SWatcher+: A Holistic Approach to Deploying Learned Cardinality Estimators in a Dynamic Setting

Yuxi Liu
Duke University
yuxi.liu@duke.edu

Vincent Capol
Duke University
vincent.capol@duke.edu

Xiao Hu
University of Waterloo
xiaohu@uwaterloo.ca

Pankaj K. Agarwal
Duke University
pankaj@cs.duke.edu

Jun Yang
Duke University
junyang@cs.duke.edu

ABSTRACT

Cardinality estimation (CE) is crucial for database query optimization. While recent advances in machine-learned CE have improved accuracy, deploying these models in dynamic databases remains challenging due to data updates and workload shifts. Frequent retraining is costly, and existing periodic retraining strategies provide little guidance on when and how to update models efficiently. We propose *Watcher*, a framework that adapts learned CE to dynamic workloads by monitoring data updates and query feedback. Instead of full retraining, Watcher maintains cardinalities for representative subqueries and selectively updates models as needed. We introduce three variants: Watcher1D and Watcher2D for single-table and two-table join cardinalities, and SWatcher+, which corrects CE estimates and monitors high-impact subqueries. Our experiments show that Watcher offers a better trade-off between query performance and overhead compared to existing approaches, making it a more practical solution for deploying learned CE in dynamic databases.

PVLDB Reference Format:
Yuxi Liu, Vincent Capol, Xiao Hu, Pankaj K. Agarwal, and Jun Yang. SWatcher+: A Holistic Approach to Deploying Learned Cardinality Estimators in a Dynamic Setting. PVLDB, 14(1): XXX-XXX, 2020. doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/louisja1/Watcher-VLDB25.

1 INTRODUCTION

Most database query optimizers rely heavily on *cardinality estimation* (*CE*) to generate efficient query plans. Recent advances in *learned CE* [9, 26–28, 31, 32, 34, 37, 48, 49], where a machine learning model is trained on the database instance and/or query execution feedback, show promise in improving the accuracy of CE. However, it remains challenging to deploy learned CE in practical settings, where the database is changing dynamically and the existing model is no longer adequate. Most learned CE models cannot be incrementally updated upon every data update or query execution. Retraining these models can be costly — not only in itself but also in the cost of acquiring up-to-date training data. When to retrain CE models is far from clear. Most previous work suggests the simple strategy of retraining periodically [5, 19, 20, 24, 49], in time or the number of data updates since the last retraining, often without clear guidance on how to set these periods. There is a need to develop a smarter framework for deploying learned CE in a dynamic setting.

A better idea is to *monitor* the database workload for clues to retrain. Ideally, it makes sense to monitor both data updates and query feedback. Ignoring data updates makes retraining unnecessarily reactive — it misses cases when data has changed significantly, but not enough queries have witnessed the effect. On the other hand, ignoring the query workload may make retraining unnecessary — often, parts of the database are rarely queried, so updates to these parts have little effect on the query workload.

These observations naturally prompt us to consider an idea analogous to incrementally maintaining data summaries (such as histograms) in response to data updates, but with more attention to parts of the database that are queried. Here, we can maintain cardinalities of subqueries that are representative of the query workload, and use them to inform retraining decisions. However, making this idea practical involves many challenges. First, monitoring incurs overhead, but its benefit is not always clear. Many CE errors may not actually impact plan quality [16]. Second, improving CE accuracy for some subqueries (e.g., only single-table selections) may sometimes lead to worse plans. This counterintuitive effect has been noted in [16] and confirmed in our experiments.[1] Finally, retraining the CE model does not guarantee more accurate CE for the queries where accurate CE does matter.

One methodological contribution of this paper is a holistic examination of the problem of deploying learned CE in a dynamic setting. Instead of focusing narrowly on how a solution improves CE accuracy, we also look at its ultimate impact on query performance, as well as its overall cost, which includes the overhead of monitoring and retraining. This holistic examination reveals that this problem deserves more attention than it has received so far from the research community, because any practical solution requires a challenging and delicate balancing act between benefit and cost. Following an iterative process of evaluation and development,

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

[1] for example, our experiments in Section 6 show that for queries of query template Q018 in DSB, injecting true CEs into single-selection subqueries causes 23% of queries to backfire, resulting in a 13% increase in end-to-end runtime compared to a baseline system using the CE model without retraining.

we have devised ways to make monitoring more efficient and beneficial beyond retraining — the results are our "watcher"-family of methods: *Watcher1D*, *Watcher2D*, and *SWatcher+*, with different trade-offs and levels of sophistication.

- Watcher1D **(Section 3)** and Watcher2D **(Section 4)** embody the basic monitoring idea. They monitor accurate cardinalities for a reservoir sample [42] of single-table selection and two-table join subqueries from the query workload, respectively. Even with the most efficient maintenance methods available, our experiments show that the practical benefit of monitoring often fails to justify its overhead.
- Going beyond basic monitoring, SWatcher+ **(Section 5)** further explores several ideas as reflected by its name. "+" in SWatcher+ is the idea of increasing the benefit of monitoring by using monitored information more proactively, to correct model estimates for query optimization, without waiting for retraining. "S" in SWatcher+ is the idea of making monitoring selective, based on the assessment of its benefit. We choose to maintain only the cardinalities of subqueries for which error correction is expected to make a tangible difference to query performance. Noting that correcting selection cardinalities alone may have little or even a negative impact, we also monitor information about joins — but instead of maintaining their exact cardinalities, which is expensive, we only cache recent "compensating correction factors."
- Through expensive experiments **(Section 6)** that holistically evaluate the benefits and costs of competing solutions, we demonstrate that SWatcher+, with a fixed configuration and minimal parameter tuning, delivers reasonable end-to-end runtimes compared to other algorithms. Specifically, SWatcher+ achieves 44.3% to 101.4% of the end-to-end runtime of periodic retraining, outperforming it in 12 out of 14 templates, even when retraining is configured with the most optimal settings observed post hoc.

2 FRAMEWORK AND PRELIMINARIES

2.1 Assumptions and Preliminaries

Our watcher-family of methods are designed to be generally applicable, without being tied to particular learned CE models and database systems. In the following, we state the assumptions made by our methods, which we believe are often met in practice.

Learned CE Model. We are given a learned model $\mathfrak{m}$ that can be (re)trained to predict the cardinalities of at least single-table selection queries (for Watcher1D and SWatcher+) and two-table equality joins with selections (for Watcher2D). Model (re)training in general can use query feedback — namely queries and their true cardinalities over the current database state — as well as the current database state if needed. Our methods apply in cases where training $\mathfrak{m}$ takes considerable time, and incrementally updating $\mathfrak{m}$ per data update or query feedback is infeasible.

Data Update Stream. Our methods assume access to the data update stream. They are notified of each update and can read its content for processing. Watcher2D may query the database for additional information as needed.

Query Stream. Our methods assume access to the stream of queries issued against the database. Additionally, it is possible to inspect a query plan π (for query Q) and its execution to obtain the cardinality of the result set produced by each of its subplans. With PostgreSQL, for example, such information is readily available with the `EXPLAIN ANALYZE` command. However, since there are many options for π, we cannot, in general, assume that the cardinality of every subquery q of the query Q is observable from the execution of π, since π may not contain a subplan that exactly corresponds to q. Hence, our method may issue additional queries to acquire subquery cardinalities; more details are given in later sections.

