Understanding and Detecting Query Performance Regression in Practical Index Tuning

Wentao Wu, Anshuman Dutt, Gaoxiang Xu, Vivek Narasayya, Surajit Chaudhuri
Microsoft Research, Redmond, USA
{wentao.wu, andut, gxu, viveknar, surajitc}@microsoft.com

Abstract

Existing index tuners typically rely on the "what if" API provided by the query optimizer to estimate the execution cost of a query on top of an index configuration. Such cost estimates can be inaccurate and may therefore lead to significant query performance regression (QPR) once the recommended indexes are materialized. This becomes a serious problem for cloud database providers, such as Microsoft's Azure SQL Database, that offer index tuning as an automated service (a.k.a. "auto-indexing"). Previous work has explored use of supervised machine learning (ML) to reduce the likelihood of QPR. However, the trained ML models have limited generalization capability when applied to new databases and workloads. We propose an alternative approach where we analyze the query plan pairs with significant QPRs and look for structural changes due to the new index configuration that could explain the QPR. We perform such study for index tuning data across many benchmark and real-world database workloads, for multiple realistic index tuning scenarios. Our study reveals that most of the significant QPRs can be attributed to a small number of common "regression patterns" characterizing the structural plan changes, and we further propose a pattern-based QPR detector accordingly. Our experimental evaluation shows that the pattern-based QPR detector can significantly outperform existing ML-based QPR detectors.

1 Introduction

Index tuning is critical to accelerating query execution in modern database systems. Existing index tuners typically rely on the "what-if" API provided by the query optimizer [6, 7, 47], as illustrated in Figure 1, that allows for estimating the execution cost of a query given a *configuration* (i.e., a set) of proposed *hypothetical* indexes, as well as their associated statistics, without actually materializing the indexes. However, what-if cost estimation is still based on query optimizer's cost model, which can be inaccurate for reasons such as cardinality estimation (CE) errors and may therefore lead to significant query performance regression (QPR) when the recommended indexes are eventually deployed [8, 59]. That is, the execution of a query becomes much slower by using the recommended indexes. QPR has been a serious problem for cloud database providers that offer index tuning as an automated service (a.k.a. "auto-indexing"). As was reported by [8], around 11% of the indexes that were automatically created by the auto-indexing service offered by Microsoft's Azure SQL Database [30] had to be reverted due to QPR. Therefore, detecting QPR *before* materializing the recommended indexes can help significantly reduce the operational cost of cloud auto-indexing service.

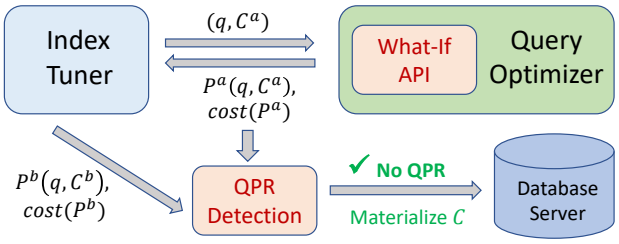


Figure 1: The architecture of cost-based index tuning with what-if query optimizer calls and QPR detection. [Notation: q, a SQL query; C^b, the existing index configuration (i.e., "before configuration"); C^a, the index configuration recommended by the index tuner (i.e., "after configuration"); P^b, the "before plan" of q on top of C^b; P^a, the "after plan" of q on top of C^a.]

We aim to develop a low-overhead technique for QPR detection. Specifically, consider a query q and the existing configuration, i.e., "before configuration", C^b, for which the index tuner proposes a new configuration, i.e., "after configuration", C^a. Even before deploying C^a, we can make a what-if call (q, C^a) to the query optimizer that returns the query plan of q for the "after configuration" C^a, as shown in Figure 1. We call this query plan the "after plan" and denote it with P^a, to distinguish it from the "before plan" P^b of q on top of the existing configuration C^b that the index tuner aims to improve over. The goal of QPR detection is to decide whether the execution time of P^a will be significantly higher than that of P^b *without executing* P^a, though the execution information of P^b is presumed available. If no QPR is detected, the configuration C^a can then be materialized for accelerating the execution of q. There has been recent work on addressing QPR detection and reduction in the context of index tuning [11, 41, 53, 60]. Most of this work applies supervised machine learning (ML) to build classification or regression models to predict/detect QPR. However, ML-based QPR detectors often exhibit poor generalization capability when evaluated on new databases and workloads that are not included in the training data, notwithstanding their nontrivial overhead.

In this paper we propose an alternative approach where we analyze the query plan pairs with significant QPRs and look for structural changes due to the new index configuration that could explain the QPR. We perform such study for index tuning data collected offline across many benchmark and real-world database

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

workloads. Our study reveals that most of the significant QPRs can be attributed to a small number of common "regression patterns" characterizing the structural plan changes, and we further propose a pattern-based QPR detector accordingly. Our experimental evaluation shows that the pattern-based QPR detector can significantly outperform existing ML-based QPR detectors.

Collection of Index Tuning Data. In the classical sense, index tuner *tunes* a given query/workload by recommending a configuration including all indexes that can improve the execution performance at once. We use the term *one-shot* tuning to represent this classic index tuning scenario that has been studied extensively in the literature, e.g., [5, 6, 47, 52]. However, in modern cloud database services, such as Microsoft's Azure SQL Database [30], indexes need to be optimized in a continuous manner [8] to adapt to evolving workloads and manage storage constraints. We therefore also collect data from two more scenarios that represent real-world index tuning applications: (1) *incremental tuning*, which constrains the index tuner in terms of the number of indexes it should return and performs tuning in an incremental manner until no more indexes can be recommended; and (2) *evolutionary tuning*, which simulates index *evolution* (e.g., deletion of existing indexes or introduction of new indexes) from a well-tuned database for reasons such as storage constraints. These two scenarios aim at capturing more QPRs that could emerge from such dynamic environments as in cloud auto-indexing services. We collect 1.2 million data points following these tuning setups, where each data point represents a pair of "before plan" P^b and "after plan" P^a. As expected, index recommendations are beneficial for a large number of plan pairs, and we highlight representative examples of such benefits in the full version of the paper [2]. However, this paper focuses primarily on the regressed cases (QPRs).

Analysis of QPRs. We then analyze the QPRs that appear in the collected index tuning data to understand their root causes. Surprisingly, we find that most of the QPRs can be attributed to a small set of *regression patterns* that are simple and easy to understand. A regression pattern characterizes some "local" change or transformation in terms of query plan structure. For example, the regressed "after plan" misses the pushdown of an aggregation (ref. Figure 4) or a bitmap filter (ref. Figure 5) that is performance critical. We further develop a taxonomy that categorizes regression patterns into two general categories: (c1) QPRs due to problematic change of access path between P^b and P^a, and (c2) QPRs due to critical optimizations that were present in P^b but missing in P^a. The simplicity of the identified QPR patterns is a strength that makes it easier to design simple (and therefore computationally more efficient) but effective pattern-based QPR detectors. The fact that there are only a handful of major QPR patterns also makes the overall task of pattern-based QPR detection addressable and manageable. More importantly, we observe that *most of the significant QPRs can be accounted for by regression patterns from the category (c1) where the regressed "after plan" P^a contains an "expensive" nested-loop join (NLJ) operator that does not appear in the "before plan" P^b (ref. Figure 2).* The emergence of such expensive NLJ is typically due to *cardinality underestimation* errors made by the query optimizer [23]: the availability of the new indexes inadvertently makes the NLJ look attractive to the query optimizer in terms of estimated cost. Although better cardinality estimation could improve query plan quality and therefore reduce the chance of QPR, the problem of accurate cardinality estimation has not yet been settled despite decades of research efforts (see [49]). State-of-the-art ML-based cardinality estimators [49] could improve cardinality estimation but with no guarantee on the accuracy. Moreover, they also incur nontrivial overhead of data collection and model training [49]. Therefore, while it may be an interesting direction for future work, we deliberately avoid using these ML-based cardinality estimators and make progress in QPR detection through an approach that can work with existing erroneous cardinality estimates.

Pattern-based QPR Detector. Motivated by the above observations, we develop a pattern-based QPR detector to identify the "expensive NLJ" regression patterns *before* the "after plan" P^a is executed. This remains a challenging problem, as we need to *precisely characterize such regression patterns* to distinguish harmful NLJs from those that are indeed beneficial. In particular, we need to estimate (1) the expensiveness of an NLJ *without executing* P^a and (2) the degree of cardinality underestimation errors rooted in the expensive NLJ, which are the primary culprit for QPR. To address (1), we develop two metrics, *local expensiveness* and *global expensiveness*. To address (2), we leverage true cardinality information contained by the "before plan" P^b, which is presumably available in the context of QPR detection for index tuning. Specifically, we develop a metric, *cost inflation factors*, to quantify the degree of cardinality underestimation errors of the left/outer and right/inner inputs of the NLJ. We then use the cost inflation factors to *recost* the NLJ as well as the entire plan [13, 55, 56]. We further try to match the logically equivalent join in P^b, and if we find such a join we recost it as well. Finally, we recompute the plan costs based on the recosted joins and infer QPRs based on the new costs. Albeit a relatively simple approach, our experimental evaluation shows that it can significantly outperform existing ML-based QPR detectors, which currently do not use the true cardinality information of the "before plan" P^b. It is non-trivial to extend existing ML model designs to include this information, which might be interesting future work. Our evaluation shows that, even without the use of sophisticated ML-based cardinality estimators, our low-overhead approach based on cost inflation factors can already detect most QPRs successfully (Section 5). ML-based cardinality estimators would further improve the results reported in this paper if their overheads could be reduced.

Contributions, Limitations, and Future Work. To summarize, this paper makes the following contributions:

(C1) We conduct an empirical QPR analysis using large amount of data collected from practical index tuning scenarios (Sections 2). To the best of our knowledge, we are not aware of any previous work on systematically understanding QPRs based on large-scale data generated by following real industrial index tuning applications.
(C2) We find that most of the QPRs can be attributed to a small set of regression patterns characterizing the structural changes between the "before plan" and the "after plan", and we further present a taxonomy of the regression patterns (Section 3).
(C3) We develop a pattern-based QPR detector based on the observation that *the majority of the significant QPRs found in our data can be attributed to the emergence of expensive NLJs in the "after plan"* (Section 4), and our experimental evaluation results demonstrate

Name	DB Size	# Queries	# Tables	# Joins	# Scans
TPC-H	sf=10	22	8	2.8	3.7
DSB	sf=10	67	24	7.7	8.8
JOB	9.2GB	108	21	7.9	2.5
STATS	223MB	91	8	3.3	4.3
Real-DY	587GB	29	7912	15.6	17
Real-LO	108GB	31	1151	8.1	8.9
Real-MS	26GB	39	474	20.2	21.7
Real-RE	100GB	21	20	6.5	7.2
Real-DW	13GB	107	20	6.3	6.9
Real-ED	210GB	36	23	8.8	8.2
Real-MP	2.9GB	127	8	1.6	2.9
Real-SE	256GB	19	3391	5.9	6.9
Real-RM	60GB	15	7	1.9	2.9
Real-SA	40GB	12	32	7.3	9.7

Table 1: Summary of database and workload properties.

Workload	#Queries	#OneShot	#Inc.	#Evol.
TPC-H	22	22	49	1,156
DSB	65	67	191	543,198
JOB	108	108	199	189,320
STATS	91	91	184	38,773
Real-DY	29	29	140	143,255
Real-LO	31	31	42	12,779
Real-MS	39	39	47	199,415
Real-RE	21	21	44	2,704
Real-DW	107	107	37	17,916
Real-ED	36	36	12	875
Real-MP	127	127	12	9,583
Real-SE	19	19	17	9,224
Real-RM	15	15	6	55
Real-SA	12	12	8	6
Total	**724**	**724**	**988**	**1,168,259**

Table 2: Summary of the index tuning data collected.

that the pattern-based QPR detector can significantly outperform state-of-the-art ML-based QPR detectors (Section 5).

While the list of regression patterns presented in this paper is based on the large-scale index tuning data we collected, it is by no means an exhaustive list—we do not rule out emergence of new regression patterns given new databases and workloads. Also, a case-by-case approach may be required to apply each specific regression pattern to practical QPR detection. For instance, if aggregation or bitmap filter pushdown appears to be the major regression pattern on a particular database workload, then one may want to design a QPR detector that focuses on finding such missed pushdowns. In this spirit, the pattern-based QPR detector developed in this paper that focuses on detecting expensive NLJs serves as such an example. Moreover, the regression patterns also provide useful clues for *correcting* the corresponding QPRs. For example, with the notation used in Figure 1, if an index $I \in C$ is the culprit of introducing a slow nested-loop join in P^a that results in a QPR of P^a over P^b, then one may *hint* the query optimizer [34] to not use the problematic index I. Exploration of such more advanced "QPR correction" mechanisms (beyond the naive mechanism of reverting all recommended indexes upon QPR [8]) is beyond the scope of this paper, which can be fertile ground for future research.

Availability. Some of the artifacts, e.g., QPR details of the public benchmark workloads, are available at [2].

2 Index Tuning Data Generation

Let $\mathcal{A}$ be an index tuner, D be a database, and W be a (multi-query) workload. Let C_0 be the *initial configuration* of the database D. Unlike most of the previous work that mainly concerns with indexes, the term "configuration" in this paper refers to *both* indexes and statistics. This is motivated by the observation that some index tuners, such as the Database Tuning Advisor (DTA) developed for Microsoft SQL Server [5], recommend both indexes and statistics with the contract that the estimated benefits of the recommended indexes are based on creating the recommended statistics simultaneously. Moreover, some database systems, such as Microsoft SQL Server, automatically update the corresponding statistics when an index is created [31]. As a result, indexes and statistics are indispensable counterparts in practical index tuning applications.

2.1 Index Tuning Scenarios

We focus on the following setups that emerge from practical index tuning scenarios for collecting index tuning data. Each data point collected represents a pair of "before plan" and "after plan" returned by the query optimizer on top of the existing configuration and the recommended configuration, respectively.

