Plan Stitch: Harnessing the Best of Many Plans

Bailu Ding Sudipto Das Wentao Wu Surajit Chaudhuri Vivek Narasayya
Microsoft Research
Redmond, WA 98052, USA
{badin, sudiptod, wentwu, surajitc, viveknar}@microsoft.com

ABSTRACT

Query performance regression due to the query optimizer selecting a bad query execution plan is a major pain point in production workloads. Commercial DBMSs today can automatically detect and correct such query plan regressions by storing previously-executed plans and reverting to a previous plan which is still valid and has the least execution cost. Such reversion-based plan correction has relatively low risk of plan regression since the decision is based on observed execution costs. However, this approach ignores potentially valuable information of efficient *subplans* collected from *other* previously-executed plans. In this paper, we propose a novel technique, Plan Stitch, that automatically and opportunistically combines efficient subplans of previously-executed plans into a valid new plan, which can be cheaper than any individual previously-executed plan. We implement Plan Stitch on top of Microsoft SQL Server. Our experiments on TPC-DS benchmark and three real-world customer workloads show that plans obtained via Plan Stitch can reduce execution cost significantly, with a reduction of up to two orders of magnitude in execution cost when compared to reverting to the cheapest previously-executed plan.

PVLDB Reference Format:
Bailu Ding, Sudipto Das, Wentao Wu, Surajit Chaudhuri, Vivek Narasayya. Plan Stitch: Harnessing the Best of Many Plans. *PVLDB*, 11(10): 1123-1136, 2018.
DOI: https://doi.org/10.14778/3231751.3231761

1. INTRODUCTION

A query's plan can change for various reasons, such as when indexes and statistics are created and dropped, when a stored procedure gets recompiled with a parameter binding different from last time, or when a query is manually tuned with hints.

As an example, in Azure SQL Database [9], customers can configure their databases to use Automated Indexing [8]. Automated Indexing continuously monitors and analyzes query workloads, and periodically recommends index configurations. As indexes are incrementally implemented, different plans get executed. We observe that tens of thousands of indexes are created and dropped over a week. This results in hundreds of thousands of instances where the optimizer chooses multiple query plans for the same query, especially for stored procedures and query templates.

Proceedings of the VLDB Endowment, Vol. 11, No. 10

DOI: https://doi.org/10.14778/3231751.3231761

The query optimizer often makes good use of the newly-created indexes and statistics, and the resulting new query plan improves in execution cost.[1] At times, however, the latest query plan chosen by the optimizer has significantly higher execution cost compared to previously-executed plans, i.e., a ***plan regresses*** [5, 6, 31].

The problem of plan regression is painful for production applications as it is hard to debug. It becomes even more challenging when plans change regularly on a large scale, such as in a cloud database service with automatic index tuning. Given such a large-scale production environment, detecting and correcting plan regressions in an *automated* manner becomes crucial. Such automated correction techniques must have low ***risk*** of making execution costs even worse due to the correction.

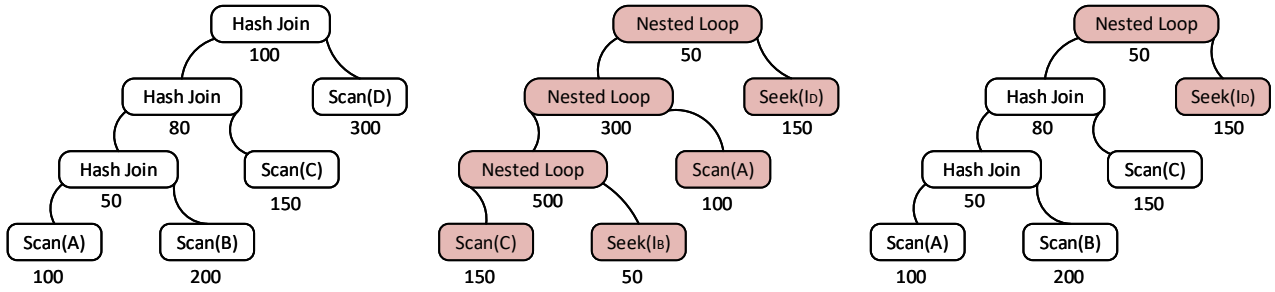
Automatic plan correction. Commercial DBMSs have recognized the importance of this problem, and indeed, the automatic plan correction (***APC***) feature in Azure SQL DB (and Microsoft SQL Server) automatically detects and corrects plan regressions [5].

When a plan regresses due to plan changes, previously-executed plans of the same query are often still ***valid***, i.e., the plans are still executable in current configuration.[2] In such cases, reverting to a cheaper previously-executed plan will resolve the regression.

APC continuously monitors aggregated execution statistics of plans. Once a plan completes execution and has statisticall-significant worse execution cost compared to earlier plans of the same query observed in recent history, the server automatically forces the optimizer to execute the query with the cheaper older plan which is still valid. When database sizes and data distributions have not changed significantly from the time when previous plans in recent history were executed, reverting to the cheaper previously-executed plan corrects the regression. Similar features are available in other commercial DMBS products as well [6].

Opportunity. This reversion-based plan correction (RBPC) is attractive in a production setting due to its low risk, since the decision is based on *observed execution cost*. However, RBPC also restricts itself to choose one overall cheapest plan from the set of previously-executed plans. With operator-level execution cost statistics, we can further identify efficient *subplans* from other plan in the set, even if the overall plan is not the cheapest. Ignoring these subplans can leave significant opportunities of further plan improvement with low risk.

[1] In this paper, we use execution cost (either CPU time and/or logical reads) as a metric for measuring the goodness of a query plan.
[2] A plan can become invalid and not executable in a configuration for various reasons, such as when the indexes used by the plan are not present in current database.



(a) Plan p_1 with no indexes

(b) Plan p_2 with I_B and I_D

(c) Plan consists of the best of p_1 and p_2 with I_B and I_D.

Figure 1: Combining subplans to get the best of multiple plans. Each operator is annotated with its execution CPU cost in seconds. p_1 is under the configuration of no indexes. p_2 and the combined plan are under the configuration of index I_B and I_D.

Example. Consider a query that joins 4 relations A, B, C, and D, and consider a specific instance where a new query plan is chosen due to new secondary indexes created on B and D. In the absence of the indexes, the optimizer chooses Hash Joins (HJs) with Table Scans on all the relations and a join order of A, B, C, and D (see p_1 in Figure 1a). In the presence of the indexes, the optimizer decides to join C and B first with Nested Loop Join (NLJ), accessing B using Index Seek. It then joins A with another NLJ and finally joins D with NLJ again using an Index Seek on D (see p_2 in Figure 1b). Based on optimizer's cost estimates, the new plan p_2 is cheaper than the previous plan p_1. However, due to cost misestimates on the joins of (C, B, A), p_2 ends up with higher execution cost.

Reverting back to p_1 corrects the increase in execution cost due to p_2. However, such an approach misses out an even better plan that combines the subplans from p_1 and p_2 (see Figure 1c). Combining the joins of (A, B, C) from p_1 and the root NLJ and Index Seek on I_D from p_2 results in a plan better than both p_1 and p_2. While the combined plan has not been executed in its entirety before, it consists of subplans with observed execution cost from previously-executed plans. Thus, executing the combined plan has a comparable low risk as reverting back to p_1.

Harnessing the best of multiple plans. We propose **Plan Stitch**, a *fully-automated* solution that opportunistically constructs a query plan from previously-executed plans of the same query. The plan constructed can be different from and cheaper than all the previously-executed plans. It is also the cheapest plan among all the plans that only consist of subplans from previously-executed plans.

Challenges. Harnessing efficient subplans from multiple plans has two challenges. First, discovering efficient subplans from multiple complicated query plans which consist of hundreds of operators requires analyzing the semantics of hundreds of subplans. Second, even after such efficient subplans are identified, combining the access paths, physical operators, and join orders from these subplans into a single valid plan is also challenging.

Plan Stitch addresses both challenges by formulating the problem as a plan search problem similar to traditional query optimization, but in a constrained search space. Every physical operator in this constrained search space must appear in a previously-executed plan of the same query with exactly the same logical expression, and it must be valid in current configuration. Plan Stitch discovers efficient subplans by comparing alternative subplans of the same logical expression in the constrained search space. It then combines these efficient subplans into the cheapest plan in execution cost with a fast, quadratic time algorithm based on dynamic programming. Finally, Plan Stitch influences the optimizer's plan choice using the plan forcing feature supported by commercial databases [6,46]. This forcing step also validates the correctness of the constructed plan.

Example. Referring back to the example in Figure 1, Plan Stitch will constrain the plan search space to only subplans executed in p_1 and p_2, such as joining A, B, C using either (A, B, C) with HJs or (C, B, A) with NLJs, but not using Merge Join or any other join order. Based on operator-level execution cost of p_1 and p_2, Plan Stitch combines the root NLJ and Index Seek on the newly created index I_D from p_2, as well as the subplan of joining (A, B, C) from p_1. It ends up with a new plan (Figure 1c) with the lowest overall execution cost possible from this constrained search space, which is cheaper than both p_1 and p_2.

Properties. Plan Stitch has the same desirable properties as RBPC: it is automatic, low-overhead, and low-risk. Similar to RBPC, Plan Stitch directly relies on observed execution cost and thus is *low in risk*, which is crucial for automatic plan correction at a large scale such as in an automatically indexed cloud-scale database service like Azure SQL DB. In contrast, most prior work on leveraging feedback from execution [1, 11, 14, 16, 43] applies feedback to general query optimization and considers subplans that have never been executed before, introducing increased risk of large estimation errors for the selected plan's execution cost. Compared with RBPC, Plan Stitch can still leverage subplans even where the previously-executed plans are invalid, e.g., due to index changes.