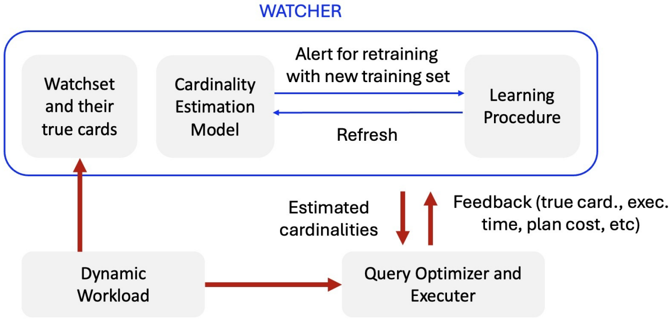

WATCHER
Watchset and their true cards
Cardinality Estimation Model
Alert for retraining with new training set
Refresh
Learning Procedure
Estimated cardinalities
Feedback (true card., exec. time, plan cost, etc)
Dynamic Workload
Query Optimizer and Executer

Figure 1: System architecture for watcher-family of methods.

Interfacing with Query Optimizer. Here, we assume a setup typical in learned CE research, where the model $\mathfrak{m}$ is external to (as opposed to integrated into) the database query optimizer. We assume that we can make a call $\mathrm{Opt}(Q, \boldsymbol{\kappa})$ to the optimizer to optimize a query Q using a set $\boldsymbol{\kappa}$ of cardinality estimates for subqueries of Q. As is commonly done in literature [8, 16, 21, 47], instead of specifying CE for all possible subqueries, which would be too costly, $\boldsymbol{\kappa}$ overrides CE only for a "relevant" subset of subqueries, while leaving the estimation of others to the database optimizer (which may depend on those specified in $\boldsymbol{\kappa}$). Furthermore, after obtaining a plan π, we assume we can ask the database system to execute it. Finally, SWatcher+ additionally assumes it can make a call $\mathrm{Cost}(\pi, \boldsymbol{\kappa})$ to the optimizer to estimate the cost of π given a set $\boldsymbol{\kappa}$ of relevant cardinality estimates, without executing π. For our PostgreSQL-based implementation, we use the same mechanism in [8, 47] for cardinality injection, and `pg_hint_plan` [30] for specifying the plan to execute (`EXPLAIN ANALYZE`) and cost (`EXPLAIN`).

2.2 Architecture and Overview

Figure 1 depicts the system architecture for the watcher family of methods. They are defined by three aspects of their behaviors: 1) processing each data update; 2) processing each query; and 3) triggering and preparing for model retraining. We briefly overview the three methods, highlighting their similarity and differences, while leaving the details to later.

All methods maintain a *watchset* of queries whose result set cardinalities are kept up to date with respect to data updates. Watchset queries are simple subqueries selected from observed queries. The cost of maintaining their cardinalities is the main source of overhead in processing data updates.

Watcher1D and Watcher2D. Both use reservoir sampling to pick watchset queries and compare their true cardinalities to estimates for an aggregate error, which, when high enough, triggers model retraining. Watcher1D only monitors single-table selection subqueries, whose cardinalities are straightforward to maintain given data updates. Watcher2D additionally monitors two-table join subqueries with optional range selections on each input table. We do not consider maintaining cardinalities for joins involving more than two tables, as two-table joins are already challenging and costly for Watcher2D. Watcher1D and Watcher2D use the information they monitor only to benefit retraining (informing its timing and providing readily available training data) and nothing else. For these two methods, the main source of overhead lies in processing data updates; query processing incurs no overhead beyond what is required to enable the database optimizer to use the CE model.

SWatcher+. This method reduces the overhead of processing data updates by limiting its watchset to selections only, and carefully deciding whether to admit/evict a subquery to/from the watchset using its potential benefit. Only subqueries for which knowing their true cardinalities tangibly improves query performance are retained in the watchset.

Moreover, to maximize the benefit of monitored information, SWatcher+ may choose to correct some estimated cardinalities of the current model when invoking the query optimizer. Selection corrections are based on the cardinalities maintained in the watchset. Join corrections, on the other hand, are based on cached information observed from recent query executions with similar signatures. Compared with cardinalities actively maintained for the watchset, cached information is less accurate, but it is cheaply available even for multiway joins involving more than two tables, and it helps avoid the pitfall of applying only selection corrections.

Overall, the query processing aspect of SWatcher+ is more elaborate than Watcher1D and Watcher2D, as it includes additional instrumentation for caching and additional optimizer calls to assess the potential benefit of candidate watchset queries. While the overhead of bookkeeping is small, computing corrections add some overhead, and the analysis needed to make smart admission/eviction decisions can be even more expensive. On the other hand, this trade-off is for reduced data processing overhead, as well as improving query performance immediately using monitored information without waiting until retraining.

Lastly, SWatcher+ changes its retraining trigger from detecting a high aggregate CE error to detecting when the watchset is too large, allowing it to cap data update processing time and realize when there are too many "important" estimates that are wrong.

3 Watcher1D

At a high level, Watcher1D maintains as its watchset $\mathcal{W}$ a reservoir sample of single-table selection subqueries observed from the query workload. For instance, consider SQL 1 below:

SQL 1: a query of TJQ013a used in Section 6

```
SELECT *
FROM store_sales, store, customer_demographics,
      household_demographics, customer_address, date_dim
WHERE s_store_sk = ss_store_sk
    AND ss_sold_date_sk = d_date_sk AND ss_hdemo_sk = hd_demo_sk
    AND cd_demo_sk = ss_cdemo_sk AND ss_addr_sk = ca_address_sk
    AND ss_sales_price BETWEEN 32.78 AND 207.53
    AND hd_dep_count BETWEEN 1 AND 4;
```

$\mathcal{W}$ may include the subqueries

$$\sigma_{32.78\leq\texttt{ss\_sales\_price}\leq 207.53}\texttt{store\_sales},$$

$$\text{and } \sigma_{1\leq\texttt{hd\_dep\_count}\leq 4}\texttt{household\_demographics}.$$

For each $q \in \mathcal{W}$, Watcher1D tracks $\widehat{\text{card}}(q)$, the cardinality of q estimated by the CE model $\mathfrak{m}$, and $\text{card}(q)$, the true cardinality of q. Watcher1D monitors an aggregate error measure computed over $\mathcal{W}$ and uses it to trigger model retraining; the training data comes conveniently from $\mathcal{W}$, which already tracks true cardinalities. In the following, we describe these steps in more detail, analyze the total cost of the approach, and discuss its pros and cons.

Processing Queries. For each query Q, we identify the set of single-table selection subqueries $\{q\}$ as candidate watchset queries. We call the optimizer with the relevant cardinalities $\hat{\boldsymbol{\kappa}}$ estimated by $\mathfrak{m}$ to obtain plan $\pi = \text{Opt}(Q, \hat{\boldsymbol{\kappa}})$. We record the estimated cardinality of each candidate watchset query q as $\widehat{\text{card}}(q)$. After executing π, we record the observed true cardinality of each candidate watchset query q as $\text{card}(q)$. Such information, when available, can be easily parsed from the output of `EXPLAIN ANALYZE`. Sometimes, the chosen π may not execute q as an independent subquery. For example, for $q = \sigma_p R$, π may use an index-based nested-loop join algorithm where R is the inner input driven by an index scan based on a join condition in Q involving R; therefore, executing π does not reveal the cardinality of $\sigma_p R$ (except when the join is between the outer table's primary key and R's foreign key reference thereto). In such cases, we formulate and execute a counting query over $\sigma_p R$ to obtain $\text{card}(q)$. These additional queries (infrequent in practice) are counted towards the overhead of the approach.

