tree expression `StreamGBAgg(MergeJoin[orders, lineitem])`. Resulting execution plans can be evaluated and compared, and the system has the ability to generate fully specified hints for use in production environments.

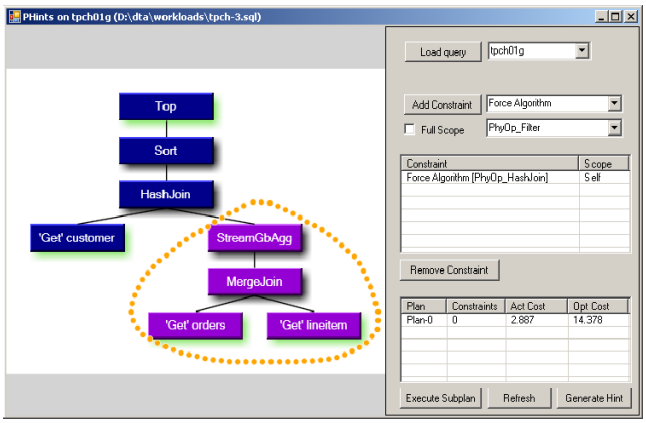


Fig. 12. A Prototype Implementation of the *Phints* Framework.

VI. Extensions and Future Work

We next discuss some extensions and open problems that were not addressed earlier to simplify the presentation.

A. Extensions to the Matching Semantics

The matching semantics for *Phints* expressions in Section III require that each leaf node in an execution subplan P matches a leaf node in the tree pattern S. In fact, any leaf node in an execution plan that satisfies the self- and subtree-constraints of an expression leaf node is deemed a valid match. We now relax this notion, which allows increased flexibility in the language, such as the possibility of matching whole execution subplans, as well as the ability to support access-path related hints. Suppose that, at least conceptually, we associate an explicit set of candidate nodes in the `PlanSpace` (not necessarily leaf nodes) to a leaf node in a *Phints* expression. Let us denote such set of candidate nodes for a pattern S the *virtual leaf nodes* for S. Then, we change the first condition in the matching semantics of *Phints* expressions from:

1) For each leaf node s in S, $M(s)$ is a leaf node in P (and those are the *only* leaf nodes in P).

to:

1) For each leaf node s in S, $M(s)$ is a virtual leaf node in P for s, and if we remove any subtree of a virtual leaf node in the execution plan, those virtual leaf nodes become the only leaf nodes in P.

Now, the implementation of `calculateMatches` in Figure 7 needs to be slightly modified by adding, to the resulting set of matches of an `OP` node n, leaf candidate matches for all nodes s in *Phints* expressions for which (i) n is a virtual leaf node of s, and (ii) n satisfies all self- and subtree-constraints of s. Of course, if we associate all leaf nodes in the `PlanSpace` as virtual leaf nodes of every leaf node in *Phints* expressions, we are back to the scenarios described in Sections III and IV. However, virtual leaf nodes enable additional functionality to the basic query model, as we discuss next.

Access-Path Hints

Perhaps more importantly, virtual leaf nodes are important when additionally dealing with index strategies. As a simple example, consider the *Phints* expression `HashJoin[R, S]`. Suppose that there is an execution plan that uses an index intersection to obtain the required tuples in R, and therefore looks like `HashJoin(Intersection(`$\mathtt{I}_R^1$`, `$\mathtt{I}_R^2$`), S)`. Unfortunately, this plan would not match the *Phints* expression because there would be an additional leaf node (either $\mathtt{I}_R^1$ or $\mathtt{I}_R^2$) which would violate the first matching condition of

Section III. To address these scenarios, we rely on the concept of virtual leaf nodes as follows. Consider a leaf node s in a *Phints* expression that has a self-constraint of the type `forceTable(T)`. In this situation, we obtain all nodes in the `PlanSpace` that correspond to execution subplans over `T` alone, and associate those nodes as the virtual leaf nodes for s. This concept makes possible to match any execution plan p that returns tuples from a table `T` (using sequential scans, index intersections, or other alternatives) to a leaf expression node s that has a constraint of type `forceTable(T)`.