2.1.1 One-shot Index Tuning. For each query $q \in W$, we run the index tuner $\mathcal{A}$ to tune the query q on top of the initial configuration C_0. Let C be the configuration returned by $\mathcal{A}$ after tuning. Moreover, let the two query plans of q on top of C_0 and C be P_0 and P, respectively. We run q on top of both C_0 and C to record the execution time t_0 and t of the two plans P_0 and P. We generate one pair of plans for the query q, which is denoted as (q, P_0, P, t_0, t). A formal algorithmic description of one-shot tuning is given in the full version of this paper [2]. One-shot tuning represents the classic offline index tuning setup that has been studied intensively in the literature, e.g., [5, 6, 47, 52].

2.1.2 Incremental Index Tuning. For each query $q \in W$, we run the index tuner $\mathcal{A}$ to tune the query q in an *iterative* manner. In each iteration, the index tuner $\mathcal{A}$ is constrained to return only one index based on the current configuration. This new index, if any, is then materialized and included into the "current configuration" of the next iteration. The iterative tuning process ends when $\mathcal{A}$ returns nothing. Let C_i be the configuration returned in the i-th iteration by the index tuner, and let P_i be the plan of q on top of C_i and t_i be the recorded execution time of P_i. We generate one pair of plans (P_{i-1}, P_i) for the query q in each iteration $i = 1, 2, ...$, which is denoted as $(q, P_{i-1}, P_i, t_{i-1}, t_i)$. A formal algorithmic description of incremental tuning can be found in the full version [2]. Incremental tuning is useful when index tuning has to be done concurrently while the database server is also processing queries, to reduce the inference or interruption of normal query processing [8].

2.1.3 Evolutionary Index Tuning. For each query $q \in W$, we run the index tuner $\mathcal{A}$ to tune the query q on top of the initial configuration C_0. We then materialize the configuration C returned by $\mathcal{A}$. For each *subset* S of C, we obtain the query plan of q on top of S and record its execution time. We include a pair of plans (q, P_1, P_2, t_1, t_2) for two different subsets S_1 and S_2 of C by *ensuring that the query optimizer's estimated cost of P^b is no less than that of P^a*, where t_1 and t_2 are the execution time of P^b and P^a, respectively. See [2] for a formal algorithmic description of evolutionary tuning.

The evolutionary index tuning setup is motivated by a common scenario that we have seen in practice: *index evolution from a well-tuned database.* Index evolution includes dropping indexes and creating new indexes, due to reasons such as changes on storage constraints. Index evolution, e.g., deletion of existing indexes, may result in QPR, and evolutionary index tuning simulates all possible outcomes of index evolution. Note that we have intentionally

Workload	#All	#QPR	%QPR	$T(P^b)$	$T(P^a)$	%Impr
TPC-H	22	1	4.55%	0.04h	0.01h	85.33%
DSB	67	2	2.99%	0.05h	0.02h	63.49%
JOB	108	14	12.96%	0.33h	0.22h	33.04%
STATS	91	3	3.30%	0.23h	0.24h	-5.95%
Real-DY	29	4	13.79%	0.58h	0.62h	-5.82%
Real-LO	31	3	9.68%	0.03h	0.02h	36.13%
Real-MS	39	1	2.56%	0.09h	0.05h	44.26%
Real-RE	21	4	19.05%	0.23h	0.27h	-17.35%
Real-DW	107	4	3.74%	0.32h	0.31h	3.14%
Real-ED	36	0	0.00%	2.43h	0.29h	88.12%
Real-MP	127	10	7.87%	0.42h	0.41h	2.38%
Real-SE	19	0	0.00%	0.00h	0.00h	80.28%
Real-RM	15	0	0.00%	0.47h	0.23h	50.58%
Real-SA	12	0	0.00%	0.22h	0.19h	13.41%
Total	**724**	**46**	**6.35%**	**5.44h**	**2.88h**	**47.16%**

Table 3: QPRs emerging in one-shot index tuning. [#All, the total number of plan pairs; #QPR, the number of plan pairs with QPRs; %QPR, the percentage of QPR, defined as $\frac{\#QPR}{\#All} \times 100\%$; $T(P^b)$, the total execution time of all "before plan" P^b; $T(P^a)$, the total execution time of all "after plan" P^a; %Impr, the percentage improvement defined as $\left(1 - \frac{T(P^a)}{T(P^b)}\right) \times 100\%$.]

Workload	#All	#QPR	%QPR	$T(P^b)$	$T(P^a)$	%Impr
TPC-H	49	5	10.20%	0.33h	0.30h	7.96%
DSB	191	18	9.42%	0.11h	0.07h	29.97%
JOB	199	28	14.07%	0.61h	0.47h	21.81%
STATS	184	9	4.89%	0.10h	0.11h	-12.81%
Real-DY	140	20	14.29%	8.73h	13.32h	-52.53%
Real-LO	42	3	7.14%	0.04h	0.02h	33.19%
Real-MS	47	5	10.64%	0.09h	0.06h	34.91%
Real-RE	44	7	15.91%	0.31h	0.27h	14.34%
Real-DW	37	3	8.11%	0.25h	0.21h	13.68%
Real-ED	12	0	0.00%	0.20h	0.02h	90.51%
Real-MP	12	2	16.67%	0.01h	0.01h	-56.32%
Real-SE	17	0	0.00%	0.00h	0.00h	38.75%
Real-RM	6	1	16.67%	0.23h	0.13h	42.43%
Real-SA	8	0	0.00%	0.12h	0.08h	31.73%
Total	**988**	**101**	**10.22%**	**11.12h**	**15.10h**	**-35.77%**

Table 4: QPRs emerging in incremental index tuning.

enforced (optimizer estimated) $\text{cost}(P_1) \geq \text{cost}(P_2)$; otherwise, a reasonable index tuner would not even recommend the configuration corresponding to P^a. A similar setup has been used in previous work [11] to generate training data for ML-based QPR detectors, though the constraint $\text{cost}(P_1) \geq \text{cost}(P_2)$ was not forced.

2.1.4 Discussion. We focused on single-query tuning in our empirical study to avoid complexity that can emerge when tuning a multi-query workload, which is a more common scenario in practice. However, it typically requires placing more constraints on the recommended indexes, such as the maximum number of indexes allowed or the maximum storage space that can be taken. These extra constraints can significantly increase the exploration space of a controlled empirical study. Our single-query tuning setups can be thought of as tuning a multi-query workload without such constraints. As a result, it actually has higher coverage in terms of the identified regression patterns (see Section 3), some of which may not appear or appear less frequently when tuning a multi-query workload with constraints. Index interaction has also been covered by single-query tuning, since the index tuning algorithm (e.g., a classic two-phase greedy search algorithm that is implemented inside DTA [5]) used for enumerating index configurations works in the same way of tuning a multi-query workload.

2.2 Results of Index Tuning Data Collected

We use standard benchmarks as well as real customer workloads in our experiments. For benchmark workloads, we use (1) a skewed version [33] of the **TPC-H** benchmark, (2) **DSB** [10], a variant of the **TPC-DS** benchmark with more variety on the data distribution, (3) the "Join Order Benchmark" (**JOB**) [24], and (4) the "Cardinality Estimation Benchmark" (**STATS**) [15]. We also use 10 real workloads. Table 1 summarizes some basic properties of the workloads, in terms of schema complexity (e.g., the number of tables), query complexity (e.g., the average number of joins and table scans contained by a query), and database/workload size. We use Microsoft SQL Server 2022 as the DBMS and use DTA as the index tuner.

Table 2 presents the statistics of the index tuning data collected. We have the same number of plan pairs as that of queries in one-shot tuning, whereas the number of plan pairs in incremental tuning increases by 36.5%. On the other hand, the number of plan pairs obtained from evolutionary tuning is significantly large, due to the exponential explosion of subset enumeration.

2.3 Distributions of QPR

We use the notation (q, P^b, P^a, t^b, t^a) to denote a general *plan pair* in the index tuning data collected, regardless of the specific index tuning scenarios, where P^b and P^a represent the "before plan" and "after plan" as defined in Figure 1, and t^b and t^a represent the execution time of P^b and P^a respectively.

A plan pair (q, P^b, P^a, t^b, t^a) is classified as a QPR if $\frac{t^a}{t^b} - 1 \geq \tau$, where τ is a regression threshold that measures the *degree* of QPR. We set $\tau = 0.5$ in our analysis, i.e., the elapsed query execution time of P^a is at least 50% longer than that of P^b.

Tables 3 and 4 present the distributions of QPRs emerging in one-shot and incremental index tuning, where we see around 6.3% and 10.2% QPRs repectively. While this may seem to suggest that the chance of QPR is relatively low in practice, *it does not mean that such QPRs are insignificant.* To the contrary, some QPRs can be considerable. To demonstrate this, Tables 3 and 4 further present the total execution time $T(P^b)$ and $T(P^a)$ of P^b and P^a for all plan pairs (P^b, P^a) in each workload as well as the percentage improvement at workload level. A negative improvement means a workload-level regression. We observe significant slowdown of the execution on certain workloads albeit a small QPR rate. For example, for incremental tuning on Real-DY, although the percentage of QPR is only around 15%, the total workload execution time is increased from 8 hours to 13.3 hours, i.e., a 52% regression.

Table 5 further presents the distribution of QPRs emerging in evolutionary index tuning. We observe around 7.4% QPR overall, which is in line with the QPR rates observed from one-shot and incremental tuning. We also observe flip of improvement/regression on some workloads. For example, while Real-DY regresses in one-shot and incremental tuning, it improves significantly in evolutionary tuning. On the other hand, JOB improves in one-shot and incremental tuning, but it regresses dramatically in evolutionary tuning.

Summary. While the chance of QPR is around 10% to 15% based on our evaluation, the impact on query execution time can be much higher. As shown in Tables 4 and 5, QPR can result in around 50% to 80% regression in terms of query execution time for certain databases and workloads. Therefore, detecting and correcting QPR is important for practical index tuning. A more complete overview of found QPRs can be found in the full version [2].

Workload	#All	#QPR	%QPR	$T(P^b)$	$T(P^a)$	%Impr
TPC-H	1,156	36	3.11%	3.32h	1.06h	68.22%
DSB	543,198	13,784	2.54%	406h	289h	28.98%
JOB	189,320	55,155	29.13%	188h	344h	-83.12%
STATS	38,773	2,743	7.07%	85.73h	87.12h	-1.62%
Real-DY	143,255	649	0.45%	5562h	3230h	41.93%
Real-LO	12,779	1,429	11.18%	71.15h	18.57h	73.91%
Real-MS	199,415	9,592	4.81%	594h	273h	53.92%
Real-RE	2,704	0	0.00%	176h	171h	2.60%
Real-DW	17,916	2,348	13.11%	113h	107h	5.14%
Real-ED	875	20	2.29%	40.61h	18.31h	54.90%
Real-MP	9,583	945	9.86%	12.29h	7.20h	41.38%
Real-SE	9,224	261	2.83%	17.65h	6.14h	65.19%
Real-RM	55	0	0.00%	4.81h	4.01h	16.79%
Real-SA	6	4	66.67%	0.04h	0.06h	-38.91%
Total	**1.17m**	**86,966**	**7.44%**	**7277h**	**4559h**	**37.35%**

Table 5: QPRs emerging in evolutionary index tuning.

Category	ID	Description
(c1)	**RP-1a**	Expensive NLJ due to new inner index seek
	RP-1b	Expensive NLJ due to reduced estimated cost
(c2)	**RP-2**	Missing critical aggregation pushdown
	RP-3	Missing critical bitmap filter pushdown

Table 6: Taxonomy of regression patterns found.

3 Regression Pattern Analysis

We analyze QPRs using the data generated by one-shot tuning and incremental tuning. The goal of this investigation is to understand the root causes of QPRs and *whether there are recurring, ubiquitous patterns across the databases and workloads.* Table 6 presents a taxonomy of the regression patterns that we found for the QPRs.

3.1 Taxonomy of Regression Patterns

We categorize the QPRs into two categories: (c1) QPRs due to problematic change of access path between P^b and P^a, and (c2) QPRs due to missing critical optimizations that were present in P^b.

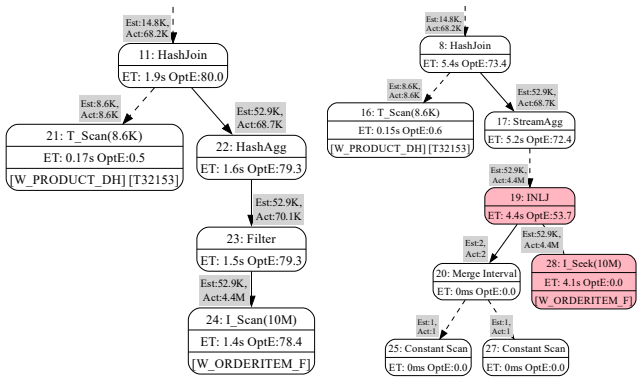
3.1.1 Problematic Change of Access Path (c1). By "change of access path", we mean one of the following situations: (1) a table access operator (e.g., a table scan, an index scan, or an index seek) in P^b has been changed in P^a; (2) the same table access operator is used but its usage pattern is changed between P^b and P^a, e.g., it serves as the inner child of a nested-loop join in P^a instead of a hash join in P^b; or (3) both a table access operator and its usage pattern are changed. There is a significant number of QPRs whose root causes can be attributed to some problematic change of access path. We see two primary patterns for QPRs that fall into this category:

- (RP-1a) P^a introduces a new expensive nested-loop join (NLJ) due to a new index seek that serves as its right/inner child;
- (RP-1b) P^a introduces a new expensive NLJ due to its reduced estimated cost by the query optimizer.

We next present examples of these regression patterns.

Example 1 (RP-1a). *Figure 2 presents an example of RP-1a. The QPR comes from the query Q-3 of Real-LO with one-shot index tuning. The "before plan" P^b does not contain any NLJ. The "after plan" P^a introduces the node 19, which is an index-based NLJ that becomes the bottleneck of query execution. Its right/inner input has huge CE error (estimated 52.9K vs. actual 4.4M rows). The NLJ is introduced due to the availability of a new inner index seek (i.e., the node 28) in P^a.*

From Example 1, QPRs come with not only change of access path but also CE errors. Intuitively, the introduction of new indexes



(a) **Before plan P^b** (b) **After plan P^a**

Figure 2: Illustration of regression pattern RP-1a. [Annotation of each operator node in a query plan tree: (1) *Est*, estimated cardinality; (2)*Act*, actual cardinality; (3) *ET*, execution time; (4) *OptE*, query optimizer's estimated cost.]