Whether each candidate q is admitted to the watchset $\mathcal{W}$ follows the standard reservoir sampling procedure. A previously watched query may be evicted to keep the size of $\mathcal{W}$ at n_{watch}.

Processing Data Updates. To process each data update concerning a row r in table R, consider $\mathcal{W}_R$, the subset of queries in the watchset $\mathcal{W}$ that are single-table selections over R. We index the queries in $\mathcal{W}$ by the tables they reference, so we can quickly retrieve $\mathcal{W}_R$. For each query $q = \sigma_p R$ in $\mathcal{W}_R$, we test its selection condition p on r. If r passes p, we increment $\text{card}(q)$ if r is being inserted, or decrement $\text{card}(q)$ if r is being deleted. This processing ensures that $\text{card}(\cdot)$ is accurate and up to date for watchset queries.

This simple linear-time approach has low overhead because of its simple implementation, which works well in practice since $\mathcal{W}_R$ is usually not large. In the case where $\mathcal{W}_R$ is large, more sophisticated data structures can be used. For example, the atomic intervals induced by n range selections involving the same column of R can be indexed in an *implicit treap* [1, 6, 39] of size $O(n)$, whose nodes are annotated by the number of rows whose values for the column fall within the corresponding subtrees. With this data structure, data updates can be processed in $O(\log n)$ time, and $\text{card}(q)$ can be computed in $O(\log n)$ time. However, adding a new range selection (when processing a new query) to the tree requires up to one additional range counting query to obtain the cardinality of newly created atomic intervals, plus $O(\log n)$ time to update the tree.

Retraining CE Model. To decide when to trigger retraining of the CE model $\mathfrak{m}$, we check an aggregate error measure, described further below, after processing each data update and query, because the performance of the CE model is generally affected by both factors. If either data distribution or query distribution is known to be stable, we may decrease the frequency of checks for data updates and queries accordingly.

For the aggregate error measure, we use the 90-th percentile of Q-error [29], computed from $\text{card}(\cdot)$ and $\widehat{\text{card}}(\cdot)$ over n_{test} queries sampled from the watchset $\mathcal{W}$. Unlike average or root-mean-square error, this quantile-based error is less sensitive to a handful of outliers. Such outliers arise often in predictions of learned CE models, but they may not indicate overall model inadequacy; our quantile-based error avoids unnecessary retraining caused by these outliers.

When the aggregate error exceeds a prescribed threshold ϵ, we trigger retraining of the CE model $\mathfrak{m}$. The training dataset is constructed by sampling n_{train} watchset queries out of $\mathcal{W}$. Since we track their up-to-date cardinalities, we do not need to rerun these queries to acquire the training data, which significantly reduces the end-to-end latency of retraining. Finally, retraining does not guarantee that the aggregate error will fall below ϵ. Whenever retraining fails to reach the target, we apply an exponential backoff on ϵ, i.e., setting a new target ϵ for the next retraining by multiplying ϵ by a factor of 1.5. If the post-training error eventually falls back to a lower previous level, the previous level can be restored.

Cost Analysis. The cost of Watcher1D has three components. First, for each data update on table R, updating the true cardinalities for the subset W_R of the watchset takes $O(|W_R|)$ time (or $O(\log|W_R|)$ with a segment tree). Second, for each query Q, besides the overhead of using a CE model to produce $\hat{\boldsymbol{\kappa}}$, obtaining accurate cardinalities from the execution incurs essentially no overhead, except for any single-table selection subquery that is not computed by the chosen plan. In the worst case, an n-way join Q may generate n extra single-table counting queries, but such cases are rare in practice. Third, retraining of the CE model incurs a cost; the cost of computing and checking the aggregate error measure is negligible, and as discussed, there is no extra query cost for generating the training dataset.

As observed in Section 6, for a setup with a query/update mix of one query for every 100 updated rows and a watchset of size $n_{\text{watch}} = 1500$, 62.7% of Watcher1D's total cost goes to processing data updates; 4.3% goes to processing queries, with 100% attributed to using a CE model in the first place, and 0% attributed to the extra counting queries; the remaining 33.0% goes to retraining. Compared to a baseline system deployed with a CE model with no retraining, Watcher1D on average is able to reduce the querying cost by about 5% on average, yet difficult to justify the maintenance overhead of $\mathcal{W}$, netting an overall regression in total cost by -1.2% to -8.5%.

Discussion. The main advantages of Watcher1D are as follows. First, for single-table selection queries sampled from the query workload, it is relatively inexpensive to monitor the data updates and maintain their true cardinalities. Second, doing so allows us to inform the timing of retraining by both data and query distributions. In contrast, many related works [19, 20, 32] rely on less reliable heuristics to trigger retraining, For example, a heuristic based on observed CE errors in executed queries alone may miss retraining when data distribution has shifted dramatically but an insufficient number of queries revealing the shift have been executed recently. On the other hand, a heuristic based only on the number of data updates accumulated may lead to unnecessary retraining if the overall data distribution shift is in fact small or limited to in parts of the data that are irrelevant to the query workload. Third, this active monitoring approach has the benefit of providing a training dataset that accurately reflects the current database state at no additional cost. In contrast, many related works [14, 19, 46] incur a high cost to run queries in order to acquire up-to-date training data.

However, there are concerns on the effectiveness of Watcher1D too. First, Watcher1D focuses solely on monitoring CE errors for single-table selection queries, while many researchers believe that the CE errors for join queries are more challenging to CE models and have a greater impact on plan quality [16, 17, 47]. Hence, the benefit of monitoring single-table selection cardinalities may be limited. Second, even though monitoring single-table selection cardinalities is inexpensive, Watcher1D incurs this cost on every data update for every watchset query involving the table being updated. Overall, this cost may not justify the limited benefit discussed earlier. Motivated by these concerns, we continue our investigation of other approaches in the following sections.

4 Watcher2D

We now consider extending Watcher1D to Watcher2D, which monitors join cardinalities too, because it is generally believed that join CE is more challenging and more influential on the quality of plans selected by optimizers than selection CE [8, 17, 43]. Specifically, Watcher2D considers two-way selection-join subqueries of the form $\sigma_{l_A \le A < u_A} R \bowtie_B \sigma_{l_C \le C < u_C} S$, where tables R and S, with local range selections on columns A and B respectively, are joined on column B. As an example, for the same query SQL 1 from Section 3, we may select the watchset query:

$$\sigma_{32.78 \le \texttt{ss\_sales\_price} \le 207.53}\texttt{store\_sales}$$
$$\bowtie_{\texttt{ss\_hdemo\_sk=hd\_demo\_sk}}\sigma_{1 \le \texttt{hd\_dep\_count} \le 4}\texttt{household\_demographics}.$$

The overall approach is similar to Watcher1D. We will focus on how to process data updates when maintaining selection-join cardinalities. As expected and as will be seen, selection-join cardinalities are more costly to maintain than selection cardinalities.

Processing Queries. For each query Q, Watcher2D additionally identifies the set of two-way selection-join subqueries of Q as candidate watchset queries. The procedure works in the same way as that of Watcher1D, with the analogous challenge that the execution of the chosen plan may not reveal the true cardinalities of such subqueries. For example, the plan may choose to join R with another table first, before joining with S, making it impossible to infer the cardinality of join between R and S from the execution. In such cases, we need to execute an additional counting query to obtain the true cardinality. With the flexibility in join reordering, Watcher2D tends to need such queries more often than Watcher1D.