To fully enable access-path hints, we add another predicate to the self-constraints of an expression node. By specifying `forceIndex(I)`, where `I` is an index name, an expression node s would only match nodes that exploit such index in the plan. The following examples might help clarify these issues:

- `*(R, S)` would match any single-table subplan over `R` joining any single-table subplan over `S`.
- `*(Intersection[`$\mathtt{I}_R^1$`, R], S)` would additionally require that the subplan over `R` is an intersection plan that uses index $\mathtt{I}_R^1$ (note the concise representation of `forceIndex` constraints, similar to those for `forceTable` constraints). The second child of the `Intersection` node is again `R`, meaning that any single-table execution plan over `R` would match, such as another index scan to intersect with $\mathtt{I}_R^1$, or even the result of intersecting multiple such indexes.

Matching Subexpressions

Consider the *Phints* expression `HashJoin[R,*]`, which matches a binary join between R and a single additional table. If we want R to truly be a prefix in the execution plan, we need to extend the expression as `HashJoin[R,*,...,*]` with as many `*` symbols as remaining tables in the query. Alternatively, we could express this more concisely as `HashJoin[R, ` $\hat{*}$`]`, where the special symbol $\hat{*}$ allows matching an arbitrary execution subplan. To enable such scenarios, we simply assign to an expression leaf node marked as $\hat{*}$ all nodes in the `PlanSpace` as its virtual leaf nodes, which results in the desired matches.

B. Open Problems

We next introduce some topics of future research.

Revisiting Interesting Candidate Matches

In Section IV-C we showed that the principle of optimality is violated for general *Phints* expressions. We then introduced the concept of interesting candidate matches to be able to correctly optimize a query under *Phints* constraints. It would be interesting to characterize a subset of the *Phints* language that is guaranteed to satisfy the principle of optimality. This open problem is not trivial due to interaction of several factors:

- The search space of the optimizer (e.g., what would happen if we do not allow bushy trees?)
- The interaction of *Phints* expressions and the input query (e.g., it seems that `HashJoin[*,*]` might be "safe" for a query that only joins two tables R and S, but is "unsafe" for the query of Figure 9).
- The structure of *Phints* expressions themselves (unary operators might result in violations of the principle of optimality even for fully constrained *Phints* expressions due to repeated unary operators in an execution plan).

Server-Side Pruning

Our architecture, described in Section V, allows DBAs to pose hints and observe results without repeatedly calling the query optimizer. This behavior is similar to that of [11], where we pay some initial cost to obtain an encoding of the search space of plans by the optimizer, and then reuse this model to have fast re-optimizations under varying physical designs. However, if we are interested in a single optimization with respect to a set of *Phints* expressions, we can think of "pushing" this constrained optimization to the DBMS engine itself. In that case, some extensions to the classical branch and bound techniques present in modern optimizers might be required. We note, however, that these extensions depend on the specific optimizer architecture.

Conditional Expressions

So far, *Phints* expressions like `HashJoin[R, S]` would not only discard merge-based join alternatives between `R` and `S`, but also force the join of these tables as part of the final execution plan. There is another set of constraints, which we call *conditional expressions*, that can be used to specify a richer set of scenarios. As an example, a conditional expression `HashJoin[R,S]` means that *if* we decide to use a join between tables `R` and `S` in the final plan, *then* the join must be hash-based. However, we are free to choose any other join ordering. Conditional expressions are very interesting and can express non-trivial constraints, but can be challenging to implement for subtree constraints. A detailed study of such expressions and other extensions is part of future work.

VII. Experimental Evaluation

We next report an experimental evaluation of the techniques described in Section IV. We implemented the interfaces described in Section V on Microsoft SQL Server 2005, and the *Phints* framework as a `C++` client, which includes the visual user interface shown in Figure 12. For our experiments we used the queries in the `TPC-H` workload [12]. The purpose of the experimental evaluation is not to show that hinting can indeed result in better execution plans, since this is a well-known and documented fact in real-life scenarios. Instead, we examine whether the evaluation of *Phints* expressions (which we showed to be very expressive in the previous sections) can be done efficiently and at low overheads.