Note that Plan Stitch can be applied even when there is no regression. Whenever a query accumulates execution statistics for multiple plans, Plan Stitch can search for opportunities to improve plan quality. Given Plan Stitch's low overhead, it is potentially worth exploring such opportunities routinely.

Our discussion so far has focused on executions of specific query instances. For parameterized queries (query templates) or stored procedures, RBPC reverts to a previously-executed plan with the cheapest execution cost averaged over all the instances of the plan. Plan Stitch follows the same paradigm and constructs the cheapest stitched plan in average execution cost. Averaging execution cost across instances optimizes for the cheapest expected execution cost. With appropriate criterion for choosing subplans, Plan Stitch can adapt to alternative objectives, such as low variance of execution cost [10]. If multiple plans can be stored for the same query, we can also stitch the plan for a specific parametrization. With parametric query optimization techniques [19, 21, 22, 27, 28], Plan Stitch can adjust the weight of previously-executed plan instances for a specific query instance based on their similarity in the selectivity space when constructing the stitched plan. Such extensions are beyond the scope of this paper.

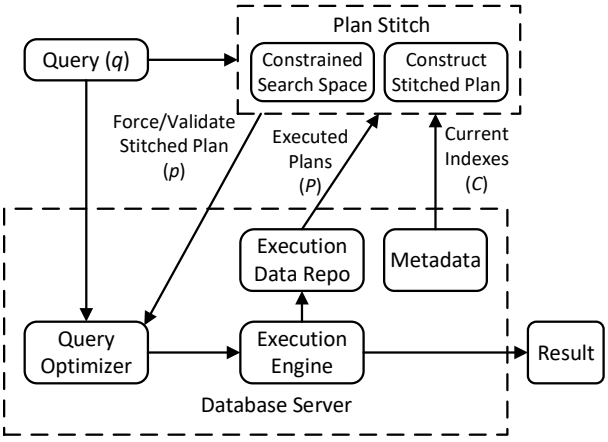


Figure 2: The architecture of the database engine with feedback collection and Plan Stitch.

Result highlights. We implement Plan Stitch in C# as a component external to Microsoft SQL Server (Section 4). We rely on existing functionality in the engine for storing a history of executed plans [37], obtaining operator-level execution cost [41, 42], and influencing the query optimizer's plan choice [36, 46]. Using industry-standard benchmark TPC-DS and three real-world customer workloads, we demonstrate that Plan Stitch outperforms a reversion-based correction technique (or RBPC) both in terms of coverage (i.e., number of queries and plans improved) as well as the reduction in execution cost. By stitching together subplans from previously-executed plans, Plan Stitch constructs new plans that are at least 10% cheaper in execution cost for up to 83% of plans across our workloads when compared to the cheapest plan found by RBPC. There are several instances where there is up to two orders of magnitude reduction in execution costs. Across all the workloads, Plan Stitch improves the quality of up to $20\times$ more plans than RBPC. In addition, constructing and validating each stitched plan is often cheaper than the cost to optimize the corresponding query by the optimizer.

Contributions. Following are the key contributions:

- We propose Plan Stitch, a novel, fully-automated, low-overhead technique that uses operator-level execution cost of previously-executed plans to opportunistically construct new plans that are cheaper in execution cost than the cheapest previous plan that is still valid, with low risk of plan regression.
- We describe an efficient, dynamic programming-based approach to construct a plan with the cheapest execution cost. We combine the observed execution cost of subplans from previously-executed plans to estimate the execution cost of stitched plans with high accuracy (Section 3).
- We implement Plan Stitch as a component layered on top of Microsoft SQL Server (Section 4). Using industry-standard TPC-DS benchmark and three real-world customer workloads, our comprehensive evaluation demonstrates that with very low overhead (Section 5), Plan Stitch results in plans which are significantly cheaper than plans found by the optimizer or RBPC.

2. OVERVIEW

Problem statement. Given a query instance q, the set of indexes $\{I_k\}$ available in the current database configuration C, and a set P of distinct query plans $\{p_i\}$ of q which have per-operator execution costs recorded from past executions, *Plan Stitch* constructs a plan p for query q that has cheapest execution cost with the constraint that each operator in p can be found in some plan $p_i \in P$.

We refer to the operation of constructing p using operators from different plans in p_i as **stitching**, and the resulting plan p as the **stitched plan**. As noted earlier in Section 1, we use execution cost measures proportional to the logical amount of work done by the query, such as CPU time consumed by the query or number of logical reads performed by the query.

Architectural overview. Figure 2 shows the logical architecture of how Plan Stitch integrates with a DBMS, how it obtains its inputs ($\langle q, P, C\rangle$), and how it influences the optimizer's plan choice with its output (p). Logically, Plan Stitch executes outside of the critical path of the query optimization and execution, either as an external client component or a background thread in the DBMS. The optimizer selects a plan for a given query and the current configuration (obtained from database metadata), which is then executed, and its execution statistics are recorded in a repository. The execution statistics includes the plan structure and its operator-level execution cost [2, 37]. Over time, the repository passively collects a history of plans, including different plans executed for the same query.

Whenever multiple plans for the query q have executed and its execution statistics are available in the repository, we can trigger Plan Stitch to search for alternative stitched plans which could reduce the execution cost compared to the plan currently chosen by the optimizer. Once triggered, Plan Stitch obtains its input plans P from the execution data repository and reads the current index configuration C from the database metadata. The final stitched plan p generated by Plan Stitch is then passed to the optimizer so that it can be used for future executions of query q.

Plan Stitch influences the optimizer to use p as q's execution plan using a widely-supported plan forcing API which allows specifying a plan hint for a query [6, 36, 46]. The plan structure fully specifies the logical tree structure and the physical operators. The query optimizer takes the hint as a constraint to prune its search space during its plan search, thus generating a query plan which conforms to the hint and is guaranteed to be a valid plan for the query.

Note that Plan Stitch's ability to find a cheaper plan depends on the presence of cheaper subplans in P which are still valid in the current configuration C. Hence, Plan Stitch is opportunistic and cannot guarantee an improved plan. Plan Stitch outputs a stitched plan only if it is estimated to be cheaper in execution cost compared to the plan returned by the optimizer.

The repository needs to age out execution feedback to ensure the accuracy of Plan Stitch's cost estimate when data changes. This is a common challenge for all feedback-based approaches. Since a query's execution cost depends on the amount of data and its distribution, it is difficult to theoretically bound the degree of inaccuracy of the execution feedback when the data changes. As a result, different heuristics have been proposed, such as to retire the execution data after a time window, invalidate the execution data when detecting a large enough fraction of the data has changed (similar to detecting out-of-date statistics and histograms [7,30]), or weigh the execution feedback based on its freshness. Such heuristics could also be applied to the execution data used in Plan Stitch.

3. PLAN STITCH

Plan Stitch has two major components: (a) using the set of previously-executed distinct[3] plans P of the same query to identify and encode a constrained search space; and (b) using per-operator

[3]We aggregate the execution cost of multiple executions of the same plan to reduce the runtime variance of execution data as well as the overhead of Plan Stitch.

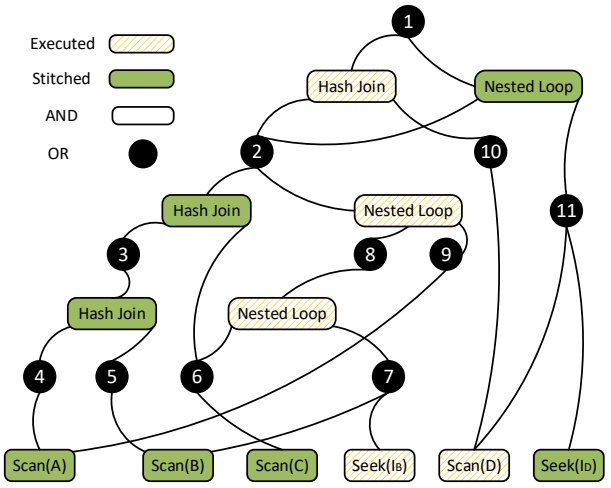


Figure 3: The AND-OR graph constructed from p_1 and p_2. It encapsulates a set of valid plans for this query that consist of only physical operators executed in p_1 or p_2, including p_1 and p_2. The final stitched plan (colored in green) differs from and is cheaper than both p_1 and p_2.

execution cost from plan p_i to construct the stitched plan with minimum total execution cost in the constrained search space. In this section, we explain these components, how Plan Stitch estimates the execution cost of the stitched plan, and the stitch algorithm.

3.1 Constrained Search Space

Plan Stitch starts by constraining its plan search space to operators that have appeared in some p_i. The major challenges in generating this constrained search space are: (a) identifying equivalent subplans from different plans p_i; and (b) compactly encoding these equivalent subplans in a structure to allow efficient search.

Identifying equivalent subplans. Every node in a query plan (along with the subplan rooted at the node) represents a logical expression with the required physical properties (such as *interesting orders* [40]), e.g., $A \bowtie B \bowtie C$ without a sort order. Two subplans are *equivalent* if they have the same logical expression and the required physical properties. A group of equivalent subplans is referred to as an *equivalent subplan group*.

To find equivalent subplans across plans in p_i, we need to decide the equivalence of the logical expressions for these subplans, which is known to be undecidable [29]. Previous work has proposed tests and greedy algorithms to match equivalent logical expressions [23,39] to enable the query optimizer to match views and detect duplicate expressions. Conceptually, such techniques can be used to identify equivalent expressions. In our implementation, we use similar heuristics that provide a reasonable balance between ease of implementation, overhead, and accuracy of matches, and use the optimizer to ensure the correctness of a stitched plan as a side-effect of plan forcing. We will discuss our implementation choices in detail in Section 4.