Processing Data Updates. To process each data update concerning a row r in table R, we need to additionally update the cardinality for any two-table watchset query joining R and another table. Let $\mathcal{W}_{RS}$ denote the subset of watch queries of the form $\sigma_{l_A \le A < u_A} R \bowtie_B \sigma_{l_C \le C < u_C} S$. The following procedure is carried out for every possible S that joins with R in the watchset.

First, we determine the subset $\mathcal{W}'_{RS}$ of $\mathcal{W}_{RS}$ for which $r.A$ passes the local selection condition $l_A \le r.A < u_A$. Only queries in $\mathcal{W}'_{RS}$

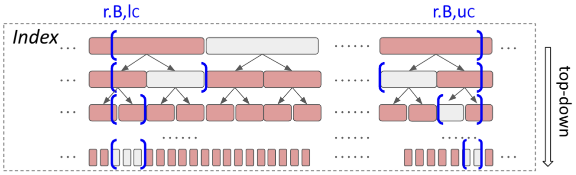


Figure 2: Watcher2D: top-down computation of one counting query. Blue parentheses is the condition $B = r.B \wedge C \geq l_C \wedge C < u_C$ or the sub-conditions generated and pushed down to the lower layers. Gray (resp. red) squares are ranges that are (resp. are not) fully covered by the conditions.

can have their cardinalities affected by r. A simple linear scan of $\mathcal{W}_{RS}$ works well to determine $\mathcal{W}'_{RS}$ in practice as $\mathcal{W}_{RS}$ is often not large. For a large $\mathcal{W}_{RS}$, we can index the $[l_A, u_A)$ intervals in an *interval tree* [38], which supports reporting of all intervals stabbed by $r.A$ in $O(\log n + k)$ time where $n = |\mathcal{W}_{RS}|$ and $k = |\mathcal{W}'_{RS}|$, and adding/removing of an interval (which happens if a query enters the watchset when processing queries) in $O(\log n)$ time.

Next, for each $q \in \mathcal{W}'_{RS}$, we determine $|\sigma_{B=r.B \wedge l_C \leq C < u_C} S|$; we add this number to $\text{card}(q)$ if r being inserted, or subtract it if r is being deleted. One option is to issue a single reporting query $\sigma_{B=r.B} S$, and scan its result rows to obtain counts for all $[l_C, u_C)$ ranges in $\mathcal{W}'_{RS}$, but doing so can be expensive when $\sigma_{B=r.B} S$ contains many result rows. Instead, we rely on an index structure built on $S(B, C)$, which can be seen as a B+tree with composite search key (B, C) where each pointer to a subtree or leaf is augmented with a total count of S rows found therein. Such counts can be incrementally maintained whenever S is updated, with minimal overhead en route from the root to the leaf containing the updated row. Given $(r.B, l_C, u_C)$, we can use this data structure to compute $|\sigma_{B=r.B \wedge l_C \leq C < u_C} S|$ by visiting counts found near the two root-to-leaf paths to $(r.B, l_C)$ and $(r.B, u_C)$, as illustrated by Figure 2. A further optimization is to process counting for all $[l_C, u_C)$ ranges in one batch: by ordering the range endpoints and sharing visits to prefixes of successive root-to-leaf paths, we avoid revisit the tree node more than once. In our implementation, instead of extending the B+tree of the database system, we use a separate data structure with some additional optimizations, described in detail in [25].

Retraining CE Model. Determining when to trigger retraining of the CE model $\mathfrak{m}$ works in the same way as in Section 3, now with two-way selection-join queries included in the watchset as well. These queries, along with their up-to-date cardinalities, can be used directly in the training set too.

Cost Analysis. Compared with Watcher1D, Watcher2D adds considerable overhead in two regards. First, when processing a query, there is a higher chance that we need to run extra counting queries to acquire true cardinalities because the execution plan did not choose these joins in its subplans; furthermore, the counting queries are generally more costly than Watcher1D because they involve joins. Second, maintaining the true cardinalities when processing data updates is far more expensive than Watcher1D. In Watcher1D, the processing cost per update is bounded by $O(n_{\text{watch}})$, independent of the database size. However, given an updated row $r \in R$, Watcher2D generally needs to probe the joining data tables to determine its effect on join cardinalities. Hence, the augmented B+tree uses space linear in the database size, and computing the cardinality changes includes a time factor logarithmic in the database size.

Even if the augmented B+tree is "almost free" assuming that the database already builds the corresponding B+tree index, the overhead of computing the cardinality changes is still significant. For example, as observed in our experiments in Section 6, for a setup where R and S contains $208,748$ and $7,200$ rows respectively, processing an updated row in R to update the cardinalities of 407 out of $1,500$ selection-join queries between R and S takes 77× longer than updating the cardinalities of the same number of selection queries on R. As discussed further in Section 6, this high overhead of monitoring join cardinalities fails to justify its potential benefit on query optimization beyond monitoring selection cardinalities.

Discussion. While we have tried to make monitoring selection-join cardinalities efficient, the inherent difficulty lies in the fact that the effect of a single row update is dependent on how this row joins with the other tables, which cannot be determined by examining query conditions alone. To reduce the overhead of monitoring and maximize its benefit, we develop a new approach below that selects what to monitor with a cost-benefit analysis, and makes use of observed and monitored true cardinalities more effectively than merely triggering model retraining.

5 SWatcher+

As discussed in Section 1, the "S" in SWatcher+ stands for the idea of selectively monitoring only selection subqueries for which CE errors indeed affect plan quality; the "+" in SWatcher+ stands for the idea of proactively incorporating monitored information during query optimization, instead of waiting for model retraining. We choose not to maintain join cardinalities because of the high overhead discussed in Section 4, but to avoid the pitfall of correcting selection CE errors alone, we also cache "compensating correction factors" for joins and apply them together with selection cardinality corrections. This section details SWatcher+, beginning with an overview below. SWatcher+ maintains the following two main data structures:

- Watchset $\mathcal{W}$. Like Watcher1D, $\mathcal{W}$ is a subset of single-table selection subqueries observed in the query workload. However, unlike Watcher1D, which simply maintains $\mathcal{W}$ as a reservoir sample and uses them in model retraining, SWatcher+ admits/evicts queries to/from $\mathcal{W}$ based on their benefit to query plan efficiency, and proactively corrects the CE model estimates with the true cardinalities monitored in $\mathcal{W}$.
- Cache $\mathcal{F}$ of compensating correction factors. The entries are indexed by the signature of an executed query plan π from which such factors were observed. More precisely, the signature has the form (J, C), where J denotes the set of tables joined by π, and C denotes the (non-empty) set of single-table selection subqueries in $\mathcal{W}$ whose monitored true cardinalities were used to correct model estimates when π was obtained. Given a signature (J, C), $\mathcal{F}$ stores a mapping, denoted $\mathcal{F}_{J,C}$, which maps each join subplan of π joining tables $J' \subseteq J$ to a compensating correction factor denoted $\mathcal{F}_{J,C}(J')$: multiplying $\mathcal{F}_{J,C}(J')$ with the CE (after correcting C) of this subplan joining J' would yield its observed