For the first set of experiments, we evaluated the performance of our implementation of the algorithms in Section IV. Each experiment was conducted over a set of *Phints* expressions denoted kP_j^i, where k is the number of simultaneous *Phints* expressions in the experiment, and P_i^j is a *Phints* expression of the form `*[`$\mathtt{T}_1$`, `$\mathtt{T}_2$`, ..., `$\mathtt{T}_j$`]`, where

exactly i out of the j leaf nodes are constrained with specific `forceTable` self-constraints in that order[6]. This family of P_i^j expressions contain almost all kinds of constraints allowed by our language. For instance, an instance of P_4^2 is `*[Lineitem,*,Part,*]`.

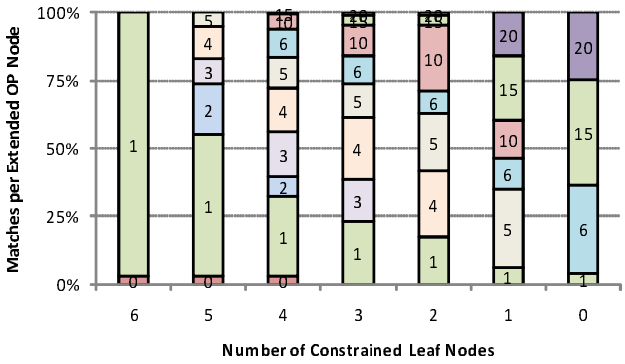
Q	$\emptyset$	$1P_4^0$	$5P_4^0$	$10P_4^0$	$1P_4^2$	$5P_4^2$	$10P_4^2$
1	0.09	0.15	0.15	0.21	0.11	0.30	0.82
2	8.94	8.79	9.00	13.05	10.44	31.16	91.47
3	1.28	1.57	3.14	7.42	2.20	13.43	49.59
4	0.48	0.62	1.53	4.25	0.90	7.21	27.78
5	6.93	7.36	10.03	16.44	9.15	37.57	124.56
6	0.03	0.03	0.03	0.04	0.03	0.06	0.14
7	3.33	3.70	5.83	7.42	4.32	13.67	39.41
8	4.67	6.41	6.91	10.58	7.21	21.83	65.72
9	8.04	8.13	9.30	12.06	13.28	40.74	130.10
10	2.11	3.55	4.16	9.03	3.29	19.88	59.93
11	0.69	0.70	0.75	0.83	0.94	4.46	15.07
12	0.15	0.20	0.37	0.89	0.28	2.70	7.48
13	0.27	0.34	0.77	2.22	0.55	6.14	13.05
14	0.12	0.12	0.16	0.16	0.16	0.78	4.30
15	0.23	0.25	0.53	1.93	0.34	2.27	9.29
16	0.33	0.34	0.39	0.49	0.45	2.69	7.16
17	2.76	2.81	4.34	9.07	4.02	28.00	99.34
18	6.88	8.91	19.32	49.78	14.33	101.80	358.13
19	0.12	0.13	0.20	0.17	0.15	0.72	3.74
20	42.28	43.15	45.60	53.00	57.20	279.51	970.94
21	2.73	2.90	3.97	6.40	3.88	16.95	52.13
22	0.35	0.39	0.63	1.15	0.50	1.79	10.56

Fig. 13. Execution time (in msecs) for evaluating *Phints* expressions.