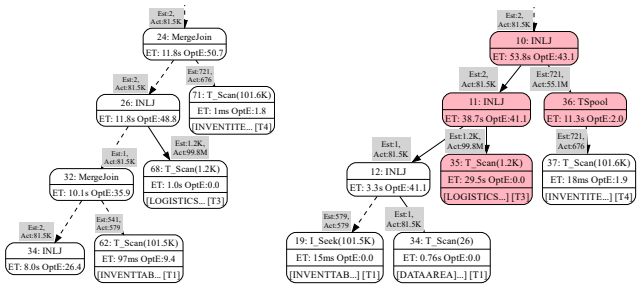
should *not* affect cardinality estimation. However, since our configuration may contain new statistics as well, they may have impact on cardinality estimation. It then raises an interesting question: *Is the QPR caused by only the new statistics, only the new indexes, or* both *the new statistics and new indexes?* To better understand this, we propose the following ablation study:

Procedure 1 (Ablation Study). *Let $C = (\mathcal{I}, \mathcal{S})$ be the configuration that results in the regressed plan P^a of a query q, where $\mathcal{I}$ represents the new indexes and $\mathcal{S}$ represents the new statistics. We only create the new statistics $\mathcal{S}$ without the new indexes $\mathcal{I}$ and let the query optimizer re-optimize the query q. We call the plan returned by the query optimizer the* intermediate plan *and denote it by $\tilde{P}^b$.*

For Example 1, we observed $\tilde{P}^b$ returned by the ablation study is *very different* from either the "before plan" P^b or the "after plan" P^a. Indeed, $\tilde{P}^b$ is even slower than P^a with a different nested-loop join as the bottleneck of query execution. P^a improves over $\tilde{P}^b$ by removing that more problematic nested-loop join and utilizing a recommended index, though it remains much slower than P^b. Since $\tilde{P}^b$ is indeed the internal view of P^b seen by some index tuners (e.g., DTA), such index tuners would think of P^a as an improvement over $\tilde{P}^b$ and therefore, incorrectly, recommend the corresponding indexes (and statistics). This example demonstrates that the introduction of new statistics can have significant impact on query optimizer's cardinality estimation and therefore plan choice as well.

Example 2 (RP-1b). *Figure 3 presents an example of RP-1b. The QPR comes from the query Q-3 of Real-DY with incremental index tuning. One bottleneck of the "after plan" P^a is the node 10 that represents an NLJ, which is much slower than the corresponding (logically equivalent) merge join (i.e., the node 24) in the "before plan" P^b. The execution time of the two operators is 53.8s and 11.8s.*

Unlike Example 1, the bottleneck NLJs in P^a (nodes 10 and 11) are not introduced due to the availability of any new inner index seek—the inner side of the join remains the same table scan in both P^b and P^a. However, the outer side of node 12 in P^a contains a new index seek (node 19) for a table that was accessed using table scan in P^b (node 62). This new index seek indirectly leads the query optimizer to introduce NLJ for node 10 in P^a since the optimizer



(a) **Before plan P^b** (b) **After plan P^a**

Figure 3: Illustration of regression pattern RP-1b.

estimated cost (i.e., 43.1) is lower than that of corresponding plan subtree rooted at node 24 in P^b (i.e., 50.7). P^a is significantly slower due to introduction of spool operator in the inner side of node 10, that creates bottleneck for pipelined execution of both NLJs (nodes 10 and 11) due to huge underestimation in the number of rebinds. Observe that node 35 in P^a is significantly slower compared to corresponding node 68 in P^b despite the same access path because P^a uses a single thread compared to 40 threads used by P^b in Figure 3(a). This change from parallel to serial execution is a side effect of change in cost estimates in the two cases.

To separate the impact of the new statistics and the new indexes, we repeated the ablation study in Procedure 1. Interestingly, in this case $\tilde{P}^b$ remains the same as P^b. This means that, even though the new statistics can affect cardinality estimation, the impact is not significant enough to change the decision made by the query optimizer. As a result, QPR would not have occurred if we only brought in the new statistics but not the new indexes.

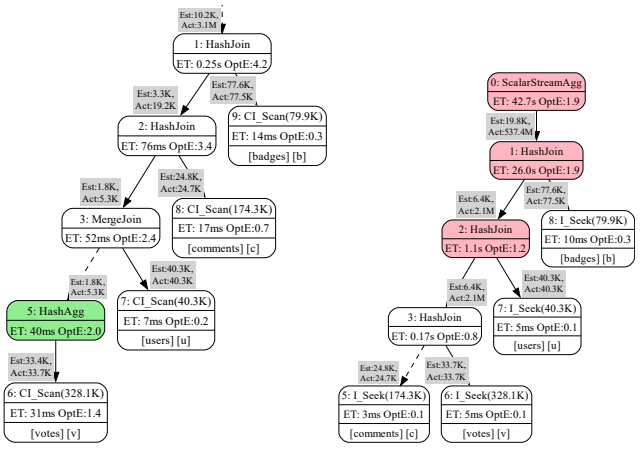
3.1.2 Missing Critical Optimizations (c2). Unlike the previous category, QPRs that fall into this category do not suffer from change in access path selection. That is, the access paths of P^a may have changed compared to P^b, but these changes are not the root causes of the QPRs. Rather, some critical optimizations that were present in P^b appeared to be missing in P^a. Again, we observe two major patterns for such QPRs:

- (RP-2) P^a misses a critical aggregation pushdown in P^b;
- (RP-3) P^a misses a critical bitmap filter pushdown in P^b.

Below, we again present examples of these regression patterns.

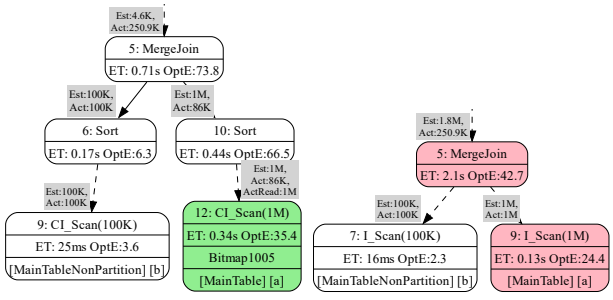
Example 3 (RP-2). *Figure 4 presents an example of RP-2. The QPR comes from the query Q-106 of STATS with one-shot index tuning. The bottleneck of the "after plan" P^a is the node 0 that represents an aggregation operator. This aggregation is much faster in the "before plan" P^b, thanks to the aggregation pushdown introduced by the node 5. The main cause of this bad decision made by the query optimizer on eliminating the aggregation pushdown is the CE errors at the join nodes 3 (6.4K estimated vs. 2.1M actual, i.e., 328× underestimation), 2 (6.4K estimated vs. 2.1M actual, i.e., 328× underestimation), and 1 (19.8K estimated vs. 0.5B actual, i.e., 25,252× underestimation) in P^a. While cardinality underestimation does present in P^b as well, it is at a much smaller scale. As a result, the amplified cardinality underestimation made the query optimizer think that the aggregation operator is cheap enough and is not worth a pushdown.*

One may ask why cardinality underestimation is amplified in the "after plan" P^a for the aggregation operator. We attribute this to the new statistics brought in by the new indexes being recommended,



(a) **Before plan P^b** (b) **After plan P^a**

Figure 4: Illustration of regression pattern RP-2.



(a) **Before plan P^b** (b) **After plan P^a**

Figure 5: Illustration of regression pattern RP-3.

as the new indexes themselves should not impact cardinality estimation. We also notice that the join order and join operator choice of P^a are different from that of the "before plan" P^b, though they are not the performance bottleneck of P^a. Again, we further performed the ablation study in Procedure 1. Interestingly, it turns out that the "before plan" P^b again remains the choice of the query optimizer in this case (i.e., $\tilde{P}^b = P^b$). This suggests that the amplified cardinality underestimation itself does not result in QPR—the aggregation pushdown remains worthwhile. It is the new indexes that further reduced the estimated cost of P^a, which misled the query optimizer to change its decision on aggregation pushdown.

Example 4 (RP-3). *Figure 5 presents an example of RP-3. The QPR comes from the query Q-147 of Real-MP with one-shot index tuning. Modern query optimizers use bitmap filter pushdowns [32] in hash join or merge join to reduce the number of rows that need to be fetched from the inner side (of the join) that match the outer side (of the join). The decision of whether a bitmap filter should be pushed down is made by the query optimizer based on its estimated selectivity. The "before plan" P^b contains a bitmap filter pushdown on the clustered index scan over the table "*MainTable*," and the actual output cardinality is 86K after bitmap filter pushdown. This bitmap filter is missing in the "after plan" P^a, and the actual output cardinality goes up to 1.0M.*

Unlike the previous examples, we do not observe significant cardinality underestimation in the "after plan" P^a of Example 4. To the contrary, there is significant cardinality *overestimation* in P^a, which suggests that the actual cost of P^a should be even less.

Interestingly, there is cardinality underestimation on the merge join (node 5) of the "before plan" P^b (i.e., 4.6K estimated output rows vs. 250.9K actual rows), and the introduction of the new statistics helps "fix" it; however, this fix goes too far that ends up with significant cardinality overestimation on the same merge join (i.e., 1.8M estimated output rows vs. 250.9K actual rows). As a result, creating (and pushing down) a bitmap filter based on a much higher estimated selectivity/cardinality is not attractive. To validate this, we further preformed the ablation study in Procedure 1. The $\tilde{P}^b$ turned out to be a "transitioning plan"—its only difference from P^b is the removal of that bitmap filter. This confirms that the missed bitmap filter pushdown optimization is indeed caused by the cardinality overestimation due to the introduction of the new statistics.

3.2 Summary and Discussion

We have the following observations based on our analysis.

Observation 1. *Most of the significant QPRs can be attributed to some regression pattern that is simple and easy to understand.*

Although our list of regression patterns in Table 6 is by no means exhaustive, it covers most of the significant QPRs observed in our data. Tables 7 and 8 further present the breakdowns of QPRs covered by individual regression patterns across the workloads, where we use RP-1 to refer to the regression pattern RP-1a or RP-1b, as they both characterize the existence of an expensive NLJ.

Observation 2. *Regression patterns typically characterize some "local change" or "local transformation" in the plan structure.*

For example, RP-1 (including both RP-1a and RP-1b) asserts the presence of a new expensive nested-loop join. RP-2 asserts the decision of pushing down an aggregation or not; similarly, RP-3 asserts the decision of pushing down a bitmap filter or not. Once a regression pattern has been detected, it is straightforward to reverse the harmful change indicated by the pattern. For example, if RP-1 is detected, we may *hint* the query optimizer [34] to not use the problematic index. On the other hand, if RP-2 or RP-3 is detected, we can simply force pushing down the corresponding aggregation or bitmap filter that is critical to the query performance, by using mechanisms such as *plan forcing* [29]. Although in this paper we do not explore potential ways of correcting QPR once some regression pattern is detected, it is an interesting direction for future work.

Observation 3. *The impact on cardinality estimation due to the introduction of new statistics can be significant enough to change the optimization decision made by the query optimizer.*

This observation is affirmed by the ablation study in Procedure 1 that highlights the impact of the new statistics. It has two possible outcomes: (1) the "intermediate plan" $\tilde{P}^b$ remains the same as the "before plan" P^b; and (2) $\tilde{P}^b$ is different from P^b. If $\tilde{P}^b = P^b$, it implies that the new statistics do not change the plan returned by the query optimizer, even if the new statistics may have impacted cardinality estimation. On the other hand, if $\tilde{P}^b \neq P^b$, the impact on cardinality estimation is significant enough to change the query optimizer's plan choice. We have seen QPR examples of both cardinality underestimation and cardinality overestimation with the new statistics. While it is intuitive that cardinality underestimation can result in QPRs, the QPRs due to cardinality overestimation are subtle (e.g., Example 4). Nevertheless, the implication here is that a regression pattern needs to account for *not only change of access paths (due to availability of new indexes) but also cardinality estimation errors (due to availability of new statistics).*

Pattern	Workload	#QPR	$T(P^b)$	$T(P^a)$
RP-1	Real-LO	3	13.08s	31.47s
RP-1	Real-MP	1	1.55s	13.85s
RP-2	STATS	1	0.53s	49.40s
RP-2	Real-RM	1	83.21s	124.33s
RP-3	Real-MP	1	1.24s	3.72s
RP-3	Real-DY	1	63.02s	128.59s

Table 7: Regression patterns in one-shot index tuning.

Observation 4. *The majority of the* significant *QPRs are attributed to the regression pattern RP-1 (including both RP-1a and RP-1b), namely, the emergence of a new expensive nested-loop join in the regressed query plan.*

This observation is evident from Tables 7 and 8, where RP-1 accounts for 23 of the QPRs while the other patterns account for 5 QPRs in total. Moreover, we further looked into the degree of QPRs in terms of their actual execution time, and we found that the QPRs due to RP-1 are *much more significant* compared to the others. Therefore, in the rest of this paper we focus on addressing QPRs that can be accounted for by RP-1. The popularity of RP-1 QPRs in the context of index tuning is not a coincidence, as it is attractive for the query optimizer to choose a nested-loop join in the presence of new indexes. Nested-loop join is powerful for accelerating query execution when there is indeed only a small number of rows that need to be fetched via index seeks. However, it becomes a risky choice in the presence of significant *cardinality underestimation.*

4 Pattern-based QPR Detector

We present a pattern-based QPR detector, based on Observation 4, namely, the majority of the significant QPRs in index tuning can be attributed to the emergence of new expensive NLJs. Although this detector is dedicated to detecting QPRs with new expensive NLJs, its underlying design principles can be applied to develop QPR detectors for other regression patterns as well (Section 4.4).

We start with a more formal characterization of such expensive NLJs (Section 4.1). We then develop an algorithmic framework to detect expensive NLJ in an *automated* manner (Sections 4.2 and 4.3).