Encoding the constrained search space. We represent the constrained search space of allowed alternative plans using an AND-OR graph [24]. The graph consists of AND and OR nodes where each node represents whether the respective subplans should be used simultaneously (AND) or are mutually exclusive (OR). Each AND node corresponds to a physical operator in a plan, e.g., Hash Join. Every OR node represents a logical expression with the required physical properties. The children of an AND node are OR nodes, representing logical expressions and required physical properties of the AND node's child subplans. The children of an OR node are AND nodes, representing the root physical operators of alternative subplans of the OR node.

To construct an AND-OR graph, for every subplan rooted at a physical operator in p_i, we find all the equivalent subplans from $p_j \in P$. With these equivalent subplans, we create an OR node representing the logical expression and the required physical properties for an equivalent subplan group. The root physical operator of each subplan in the group corresponds to a child AND node of the OR node.

Example. Consider the example query joining relations A, B, C, and D discussed in Section 1. Figure 3 shows the AND-OR graph constructed from the two alternative query plans under the current configuration of indexes on B and D (OR nodes are numbered black circles; AND nodes are rounded rectangles. See Figures 1a and 1b for the original plans). Every physical operator in the AND-OR graph, represented by an AND node, has been executed with exactly the same logical expression from either p_1 or p_2. For example, the root OR node 1 of the graph has two alternatives, the root HJ from p_1 and the root NLJ from p_2. Similarly, the left subplans of the two root physical operators have the same logical expression, i.e., joining A, B, and C, and they share the same OR node 2. This OR node has two alternatives: joining (A, B, C) with HJs from p_1 and joining C, B, A with NLJs from p_2.

Note that not all the leaf operators from previously-executed plans will appear in the corresponding AND-OR graph. If a leaf operator uses an access path that is not available in current configuration, it is not valid and does not appear in the graph. For instance, if we drop the index I_D from the configuration in our example, the Index Seek I_D from p_2 will not be valid and thus the corresponding AND node will be removed from the AND-OR graph in Figure 3.

3.2 Constructing the Stitched Plan

There are two important properties of the AND-OR graph to enable an efficient search. First, the graph is acyclic which enables Plan Stitch to construct the cheapest plan recursively from the bottom up. Since each plan is tree-structured, the physical operators and the subplans rooted at these operators are partially ordered within a plan.[4] Consequently, the operators are partially ordered across plans; otherwise, there will exist two equivalent operators in the same plan, where one operator is the ancestor of the other, which is not possible. Since all the operators are partially ordered across plans, the AND-OR graph is acyclic.

Second, there is at least one OR node: the root OR node shared by all the plans of the query. Because all the plans execute the same query, the plans themselves are logically equivalent and their root physical operators share the same root OR node. This implies that the AND-OR graph embeds all the previously-executed plans that are still valid from the same OR root. Thus, the cheapest plan constructed from this constrained search space costs no higher than the cheapest plan among all the valid previously-executed plans.

Stitching plans. Intuitively, we construct the cheapest plan from leaf AND nodes to the root OR node by stitching the cheapest subplan for each AND node and each OR node in the AND-OR graph using dynamic programming. At the leaf-level, each node is an AND node and the corresponding operator becomes the cheapest subplan of the AND node by itself. The cheapest stitched subplan of an OR node is the cheapest one from the cheapest stitched subplans of all its alternative AND nodes. To get the cheapest stitched plan rooted at

[4]Here we assume that every physical operator in a plan either changes the logical expression or the physical properties, i.e., sort order, of the subplan rooted at the operator.

an AND node, we take the cheapest stitched subplans of its children OR nodes and stitch them as child subplans to the corresponding root physical operator of the AND node. Finally, the cheapest plan of the query is the cheapest stitched subplan at the root OR node.

Costing stitched plans. To stitch the cheapest subplan, Plan Stitch compares the execution cost of alternative stitched subplans. Since a stitched subplan may not have been executed in its entirety from one previously-executed plan, Plan Stitch combines the observed execution cost of the operators in the stitched subplan. When stitching the cheapest subplan of a child OR node to a parent AND node, Plan Stitch estimates the execution cost of the resulting stitched subplan by combining the cost of child subplans with that of the cost of the parent, similar to how the query optimizer combines its cost estimates and propagates them through the plan:

$$stitchedSubUnitCost(opCost, execCount, \{(childSubUnitCost, childExecCount)\})$$

stitchedSubUnitCost estimates the execution cost of executing a stitched subplan rooted at a physical operator *op* once. It takes four inputs: the *observed* execution cost of *op*, how many times *op* is executed in its *original* plan, the *estimated* execution cost of executing each *stitched* child subplan once, and how many times each of its *original* child subplans executes. Note that most subplans are executed only once in a plan, except if the subplan is a descendant of the inner side of some Nested Loop Join or Apply [4] operators.

By constraining the search to operators that have been executed, Plan Stitch can combine and estimate the execution cost of the stitched plan with high accuracy, which is verified in our experiments and thus confirms its low risk. Note that this applies to all logical cost measures, such as CPU time and logical reads.

Assumptions in costing. To simplify combining execution costs for the stitched plan, we make a few assumptions and approximations. First, we assume that the execution cost of an operator is transferrable from one plan to another, provided its input and output are unchanged and it is performing the same logical operation. This follows from the observation that the operator implementations are deterministic. Second, for subplans with multiple executions, we divide the execution costs evenly among executions, ignoring start-up overhead for the first execution. While this approximation can introduce errors in cost estimation, as we will see in the experimental evaluation (Section 5.3), in practice across a variety of workloads, we did not observe this factor introducing noticeable errors in our costing.

Example. Consider the example in Figure 3. We start from the leaf nodes, where OR node 7 and 11 have alternative choices. As the cheapest subplan of the group, OR node 7 chooses $Seek(I_B)$ and OR node 11 chooses $Seek(I_D)$. Similarly, at children AND nodes of OR node 2, the cheapest stitched subplan rooted at the HJ is cheaper than its equivalent stitched subplan rooted from the NLJ, and thus the former becomes the cheapest stitched subplan of OR node 2. Continue to stitch the cheapest subplans for AND nodes and OR nodes upwards, eventually, at the root OR node 1, it compares the cheapest stitched subplan from the top HJ and the top NLJ and chooses the latter. Figure 3 shows the resulting cheapest stitched plan colored in green.

3.3 Stitch Algorithm

Algorithm 1 outlines the dynamic programming algorithm that constructs the cheapest stitched plan from bottom up, similar to the System R optimizer [40].

Let G be an AND-OR graph constructed from previously-executed plans with all invalid operators removed. For each AND node and, we maintain two states: the cheapest stitched subplan $bestSubplan(and)$ with the corresponding root operator op and the estimated execution cost $bestSubUnitCost(and)$ of executing $bestSubplan(and)$ once. For each OR node or and its corresponding equivalent subplan group $g(or)$, we maintain the cheapest stitched subplan in the group as $bestSubInGp(g(or))$, which is the cheapest stitched subplan among all the $bestSubplan(and)$, where and is a child AND node of or.

Input: A set of query plans P, a set of indexes I present in current configuration, and the AND-OR graph G constructed from P
Output: The cheapest stitched query plan

```
// Order the equivalent subplan groups from bottom to top
1  G' ← GetOrderedSubplanGroups(G)
   // Construct the cheapest stitched plan
2  for g(or) in G' do
3      bestSubInGp(g(or)) ← NULL
4      bestCost ← ∞
5      for and in or's child AND nodes do
6          if and is leaf operator then
7              bestSubUnitCost(and) ← opUnitCost(op)
8              bestSubplan(and) ← A single operator op
9          else
10             bestSubplan(and) ← op
11             for or_k in and's child OR nodes do
12                 bestSubplan(and) ← Stitch
                     bestSubInGp(g(or_k)) to op
13                 and_k ← subplan root of
                     bestSubInGp(g(or_k))
14             end
15             bestSubUnitCost(and) ←
                 stitchedSubUnitCost(opCost(op),
                 execCount(op), {bestSubUnitCost(and_k),
                 execCount(op, or_k)})
16         if bestSubInGp(g(or)) = NULL or
             bestCost > bestSubUnitCost(and) then
17             bestSubInGp(g(or)) ← bestSubplan(and)
18             bestCost ← bestSubUnitCost(and)
19     end
20 end
21 return bestSubInGp(g_root)
```

Algorithm 1: Construct the cheapest stitched query plan using dynamic programming

We stitch the subplans from bottom up (line 1 in Algorithm 1). If op is a leaf operator (line 6-8 in Algorithm 1), the $bestSubplan(op)$ is op itself and $bestSubUnitCost(op)$ is the cost of executing op once.

If op is an internal operator (line 10-15 in Algorithm 1), for the equivalent subplan group of every OR child node $g(or_k)$, we stitch $bestSubInGp(g(or_k))$ to op. We compute $bestSubUnitCost$ with the costing API as $stitchedSubUnitCost(opCost(op), execCount(op), \{bestSubUnitCost(and_k), execCount(op, or_k)\})$, where $execCount(op, or_k)$ is how many times the child subplan rooted at or_k is executed when the subplan rooted at op is executed once.

After constructing the cheapest stitched subplan rooted at op, we update $bestSubInGp(g(or))$ accordingly (line 16-19 in Algorithm 1). Finally, we return the cheapest stitched subplan of the root group (i.e., $bestSubInGp(g_{root})$) as the cheapest stitched plan from this AND-OR graph G (line 21 in Algorithm 1).

Time complexity. Algorithm 1 is quadratic in the number of plans and the number of operators in the plan, assuming the time to decide the equivalence of two subplans is a constant using our heuristics. Let N be the number of plans and M be the maximal number of operators in a plan among all the plans in P, we show that:

The worst-case running time for the plan stitch algorithm is $\Theta((NM)^2)$.

Because the total number of operators in all the plans is at most NM, the size of each equivalent subplan group is at most NM. So updating the cheapest stitched plan for each group takes $O(NM)$. Since there is at most NM equivalent subplan groups, computing the cheapest stitch subplans for all groups takes $O((NM)^2)$. Now we show a case where the upper bound is tight.