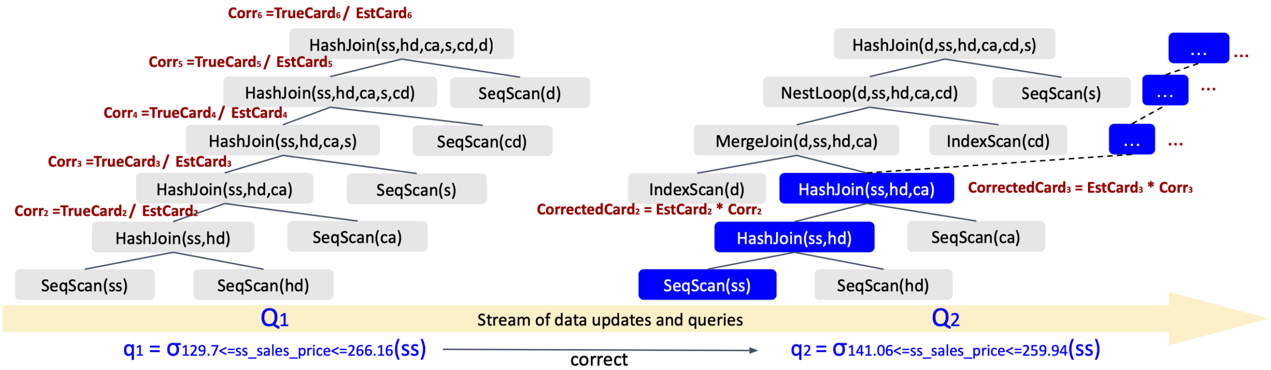

Corr₆ =TrueCard₆ / EstCard₆
HashJoin(ss,hd,ca,s,cd,d)
Corr₅ =TrueCard₅ / EstCard₅
HashJoin(ss,hd,ca,s,cd)
SeqScan(d)
Corr₄ =TrueCard₄ / EstCard₄
HashJoin(ss,hd,ca,s)
SeqScan(cd)
Corr₃ =TrueCard₃ / EstCard₃
HashJoin(ss,hd,ca)
SeqScan(s)
Corr₂ =TrueCard₂ / EstCard₂
HashJoin(ss,hd)
SeqScan(ca)
SeqScan(ss)
SeqScan(hd)
HashJoin(d,ss,hd,ca,cd,s)
NestLoop(d,ss,hd,ca,cd)
SeqScan(s)
MergeJoin(d,ss,hd,ca)
IndexScan(cd)
IndexScan(d)
HashJoin(ss,hd,ca)
CorrectedCard₃ = EstCard₃ * Corr₃
CorrectedCard₂ = EstCard₂ * Corr₂
HashJoin(ss,hd)
SeqScan(ca)
SeqScan(ss)
SeqScan(hd)
...
...
...
...
...
...
Q1
Stream of data updates and queries
Q2
q1 = σ129.7<=ss_sales_price<=266.16(ss)
correct
q2 = σ141.06<=ss_sales_price<=259.94(ss)

Figure 3: SWatcher+: example of correcting both the single-table selection and the join cardinalities.

result cardinality. We also record some basic information with the entry, namely the timestamp when the entry is cached, the query that created this entry, and its execution/planning times.

To process a data update, SWatcher+ essentially follows the same procedure as Watcher1D in Section 3. To process a query, SWatcher+ consults $\mathcal{W}$ and $\mathcal{F}$ to correct any CE model estimates before query optimization. Based on the feedback from query execution, it updates information in $\mathcal{W}$ and $\mathcal{F}$ if applicable. Then, potentially with additional optimizer calls (but no query execution), it decides whether to admit/evict subqueries to/from $\mathcal{W}$. We detail the steps in Section 5.1. As for model retraining, SWatcher+ decides its timing based on the number of watchset queries — a large number indicates that many CE errors are impacting query performance, so retraining may be beneficial. To construct the training set, SWatcher+ uses its watchset queries as in Section 3, but it also adds queries randomly sampled from the workload, which must be executed to acquire true cardinalities. This training set is then used to seed the watchset going forward. We provide details in Section 5.2. Finally, Section 5.3 concludes with cost analysis and discussion.

5.1 Processing Queries

Pre-Execution. Upon receiving a query Q, SWatcher+ first uses the CE model $\mathfrak{m}$ to obtain the cardinality estimates $\hat{\boldsymbol{\kappa}}$. Additionally, however, SWatcher+ corrects $\hat{\boldsymbol{\kappa}}$ using $\mathcal{W}$ and $\mathcal{F}$ as follows. First, for each single-table selection subquery $q = \sigma_p R$ of Q, if we can find a watchset query $\sigma_{p'} R$ in $\mathcal{W}_R$ where p' matches p exactly or approximately, we correct the estimate for q in $\hat{\boldsymbol{\kappa}}$. We say two conditions p and p' approximately match if $\mathrm{sim}(p, p')$, their Jaccard similarity [12] (computed under practical assumptions), is no less than a prescribed threshold θ_{sim}. For example, for range conditions $p = [l, u)$ and $p' = [l', u')$ over the same column of R, we compute $\mathrm{sim}(p, p')$ as $\frac{|\min(u,u') - \max(l,l')|}{|\max(u,u') - \min(l,l')|}$ if p intersects p, or 0 otherwise. If there are multiple watchset queries in $\mathcal{W}_R$ whose conditions match p approximately, we pick the query $\tilde{q}$ with condition $\tilde{p}$ having the highest similarity with p. Then, we correct the model estimate $\widehat{\mathrm{card}}(q)$ as $\widehat{\mathrm{card}}(p \wedge \neg\tilde{p}) + \mathrm{sim}(p \wedge \tilde{p}, \tilde{p}) \cdot \mathrm{card}(\tilde{q})$, where the first term estimates the cardinality for the portion of p outside $\tilde{p}$, and the second term scales the true cardinality for $\tilde{p}$ down to its intersection with p. Note that if p and $\tilde{p}$ match exactly (i.e., $\mathrm{sim}(p, \tilde{p}) = 1$), this correction amounts to replacing $\widehat{\mathrm{card}}(q)$ with $\mathrm{card}(\tilde{q})$.

Second, SWatcher+ further corrects join cardinality estimates in $\hat{\boldsymbol{\kappa}}$ according to $\mathcal{F}$ as needed. Let J_Q denote the set of tables joined by Q and C_Q the set of watchset queries used in the correction step above. Join CE correction may be needed if C_Q is not empty. We look in $\mathcal{F}$ entries indexed by (J_Q, C_Q). If there is no exact match, we settle for a set of indices $\{(J_i, C_i)\}$ in $\mathcal{F}$ such that the J_i's are disjoint, each C_i is exactly C_Q restricted to those over the tables in J_i, and J_Q is maximally covered by the union of the J_i's.[2] Recall that for each (J_i, C_i), $\mathcal{F}_{J_i, C_i}$ remembers the compensating correction factors for subplans joining subsets of J_i. We perform correction in a bottom-up fashion. With corrections to single-table selection CEs made earlier in the first step, for any join involving two tables J' where at least one input cardinality is corrected, we obtain the revised output CE (due to input CE correction), multiply it by $\mathcal{F}_{J_i, C_i}(J')$, the compensating correction factor recorded for J', and use the result as the corrected CE. We then repeat this process for three-table joins, four-table joins, etc., until the join involving all of J_i.

Let $\tilde{\boldsymbol{\kappa}}$ be the final set of cardinality estimates with all corrections (if any), in contrast to the set $\hat{\boldsymbol{\kappa}}$ of uncorrected cardinality estimates. SWatcher+ invokes $\mathrm{Opt}(Q, \tilde{\boldsymbol{\kappa}})$ to obtain the execution plan π, and records its estimated cost, $\mathrm{Cost}(\pi, \tilde{\boldsymbol{\kappa}})$, and planning time, $\mathsf{T}_{\mathrm{Opt}}(Q)$.