4.1 Characterization of Expensive NLJ

Observation 4 itself is far from actionable for QPR detection in practice. Indeed, a naive solution here could be to forbid the use of nested-loop joins. However, this will forfeit most of the benefits brought in by index tuning as well. Clearly, not all nested-loop joins are harmful, and the challenge is to identify which ones are problematic or risky *without executing the "after plan"* P^a.

To estimate the *expensiveness* of a nested-loop join, we define two metrics, *local expensiveness* and *global expensiveness*, as follows.

Definition 1 (Local Expensiveness). *Let J be a nested-loop join contained by the "after plan" P^a in QPR detection. The local expensiveness of J is defined as $l(J, P^a) = \frac{\text{cost}(J)}{\text{cost}(P^a)}$, where* cost$(J)$ *represents the estimated cost of the plan* subtree *under the join J.*

A nested-loop join J is *locally expensive* if $l(J, P^a) > \tau_l$, where $0 \leq \tau_l \leq 1$ is some threshold. Local expensiveness characterizes how significant the execution cost of a nested-loop join is *inside*

Pattern	Workload	#QPR	$T(P^b)$	$T(P^a)$
RP-1	Real-DY	8	495.67s	11824.84s
RP-1	Real-ED	1	2.85s	5.77s
RP-1	JOB	2	4.23s	12.36s
RP-1	Real-LO	3	13.32s	33.67s
RP-1	Real-RE	3	1.32s	14.63s
RP-1	STATS	1	0.37s	3.34s
RP-1	Real-MP	1	4.00s	25.80s
RP-2	STATS	1	0.53s	39.57s

Table 8: Regression patterns in incremental index tuning.

the query plan. Ideally, one should use the *actual* execution time instead of query optimizer's estimated cost. Unfortunately, this is impossible in practice because the execution time of the "after plan" P^a is unknown when QPR detection needs to be performed. Thus, local expensiveness can be inaccurate. For example, we may miss a locally expensive NLJ J if cost(J) is underestimated and a relatively expensive operation follows. However, in general, we would expect a bottom-up propagation of cost estimation errors [17]. That is, if cost(J) is underestimated, then the costs of higher-level operations are likely underestimated too. If so, the ratio between cost(J) and cost(P^a), i.e., the local expensiveness of J, will be relatively stable.

Definition 2 (Global Expensiveness). *Let J be a locally expensive nested-loop join, and let q be the corresponding query in the workload W where J comes from. Let $t^b(q)$ be the execution time of the "before plan" P^b of q, which is presumably available before QPR detection starts. The global expensiveness of J is defined as* $g(J, q) = percentile(t^b(q), \{t^b(q')\}_{q' \in W})$.

A nested-loop join J is *globally expensive* if the corresponding query q satisfies $g(J, q) > \tau_g$, where $0 \leq \tau_g \leq 1$ is some threshold. Intuitively, global expensiveness measures the relative execution cost of a query *at workload level*. Specifically, $t^b(q')$ means the execution time of the "before plan" of q', which is presumed available when performing QPR detection. $\{t^b(q')\}$ represents the *distribution* of the "before plan" execution time w.r.t. all queries of a workload W. Essentially, we use the percentile of $t^b(q)$ in this "before plan" execution time distribution as our definition of the global expensiveness of q. On the other hand, it is possible that a query q is globally expensive but actually not so under the new configuration, i.e., when considering the distribution of the "after plan" execution time of all workload queries. Unfortunately, this latter "after plan" execution time distribution is unknown when performing QPR detection. This is indeed a limitation of our current definition of global expensiveness, which we leave for future work.

4.2 Regression Pattern by Expensive NLJ

We define the regression pattern based on expensive NLJ as $J = \mathcal{P}(P^b, P^a)$ over a pair of "before plan" P^b and "after plan" P^a, with respect to a given local expensiveness threshold τ_l and a given global expensiveness threshold τ_g:

(1) The nested-loop join J appears in P^a but not in P^b;
(2) The nested-loop join J is *both* locally and globally expensive;
(3) The right/inner side of the nested-loop join is a table access operator (with perhaps filters but no other operators, e.g., join, on top of it), i.e., it is a "left deep" nested-loop join.

If there are multiple expensive nested-loop joins in P^a, we will only return the "deepest" one in the plan tree (where the root node of the plan tree receives a depth of zero).

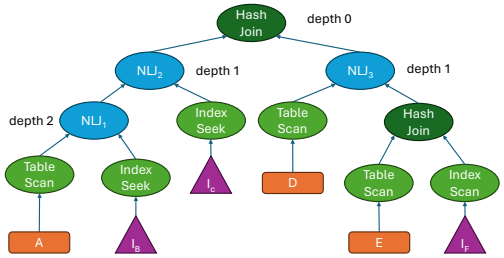


Figure 6: Illustration of the expensive NLJ pattern.

Example 5 (Expensive NLJ Pattern). *Figure 6 presents an example query plan that contains three nested-loop joins NLJ_1, NLJ_2, and NLJ_3. Suppose that all of them pass the local and global expensiveness thresholds. NLJ_3 does not match the expensive NLJ pattern because it is not "left deep." Both NLJ_1 and NLJ_2 are "left deep," but only NLJ_1 matches the pattern as it is the "deepest" one in the query plan.*

We choose to focus on "left deep" nested-loop join following the observations on the simplicity (i.e., Observation 1) and locality (i.e., Observation 2) of regression patterns. Compared to more complicated "bushy" nested-loop join (e.g., NLJ_3 in Example 5), "left deep" nested-loop join is easier to define and detect. The impact of an index is also more direct on "left deep" nested-loop join, which makes it easier to understand and correct the corresponding QPR with remediation actions. Moreover, we choose to focus on the *deepest* expensive "left deep" nested-loop join if there are multiple candidates, because the (local) expensiveness of higher-level joins may be a consequence of expensive joins below.

Algorithm 1 presents the details of automating the process of matching the expensive NLJ pattern in a given plan pair (P^b, P^a) of a query q. We start by looking for all nested-loop joins that appear in the plan P^a (line 2). For each of the nested-loop joins J found, we simply check whether (1) J is "left deep," (2) J does not appear in the plan P^b, and (3) J is expensive; if so, we only keep the one with the maximum depth (lines 3 to 9).

4.3 QPR Detection Algorithm

Our QPR detection algorithm based on the expensive NLJ pattern can be broken down into three major steps: (1) *match* the expensive NLJ pattern using Algorithm 1; (2) *measure* the degree of potential QPR based on the notion of *cost inflation factors*; and (3) *recost* the "before plan" and "after plan" using the cost inflation factors and *predict* QPR based on the recomputed plan costs.

4.3.1 Cost Inflation Factors. Formally, let J be an expensive nested-loop join operator found in the "after plan" P^a, and let O_l and O_r be its left/outer and right/inner input operators. Moreover, let O_l^m and O_r^m be the corresponding *match* (i.e., logically equivalent operator) of O_l and O_r in the "before plan" P^b, respectively.

Definition 3 (Cost Inflation Factors). *The cost inflation factors of the left and right inputs of J are defined as*

$$f_l = \max\{\frac{ActCard(O_l^m)}{EstCard(O_l^m)}, 1\} \text{ and } f_r = \max\{\frac{ActCard(O_r^m)}{EstCard(O_r^m)}, 1\}.$$

Here, *EstCard* and *ActCard* represent the *estimated* and *actual* cardinality, respectively. Intuitively, cost inflation factors measure, in an approximate way, the impact of *cardinality underestimation*

Algorithm 1: MatchExpensiveNLJ(q, P^b, P^a).

Input: (P^b, P^a), a pair of plans to detect QPR; q, the corresponding query in the workload W; τ_l, the threshold for local expensiveness; τ_g, the threshold for global expensiveness.
Output: J, the expensive nested-loop join found.

```
1  J ← null;
2  𝒥 ← GetAllNLJs(P^a);
3  foreach nested-loop join J' ∈ 𝒥 do
4      if J' is "left deep" and J' ∉ P^b then
5          l(J', P^a) ← cost(J') / cost(P^a);
6          g(J', q) ← percentile(t^b(q), {t^b(q')}_{q'∈W});
7          if l(J', P^a) > τ_l and g(J', q) > τ_g then
8              if depth(J') > depth(J) then
9                  J ← J';
10 return J;
```

Algorithm 2: ComputeCostInflationFactors(J, P^b, P^a).

Input: (P^b, P^a), the plan pair; J, the expensive NLJ found in P^a.
Output: f_l, the cost inflation factor of the left/outer input of J; f_r, the cost inflation factor of the right/inner input of J.

```
1  O_l ← LeftChild(J), O_r ← RightChild(J);
2  f_l ← 1, f_r ← 1;
3  O_l^m ← Match(O_l, P^b), O_r^m ← Match(O_r, P^b);
4  if O_l^m is not null then
5      f_l ← max{ ActCard(O_l^m) / EstCard(O_l^m), 1};
6  if O_r^m is not null then
7      f_r ← max{ ActCard(O_r^m) / EstCard(O_r^m), 1};
8  return f_l, f_r;
```

on the execution cost of the join. Moreover, once again we assume that we have obtained execution information (in particular, true cardinality information) of the "before plan" P^b.

4.3.2 Recosting of the Join and the Plan. The presence of cardinality underestimation makes it necessary to recompute the costs of the "before plan" and the "after plan" to reevaluate the likelihood of QPR. We conduct this *recosting* process [13, 55, 56] based on the cost inflation factors obtained by Algorithm 2.

Specifically, Algorithm 3 presents the details of plan recosting, which employs Algorithm 4 as a subroutine to recost the join operators. We start by seeking a match (i.e., logically equivalent operator) of the expensive NLJ operator J in P^b (line 2). If such a match J' is found, we call Algorithm 4 to recost J' based on its own cost inflation factors f_l' and f_r', and we recompute the cost of P^b by replacing the old cost of J' with its new cost (lines 3 to 8). Similarly, we recost J based on the given cost inflation factors f_l and f_r and recompute the cost of P^a accordingly (lines 9 to 11).

The recosting of the join operators J and J', as illustrated in Algorithm 4, works as follows. We recompute the cost of a join based on its type. If it is a nested-loop join, we increase the cost of the right/inner side and the residual cost (i.e., the cost of the join operator itself excluding the costs of the left and right inputs) by a factor of $f_l \cdot f_r$, while keeping the cost of the left/outer side unchanged (line 3). This is easy to understand, as the cost inflation factors quantify the degree of cardinality underestimation on the left and right inputs of the join. Therefore, for each iteration of the nested-loop join, the cost of the inner side is roughly increased by a factor of f_r. On the other hand, the number of iterations is boosted by a factor of f_l. This justifies the recosting formula of the nested-loop join. Meanwhile, for other types of join, such as hash join or merge join, that do not require multiple accesses of the right/inner side, we increase only the residual cost of the join by a factor of $f_l \cdot f_r$ but not the cost of the right/inner side.

4.3.3 Putting It Together. Algorithm 5 presents the details of the pattern-based QPR detection algorithm. We start by matching the expensive NLJ pattern using Algorithm 1 (line 1). We report no QPR if we fail to find any expensive NLJ (lines 2 to 3). Otherwise, we compute the cost inflation factors using Algorithm 2. We again report no QPR if there is no cardinality underestimation, i.e., $f_l \leq 1$ and $f_r \leq 1$ (lines 4 to 6). Otherwise, we recost both plans P^b and P^a with the cost inflation factors, using Algorithm 3, and we report QPR if recost(P^a) > recost(P^b) (lines 7 to 12).

Discussion. The QPR detection algorithm in Algorithm 5 is limited by the fact that it relies on finding matches in the "before plan" P^b. It can result in both *false positives* and *false negatives*:

- (*False Positives*) Consider a case where we have either $f_l > 1$ or $f_r > 1$ but we cannot find a match for J in P^b. As a result, the cost of P^a is increased after plan recosting, whereas P^b cannot be recosted by Algorithm 3 and thus cost(P^b) remains the same. However, it is likely that cost(P^b) should have been increased, too, as the existence of cardinality underestimation in P^a suggests that there may be cardinality underestimation in P^b as well. This possibility is currently ignored by Algorithm 5. Consequently, if Algorithm 5 reports QPR in this case, it may be a *false positive* due to the potential underestimation of cost(P^b).
- (*False Negatives*) Consider another case where we cannot find a match for either O_l or O_r. As a result, we may miss potential cardinality underestimation and therefore the recomputed cost of J may be less than it should have been. When this happens, if we can find a match J' in P^b for J, then it creates an unfair situation as we can use actual cardinality for recosting J' but not J. Therefore, we may make the cost of J' (and thus the plan P^b) higher but not the cost of J (and thus the plan P^a). Consequently, if Algorithm 5 reports no QPR in this case, it may be a *false negative* due to the potential underestimation of cost(P^a).

4.4 Other Regression Patterns

While it is not our goal in this paper to provide a comprehensive list of regression patterns and their corresponding pattern-based QPR detectors, the principles and techniques developed can be applied to the development of QPR detectors based on regression patterns other than RP-1. For example, a QPR detector based on RP-2 or RP-3 would be to monitor any aggregation or bitmap filter pushdowns that were present in P^b but missing in P^a, while also considering the degree of cardinality estimation (CE) errors.

Although this case-by-case approach is effective for the QPR patterns covered in the present study, a more general approach remains interesting. There are two basic elements in such a general approach: (1) specification of the structural change between the "before plan" and the "after plan" and (2) quantification of the CE error. From this point of view, we can retain the skeleton of Algorithm 5 and only replace the three function calls `MatchExpensiveNLJ()`,

Algorithm 3: RecostPlan(J, P^b, P^a).

Input: (P^b, P^a), the plan pair; J, the expensive NLJ operator.
Output: recost(P^b), the recomputed cost of plan P^b; recost(P^a), the recomputed cost of plan P^a.

recost(P^b) ← cost(P^b), recost(P^a) ← cost(P^a);
J' ← Match(J, P^b);
if J' *is not null* **then**
  O'_l ← LeftChild(J'), O'_r ← RightChild(J');
  f'_l ← max{$\frac{\text{ActCard}(O'_l)}{\text{EstCard}(O'_l)}$, 1}, f'_r ← max{$\frac{\text{ActCard}(O'_r)}{\text{EstCard}(O'_r)}$, 1};
  recost(J') ← RecostJoin(J', f'_l, f'_r);
  residual(P^b) ← cost(P^b) − cost(J');
  recost(P^b) ← residual(P^b) + recost(J');
recost(J) ← RecostJoin(J, f_l, f_r);
residual(P^a) ← cost(P^a) − cost(J);
recost(P^a) ← residual(P^a) + recost(J);
return recost(P^b), recost(P^a);

`ComputeCostInflationFactors()`, and `RecostPlan()` with implementations customized for detecting different QPR patterns.