Let there be N query plans. Each plan is a perfect binary tree[5] with $M = 2^k - 1$ operators. Let all the interior nodes be join operators and each leaf node be an index access operator on the same table. Since there are $M = 2^k - 1$ operators in a plan, 2^{k-1} operators are leaf operators. Let the leaf operators in all the plans be $op_1, op_2, \cdots, op_{2^{k-1}N}$. Let each op_i access the same table with an index of key columns $a_1, a_2, \cdots, a_i$ and sort order $a_1, a_2, \cdots, a_i$. Thus, the subplan of leaf operator op_i has at least $(2^{k-1}N) - i$ equivalent subplans, i.e., $op_{i+1}, op_{i+2}, \cdots, op_{2(k-1)N}$. So computing the best stitched subplans in all the groups takes at least $\sum_{i=1}^{2^{k-1}N} 2^{k-1}N - i + 1 = 2^{k-2}N(2^{k-1}N + 1) = \frac{(M+1)N((M+1)N+2)}{8}$. Therefore, the time complexity $O((NM)^2)$ is tight and the worst-case running time is $\Theta((NM)^2)$.

In TPC-DS benchmark and real-world customer workloads used in our experiments, the number of operators in a query plan is no more than a few hundred, and we expect to stitch a handful of previously-executed plans for each query. In addition, the equivalent subplan matches are usually sparse, and the algorithm will rarely reach its worse-case running time. That is, in practice, the overhead of Algorithm 1 is much less than optimizing the corresponding query given plans with a few hundred of operators and a handful of plans to stitch.

4. IMPLEMENTATION

We implement Plan Stitch in C# as a component on top of Microsoft SQL Server without requiring any changes to the engine. SQL Server stores previously-executed plans and their execution statistics, including plan structures, properties of operators (e.g., sort columns), and operator-level execution costs [37, 41]. Plan Stitch passively monitors query executions and triggers the search for alternative plans when execution statistics for multiple plans for the same query are available. Our implementation follows the logical architecture in Figure 2, where we force the stitched plan using the plan hinting API supported in SQL Server [36, 46].

As described in Section 3, conceptually, we can implement Plan Stitch inside SQL Server. When stitching plans inside the engine, we can leverage view matching and transformation rules to match subplans [23, 39], constrain the search space, and force the plan.

Implementing Plan Stitch inside the engine, however, incurs significant engineering overhead. Because SQL Server query optimizer prunes search space aggressively, it explores different search space for the same query under different configurations. Persisting and aligning the in-memory data structures of plans across query optimization boundary to construct the `AND-OR` graph become an engineering challenge. Moreover, integrating the execution cost into the optimizer requires a redesign of its costing framework.

As a proof of concept, we choose to implement Plan Stitch external to SQL Server, using heuristics for subplan match and the existing mechanism to force and validate the stitched plan.

Matching equivalent subplans. We use the following heuristics to determine matches between subplans: (a) rule out subplans which can never be equivalent (e.g., different joined tables, or not matching interesting orders), (b) consider candidate matches where the necessary conditions are met (e.g., the joined tables, sort order of output columns, etc.), (c) match expressions computed in the query wherever possible by comparing the expression trees. For sort orders, as long as the prefix of the sort order of a subplan satisfies the required sort order of the `OR` node, the subplan matches the sort order of the corresponding `OR` node. In addition, when the optimizer generates single-threaded (i.e., serial) and multi-threaded (i.e., parallel) plans for the same query, we also need to consider the serial or parallel mode as required physical properties, since equivalent serial and parallel nodes are not interchangeable across plans.

Forcing the stitched plan. When a plan is forced for a query, especially when the plan is externally provided, the optimizer must ensure that the plan is correct, i.e., it returns the result required by the query and the plan is executable under current database configuration. The optimizer performs this validation by constraining its search space to only explore plans satisfying the specified plan structure. Similar to Plan Stitch, such a constraint enforcement requires the optimizer to find equivalence of expressions in the specified plan and expressions that appear in the query. Since determination of such equivalence is undecidable [29], the optimizer also relies on heuristics to perform this match. SQL Server's query optimizer uses a large collection of transformation rules which are used to rewrite an expression into equivalent logical expressions, which the optimizer tries to match. Such an approach ensures that there are no false positives in matches, though it is possible to have false negatives. Therefore, if a specified plan is successfully forced, the plan is guaranteed to be correct. However, the optimizer might fail to force some valid plans due to false negatives in matches. We observed multiple cases where either the original plan itself or the cheapest previously-executed plan that is still valid in current configuration cannot be forced.

Handling errors in validation. The stitched plan generated by Plan Stitch can fail the forcing and validation step either due to shortcomings of the equivalence matching in Plan Stitch or the optimizer. Plan Stitch handles such failed validations with ***two-stage stitch***. When a stitched plan is invalid and the corresponding plan forcing fails, Plan Stitch makes a second attempt with *sparse stitch*. A sparse stitch performs a sparser match by removing equivalent subplan groups rooted at operators such as Bitmap[6], and Compute Scalar, which introduces expressions to make matching harder. Intuitively, by reducing the number of equivalent subplan groups, we decrease the chance of making mistakes in constructing the constrained search space and hopefully increase the chance of producing a valid stitched plan.

It is possible to iteratively eliminate subplan groups to make the stitch sparser until we find a valid plan which can be forced. However, in practice, we found the incremental benefits of sparser stitches in improving the probability of a successful validation decreases, and two stages provide a good trade-off between the overhead and success rate. For the several workloads used in our experiments, 75%-100% of the initial stitched plans are successfully forced, and the two-stage stitch increases the rate by up to 9%. If

[5]In a perfect binary tree, all interior nodes have two children and all leaves have the same depth.

[6]Bitmap represents a filtering bitmap created by a HashMap operator from its outer relation and is pushed down to its inner relation.

Table 1: Statistics of TPC-DS benchmark and real-world customer workloads. Included queries are subsets of queries with at least two distinct plans. Average indexes is the number of indexes referenced by the plans for the query averaged over the entire workload.

Workload & Statistics	TPC-DS	Cust1	Cust2	Cust3
DB size (GB)	87.7	44.6	493	93
Tables	24	23	349	22
Queries	99	25	38	26
Included Queries	76	24	27	11
Avg. Joins	8	8.2	18.7	7.2
Avg. Plans	8.4	7.9	8.4	6.6
Avg. Indexes	4.5	3.9	6.2	3.1

the sparse stitch still results in an invalid stitched plan, Plan Stitch uses the cheapest previously-executed plan which is valid.

Note that a sparse stitch does not necessarily sacrifice the quality of the stitched plan. Even if the sparse stitch does not identify a cheaper alternative subplan, it can still benefit from that subplan by stitching a subplan that contains this cheaper subplan.

5. EXPERIMENT

In this section, we evaluate Plan Stitch compared to reversion-based plan correction (**RBPC**) on TPC-DS benchmark (10 GB) [44] and three real-world customer workloads. Table 1 shows some summary statistics of the workloads. We use the CPU time consumed during execution as the measure of execution cost.

The major questions our evaluation focuses on are:

- **Plan Quality** (Section 5.2). How much improvement in plan execution cost does Plan Stitch bring compared to RBPC? How much is the risk of plan regression using Plan Stitch?
- **Cost Estimation** (Section 5.3). How accurate is the stitched plan's estimated execution cost compared to true execution cost?
- **Coverage** (Section 5.4). How many queries and plans can Plan Stitch improve?
- **Overhead** (Section 5.5). What is the overhead of Plan Stitch?
- **Stitched Plan Analysis** (Section 5.6). How different is the stitched plan compared to the optimizer's plan? How many previously-executed plans are used for the stitched plan? Why does the optimizer miss the cheaper stitched plan in its optimization?
- **Parameterized Queries** (Section 5.7). How much does Plan Stitch improve in aggregated execution cost of query instances?
- **Data Changes** (Section 5.8). How much does cost estimation in Plan Stitch degrade when data changes?

5.1 Experimental Setup

We consider the setup where an automated indexing solution, like Automated Indexing in Azure SQL DB [8, 9], is analyzing the workload and incrementally implementing and/or reverting indexes, which results in multiple distinct plans for different subsets of index configurations. To simulate the setup, we use Database Engine Tuning Advisor (DTA) in Microsoft SQL Server to tune each query, and then select different subsets of the recommended indexes to obtain plans and execution costs which are recorded in the execution data repository described in Section 2.

We use each tuned query and the plans executed for the query under different configurations as input of Plan Stitch. For each query q, we generate up to 10 configurations $\{c_i\}$, and each different configuration c_i can result in a different query plan p_i chosen by the optimizer. For each c_k, p_k becomes the original plan, and $\{p_i\}$ become the previously-executed plans under different configurations. For each configuration c_k, Plan Stitch constructs a new plan p' under c_k using $\{p_i\}$. Thus, p' is constrained to the same configuration c_k (e.g., the same set of indexes) as p_k. We validate and execute a stitched plan with our two-stage stitch (Section 4) whenever the estimated execution cost of the stitched plan is significantly cheaper than the original plan, i.e., $\geq 10\%$.

As a baseline, we implement RBPC, where if the original plan is more expensive than a previously-executed plan that is still valid, we revert back to the older cheap plan. In our experiment, a previous plan p_i is not valid only when it accesses indexes that are not in current configuration c_k.

We execute each plan and each stitched plan in isolation (i.e., no concurrent queries) using Microsoft SQL Server and use the average CPU time of five executions in our analysis. All the experiments are run on machines with the same hardware specification. Each machine has Intel Xeon CPU $E5-2660$ v3 2.6GHz, 192GB memory, a 6.5TB hard disk, and running Windows Server 2012 R2.