Example 5.1. As shown in Figure 3, in the stream of data updates and queries, query Q_2 arrives after Q_1, whose single-selection subquery q_1 is still in $\mathcal{W}$, i.e., admitted and not evicted yet. Suppose that q_1's condition has the highest similarity with q_2's, i.e., $\frac{259.94 - 141.06}{266.16 - 129.7} = 87.12\%$ exceeding the threshold $\theta_{\mathrm{sim}} = 80\%$. First, $\mathrm{card}(q_1)$ is utilized to correct $\widehat{\mathrm{card}}(q_2)$, or equivalently $\hat{\boldsymbol{\kappa}}[\mathsf{ss}]$, obtained by the CE model $\mathfrak{m}$ and enter the corrected CE in $\tilde{\boldsymbol{\kappa}}[\mathsf{ss}]$, where $[\cdot]$ indexes the coordinate corresponding to the query fragment. Second, the CE for $\sigma(\mathsf{ss}) \bowtie \mathsf{hd}$ is then based on the join selectivity between ss and hd estimated by the optimizer stats and $\tilde{\boldsymbol{\kappa}}[\mathsf{ss}]$. This estimate is further refined by the compensating corrections factors in $\mathcal{F}_{\{\mathsf{ss},\mathsf{hd},\mathsf{ca},\mathsf{s},\mathsf{cd},\mathsf{d}\},\{q_1\}}(\mathsf{ss} \bowtie \mathsf{hd})$. It was cached after executing Q_1 with the plan illustrated by the plan tree on the left in Figure 3. Then, we update $\tilde{\boldsymbol{\kappa}}[\mathsf{ss} \bowtie \mathsf{hd}]$. This process is repeated for $\mathsf{ss} \bowtie \mathsf{hd} \bowtie \mathsf{ca}$, $\mathsf{ss} \bowtie \mathsf{hd} \bowtie \mathsf{ca} \bowtie \mathsf{s}$, and beyond,

[2] This join correction procedure tries to ensure some degree of consistency when applying corrections. Inconsistency and over-correction may arise if we correct one join cardinality using factors cached from multiple previous executions; therefore, we require the J_i's be disjoint. Furthermore, because the observed correction factors are highly dependent on the set of corrected selections, we require C_i to be consistent with the selection corrections applied to Q.

using the compensating corrections factor stored in the the index $(\{\mathsf{ss}, \mathsf{hd}, \mathsf{ca}, \mathsf{s}, \mathsf{cd}, \mathsf{d}\}, \{q_1\})$. At the end, $\tilde{\boldsymbol{\kappa}}$, corresponding to the bottom-up path in blue on the right plan tree, is injected to get the optimal plan $\mathrm{Opt}(Q_2, \tilde{\boldsymbol{\kappa}})$. Notably, as shown in Figure 3, it is not necessarily the same as the plan for Q_1, even though the correction factors are derived from it.

Post-Execution. By observing the execution of π, SWatcher+ first performs some basic bookkeeping. Let $\mathsf{T}_{\mathsf{Exec}}(\pi)$ denote the total execution time. If any correction was done, we update the cache $\mathcal{F}$ of compensating correction factors: a new mapping $\mathcal{F}_{J_Q, C_Q}$ will be created, replacing any previous mapping keyed by the same signature. For each join subplan of π joining tables $J' \subseteq J_Q$, we record $\mathcal{F}_{J_Q, C_Q}(J')$, the compensating correction factor for J', as the observed true result cardinality of the join subplan, divided by the estimate with selection corrections in C_Q but no join corrections otherwise. We also record some other basic information with the entry, including the creation timestamp, the query Q, and its observed execution/planning times.

Next, for each single-table selection subquery $q = \sigma_p R$ of Q, SWatcher+ decides whether to admit q as a watchset query (if $q \notin \mathcal{W}$) or evict it (if $q \in \mathcal{W}$). Conceptually, q is worth monitoring if correcting its CE by its true cardinality leads to a faster query plan by a large margin — at least a speedup of θ_{gain}. Now that we already have q's true cardinality from the execution (or by an extra counting query if needed, as discussed in Section 3), we could test this admission condition by reoptimizing Q with this correction to q, executing the new plan, and measure its speedup over the plan obtained by using no CE correction. However, the overhead would be unacceptable. Instead, we estimate the potential improvement without any new query execution. To further reduce the need to reoptimize, we heuristically apply quick checks using two thresholds to eliminate any Q not worthy of monitoring: $\theta_{\mathsf{T}_{\mathsf{Exec}}}$, minimum execution time for Q, and $\theta_{\mathsf{T}_{\mathsf{Opt}}}$, maximum optimization time for Q. Intuitively, there is little benefit to meddle with a query that already runs fast, and there is too much overhead to meddle with a query whose optimization takes long. The details are explained below.

Suppose q is not a watchset query. 1) Quick checks: If $\mathsf{T}_{\mathsf{Exec}}(\pi) < \theta_{\mathsf{T}_{\mathsf{Exec}}}$ or $\mathsf{T}_{\mathsf{Opt}}(Q) > \theta_{\mathsf{T}_{\mathsf{Opt}}}$, do not admit q. 2) Speedup check: Let $\boldsymbol{\kappa}^{\star}$ denote the set of true cardinalities observed from the execution of π, and $\hat{\boldsymbol{\kappa}}$ the set of model-estimated cardinalities. Let $\boldsymbol{\kappa}'$ denote the set of estimates formed by the observed true cardinalities for q and all join subplans of π involving q, plus uncorrected cardinality estimates for all other single-table subplans. We invoke the optimizer to obtain the baseline plan $\hat{\pi} = \mathrm{Opt}(Q, \hat{\boldsymbol{\kappa}})$, as well as the plan $\pi' = \mathrm{Opt}(Q, \boldsymbol{\kappa}')$, which would be obtained by admitting q (alone). Comparing their costs under $\boldsymbol{\kappa}^{\star}$ yields the estimated speedup: $\mathsf{gain}(q) = \mathrm{Cost}(\hat{\pi}, \boldsymbol{\kappa}^{\star}) / \mathrm{Cost}(\pi', \boldsymbol{\kappa}^{\star})$. We admit q if and only if $\mathsf{gain}(q) \geq \theta_{\mathsf{gain}}$. When admitting q into $\mathcal{W}$, we record its observed true cardinality $\mathsf{card}(q)$ and the speedup $\mathsf{gain}(q)$.

Suppose q is a watchset query. We proceed with the following steps. 1) quick checks: If $\mathsf{T}_{\mathsf{Exec}}(\pi) \cdot \mathsf{gain}(q) < \theta_{\mathsf{T}_{\mathsf{Exec}}}$ or $\mathsf{T}_{\mathsf{Opt}}(Q) > \theta_{\mathsf{T}_{\mathsf{Opt}}}$, evict q. Note that $\mathsf{T}_{\mathsf{Exec}}(\pi) \cdot \mathsf{gain}(q)$ estimates the running time without the benefit of watching q using its last recorded speedup. 2) Speedup check: We proceed to compute $\mathsf{gain}(q)$ in the exact way described above, and evict q if and only if $\mathsf{gain}(q) < \theta_{\mathsf{gain}}$. If q is evicted, we also purge the entries in $\mathcal{F}$ indexed by any (J, C) where $q \in C$. If q remains in $\mathcal{W}$, we update $\mathsf{gain}(q)$.