5 Experimental Evaluation

We evaluate the pattern-based QPR detector proposed in Section 4 and report the experimental evaluation results in this section.

5.1 Experiment Settings

We focus on detection of significant QPRs emerging in one-shot, incremental, and evolutionary tuning by setting the regression threshold $\tau = 0.5$ (i.e., 50% QPR).

5.1.1 Evaluation Metrics. We use the following metrics to evaluate the performance of a QPR detector.

The first set of metrics are standard based on the viewpoint of treating QPR detection as a binary classification problem: (1) *precision*, (2) *recall*, (3) *accuracy*, and (4) *F1 score*.

The second set of metrics are to address the limitation of the binary classification view of QPR detection, as some QPRs can be much worse than the others: (1) *time of the "before plan"* P^b, (2) *time of the "after plan"* P^a, (3) *time of* P_{pred}, and (4) *time of* P_{best}. Here, P_{pred} is the plan chosen based on the output of the QPR detector h. That is, $P_{\text{pred}} = P^b$ if h predicts a QPR, and $P_{\text{pred}} = P^a$ otherwise. P_{best} is the plan chosen based on the output of an oracle (i.e., a perfect QPR detector) that always makes the right prediction. That is, $P_{\text{best}} = P^b$ if $t^b < t^a$, where t^b and t^a are the execution time of P^b and P^a respectively, and $P_{\text{best}} = P^a$ otherwise.

5.1.2 QPR Detectors. We evaluate the pattern-based QPR detector proposed in Section 4, as well as three state-of-the-art ML-based QPR detectors: (1) *AI meets AI* (AMA) [11], (2) *TreeCNN* (TCNN) [27], and (3) *QueryFormer* (QF) [60].

5.2 ML-based QPR Detection

The main difference between the three ML-based QPR detectors AMA, TCNN, and QF lies in the *feature representation of a query plan*. Specifically, AMA carefully selects features that are important for characterizing the execution profiles of individual operators in the query plan (e.g., estimated number of input and output rows, estimated number of input and output bytes, estimated execution cost, etc.). Such operator-level features are further aggregated w.r.t. the plan tree structure to form a vector representation of the query plan. On the other hand, both TCNN and QF adopt more advanced technologies to encode a query plan into its vector representation. In more detail, TCNN leverages *tree convolution* [35] that adapts the well-known *convolutional neural network* (CNN) [26] to work for tree-structured data, whereas QF leverages *tree transformer* that adapts the well-known *transformer* architecture [48] to encode query plan tree. Therefore, we can use a uniform framework to evaluate all these three ML-based QPR detectors.

Algorithm 4: RecostJoin(J, f_l, f_r).

Input: J, the join operator; f_l, the cost inflation factor of the left/outer child of J; f_r, the cost inflation factor of the right/inner child of J.
Output: recost(J), the new cost of J.

residual(J) ← cost(J) − outerChildCost(J) − innerChildCost(J);
if J *is nested-loop join* **then**
  recost(J) ← outerChildCost(J) + $f_l \cdot f_r \cdot$ innerChildCost(J) + $f_l \cdot f_r \cdot$ residual(J);
else
  recost(J) ← outerChildCost(J) + innerChildCost(J) + $f_l \cdot f_r \cdot$ residual(J);
return recost(J);

Given a pair of plans (P^b, P^a), we first convert P^b and P^a into their feature vectors $\vec{\mathbf{P}}^b$ and $\vec{\mathbf{P}}^a$ using the plan encoder provided by the corresponding ML-based QPR detector. Following [11], we then take the difference $\vec{\mathbf{x}} = \vec{\mathbf{P}}^a - \vec{\mathbf{P}}^b$ as the input to train a binary classifier h as the QPR detector. For fair comparison, we use the same classifier h for AMA, TCNN, and QF plan representations. Specifically, h is a 4-layer fully-connected deep neural network, where each hidden layer contains 64 neurons and uses `ReLU` as the activation function. A similar architecture has been used in [60].

5.2.1 Implementation and Evaluation Setups. We implement AMA, TCNN, and QF using PyTorch, and we use an Nvidia RTX A6000 GPU for model training and inference. For model training, we use the Adam optimizer [19] with 100 epochs and batch size of 32.

We use a "leave one out" setup for evaluating the ML-based technologies [11]. Specifically, let $\mathcal{W}$ be the set of all workloads. For each workload $W \in \mathcal{W}$, we use all index tuning data collected for the other workloads $\mathcal{W}_{-W} = \mathcal{W} - \{W\}$ to train an ML model $\mathcal{M}$ and test it using the index tuning data collected for W.

5.2.2 Results. Figure 7(a) and 7(b) present results on the one-shot and incremental index tuning data in terms of prediction accuracy of the ML-based QPR detectors. We were not able to finish training TCNN and QF within reasonable time (i.e., 48 hours) on the evolutionary index tuning dataset. We observe that TCNN and QF perform better than AMA in terms of the "accuracy" metric. However, it does not suggest that TCNN and QF are more effective binary classifiers, because their F1 scores are much lower than that of AMA. In fact, in almost all cases the F1 scores of TCNN and QF are zero, which means that they are not able to capture any QPR. In other words, they behave the same as a degenerated QPR predictor that simply says there is no QPR. Overall, all three ML-based QPR detectors show unsatisfactory performance. There are several potential reasons for this observation. First, the one-shot and incremental index tuning datasets are relatively small and therefore sophisticated plan encodings such as TCNN and QF are perhaps not worthwhile and more likely to overfit. Second, the fact that QPR

Algorithm 5: Pattern-based QPR detection.

Input: (P^b, P^a), the pair of plans; q, the corresponding query.
Output: *true*, if (P^b, P^a) is a QPR; *false*, otherwise.

```
J ← MatchExpensiveNLJ(q, P^b, P^a);
if J is null then
    return false;
f_l, f_r ← ComputeCostInflationFactors(J, P^b, P^a);
if f_l ≤ 1 and f_r ≤ 1 then
    return false;
// We have either f_l > 1 or f_r > 1;
recost(P^b), recost(P^a) ← RecostPlan(J, P^b, P^a);
if recost(P^a) > recost(P^b) then
    return true;
else
    return false;
```

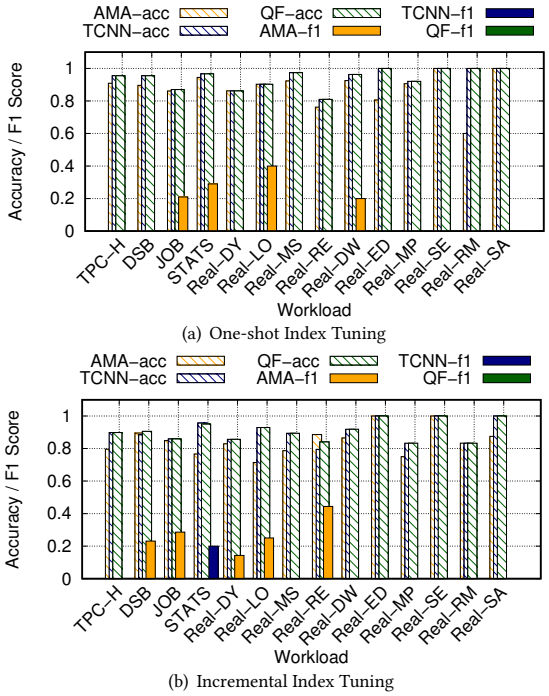
is a relatively infrequent event makes the classification problem more challenging (e.g., a naive classifier such as the degenerated one can achieve high accuracy but fail miserably in terms of F1 score). This is related to the well-known "learning from imbalanced data" challenge in the literature [21]. Indeed, we have tried to "re-balance" the data by giving the regressed cases higher weights in the loss function when training the ML-based classifiers but we still see underwhelming results as shown in Figures 7(a) and 7(b). Third, the "leave one out" setup is arguably the worst-case scenario for ML-based classifiers, as the training and test datasets may not follow the same distribution. Indeed, our results here resonate with the observations in [11], where the AMA classifier shows similar results to the degenerated classifier under the "leave one out" setup.

5.3 Pattern-based QPR Detection

We now evaluate the pattern-based QPR detector proposed in Section 4. We set the local expensiveness threshold $\tau_l = 0.1$ and the global expensiveness threshold $\tau_g = 0.1$ in our evaluation, which are the default settings of the pattern-based QPR detector. We use AMA as the baseline of ML-based detectors to compare with.

5.3.1 One-shot Index Tuning. Figure 8(a) presents the (percentage) improvement of the total execution time by using the plan suggested by the QPR detector, i.e., P_{pred}, w.r.t. to the plan over the existing configuration, i.e., P^b, for one-shot index tuning. We observe that the pattern-based QPR detector significantly outperforms AMA on the workloads JOB, Real-RE, Real-ED, and Real-RM, while having similar performance on the other workloads (except Real-LO, where AMA outperforms the pattern-based QPR detector). In fact, the pattern-based QPR detector achieves similar performance to the best possible, i.e., P_{best}, on the workloads TPC-H, JOB, DSB, Real-MS, Real-ED, Real-MP, Real-SE, Real-RM, and Real-SA. For example, the improvements on JOB by the pattern-based QPR detector and AMA are 53% and 40%, where the best possible improvement is 59%. Meanwhile, the performance of using the plan suggested by AMA is often inferior to the default approach of always trusting the index tuner by using P^a, for example, on Real-ED and Real-RM.

Figure 9(a) further compares the pattern-based QPR detector with AMA in terms of the prediction/classification accuracy. We observe that the pattern-based QPR detector significantly outperforms AMA in terms of both accuracy and F1 score on the workloads JOB, STATS, DSB, Real-LO, Real-RE, Real-DW, Real-ED, and Real-RM. AMA has



(a) One-shot Index Tuning

(b) Incremental Index Tuning

Figure 7: Comparison of ML-based regression detectors in terms of prediction accuracy and F1 score.

an advantage only on Real-SA in terms of accuracy. Some F1 scores are zero (i.e., either precision or recall is zero) and thus not visible.

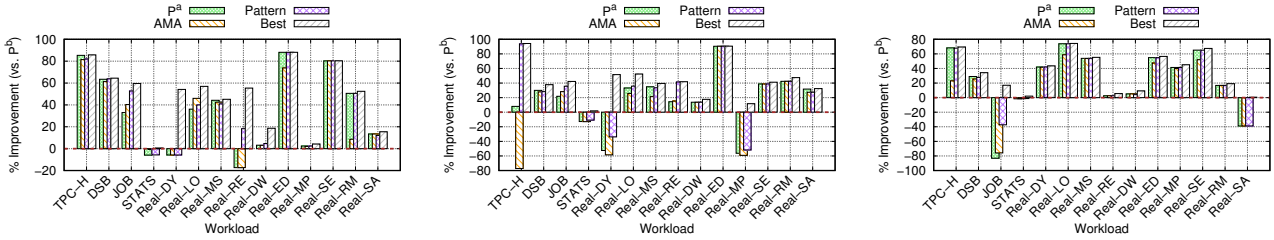
5.3.2 Incremental Index Tuning. Figures 8(b) and 9(b) compare the pattern-based QPR detector against AMA for incremental index tuning. Again, the pattern-based QPR detector significantly outperforms AMA on workloads such as TPC-H, JOB, Real-DY, Real-LO, Real-MS, Real-RE, and Real-MP. We also observe that AMA is sometimes even significantly worse than the default approach of using P^a (e.g., on TPC-H, Real-DY, Real-LO, and Real-MS). Meanwhile, the pattern-based QPR detector achieves the best possible on workloads such as TPC-H, Real-RE, and Real-ED. For example, the time improvements on Real-RE by the pattern-based QPR detector and AMA are 41.5% and 15.3%, where the best possible is 41.8%.

5.3.3 Evolutionary Index Tuning. Figures 8(c) and 9(c) further compare the two QPR detectors in the context of evolutionary index tuning. As shown in Figure 8(c), the pattern-based QPR detector significantly outperforms AMA on TPC-H, JOB, Real-LO, Real-ED, and Real-SE in terms of improved plan execution time, while their performances on the other workloads are similar. The improvement achieved by the pattern-based QPR detector is similar to the best possible on most workloads (except JOB). On the other hand, AMA remains inferior to the default approach of using P^a on workloads such as TPC-H, Real-LO, Real-ED, and Real-SE.

5.4 Analysis of Pattern-based QPR Detector

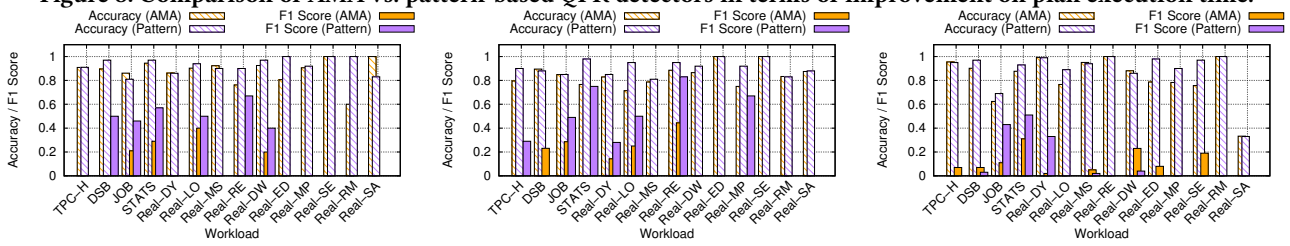
We further perform more detailed analysis of the pattern-based QPR detector to understand the impact of (1) the local and global expensiveness thresholds and (2) the cost inflation factors.

5.4.1 Local and Global Expensiveness Thresholds. We are interested in the potential of the pattern-based QPR detector by varying the local and global expensiveness thresholds. For this sake



(a) One-shot Index Tuning (b) Incremental Index Tuning (c) Evolutionary Index Tuning

Figure 8: Comparison of AMA vs. pattern-based QPR detectors in terms of improvement on plan execution time.