5.2 Plan Quality Improvement

We first evaluate the improvement of plan quality of Plan Stitch. We execute a stitched plan when it is *estimated* to be at least 10% cheaper than the original plan and compare the *true execution cost* of the stitched plan and the cheapest valid plan (RBPC). Since the estimated execution cost of a stitched plan is always at least as cheap as the cheapest valid plan, Plan Stitch improves all the test cases where RBPC can improve the plan by at least 10% as well.

Figures 4, 5, 6, and 7 show the percentage improvement of the stitched plan's execution cost over RBPC for TPC-DS benchmark and real-world customer workloads. Improvement (%) reported is: $\frac{(CPUTime_{stitched}-CPUTime_{RBPC})\times 100.0}{CPUTime_{RBPC}}$. We report the distribution of improvements using an equi-width histogram, with the x-axis labels being the lower bin boundary and y-axis representing the percentage of cases where Plan Stitch was invoked. For example, Figure 4 shows that, 16% of the stitched plans are $0\%-10\%$ cheaper in execution cost than the plan chosen by RBPC, 16% of them are $10\%-20\%$ cheaper, etc. Compared with the plan chosen by RBPC, Plan Stitch further reduces the execution cost by at least 10% for at least 40% of stitched plans across all the workloads, with a maximum of 83% of stitched plans in Cust3 workload. In particular, Plan Stitch further reduces the execution cost of the plan by least 50% for 28% of the improved plans in Cust2 workload.

Figure 8 shows the percentage of stitched plans where the execution cost is at least 10% worse than RBPC. This happens when the execution cost of the stitched plan is underestimated, *and* such underestimate outweighs the improvement margin of the stitched plan. Across all our workloads, the percent of stitched plans that regress more than 10% compared to RBPC is less than 2.7%. Thus, Plan Stitch has a comparable low risk of regression with RBPC.

5.3 Cost Estimation

Plan Stitch needs to estimate the cost of the stitched plan since it may not have been executed in its entirety before. Even though Plan Stitch combines costs of subplans observed from previous executions, it makes some simplifying assumptions (Section 3.2). We evaluate how much error is introduced due to those assumptions by comparing Plan Stitch's estimates with measured execution costs after the stitched plans are forced and executed.

Figures 9, 10, 11, and 12 show the distribution of cost estimation errors of stitched plans for TPC-DS benchmark and real-world customer workloads. We compute the ratio of execution cost over estimated cost of stitched plans in percentage, excluding the plans where both execution cost and cost difference are small (< 100ms) to reduce noise from runtime variance. Across the workloads, the

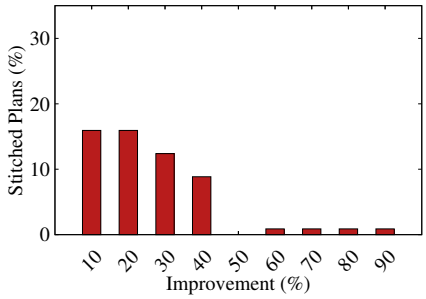


Figure 4: Improvement over RBPC for TPC-DS

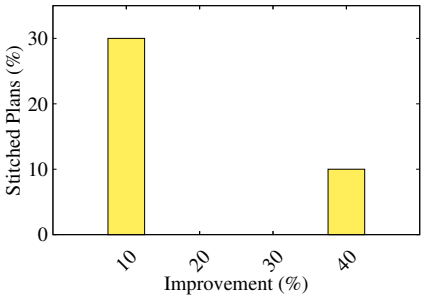


Figure 5: Improvement over RBPC for Cust1

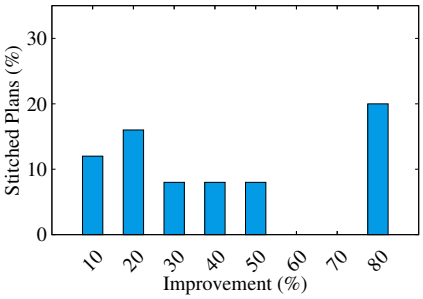


Figure 6: Improvement over RBPC for Cust2

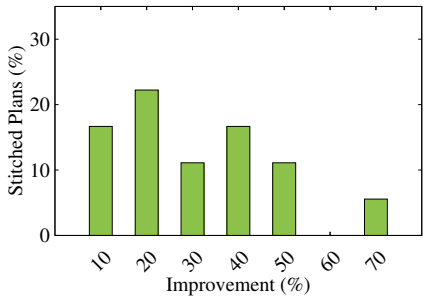


Figure 7: Improvement over RBPC for Cust3

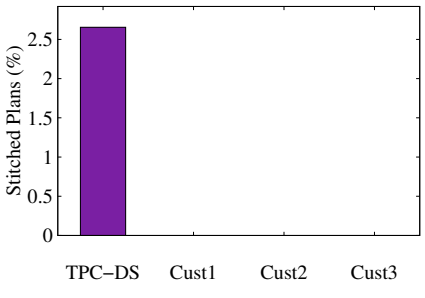


Figure 8: Regression over RBPC

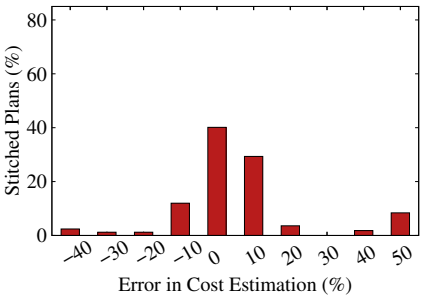


Figure 9: Cost estimation for TPC-DS

estimation is within 20% for at least 70% of stitched plans, with a maximum of 85% in Cust1 workload. In all our real-world customer workloads, the misestimate is less than 50%. Thus, our costing assumptions described in Section 3 mostly hold in practice.

We analyze the small fraction of outliers (8%) in TPC-DS where the misestimates are more than 50% and discover that the misestimates mostly come from implementation artifacts of SQL Server.

First, the plan executed with plan forcing can be slightly different from the specified stitched plan, which makes our cost estimation inaccurate. While Plan Stitch specifies the *full* stitched plan to SQL Server, the optimizer, however, only uses the "skeleton" of the logical structure of the specified plan to constrain its plan search, ignoring operators that are not critical for the logical plan and leaving it to the optimizer to decide the rest of the plan structure. For example, Bitmap and Parallelism[7] are ignored from the "skeleton", and Sort and Compute Scalar operators can be rearranged.

Second, the variance of runtime factors can lead to errors in cost estimation. For example, when an operator executes with insufficient memory, it spills to disk. The execution cost of the spill depends on the memory allocated, which is unpredictable and can be different from the allocation from previous-executed plans. Another example is when the outer side of a Hash Join produces no tuples. Since SQL Server executes the outer and inner side of the Hash Join in parallel, the time to stop the execution of the inner side depends on thread scheduling at runtime.

5.4 Coverage

In this section, we analyze how many plans and queries have been improved with Plan Stitch. Figure 13 shows the percent of plans with a reduction of at least 10% in execution cost with Plan Stitch and RBPC among all the test cases of each workload. Plan Stitch improves up to 20× more plans compared with RBPC across the workloads. Because a query can have multiple plans improved with different configurations, we also analyze how the improved plans are distributed over the queries. Figure 14 shows the percent of queries which have at least one plan with 10% or more reduction in execution cost. Plan Stitch covers up to 6× additional queries compared with RBPC.

While both Plan Stitch and RBPC opportunistically improve plan quality with the same set of previously-executed plans, Plan Stitch improves the plan quality of significantly more plans and queries compared with RBPC. Since there are still many opportunities for plan quality improvement even without plan regression, it is worth exploring using Plan Stitch routinely in query executions.

5.5 Overhead

In this section, we analyze the overhead of constructing and forcing stitched plans with Plan Stitch, by breaking it into three major components: stitching a plan from previously-executed plans (*Stitch*), preparing a plan XML of the stitched plan for plan forcing (*Xml*), and validating the stitched plan with the optimizer (*Validate*). This includes the total overhead of the two-stage fall-back of Plan Stitch as described in Section 4.

Figure 15 shows the overhead of Plan Stitch and its break down, compared with the time spent on optimizing the corresponding query by the optimizer without Plan Stitch, averaged over each workload. We compare the overhead of Plan Stitch to that of optimizing the same query because Plan Stitch re-optimizes an executed plan of the query leveraging past executions of the query. Overall, the overhead of Plan Stitch is less than 88% of the time compared with optimizing the corresponding query with the optimizer.

We investigate the stitched plans in our customer workload Cust1, where the overhead is significantly higher than other workloads. In this workload, many queries have multiple self-joins of the same table, which leads to more candidates of potential equivalent sub-

[7]Parallelism operator either distributes or gather tuple streams for multi-threaded processing.