5.2 Retraining CE Model

After admitting a new watchset query, SWatcher+ checks the size of the watchset. If $|\mathcal{W}|$ exceeds a prescribed threshold $\theta_{n_{\mathsf{watch}}}$, we trigger retaining of the CE model $\mathfrak{m}$. To obtain the training set, we first add all watchset queries with their up-to-date true cardinalities, as in Section 3. Since the watchset is small and no longer an unbiased sample of the query workload in this case, we augment the training set by randomly sampling queries from the workload, until we reach n_{train}, the desired size of the training set. We assume SWatcher+ has access to some representation of the query workload to sample from, such as a query log.[3] However, since these queries have not been monitored, we must execute them over the current database state to acquire true cardinalities for training.

After retraining $\mathfrak{m}$, SWatcher+ re-initializes $\mathcal{W}$ to reflect the new $\mathfrak{m}$. For each single-table selection subquery q of a query Q involved in the training set, we decide whether to watch q using a process nearly identical to that discussed in Section 5.1 — specifically the case when q is not yet a watchset query. Ideally, for the quick checks, we would like to know $\mathsf{T}_{\mathsf{Exec}}(\pi)$ and $\mathsf{T}_{\mathsf{Opt}}(Q)$, where π is the query plan obtained under the new CE model. To avoid another query execution, however, we simply use earlier observed timings as surrogates (in any case, SWatcher+ will eventually reexamine the decision when updated information is available). For a subquery q that was in $\mathcal{W}$ prior to retraining, we look for the entry in $\mathcal{F}$ indexed by (J, C) where $q \in C$ and the timestamp is the latest, and use its recorded query as Q and timings for $\mathsf{T}_{\mathsf{Exec}}(\pi)$ and $\mathsf{T}_{\mathsf{Opt}}(Q)$. For a query Q added to the training set by workload sampling, we use the timings when running Q to acquire training data earlier. Finally, we clear the cache $\mathcal{F}$.

Note that re-initialization of $\mathcal{W}$ does not guarantee that $|\mathcal{W}|$ fall below $\theta_{n_{\mathsf{watch}}}$ following retraining. In the unlikely event where the threshold is immediately violated, we apply exponential backoff on $\theta_{n_{\mathsf{watch}}}$, similar to the process described in Section 3.

5.3 Cost Analysis and Discussion

Like Watcher1D and Watcher2D, the cost of SWatcher+ has three components. First, SWatcher+ greatly reduces the cost of processing data updates. Not only does it avoid the costly maintenance of join cardinalities as in Watcher2D, but it is also more selective than Watcher1D in what selection cardinalities to maintain. For example, the watchset size of SWatcher+ is typically 90% less than Watcher1D 's in our experiments in Section 6. Second, for the cost of of processing queries, the extra overhead in making CE corrections using $\mathcal{W}$ and $\mathcal{F}$ and bookkeeping is minimal. However, for a query Q containing k single-table selection subqueries, the speedup check in deciding whether these selection subqueries are worth watching can result in up to $k+1$ $\mathrm{Opt}(\cdot)$ calls (one to get the baseline plan $\hat{\pi}$ and one for correcting each subquery) plus $k+1$ $\mathrm{Cost}(\cdot)$ calls (to estimate their costs under true cardinalities). Luckily, in practice, the quick checks help eliminate 40% to 80% of these calls, as observed in our experiments in Section 6. Note that an extra counting query may be also needed if the plan executed did not use a selection subquery, but such cases are rare, as discussed in

[3] In the worst case, SWatcher+ can separately maintain a reservoir sample of the past queries. In this case, unlike Section 3, since we do not maintain their cardinalities, the overhead is negligible.

Section 3. On the other hand, by proactively applying CE corrections, SWatcher+ hopes to improve query plans and reduce query execution times more effectively than Watcher1D and Watcher2D. Third, retraining of the CE model is more expensive in SWatcher+ than Watcher1D and Watcher2D, because SWatcher+ may need to execute additional queries to construct a large enough training set, plus additional optimizer calls (linear in the size of the training set) to re-initialize the watchset. Section 6 will demonstrate the overall effectiveness of SWatcher+ in comparison with other approaches.

Although SWatcher+ has been presented with several parameters, including θ_{sim}, $\theta_{\text{T}_{\text{Exec}}}$, $\theta_{\text{T}_{\text{Opt}}}$, θ_{gain}, and $\theta_{n_{\text{watch}}}$, we note that setting them is straightforward and does not require extensive tuning. Specifically, users can leave θ_{sim}, $\theta_{\text{T}_{\text{Exec}}}$, $\theta_{\text{T}_{\text{Opt}}}$, and θ_{gain} at their default settings, discussed in detail in Section 6, though we at least examine an alternative setting of θ_{gain} to evaluate the sensitivity of SWatcher+ to its setting. Currently, setting of $\theta_{n_{\text{watch}}}$ is primarily driven by users' desire to cap the overhead in processing data updates. We believe it is possible to extend SWatcher+ to avoid this parameter altogether: with no bound on $|\mathcal{W}|$, we would control the watchset contents by tracking both the benefit (to query performance) and cost (incurred for data updates) of each watchset query to ensure positive net benefit over time; model retraining would then be forced if "thrashing" of $\mathcal{W}$ is observed. We leave further investigation as future work.

6 EXPERIMENT

6.1 Setup

We implemented the Watcher algorithms in Python3 and evaluated their performance on the DSB benchmark [3]. PostgreSQL V16.2 is used as the default cost-based query optimizer, with cardinality injection implemented using the methods in [8, 47], and plan hinting facilitated by `pg_hint_plan` [30]. For the black-box learned cardinality estimator, we select MSCN [14], a popular query-driven estimator known for its low model complexity, fast inference times, adequate estimation accuracy, and support of join cardinality estimation. These characteristics, as demonstrated in [8, 43], make MSCN a suitable choice for our framework. Execution times are measured as the median of three repetitions throughout our experiments to reduce noise. All experiments are conducted on a Linux server with 8 Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz processors and 1TB of disk storage. Our codes are available at https://github.com/louisja1/Watcher-VLDB25.

Data updates. The original tables are generated with a scaling factor of 2, while all other parameters are set to the default values. Under the DSB schema, we identify the following fact tables as dynamic tables, each associated with a timestamp attribute: tables `ss, cs, ws` with attributes `*_sold_date_sk`; tables `sr, cr, wr` with attributes `*_returned_date_sk`, and table `inv` with attribute `inv_date_sk`. These timestamp attributes reference the primary key `d_date_sk` in table `date_dim`, representing the surrogate key for dates. To simulate a dynamic workload, these tables are modified by implementing a shifting 60-day window: tuples corresponding to the oldest date are deleted, and tuples for the next date are inserted. This approach naturally leverages the inherent data skewness in the DSB benchmark to simulate data shifts during updates. The database instance started with `*_date_sk between 2450815 and 2450874`. After all data updates, the final database reflects a 60-day forward shift with `*_date_sk between 2450875 and 2450934`, representing a completely different state. The dimension tables remain unchanged throughout. The third column of Table 1 lists the number of data updates.

Table 1: query templates and the associated information. The templates cover a wide range of features, including FK-FK joins, inequality joins, band joins, and self-joins. †: the percentiles (P50, P75) and averages (Avg.) are measured over the initial query set. $\theta_{\text{T}_{\text{Exec}}}$ is the configuration for SWatcher+. The last three columns are presented in ms. $*$: adjusted based on observations.