(a) One-shot Index Tuning (b) Incremental Index Tuning (c) Evolutionary Index Tuning

Figure 9: Comparison of AMA vs. pattern-based QPR detectors in terms of prediction accuracy and F1 score.

we study the *optimal* settings of the thresholds. Specifically, we perform a "grid search" in the space of $(\tau_l, \tau_g) \in \mathcal{L} \times \mathcal{G}$, where $\mathcal{L} = \mathcal{G} = \{0.1, 0.2, 0.5, 0.8, 0.9\}$. Figures 10(a), 10(b), and 10(c) present the results of the optimal thresholds for one-shot, incremental, and evolutionary index tuning, respectively. For one-shot tuning and incremental tuning, optimal thresholds only make a significant difference on Real-DY. On the other hand, for evolutionary tuning optimal thresholds only make a significant difference on JOB. It remains future work to explore ways of finding the optimal thresholds without an exhaustive search.

5.4.2 Cost Inflation Factors. Our way of computing cost inflation factors is *best-effort*: the cost inflation factor of the left/right input of the expensive nested-loop join remains 1 if we cannot find the corresponding match in the "before plan" P^b. To understand the impact of this limitation, we study a hypothetical case where we use the true left/right input cardinality to compute the cost inflation factor if we cannot find a match. The results of using the cost inflation factors based on true cardinality, in combination with using the optimal local and global thresholds, are presented as 'Opt-TC' in Figures 10(a), 10(b), and 10(c). We observe that leveraging true cardinality can further improve the pattern-based QPR detector in certain cases, e.g., on Real-DY for incremental tuning and JOB for evolutionary tuning. This suggests that one direction for further improvement of the pattern-based QPR detector is to improve the cardinality estimation for those operators with no match in P^b.

5.5 Other Evaluation Results

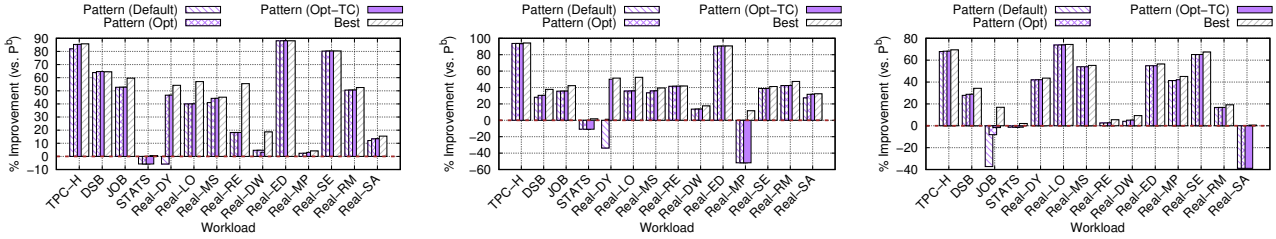
5.5.1 Generality of Regression Pattern. Some of the observations and results (in particular, the QPR pattern due to emergence of expensive NLJs) are not restricted to Microsoft SQL Server. First, NLJs are supported by almost all database systems. Second, the QPR pattern related to expensive NLJs also characterizes the roles of cardinality estimation (CE) errors, which are well-known general issues beyond a specific database system (e.g., see [24, 56] for studies of CE errors on top of PostgreSQL). To validate this, we create the same indexes recommended by DTA on top of PostgreSQL databases. We then check if the QPR pattern based on the emergence of new expensive NLJs occurs as well. We use PostgreSQL 17.4 running on a standard Azure D16s-v3 VM.

Figure 11 presents the validation results on the four benchmark workloads TPC-H, DSB, JOB, and STATS. Here, we compare the percentage of QPRs with new emerging expensive NLJs. We have two main observations. First, for most of the cases tested, around 60% to 100% of the QPRs contain new expensive NLJs. Second, this percentage coverage is consistent across PostgreSQL and Microsoft SQL Server, demonstrating the generality of the QPR pattern.

5.5.2 Decoupling Indexes from Statistics. Following our ablation study (Procedure 1) in Section 3.1, a "statistics only" scenario is itself interesting, as having extra data statistics available could, in theory, greatly improve the plans (without extra indexes). To shed some light on the sheer impact of new statistics, we extend our ablation study to all plan pairs collected from one-shot and incremental index tuning scenarios. Figures 12(a) and 12(b) present the time improvement by the "intermediate plan" $\tilde{P}^b$ over the "before plan" P^b. We also include the "after plan" P^a for comparison. Interestingly, it is not guaranteed that the availability of new statistics will result in better plans. Although $\tilde{P}^b$ indeed leads to significant improvements for some workloads (e.g., JOB), it also causes significant regressions for some other workloads (e.g., Real-LO). This raises the question of recommending statistics that can improve query execution *without causing regression*, which we leave for future work.

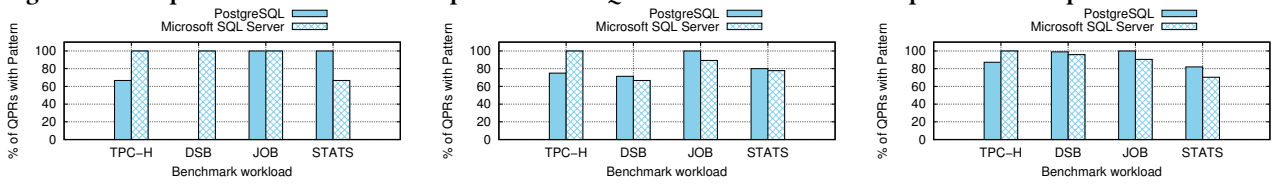
6 Related Work

Index Tuning. Much work has been devoted to index tuning in the past decades (see [44] for a recent survey). A classic setup is *offline* index tuning (e.g., [3–6, 9, 18, 20, 39, 43, 45, 47, 50–52, 57]), where the index tuner is given a static workload of queries and the goal is to come up with an index configuration that minimizes the workload execution cost (subject to certain constraints such as storage bound). Offline index tuners rely on the what-if query optimizer call to estimate the execution cost of a query given an index configuration, which can be inaccurate and result in QPR after the index configuration is materialized. On the other hand, there is also a prominent line of recent work towards *online* index tuning (e.g. [22, 37, 38, 40]), where the index tuner needs to deal with dynamic workloads with new queries coming from time to time. Online index tuning is a more challenging problem and existing work has been focusing on solutions using reinforcement



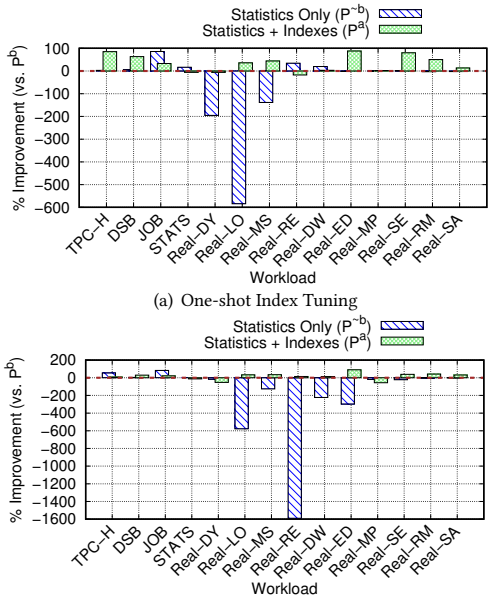
(a) One-shot Index Tuning (b) Incremental Index Tuning (c) Evolutionary Index Tuning

Figure 10: Comparison of variants of the pattern-based QPR detector in terms of improvement on plan execution time.



(a) One-shot Index Tuning (b) Incremental Index Tuning (c) Evolutionary Index Tuning

Figure 11: Validation of the generality of the regression pattern based on new emerging expensive NLJs.



(a) One-shot Index Tuning

(b) Incremental Index Tuning

Figure 12: Comparison of the improvement in execution time by using the "intermediate plan" $\tilde{P}^b$ and the "after plan" P^a.

learning (RL) technologies with actual query execution time as feedback to build reward functions that guide the RL search process. Using actual query execution time reduces the chance of QPR but is significantly more expensive compared with using what-if calls.

Query Performance Regression. QPR is an averse problem in practice. One prominent cause of QPR is plan change due to bad plan choice made by the query optimizer. QPR emerging in index tuning, in particular, falls into this category and is more costly given the nontrivial overhead of running the index tuner and creating the recommended indexes in addition to the query execution time itself. QPR after index tuning means all tuning efforts are wasted and the recommended indexes have to be dropped to bring the query execution time back to normal [8]. Existing approaches to QPR detection in the context of index tuning mainly adopt machine learning (ML) technologies [11, 41, 53, 60]. These approaches often suffer from limited generalization capability when facing new databases and workloads. The pattern-based QPR detector studied in this paper provides an alternative to these ML-based approaches with better generalizability. On the other hand, there has also been work on QPR *correction* in the context of index tuning [12], which is an interesting but orthogonal direction.

Cardinality Estimation. The pattern-based QPR detector proposed in this paper relies on accurate cardinality information available from execution feedback in the "before plan." Moreover, the evaluation results in Section 5.4.2 further demonstrate the potential improvement of the pattern-based QPR detector by fixing cardinality estimation errors. There has been extensive work in the literature on improving cardinality/selectivity estimation accuracy, and we refer the readers to recent benchmark studies [15, 49] for an overview of progresses in this area. As we mentioned in Section 5.4.2, it remains interesting future work to integrate these more advanced cardinality estimation technologies into the pattern-based QPR detector to improve the calculation of the cost inflation factors when exact matching fails in the "before plan."

Cost Modeling. We have used query plan recosting [13, 55, 56] in the pattern-based QPR detector, based on simple cost formulas crafted by following the execution logic of the NLJ and other join operators. It is well-known that query optimizer's cost modeling can be inaccurate, and there has been considerable amount of work on improving cost modeling (e.g., [1, 14, 16, 25, 27, 28, 36, 42, 46, 54, 55, 58]). While the simple cost modeling techniques used for the pattern-based QPR detector show reasonable results in our evaluation, it remains interesting future work to leverage more advanced cost modeling techniques for further improvement.

7 Conclusion

We have proposed a pattern-based QPR detector based on learnings from an in-depth study of QPRs emerging from real-world index tuning scenarios. The design of the pattern-based QPR detector is motivated by the observation that most of the significant QPRs can be attributed to expensive nested-loop joins with underestimated input cardinalities. We have evaluated the pattern-based QPR detector on top of both industrial benchmarks and real customer workloads. Our evaluation results show that the pattern-based QPR detector exhibits better generalizability than state-of-the-art ML-based QPR detectors when applied to new databases and workloads.

References

[1] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. 2012. Learning-based Query Performance Modeling and Prediction. In *ICDE*.

[2] Anonymous authors. 2025. Understanding and Detecting Query Performance Regression in Practical Index Tuning (Extended Version). https://1drv.ms/f/c/825796278ed48a9e/EuqYl4iRmt1KjFB6goqUSDMBC_xm6Hh-V9K0UD7jFEKkNA?e=VQCXbe.

[3] Matteo Brucato, Tarique Siddiqui, Wentao Wu, Vivek Narasayya, and Surajit Chaudhuri. 2024. Wred: Workload Reduction for Scalable Index Tuning. *Proc. ACM Manag. Data* 2, 1, Article 50 (2024), 26 pages.

[4] Nicolas Bruno and Surajit Chaudhuri. 2005. Automatic Physical Database Tuning: A Relaxation-based Approach. In *SIGMOD*. 227–238.

[5] Surajit Chaudhuri and Vivek Narasayya. 2020. Anytime Algorithm of Database Tuning Advisor for Microsoft SQL Server.

[6] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*. 146–155.

[7] Surajit Chaudhuri and Vivek R. Narasayya. 1998. AutoAdmin 'What-if' Index Analysis Utility. In *SIGMOD*. 367–378.

[8] Sudipto Das et al. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *SIGMOD*. 666–679.

[9] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. 2011. CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads. *Proc. VLDB Endow.* 4, 6 (2011), 362–372.

[10] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek R. Narasayya. 2021. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems. *Proc. VLDB Endow.* 14, 13 (2021), 3376–3388.

[11] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *SIGMOD*. 1241–1258.

[12] Bailu Ding, Sudipto Das, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2018. Plan Stitch: Harnessing the Best of Many Plans. *Proc. VLDB Endow.* 11, 10 (2018), 1123–1136.

[13] Anshuman Dutt, Vivek R. Narasayya, and Surajit Chaudhuri. 2017. Leveraging Re-costing for Online Optimization of Parameterized Queries with Guarantees. In *SIGMOD*. 1539–1554.

[14] Archana Ganapathi, Harumi A. Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael I. Jordan, and David A. Patterson. 2009. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *ICDE*.

[15] Yuxing Han et al. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *Proc. VLDB Endow.* 15, 4 (2021), 752–765.

[16] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction. *Proc. VLDB Endow.* 15, 11 (2022), 2361–2374.

[17] Yannis E. Ioannidis and Stavros Christodoulakis. 1991. On the Propagation of Errors in the Size of Join Results. In *SIGMOD*. 268–277.

[18] Andrew Kane. 2017. Introducing Dexter, the Automatic Indexer for Postgres. https://medium.com/@ankane/introducing-dexter-the-automatic-indexer-for-postgres-5f8fa8b28f27.

[19] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR*.

[20] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms. *Proc. VLDB Endow.* 13, 11 (2020), 2382–2395.

[21] Bartosz Krawczyk. 2016. Learning from imbalanced data: open challenges and future directions. *Prog. Artif. Intell.* 5, 4 (2016), 221–232.

[22] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2020. An Index Advisor Using Deep Reinforcement Learning. In *CIKM*. 2105–2108.

[23] Kukjin Lee, Anshuman Dutt, Vivek R. Narasayya, and Surajit Chaudhuri. 2023. Analyzing the Impact of Cardinality Estimation on Execution Plans in Microsoft SQL Server. *Proc. VLDB Endow.* 16, 11 (2023), 2871–2883.

[24] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.