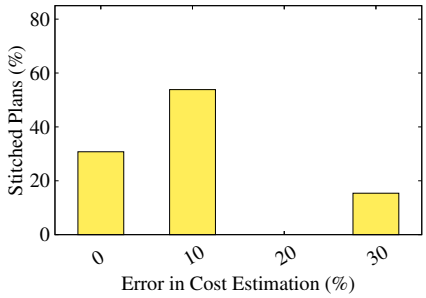


Figure 10: Cost estimation for Cust1

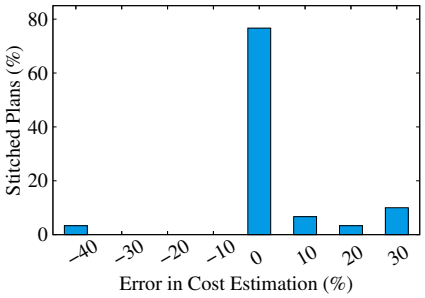


Figure 11: Cost estimation for Cust2

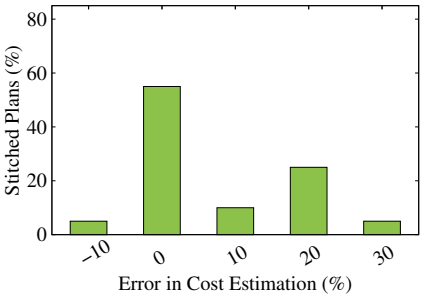


Figure 12: Cost estimation for Cust3

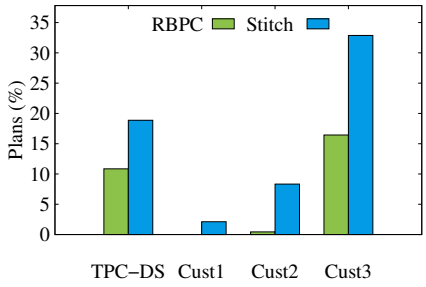


Figure 13: Plan coverage

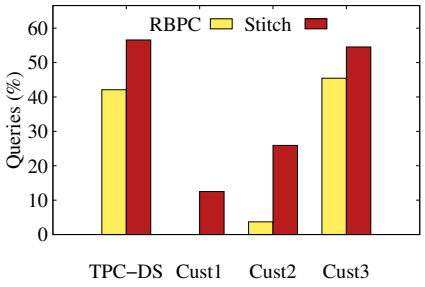


Figure 14: Query coverage

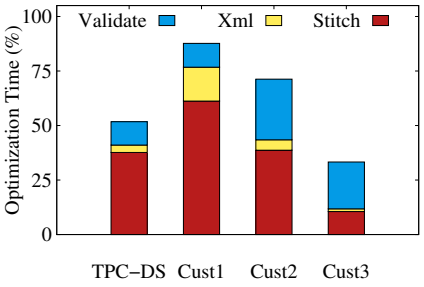


Figure 15: Overhead of Plan Stitch

plans among previously-executed plans than in other workloads. Using our heuristics to construct the constrained search space becomes more expensive, and consequently, the stitch algorithm has more overhead than that in other workloads.

5.6 Stitched Plan Analysis

In this section, we analyze the resulting stitched plans in detail. First, how does the stitched plan differ from the original plan? We classify plan changes into four categories: leaf operators (*Leaf*), internal operators (*Inter*), join order (*JoinO*), and plan structure (*Struct*). Comparing the original and stitched plans, if their sets of physical leaf operators (e.g., Table Scan on A) are different, we say the leaf operators change. Internal operators change when either the physical (e.g., Hash Join) or logical operator (e.g., Inner Join) changes for internal operators, or when the plan structures are different. A plan structure change happens when different query transformation rules apply (e.g., aggregate push down or join order change). We also show a special kind of plan structure change – the join order change. The four categories of changes are not exclusive and could all happen in one stitched plan.

Figure 16 shows the percentage of stitched plans with each category of changes. Across the workloads, in at least 94% of stitched plans, the leaf operators change; in at least 83% of stitched plans, the internal operators are different. The changes of leaf operators and some internal operators are "local", since we can correct the original plan by replacing the corresponding operator. However, for a significant number of stitched plans across the workloads, the changes are substantial: more than 63% have plan structural changes, including more than 33% with different join orders. Such substantial changes require careful examination and correction of the plan structure to ensure the validity of the query semantics if done manually. This is precisely the challenge of combining efficient subplans into a valid plan, which is addressed with an automatic manner in Plan Stitch.

Second, how many previously-executed plans actually contribute to the stitched plan? Figure 17 shows the distribution. A significant number of stitched plans only consist of subplans from a small number of previously-executed plans: more than 53% from up to 4 plans across our workloads. We empirically observed that simply stitching the original plan with the cheapest valid plan does not necessarily improve the plan quality beyond the valid plan itself. Instead, Plan Stitch benefits from the diverse subplans of previously-executed plans, even if the plans themselves are more expensive in execution cost than the original plan. This confirms the motivation of Plan Stitch that even a suboptimal plan has efficient subplans that can help improve plan quality.

Finally, since the original plan and the stitched plan always both exist in the optimizer's search space – one is found during normal optimization and the other is found with plan forcing, why does the optimizer fail to return the cheaper stitched plan in its optimization? This can be either a cost misestimate where the optimizer thinks the stitched plan is more expensive, or a search strategy issue where the optimizer neglects such a plan. Figures 18, 19, 20, 21 show the optimizer's estimated cost of the stitched plan over that of the original plan. When the ratio is less than 1, the optimizer estimated the stitched plan to be cheaper than the original plan and thus it has an issue in its search strategy; otherwise, the optimizer thinks the original plan is cheaper and thus it is a cost misestimate.

Perhaps surprisingly, for a significant number of plans (10%-22%), the optimizer estimates the stitched plan to be cheaper (i.e., the ratio < 1.0). However, the estimated cost difference for such plans is mostly small, i.e., the ratio > 0.9. The optimizer misses plans with slightly lower cost estimate when it aggressively prunes its search space under a limited time budget for optimization. For majority of the plans, the optimizer misestimates the costs and finds the original plans cheaper, which is caused by the fundamental challenges of cardinality estimation and cost modeling.

5.7 Parametric Queries

In this experiment, we compare Plan Stitch to RBPC on parametric TPC-DS benchmark. We generate 50 query instances with different parameter bindings for each tuned query in TPC-DS bench-

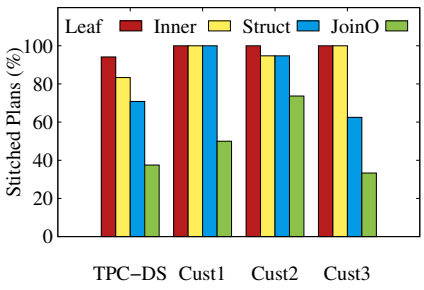


Figure 16: Stitched plans with different types of changes

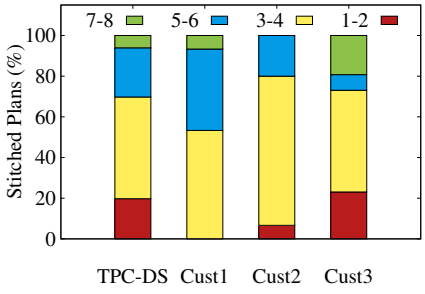


Figure 17: Number of plans contributing to stitched plans

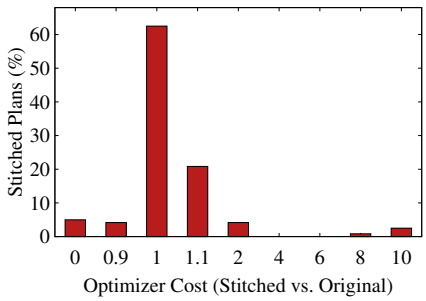


Figure 18: Query optimizer's cost estimation for TPC-DS

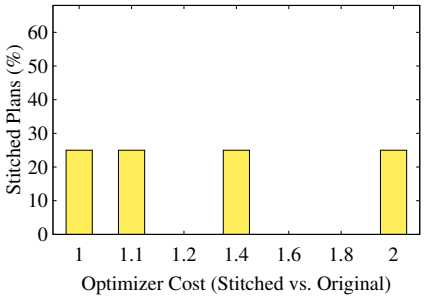


Figure 19: Query optimizer's cost estimation for Cust1

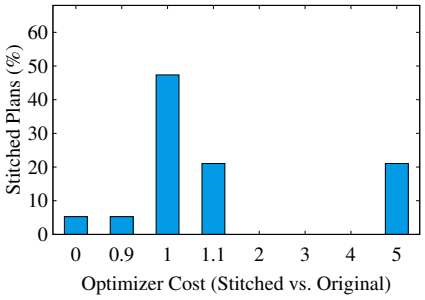


Figure 20: Query optimizer's cost estimation for Cust2

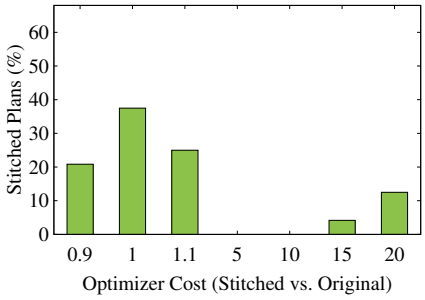


Figure 21: Query optimizer's cost estimation for Cust3

mark using dsqgen [45]. For each query and configuration, we randomly select 5 instances from the 50 instances and execute them with the same plan chosen by the optimizer. To ensure the 5 instances of different parameters share the same query plan under the configuration, we compile the first query instance and cache its plan. Microsoft SQL Server will use the cached plan to execute other instances of the same query.[8]

As mentioned in Section 1, we average the execution cost over multiple instances for each plan of a query template. For RBPC, we average the execution cost statistics over instances of a plan as the execution cost of the plan. For Plan Stitch, we average both the operator-level execution cost and the number of executions of subplans over instances of a plan. We assume the execution cost and the number of executions of a subplan are independent. With this independence assumption, Plan Stitch constructs the cheapest stitched plan in average execution cost.

For parametric queries, we are interested in both the execution cost of queries and the variance of the execution cost over different instances of the same query. Figure 22 shows the improvement of plan quality using Plan Stitch compared to RBPC. For 61% of the stitched plans, Plan Stitch further reduces the execution cost by at least 10% compared to RBPC, with a reduction of more than $2\times$ for 4% of the improved plans. For a small percent of stitched plans (5.6%), the stitched plan regresses more than 10% compared to the plan from RBPC. Such regressions are results of execution cost misestimates of stitched plans due to reasons discussed in Section 5.3. Figures 23 and 24 show the distribution of the standard deviation of the improved plans for Plan Stitch and RBPC. The standard deviation is less than 2% for more than 95% of the improved plans, and the deviation of Plan Stitch is similar to RBPC. Thus, Plan Stitch improves the plan quality without introducing more variance of execution cost compared to RBPC.

[8]Some queries have different semantics for different instances in parameterized TPC-DS, e.g., when a column name is a parameter. We ensure the selected instances have the same semantics.