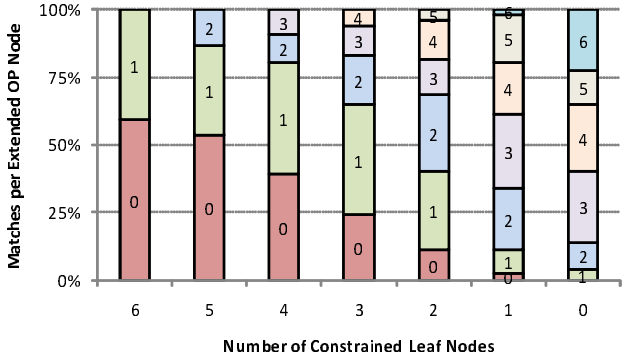
Figure 13 summarizes the results of this experiment when varying the number of expressions from either P_4^0 (i.e., 4 leaf nodes fully specified) and P_4^2 (i.e., 4 leaf nodes for which 2 match any table in the queries). We can see that when there are no constraints (first column in the figure), each evaluation of our algorithm takes below 10 msecs for all but one query. As we increase the number of *Phints* expressions from the P_4^0 family, the evaluation time increases but not very much (evaluation time increases from 1x to 1.7x when going from no constraints to a single *Phints* expression, and from 1.2x to 8.9x when going from one to ten *Phints* expressions). When considering expressions from the P_4^2 family, the evaluation cost increases more steeply than before, because unconstrained nodes can match many nodes in the `PlanSpace`, and all such matches need to be propagated in the tree (see, for instance, the performance of queries 18 and 20, which are associated with the largest `PlanSpace` graphs). We note, however, that even for 10 simultaneous *Phints* expressions that contain all kinds of constraints, the performance of our algorithms was in all cases below one second (and in most of the cases around tens of milliseconds), which makes it a viable solution for interactive specification of constraints and performance tuning.

We now analyze more closely query `TPCH-5`, which is defined over 6 tables and therefore can result in interesting interactions of *Phints* expressions. The `PlanSpace` for this query contains $1,971$ `OP` nodes and 531 `OR` nodes (the time taken to produce `PlanSpace` structures is roughly 1x to 3x that of a plain optimization call; see [11] for additional details). Figure 14(a) shows the distribution of the number of candidate matches that our algorithm found for each extended `OP` node

[6]Note that these *Phints* expressions were chosen to illustrate trends in our matching algorithms, but might not necessarily be "interesting" ones.



(a) Expressions from the P_6^j family.



(b) Expressions from the P_6^j family (with order).

Fig. 14. Number of Matches per extended `OP` node.

as a function of P_i^j. Each bar in the figure corresponds to the evaluation of a *Phints* expression P_6^j, for varying j values from 6 (which corresponds to a fully specified expression) down to 0 (in which no node is constrained). We can see that for P_6^0 the vast majority of extended `OP` nodes contains a single match (the very few nodes with no matches correspond to some non-traditional execution subplans that contain two independent mentions of table `lineitem`). As we move to expressions with more unconstrained nodes, intermediate `OP` nodes contain a larger number of partial matches. As an extreme example, consider P_6^0 = `*(*,*,*,*,*,*)`, and an execution plan `lineitem` $\bowtie$ `orders`. The algorithm would consider that this subplan matches any subset of nodes in the *Phints* expression (e.g., $\{s_0, s_3\}$, $\{s_3, s_4\}$, and $\{s_5, s_1\}$, where s_i denotes the i-th `*` node). Perhaps not surprisingly, the number of matches in this example is consistent with $\binom{6}{j}$.

Figure 14(b) repeats the previous experiment but adding an order constraint to each expression. We can see that the trends remain similar (i.e., there are more matches when the expressions are less constrained), but the number of matches is smaller due to the order constraint, which eliminates any subset of joined tables to be part of a candidate match.

VIII. Related work

All commercial DBMSs support (to varying degrees) the ability to supply hints to the query optimizer, and thus allow DBAs to influence the choice of the best execution plan.

We next briefly review some of the features present in these systems, and refer the reader to the respective white papers or user manuals (i.e., [7], [13], [14]) for more details. As mentioned in the introduction, existing query hints in current DBMSs cover a wide variety of alternatives that influence not only the choice of plans but also certain key aspects of query execution, such as limiting the degree of parallelism for the query, or setting the appropriate isolation level during query execution. While such hints are very important, in this paper we addressed only the class of hints that can be used to influence the optimizer to pick a "different" plan.

Hinting mechanisms in most systems typically include support for specifying an access path for a table (i.e. force the use of an index for a particular table) and also the ability force the join order of the plan based on the order of the tables listed in the SQL string. Microsoft SQL Server [13] has the ability to force the use of a particular join algorithm throughout a query block (letting the optimizer pick the right join order) or fix individual join algorithms in a query block (however simultaneously fixing a specific join order). Microsoft SQL Server also supports forcing a full plan by using the `USE PLAN` xml interface [2]. This is particularly useful to revert back to a previously used plan (to avoid a performance regression), but requires that the full plan being specified in the hint.