[25] Jiexing Li, Arnd Christian König, Vivek R. Narasayya, and Surajit Chaudhuri. 2012. Robust Estimation of Resource Consumption for SQL Queries using Statistical Techniques. *Proc. VLDB Endow.* 5, 11 (2012), 1555–1566.

[26] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E. Alsaadi. 2017. A survey of deep neural network architectures and their applications. *Neurocomputing* 234 (2017), 11–26.

[27] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (2019), 1705–1718.

[28] Ryan C. Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.* 12, 11 (2019), 1733–1746.

[29] Microsoft. 2025. Apply a Fixed Query Plan to a Plan Guide. https://learn.microsoft.com/en-us/sql/relational-databases/performance/apply-a-fixed-query-plan-to-a-plan-guide?view=sql-server-ver16.

[30] Microsoft. 2025. Azure SQL Database. https://azure.microsoft.com/en-us/products/azure-sql/database.

[31] Microsoft. 2025. CREATE INDEX (Transact-SQL). https://learn.microsoft.com/en-us/sql/t-sql/statements/create-index-transact-sql?view=sql-server-ver16.

[32] Microsoft. 2025. Intro to Query Execution Bitmap Filters. https://techcommunity.microsoft.com/t5/sql-server-blog/intro-to-query-execution-bitmap-filters/ba-p/383175.

[33] Microsoft. 2025. Program for TPC-H Data Generation with Skew. https://www.microsoft.com/en-us/download/details.aspx?id=52430.

[34] Microsoft. 2025. Table hints (Transact-SQL). https://learn.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-table?view=sql-server-ver16.

[35] Lili Mou et al. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *AAAI*. 1287–1293.

[36] Debjyoti Paul, Jie Cao, Feifei Li, and Vivek Srikumar. 2021. Database Workload Characterization with Query Plan Encoders. *PVLDB* 15, 4 (2021), 923–935.

[37] R. Malinga Perera, Bastian Oetomo, Benjamin I. P. Rubinstein, and Renata Borovica-Gajic. 2021. DBA bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees. In *ICDE*. 600–611.

[38] R. Malinga Perera, Bastian Oetomo, Benjamin I. P. Rubinstein, and Renata Borovica-Gajic. 2022. HMAB: Self-Driving Hierarchy of Bandits for Integrated Physical Database Design Tuning. *Proc. VLDB Endow.* 16, 2 (2022), 216–229.

[39] Rainer Schlosser, Jan Kossmann, and Martin Boissier. 2019. Efficient Scalable Multi-attribute Index Selection Using Recursive Strategies. In *ICDE*. 1238–1249.

[40] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. The Case for Automatic Database Administration using Deep Reinforcement Learning. *CoRR* abs/1801.05643 (2018).

[41] Jiachen Shi, Gao Cong, and Xiaoli Li. 2022. Learned Index Benefits: Machine Learning Based Index Performance Estimation. *PVLDB* 15, 13 (2022), 3950–3962.

[42] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. In *SIGMOD*. 99–113.

[43] Tarique Siddiqui, Saehan Jo, Wentao Wu, Chi Wang, Vivek R. Narasayya, and Surajit Chaudhuri. 2022. ISUM: Efficiently Compressing Large and Complex Workloads for Scalable Index Tuning. In *SIGMOD*. 660–673.

[44] Tarique Siddiqui and Wentao Wu. 2023. ML-Powered Index Tuning: An Overview of Recent Progress and Open Challenges. *SIGMOD Rec.* 52, 4 (2023), 19–30.

[45] Tarique Siddiqui, Wentao Wu, Vivek R. Narasayya, and Surajit Chaudhuri. 2022. DISTILL: Low-Overhead Data-Driven Techniques for Filtering and Costing Indexes for Scalable Index Tuning. *Proc. VLDB Endow.* 15, 10 (2022), 2019–2031.

[46] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *Proc. VLDB Endow.* 13, 3 (2019), 307–319.

[47] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE*. 101–110.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NIPS*. 5998–6008.

[49] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready For Learned Cardinality Estimation? *Proc. VLDB Endow.* 14, 9 (2021), 1640–1654.

[50] Xiaoying Wang, Wentao Wu, Vivek R. Narasayya, and Surajit Chaudhuri. 2025. Esc: An Early-Stopping Checker for Budget-aware Index Tuning. *Proc. VLDB Endow.* 18, 5 (2025), 1278–1290.

[51] Xiaoying Wang, Wentao Wu, Chi Wang, Vivek R. Narasayya, and Surajit Chaudhuri. 2024. Wii: Dynamic Budget Reallocation In Index Tuning. *Proc. ACM Manag. Data* 2, 3 (2024), 182.

[52] Kyu-Young Whang. 1985. Index Selection in Relational Databases. In *Foundations of Data Organization*. 487–500.

[53] Wentao Wu. 2025. Hybrid Cost Modeling for Reducing Query Performance Regression in Index Tuning. *IEEE Trans. Knowl. Data Eng.* 37, 1 (2025), 379–391.

[54] Wentao Wu, Yun Chi, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Towards Predicting Query Execution Time for Concurrent and Dynamic Database Workloads. *Proc. VLDB Endow.* 6, 10 (2013), 925–936.

[55] Wentao Wu, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In *ICDE*. 1081–1092.

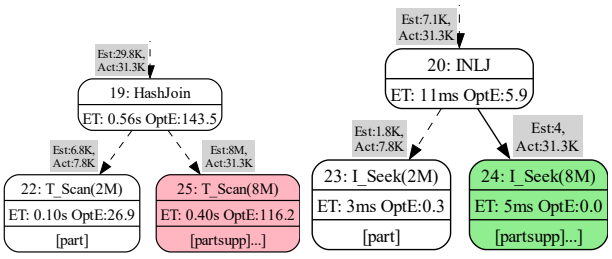
[56] Wentao Wu, Jeffrey F. Naughton, and Harneet Singh. 2016. Sampling-Based Query Re-Optimization. In *SIGMOD*. 1721–1736.

[57] Wentao Wu, Chi Wang, Tarique Siddiqui, Junxiong Wang, Vivek R. Narasayya, Surajit Chaudhuri, and Philip A. Bernstein. 2022. Budget-aware Index Tuning with Reinforcement Learning. In *SIGMOD*. 1528–1541.

[58] Wentao Wu, Xi Wu, Hakan Hacigümüs, and Jeffrey F. Naughton. 2014. Uncertainty Aware Query Execution Time Prediction. *PVLDB* 7, 14 (2014), 1857–1868.

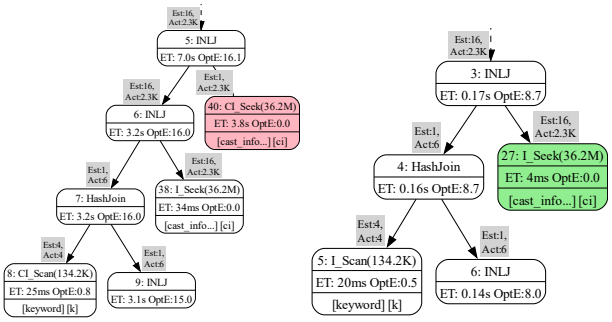
[59] Ritwik Yadav, Satyanarayana R. Valluri, and Mohamed Zaït. 2023. AIM: A practical approach to automated index management for SQL databases. In *ICDE*.

[60] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: A Tree Transformer Model for Query Plan Representation. *Proc. VLDB Endow.* 15, 8 (2022), 1658–1670.



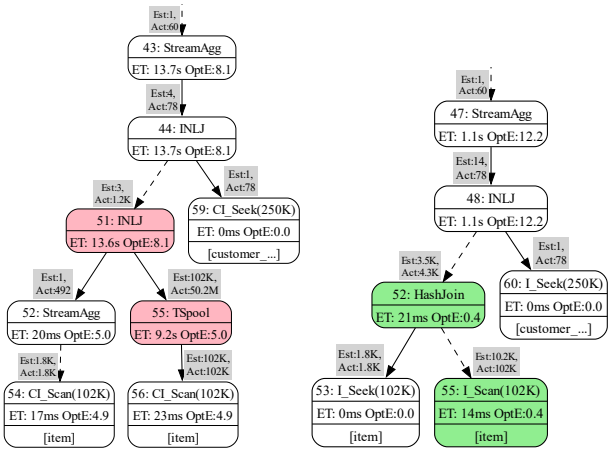
(a) **Before plan P^b** (b) **After plan P^a**

Figure 13: Illustration of the "gain pattern" GP-1.



(a) **Before plan P^b** (b) **After plan P^a**

Figure 14: Illustration of the "gain pattern" GP-2.



(a) **Before plan P^b** (b) **After plan P^a**

Figure 15: Illustration of the "gain pattern" GP-3.

A Beneficial Indexes and Patterns

Index recommendations are often beneficial as they help the query optimizer in finding more efficient ways to access the base tables. Below, we present a few representative examples of such improvements, and compare the plans before and after implementing the index recommendation. We call these beneficial indexes and their patterns the "*gain patterns*" (GPs).

A.1 Avoiding Expensive Scan (GP-1)

In Figure 13(a), we show the bottleneck operators for query 2 of TPC-H. The most expensive operator of the "before plan" is the `Table Scan` operator for the base table partsupp with total cardinality of 8 million (node 25, highlighted in pink color), although only 31K rows satisfy the filter and join predicates. The "after plan," as shown in Figure 13(a), uses the recommended index to efficiently access the corresponding table with an `Index Seek` operator (node 24, highlighted in green color). We find many such examples where index helps in avoiding expensive scan on a large table.

A.2 Avoiding Extra Lookup (GP-2)

As a second example for beneficial indexes, we show the "before plan" and "after plan" for query 26b from JOB in Figure 14. Observe that the "before plan" first uses an index on a foreign key column of the cast_info table to evaluate the join predicate (node 38) and then accesses the clustered index on cast_info to access other columns that are required to join with the remaining tables.[1] This access to the clustered index turns out to be expensive (see node 40, highlighted in pink color). The "after plan" uses the recommended index with included columns to retrieve all required columns from cast_info (node 24, highlighted in green color) and helps avoid the extra expensive lookup from the clustered index.

A.3 Avoiding Spool Operator (GP-3)

In the third example shown in Figure 15, we find that the "before plan" for DSB query 60 uses a `Table Spool` operator (node 55) together with clustered index scan (node 56) to join with a relatively small table item (with 100K rows). When executed, the nested-loop join with the spool operator (nodes 51 and 55) turns out to be expensive as the number of rebinds are higher at run-time (highlighted in pink color). The recommended index leads to an "after plan" that avoids the use of spool operator and executes the join faster using `Hash Join` with non-clustered indexes (nodes 52 and 55, highlighted in green color).

B An Overview of Found QPRs

Figures 16 to 29 present the distributions of the percentage improvement/regression regarding query plan execution time (by comparing the "after plan" P^a with the "before plan" P^b) over all workloads studied for the three index tuning scenarios, namely, one-shot tuning, incremental tuning, and evolutionary tuning.

We observe that the chance of regression is relatively small compared to that of improvement, which is expected as index tuning in general accelerates query execution. However, when regression happens, it can be significant—we observe a considerable fraction of regressions with a slowdown of more than 2× in terms of query execution time on certain workloads. Examples include one-shot tuning of JOB (Figure 18(a)), incremental tuning of TPC-H (Figure 16(b)), evolutionary tuning of Real-LO (Figure 21(c)), and more.

C Algorithm Details for Index Tuning Scenarios

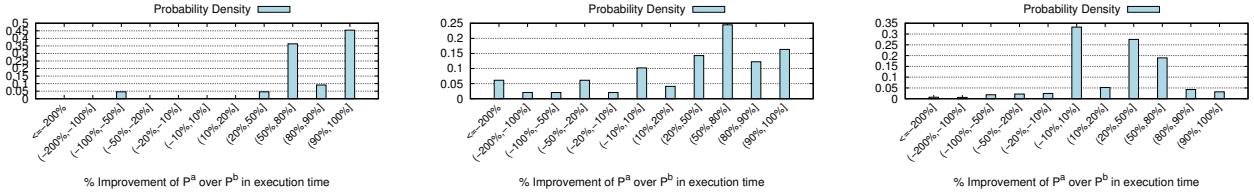
C.1 One-shot Index Tuning

Algorithm 6 presents the algorithmic details of the data generation process of one-shot index tuning.

C.2 Incremental Index Tuning

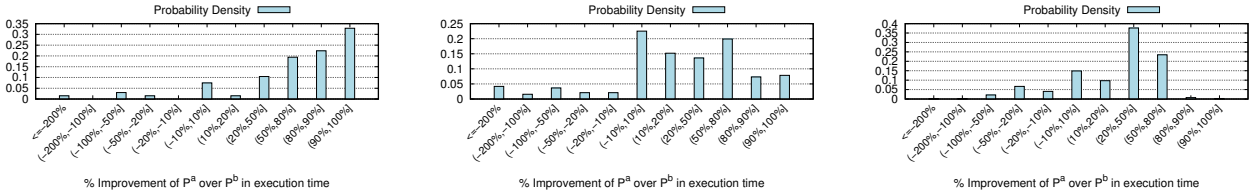
Algorithm 7 presents the algorithmic details of the data generation process of incremental index tuning.

[1] This is also called a `Key Lookup` in Microsoft SQL Server.



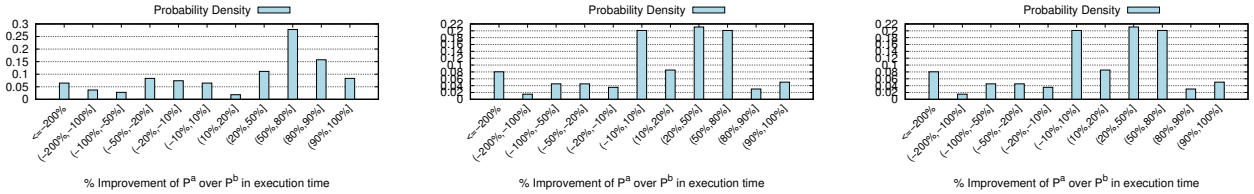
(a) One-shot Index Tuning (b) Incremental Index Tuning (c) Evolutionary Index Tuning

Figure 16: Distribution of performance improvement/regression for TPC-H plan pairs collected.