5.8 Data Changes

In this section, we evaluate how the quality of the stitched plan can degrade when the data changes and the execution feedback is less accurate. Stale execution feedback can increase errors in Plan Stitch's cost estimation, which can increase the risk of regression with the resulting stitched plan. We measure the accuracy of Plan Stitch's cost estimation and compare the stitched plan to RBPC's plan in the presence of data changes.

We consider three types of operations that change the data distribution or the database size: update, insertion, and deletion. Updates change the data distribution, while insertions and deletions change both the size and data distribution of the database.

We populate a TPC-DS database based on its specification [44] using dsdgen [45] with scale factor 10 (i.e., $10g$) as the original database. Based on the original database, we create 5 changed databases with the three types of operations, including three databases with different degrees of updates, one database decreased in size, and one database increased in size.

Based on the data maintenance specification of TPC-DS benchmark, we generate three refresh runs (with insertions and deletions) as three streams of updates. Starting from the original TPC-DS database ($10g$), we apply the three streams sequentially and get three snapshots of the database: $u1$ with only the first update stream applied to $10g$, $u12$ with the first two update streams applied to $10g$, and $u123$ with all three update streams applied to $10g$. We simulate database size change by populating TPC-DS databases with scale factor 9 (i.e., $9g$) and 11 (i.e., $11g$), again based on the TPC-DS benchmark specification.

As with previous experiments, for each query, we execute the plans under multiple configurations on the original database $10g$

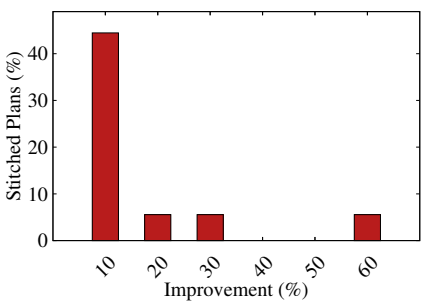


Figure 22: Improvement over RBPC for parametric TPC-DS

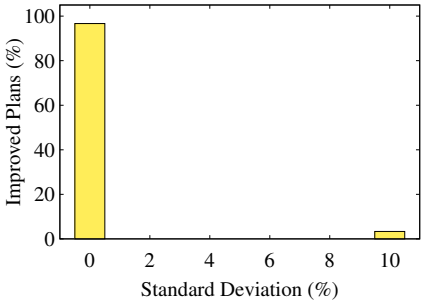


Figure 23: Standard deviation of RBPC for parametric TPC-DS

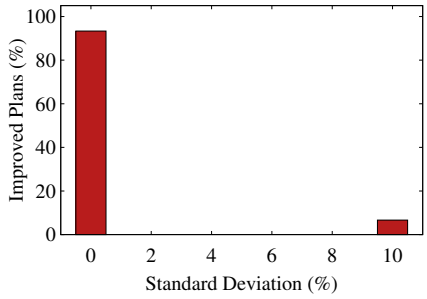


Figure 24: Standard deviation of Plan Stitch for parametric TPC-DS

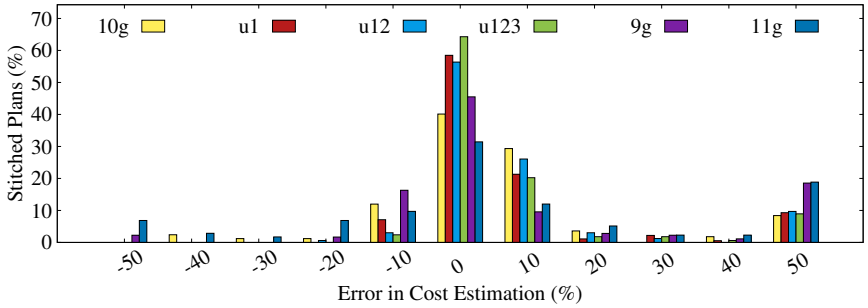


Figure 25: Cost estimation for TPC-DS databases with and without data changes

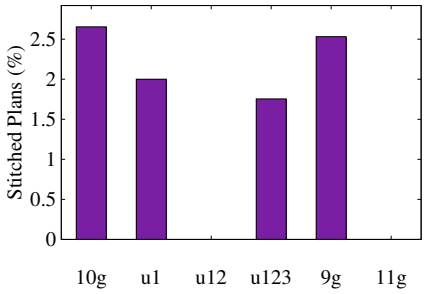


Figure 26: Regression for original and changed TPC-DS databases over RBPC

and collect the execution feedback. To simulate the scenario where execution feedback becomes stale after data changes, Plan Stitch constructs stitched plans based on the execution feedback from the *original database*, and executes them *after* data changes have been applied. Similarly, RBPC chooses plans based on stale execution feedback and executes them on the changed databases.

We evaluate Plan Stitch's cost estimation with stale execution feedback. As reported in Section 5.3, Plan Stitch can have cost estimation errors even without data changes. To isolate the effect of data changes, we compare the percent of stitched plans with different degrees of cost estimation errors with and without data changes.

Figure 25 shows the cost estimation errors using Plan Stitch when forcing the stitched plans on the original and the changed TPC-DS databases. On the databases with updates alone ($u1, u12, u123$), the percent of stitched plans with $< 20\%$ error in cost estimation is similar to that when forcing the stitched plans on the original database (i.e., 81%). With more significant data changes, however, we observe noticeable increase in cost estimation errors. When the stitched plans execute on the $9g$ database, only 70% of them has $< 20\%$ error in cost estimation. The number is even less on the $11g$ database, i.e., 52%.

Figure 26 shows the percent of stitched plans that regress significantly ($> 10\%$ in CPU cost) compared with RBPC. While the error in cost estimate can increase with data changes, the chance of regression compared with RBPC is still low for all five databases. There are three main reasons that keep the regression rate low. First, Plan Stitch constructs plans that are significantly cheaper than RBPC on the original database, which provides a buffer to tolerate the increased error in cost estimation even when the execution feedback becomes inaccurate on the changed databases. Second, while the cost estimate has large error, the rank order of the plans can stay the same, e.g., the original plan and the plan chosen by RBPC also become more expensive. So the error in cost estimate does not necessarily lead to a relatively more expensive plan. Finally, with stale execution feedback, the plan quality with RBPC can degrade more than that of Plan Stitch. Thus, Plan Stitch can even regress less in the changed databases than in the original database compared with RBPC.

5.9 Summary

We summarize our key findings with our experiments as below:

- **Plan Quality**. Compared to reversion-based plan correction, Plan Stitch further reduces the execution cost by at least 10% for up to 83% of the improved plans across our workloads, with a reduction of at least 50% for 28% of the improved plans in one workload. That is, by combining subplans from different plans, Plan Stitch can significantly reduce execution costs over the plans returned by the optimizer. Plan Stitch also has a low risk of plan regression, where among all the workloads, only 2.7% of the stitched plans regresses for more than 10% in execution cost in one workload among all the workloads.
- **Cost Estimation**. Estimated execution cost is within 20% of the true execution cost for at least 70% of the stitched plans across all the workloads. This confirms that our costing assumptions in Section 3 mostly hold in practice.
- **Coverage**. Plan Stitch improves plan quality for up to 33% of plans across our workloads, with up to $20\times$ additional plan coverage compared with RBPC. This confirms that there are abundant opportunities of improving plan quality by combining efficient subplans from multiple plans.
- **Overhead**. Plan Stitch constructs plans with less than the time taken to optimize the corresponding query by the optimizer.
- **Stitched Plan Analysis**. More than 63% of the stitched plans have different plan structures than the original plans. This supports our initial claim that combining efficient subplans manually is challenging. We also show that Plan Stitch requires less

than four previously-executed plans to construct the cheapest stitched plans for up to 80% of the cases across our workloads. Finally, we find that the optimizer misses the cheaper stitched plan mostly due to errors in its cost estimates.
- **Parameterized Queries**. Compared to reversion-based plan correction, Plan Stitch further improves plan quality without introducing noticeable variance for parameterized queries.
- **Data Changes.** With data changes, the execution feedback becomes inaccurate. While the error in cost estimate of Plan Stitch can increase if the data changes significantly, Plan Stitch can still improve the plan quality and keep the risk of regression low.

6. RELATED WORK

Execution feedback to improve plan quality. Using execution feedback to improve plan quality has been an area of active research for the past few decades. Previous work can be broadly classified into three categories: (a) using observed cardinality of a query's expressions to improve optimization of the same query; (b) using observed cardinality to improve summary statistics to help many queries; and (c) improving the optimizer's cost model.

Chen et al. [17] and Stillger et al. [43] collect cardinality feedback to improve cost estimates and hopefully the end-to-end plan quality. Chaudhuri et al. [16] extends the query execution framework to proactively obtain additional cardinality feedback by evaluating alternative subexpressions that are not in the final plan. One important observation in previous work is that since the search space of the optimizer is huge, collecting execution feedback of every subexpression is infeasible. In such cases, partial feedback, i.e., using true cardinality of some expressions and estimated cardinality for others, introduces inconsistencies which can lead to bias in subexpression selection and plans worse in execution compared to plans obtained with estimated cardinalities only. Markl et al. [33] proposes a consistent selectivity estimation method based on maximum entropy. Wu et al. [47] iteratively samples subexpression cardinalities and requests the optimizer for a new query plan with updated cardinality until the output plan reaches a fixed point. Herodotou et al. [26] enumerates the plan space and executes plans to get true cardinalities until a desired plan is found.

Another line of work focuses on using observed cardinality to improve the summary statistics, such as histograms [1,11,14]. While the approaches in category (a) primarily focuses on reusing the cardinality for future optimizations of the same query, these approaches can improve cardinality estimates for expressions from multiple queries, thus providing broader benefit.

The third category of work focuses on the optimizer's cost model (assuming better cardinality estimates are available). Wu et al. [25] carefully tunes the optimizer's cost model for better cost estimation. Data-driven, sophisticated machine learning techniques are also studied to predict query execution cost [3, 20, 32].