Oracle hints [7] can specify the first join to be used in the execution plan using `/* LEADING(T1, T2) */`. It can also specify the join algorithm to be used for a particular pair of tables (for example, `USE_NL(Table1, Table2)`). While Oracle hints have the ability to scope a hint to a particular query block in a multi-block SQL query, they do not support any finer scopes over an arbitrary set of tables.

DB2's optimization guidelines [15] provide mappings between `SQL` statements and optimization hints through an XML representation (note that the XML representation is used to encode the mapping, but not to constrain the `PlanSpace` itself). As with Oracle hints, some DB2 hints can be additionally applied at a predicate level, but in general this is not as powerful or unified as our *Phints* language (e.g., see Section III-A).

We note that there is not a unified abstraction of hints even within any single system. The existing set of hints supported by each product is largely a toolkit of techniques that are considered important and that have been incorporated in their optimizers. We therefore believe ours is the first work to take a more principled approach towards plan hints by working on an abstraction on the search space of the optimizer.

In Section III we outlined the language *Phints*, which enables the specification of constraints on top of tree expressions. While "tree pattern matching" is conceptually similar to evaluating XML queries by specifying patterns that have some specified tree structured relationships [6], there are significant differences, both in the semantics of matching and in the objective function, as discussed in detail in Section III.

Reference [11] introduces *Configuration-Parametric Query Optimization*, which is a light-weight mechanism to re-optimize queries for different physical designs at very low overhead. By issuing a single optimization call per query, [11] is able to generate a compact representation of the `PlanSpace` that can then produce very efficiently execution plans for the input query under arbitrary configurations. We use the ideas of [11] to implement the interface described in Section V that generates the `PlanSpace` for a given input query.

IX. Conclusions

Optimizer hinting mechanisms have served as a valuable tool for experienced DBAs to influence the optimizer's choice of execution plan. Unfortunately, existing mechanisms available in commercial DBMS are an ad-hoc collection of hinting options. In this paper, we have proposed *Phints*, which provides a clean framework to express and capture an important class of optimizer hints. Additionally, our framework offers DBAs the ability to express a much richer class of constraints (at a finer granularity of subexpressions) that go beyond what traditional hinting mechanisms offer. In the future, we plan to investigate similar approaches to an even wider class of query hints beyond those that are targeted at the optimizer.

References

[1] S. Christodoulakis, "Implications of certain assumptions in database performance evauation," *ACM TODS*, vol. 9, no. 2, 1984.

[2] B. A. Patel, "Forcing query plans," 2005. [Online]. Available: `http://www.microsoft.com/technet/prodtechnol/-sql/2005/frcqupln.mspx`

[3] G. Graefe, "The Cascades framework for query optimization," *Data Engineering Bulletin*, vol. 18, no. 3, 1995.

[4] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1979.

[5] R. Bellman, *Dynamic Programming*. Princeton University Press, 2003.

[6] M. F. Fernandez, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh, "XQuery 1.0 and XPath 2.0 Data Model (XDM)," 2007. [Online]. Available: `http://www.w3.org/TR/2007/REC-xpath-datamodel-20070123`

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

[8] T. Neumann and G. Moerkotte, "An efficient framework for order optimization," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2004.

[9] S. Borzsonyi, D. Kossmann, and K. Stocker, "The skyline operator," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2001.

[10] K. Billings, "A TPC-D Model for Database Query Optimization in Cascades. Ms. thesis, Portland State University," 1996.

[11] N. Bruno and R. Nehme, "Configuration-parametric query optimization for physical design tuning," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2008.

[12] TPC Benchmark H, "Decision support." [Online]. Available: `http://www.tpc.org`

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

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

[15] "Optimizer profiles and guidelines overview," 2008. [Online]. Available: `http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp?topic=/com.ibm.db2.udb.admin.doc/doc/c0024522.htm`