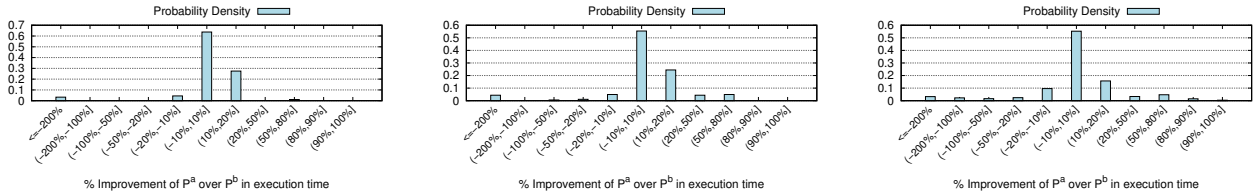
(a) One-shot Index Tuning (b) Incremental Index Tuning (c) Evolutionary Index Tuning

Figure 17: Distribution of performance improvement/regression for DSB plan pairs collected.



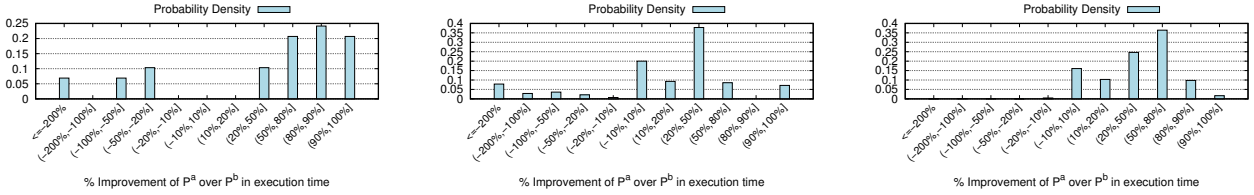
(a) One-shot Index Tuning (b) Incremental Index Tuning (c) Evolutionary Index Tuning

Figure 18: Distribution of performance improvement/regression for JOB plan pairs collected.



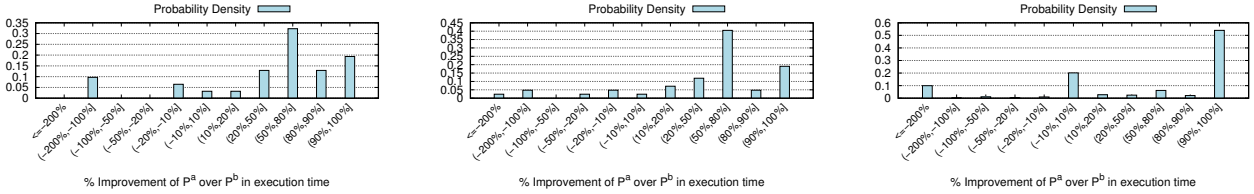
(a) One-shot Index Tuning (b) Incremental Index Tuning (c) Evolutionary Index Tuning

Figure 19: Distribution of performance improvement/regression for STATS plan pairs collected.



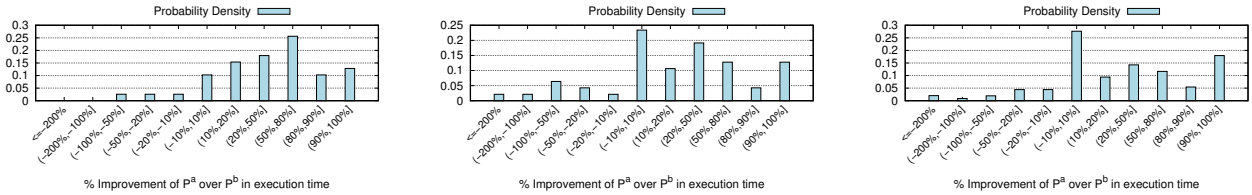
(a) One-shot Index Tuning (b) Incremental Index Tuning (c) Evolutionary Index Tuning

Figure 20: Distribution of performance improvement/regression for Real-DY plan pairs collected.



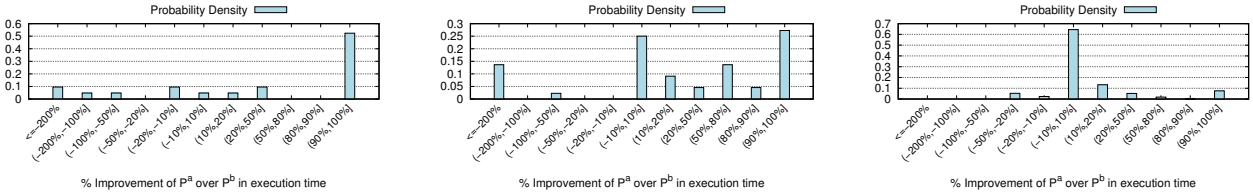
(a) One-shot Index Tuning (b) Incremental Index Tuning (c) Evolutionary Index Tuning

Figure 21: Distribution of performance improvement/regression for Real-LO plan pairs collected.



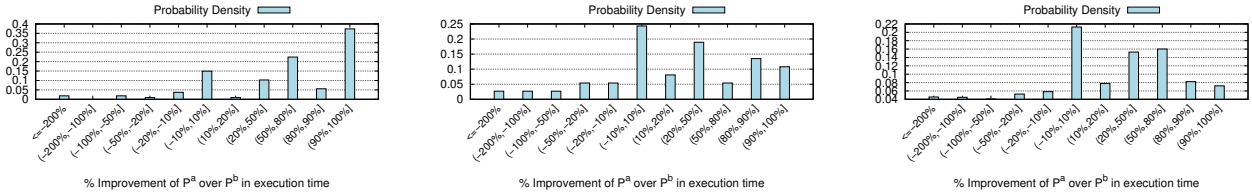
(a) One-shot Index Tuning (b) Incremental Index Tuning (c) Evolutionary Index Tuning

Figure 22: Distribution of performance improvement/regression for Real-MS plan pairs collected.



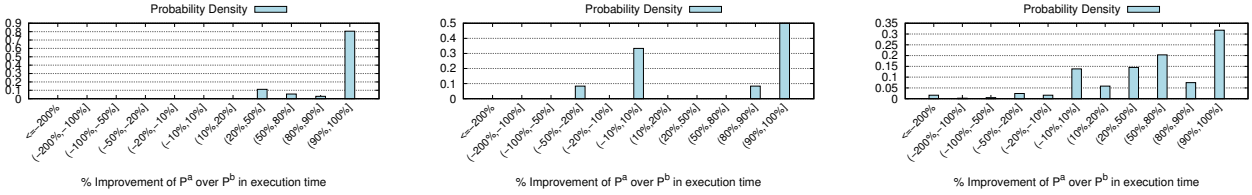
(a) One-shot Index Tuning (b) Incremental Index Tuning (c) Evolutionary Index Tuning

Figure 23: Distribution of performance improvement/regression for Real-RE plan pairs collected.



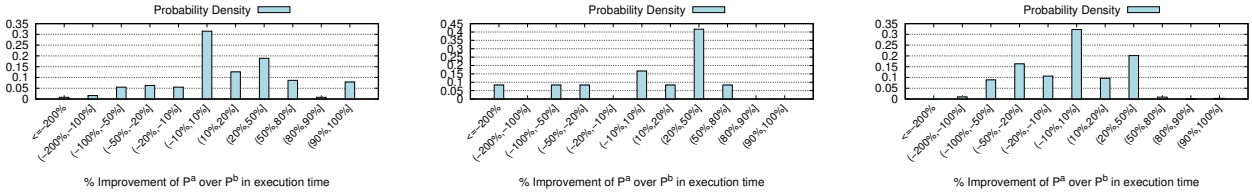
(a) One-shot Index Tuning (b) Incremental Index Tuning (c) Evolutionary Index Tuning

Figure 24: Distribution of performance improvement/regression for Real-DW plan pairs collected.



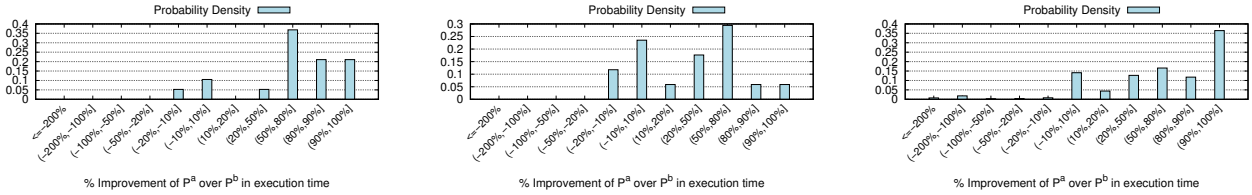
(a) One-shot Index Tuning (b) Incremental Index Tuning (c) Evolutionary Index Tuning

Figure 25: Distribution of performance improvement/regression for Real-ED plan pairs collected.



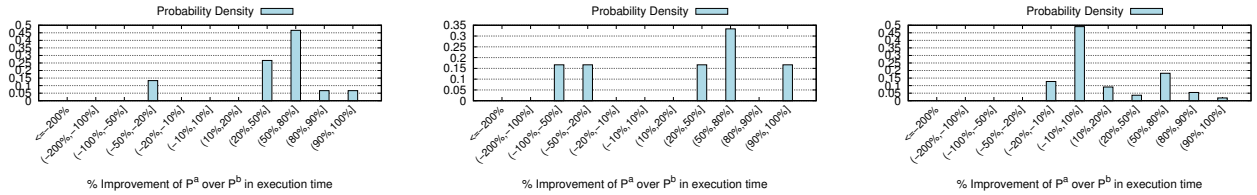
(a) One-shot Index Tuning (b) Incremental Index Tuning (c) Evolutionary Index Tuning

Figure 26: Distribution of performance improvement/regression for Real-MP plan pairs collected.



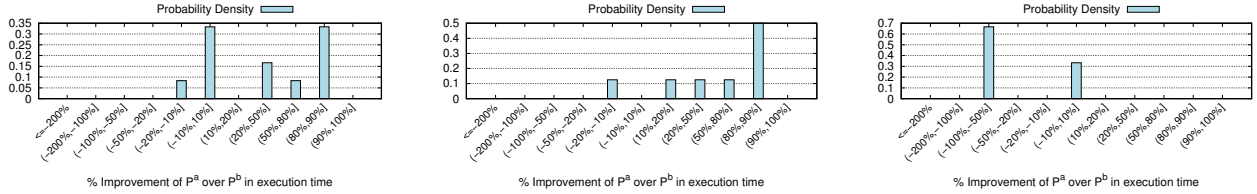
(a) One-shot Index Tuning (b) Incremental Index Tuning (c) Evolutionary Index Tuning

Figure 27: Distribution of performance improvement/regression for Real-SE plan pairs collected.



(a) One-shot Index Tuning (b) Incremental Index Tuning (c) Evolutionary Index Tuning

Figure 28: Distribution of performance improvement/regression for Real-RM plan pairs collected.



(a) One-shot Index Tuning (b) Incremental Index Tuning (c) Evolutionary Index Tuning

Figure 29: Distribution of performance improvement/regression for Real-SA plan pairs collected.

Algorithm 8: Evolutionary index tuning.

Input: $\mathcal{A}$, the index tuner; W, the workload; C_0, the initial configuration of the database.
Output: D_W^{evol}, the data collected for evolutionary index tuning.

$D_W^{\text{evol}} \leftarrow \emptyset$;
foreach *query* $q \in W$ **do**
  $C \leftarrow \text{TuneQuery}(\mathcal{A}, q, C_0)$;
  Materialize C;
  $\mathcal{P} \leftarrow \emptyset$;
  foreach *subset* $S \subset C$ **do**
    $P_S \leftarrow \text{GetPlan}(q, S)$, $t_S \leftarrow \text{RunQuery}(q, P_S)$;
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{(q, P_S, t_S)\}$;
  foreach *pair* $(q, P_1, P_2, t_1, t_2) \in \mathcal{P} \times \mathcal{P}$ **do**
    Sort the pair to make sure $\text{cost}(P_1) \geq \text{cost}(P_2)$;
    $D_W^{\text{evol}} \leftarrow D_W^{\text{evol}} \cup \{(q, P_1, P_2, t_1, t_2)\}$;
return D_W^{evol};

Algorithm 6: One-shot index tuning.

Input: $\mathcal{A}$, the index tuner; W, the workload; C_0, the initial configuration of the database.
Output: D_W^{os}, the data collected for one-shot index tuning.

$D_W^{\text{os}} \leftarrow \emptyset$;
foreach *query* $q \in W$ **do**
  $P_0 \leftarrow \text{GetPlan}(q, C_0)$, $t_0 \leftarrow \text{RunQuery}(q, P_0)$;
  $C \leftarrow \text{TuneQuery}(\mathcal{A}, q, C_0)$;
  Materialize C;
  $P \leftarrow \text{GetPlan}(q, C)$, $t \leftarrow \text{RunQuery}(q, P)$;
  $D_W^{\text{os}} \leftarrow D_W^{\text{os}} \cup \{(q, P_0, P, t_0, t)\}$;
return D_W^{os};

Algorithm 7: Incremental index tuning.

Input: $\mathcal{A}$, the index tuner; W, the workload; C_0, the initial configuration of the database.
Output: D_W^{inc}, the data collected for incremental index tuning.

$D_W^{\text{inc}} \leftarrow \emptyset$;
foreach *query* $q \in W$ **do**
  for $i = 1, \ldots$ **do**
    $P_{i-1} \leftarrow \text{GetPlan}(q, C_{i-1})$, $t_{i-1} \leftarrow \text{RunQuery}(q, P_{i-1})$;
    $I_i \leftarrow \text{TuneQuery}(\mathcal{A}, q, C_{i-1})$;
    if I_i *is null* **then**
      Break;
    Materialize I_i;
    $C_i \leftarrow C_{i-1} \cup \{I_i\}$;
    $P_i \leftarrow \text{GetPlan}(q, C_i)$, $t_i \leftarrow \text{RunQuery}(q, P_i)$;
    $D_W^{\text{inc}} \leftarrow D_W^{\text{inc}} \cup \{(q, P_{i-1}, P_i, t_{i-1}, t_i)\}$;
return D_W^{inc};

C.3 Evolutionary Index Tuning

Algorithm 8 presents the algorithmic details of the data generation process of evolutionary index tuning.