Plan Stitch differs from the above approaches because we rely directly on observed execution cost without decoupling cost estimation into cardinality estimation and cost modeling. Thus, Plan Stitch can circumvent inaccuracies in cardinality estimates, cost modeling, or both. By using observed execution cost, Plan Stitch results in very low risk of identifying a stitched plan which is even more expensive than the original plan returned by the optimizer. Such low risk is crucial for an automated solution in a production environment. Note, however, since Plan Stitch limits itself to only observed subplans, it cannot use the feedback to cost or explore unseen subplans which may be even more efficient.

Plan regression correction. As noted in Section 1, Plan Stitch improves upon reversion-based plan correction [5, 6] techniques used in commercial DBMSs. Plan Stitch preserves the desirable properties of low risk and low overhead while leveraging efficient subplans from previous plans to further improve plan quality, resulting in plans which are often $2\times$ to two orders of magnitude better compared to just using the cheapest previously-executed plan.

Query hinting. Commercial databases expose query hints to restrict and/or influence the search space of the optimizer. Microsoft SQL Server supports query hints to influence the join order, access paths, up to providing the entire plan [38]. Oracle Database and IBM DB2 provide similar functionalities [18, 34]. Bruno et al. [15] proposes a general language to specify a rich set of constraints to influence the optimizer to pick plans. While these hinting techniques provide a way to influence the optimizer, the task of identifying which hint is appropriate to improve a given query's execution cost still relies on human experts, such as a DBA. Such manual tuning is labor-intensive, time consuming, and often error-prone. Plan Stitch fills this void by automatically identifying good subplans from previously-executed plans, thus it eliminates manual tuning and is applicable to complex applications at the scale of millions of databases in a cloud platform such as Azure SQL DB.

Exploring alternative plans. `AND-OR` graph [24] represents a search space that allows the exploration of alternative plans. Bruno et al. [12], Chaudhuri et al. [13], and Dash et al. [35] encode a plan with an `AND-OR` graph and modify leaf `AND` nodes of the graph to reuse the internal plan structure and the query optimizer's cost, and thus reduce the overhead of physical configuration tuning. Sudarshan et al. [27] optimizes parametric queries and reduces the number of optimizer calls by caching plans and reusing optimizer's cost for shared subexpressions. Plan Stitch focuses on execution cost and improves plan quality with execution feedback. It constructs the cheapest plan in execution cost, which can be different from previously executed plans with both local and structural changes.

7. CONCLUSION

We propose Plan Stitch, a novel, fully-automated, low-overhead technique that uses operator-level execution cost statistics of previously-executed plans to opportunistically construct new plans that are cheaper in execution cost than the cheapest valid previous plan, with low risk of plan regression. We implement Plan Stitch on top of Microsoft SQL Server without any changes to the DBMS.

We comprehensively evaluate Plan Stitch on both industry-standard TPC-DS benchmark and three real-world customer workloads. Our evaluation shows that Plan Stitch significantly improves the plan quality compared with reversion-based plan correction (RBPC), with a reduction in execution cost of up to two orders of magnitude for some plans. In addition, Plan Stitch has a wider applicability than RBPC, improving the quality of up to $20\times$ more plans. Moreover, Plan Stitch is often able to leverage previously-executed plans to find new plans which are significantly cheaper than the plan returned by the optimizer with an overhead lower than the time taken to optimize the corresponding query by the optimizer. Given its benefits and low overhead, it is worth exploring the potential opportunities of improving plan quality when multiple executed plans are available using Plan Stitch, with or without query regression.

Acknowledgments

The authors would like to thank the anonymous reviewers and Anshuman Dutt for their valuable feedback.

8. REFERENCES

[1] A. Aboulnaga and S. Chaudhuri. Self-tuning Histograms: Building Histograms Without Looking at Data. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 181–192, 1999.

[2] Access Plan in IBM DB2. https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/com.ibm.db2.luw.admin.explain.doc/doc/c0021356.html.

[3] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. Learning-based Query Performance Modeling and Prediction. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, pages 390–401, 2012.

[4] APPLY Operator in Microsoft SQL Server. https://social.technet.microsoft.com/wiki/contents/articles/6525.apply-operator-in-sql-server.aspx.

[5] Automatic Plan Correction in Microsoft SQL Server 2017 and Azure SQL Database. https://docs.microsoft.com/en-us/sql/relational-databases/automatic-tuning/automatic-tuning.

[6] SQL Plan Management with Oracle Database 12c. http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-sql-plan-mgmt-12c-1963237.pdf.

[7] Automatic Statistics Update in Microsoft SQL Server. https://docs.microsoft.com/en-us/sql/relational-databases/statistics/statistics.

[8] Automatic Index Tuning in Azure SQL Database. https://blogs.msdn.microsoft.com/sqlserverstorageengine/2017/05/16/automatic-index-management-in-azure-sql-db/.

[9] Azure SQL Database. https://azure.microsoft.com/en-us/services/sql-database/.

[10] B. Babcock and S. Chaudhuri. Towards a Robust Query Optimizer: A Principled and Practical Approach. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 119–130, 2005.

[11] N. Bruno and S. Chaudhuri. Exploiting Statistics on Query Expressions for Optimization. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 263–274, 2002.

[12] N. Bruno and S. Chaudhuri. Automatic Physical Database Tuning: A Relaxation-based Approach. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 227–238, 2005.

[13] N. Bruno and S. Chaudhuri. To Tune or Not to Tune?: A Lightweight Physical Design Alerter. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 499–510, 2006.

[14] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: A Multidimensional Workload-aware Histogram. *SIGMOD Rec.*, 30(2):211–222, May 2001.

[15] N. Bruno, S. Chaudhuri, and R. Ramamurthy. Power Hints for Query Optimization. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 469–480, 2009.

[16] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. A pay-as-you-go framework for query execution feedback. *PVLDB*, 1(1):1141–1152, 2008.

[17] C. M. Chen and N. Roussopoulos. Adaptive Selectivity Estimation Using Query Feedback. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 161–172, 1994.

[18] Creating and deploying optimization hints for SQL statements that run on DB2 for z/OS. https://www.ibm.com/support/knowledgecenter/SS7LB8_4.1.0/com.ibm.datatools.qrytune.sngqry.doc/topics/reviewingvph.html.

[19] A. Dutt, V. Narasayya, and S. Chaudhuri. Leveraging Re-costing for Online Optimization of Parameterized Queries with Guarantees. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1539–1554, 2017.

[20] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 592–603, 2009.

[21] S. Ganguly. Design and Analysis of Parametric Query Optimization Algorithms. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 228–238, 1998.

[22] A. Ghosh, J. Parikh, V. S. Sengar, and J. R. Haritsa. Plan Selection Based on Query Clustering. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 179–190, 2002.

[23] J. Goldstein and P.-A. Larson. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. *SIGMOD Rec.*, 30(2):331–342, May 2001.

[24] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 209–218, Washington, DC, USA, 1993.

[25] H. Hacigumus, Y. Chi, W. Wu, S. Zhu, J. Tatemura, and J. F. Naughton. Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable? In *Proceedings of the 2013 IEEE International Conference on Data Engineering*, pages 1081–1092, 2013.

[26] H. Herodotou and S. Babu. Xplus: a sql-tuning-aware query optimizer. *PVLDB*, 3(1-2):1149–1160, 2010.

[27] A. Hulgeri and S. Sudarshan. AniPQO: Almost Non-intrusive Parametric Query Optimization for Nonlinear Cost Functions. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 766–777, 2003.

[28] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric Query Optimization. *The VLDB Journal*, 6(2):132–151, May 1997.

[29] Y. E. Ioannidis and R. Ramakrishnan. Containment of Conjunctive Queries: Beyond Relations As Sets. *ACM Trans. Database Syst.*, 20(3):288–324, Sept. 1995.

[30] P.-A. Larson, W. Lehner, J. Zhou, and P. Zabback. Cardinality Estimation Using Sample Views with Quality Assurance. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 175–186, 2007.

[31] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How Good Are Query Optimizers, Really? *PVLDB*, 9(3):204–215, Nov. 2015.

[32] J. Li, A. C. König, V. Narasayya, and S. Chaudhuri. Robust Estimation of Resource Consumption for SQL Queries Using Statistical Techniques. *PVLDB*, 5(11):1555–1566, July 2012.

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

[34] Database Performance Tuning Guide - Using Optimizer Hints. https://docs.oracle.com/cd/B19306_01/server.102/b14211/hintsref.htm#i8327.

[35] S. Papadomanolakis, D. Dash, and A. Ailamaki. Efficient Use of the Query Optimizer for Automated Physical Design. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 1093–1104, 2007.

[36] Microsoft SQL Server Plan Guides. https://docs.microsoft.com/en-us/sql/relational-databases/performance/plan-guides.

[37] Microsoft SQL Server Query Store. https://docs.microsoft.com/en-us/sql/relational-databases/performance/monitoring-performance-by-using-the-query-store.

[38] Microsoft Query Hints. https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-query.

[39] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 249–260, 2000.

[40] 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 1979 ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979.

[41] Statistics XML in Microsoft SQL Server. https://docs.microsoft.com/en-us/sql/t-sql/statements/set-statistics-xml-transact-sql.

[42] Showplan XML Schema in Microsoft SQL Server. schemas.microsoft.com/sqlserver/2004/07/showplan/.

[43] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 19–28, 2001.

[44] TPC Benchmark DS: Standard Specification v2.6.0. http://www.tpc.org/tpcds/.

[45] TPC-DS Tools. http://www.tpc.org/tpc_documents_current_versions/current_specifications.asp.

[46] Microsoft SQL Server USE PLAN Query Hint. https://technet.microsoft.com/en-us/library/ms186954(v=sql.105).aspx.

[47] W. Wu, J. F. Naughton, and H. Singh. Sampling-Based Query Re-Optimization. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1721–1736, 2016.