Template	# Tables	# DUs	P50/P75 T_{Exec}†	Avg. T_{Opt}†	$\theta_{\text{T}_{\text{Exec}}}$
Q013a	6	0.30M	459/542	3	400
Q013b	6	0.30M	483/740	3	400
Q018	6	0.14M	465/487	3	450
Q019	6	0.30M	206/297	3	250
Q040	5	0.14M	2/2	3	700*
Q072	9	1.30M	435/665	16	700
Q085a	8	0.08M	1/2	8	5
Q085b	8	0.08M	1/4	8	5
Q099	5	0.14M	54/83	2	80
Q100	8	0.30M	928/1260	7	600
Q101	9	0.41M	38/46	67	50
Q102	9	1.55M	45/47	95	50
TJQ013a	6	0.30M	80/143	3	400
TJQ013b	6	0.30M	96/472	4	400

Query templates. We modify the single-block SPJ templates from DSB [4] as follows: (i) filtered out templates that did not involve any dynamic tables or lacked selection conditions on them; (ii) limited the number of selection conditions to at most two, ensuring query performance was significantly influenced by the cardinality estimation of these conditions; and (iii) removed templates with complex predicates such as `LIKE` or disjunctions. These modifications resulted in 12 query templates (denoted with a 'Q,' prefix) for single-selection experiments and 2 query templates (denoted with a 'TJQ,' prefix) for two-table join experiments. The numerical IDs of the templates are retained from DSB. A summary of these templates is provided in Table 1, and the full SQLs can be found in [25]. Query instances generated from these templates differ only in the parameters of their range predicates, which are randomly and uniformly sampled from the domain of the selection columns. For instance, a query from Q100 includes the selection `ss_sales_price / ss_list_price between 0.49 and 0.89`.

Initial query set & dynamic workload. We assume a fixed query template for each experimental trial. This setup ensures that the CE model focuses on a single template, isolating data updates are the primary cause of CE model regression. Under this assumption, we first generate 1500 queries as the initial query set, as well as the associated information including single-table true cardinalities (required by Watcher1D, SWatcher+), two-table join true cardinalities (required by Watcher2D), as well as the execution time, planning time, and plan cost for executing the entire query on the initial database instance (required by SWatcher+). Then, we generate 3000 queries (or Qs in short) for the dynamic workload and explore three different mixing rates with data updates (or DUs in short):

- 6-batch setting (e.g., Q013a, TJQ013b): six uniform batches of DUs-then-Qs, concretely 6 × 10-days DUs then 500 Qs;
- 3-batch setting (e.g., Q013a-3B, TJQ013b-3B): three uniform batches of DUs-then-Qs, concretely 3 × 20-days DUs then 1000 Qs;

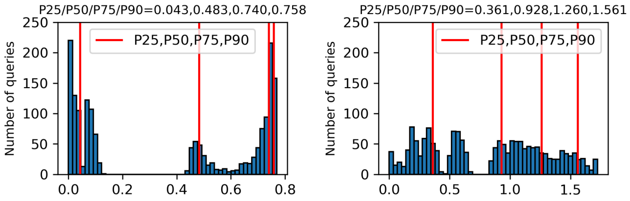
Moreover, for each of these workloads, we pre-compute the true cardinality for each query and record the time cost of the corresponding counting query. This ensures that whenever an algorithm requires it but does not obtain it for free, it can access the pre-computed value with the cost.

Algorithms & parameters setup. The training set size is set to be 1500 for all cases. All algorithms monitor the cardinality estimates over the selection(s) in each query template. The following algorithms are evaluated:

- **True**: it is an idealized (but practically impossible) algorithm that always injects true cardinalities to the monitored parts, with no accounting of overhead.
- **Baseline**: it relies entirely on the initial CE model's estimates, which remain unchanged and never retrained.
- **Periodic**: it simulates periodic retraining of the CE model after every X-days of data updates, where $X \in \{10, 20, 30\}$ captures scenarios of timely or delayed retraining. For instance, in a 6-batch setting, $X = 10$ reflects an ideal scenario where retraining is triggered immediately after every batch of data updates, while $X = 20$ means that the 1st, 3rd, and 5th query batches are handled by CE models affected by 10-days data updates.
- **Watcher1D**: it sets $n_{\text{watch}} = n_{\text{test}} = n_{\text{train}} = 1500$, and ϵ is $\{90, 95\}$-th percentile of the Q-error in the initial query set.
- **Watcher2D**: it shares these configurations and leverages LevelDB [7], an on-disk key-value store package, for composite key-to-count indices, with both the fan-out and the maximum levels set to 5. Further details are available in [25].
- **SWatcher+**: we have a sensitivity analysis (more details in [25]) that does not show much difference in the choice of θ_{sim}, $\theta_{\mathrm{T}_{\text{Opt}}}$, $\theta_{\mathrm{T}_{\text{Exec}}}$, θ_{gain}, and $\theta_{n_{\text{watch}}}$ across all the templates. We only show part of the analysis in Tables 2 and 3 for Q013b. And we stick to the setup $\theta_{\text{sim}} = 80\%$, $\theta_{n_{\text{watch}}} = 20\%$ the number of queries in the workload, $\theta_{\mathrm{T}_{\text{Opt}}} = 10\% \cdot \theta_{\mathrm{T}_{\text{Exec}}}$, $\theta_{\text{gain}} = 101\%$, $\theta_{\mathrm{T}_{\text{Exec}}}$ as detailed in Table 1 and with correction on two-table join subqueries, which yields the best in overall. Notably, $\theta_{\mathrm{T}_{\text{Exec}}}$ is chosen as a clear boundary between slow and fast queries when such a distinction is evident, as illustrated in Figure 4. For relatively uniform cases, $\theta_{\mathrm{T}_{\text{Exec}}}$ is set around the 50th or 75th percentile. In Q040, about 20% of queries exceed 700ms after processing data updates, so we increase $\theta_{\mathrm{T}_{\text{Exec}}}$ accordingly.

6.2 Results on End-to-End Runtime

For each single-selection query template, we measure the end-to-end runtime for each algorithm and report the results using their configuration optimized for the lowest runtime. Notably, we report the Periodic with the best-performing X setting observed post hoc, which is impractical in real deployments. Moreover, achieving the best trade-off between the execution time and overhead requires different X-values for different templates in our experiments. We present the results for the 6-batch setting in Figure 5, and those for the 3-batch setting in [25]. In Figure 5, the end-to-end runtimes are all normalized against Baseline's runtime, clearly comparing performance improvements. The results are summarized into four groups based on key observations (later, we will focus on Group 1 and Group 2 respectively):



(a) Q013b **(b) Q100**

Figure 4: Query latency (sec.) in the initial query set for Q013b and Q100 (only the fast 95% is displayed in the histograms).

Group 1: Q013a, Q013b, Q100. These templates exhibit a noticeable performance gap of approximately 10% between True and Baseline, indicating that injecting true cardinalities into the monitored subqueries of these templates significantly improves query execution times. Among all algorithms, SWatcher+ achieves the lowest end-to-end runtime, outperforming Baseline, Periodic, and Watcher1D. By applying corrections to the CE model's estimates rather than retraining, SWatcher+ achieves total query execution times that are often closer to those achieved with true cardinalities injected than those obtained using more up-to-date CE models from Periodic. Conversely, Watcher1D incurs much higher overhead due to the cost of maintaining a large $\mathcal{W}$, which outweighs the savings in query execution times. A detailed analysis is provided through an ablation study of various parameters and a breakdown of componentized overhead for each algorithm in Section 6.3.

Group 2: Q018, Q040, Q072. For this group, injecting true cardinalities into single-selection subqueries can occasionally backfire. This happens because PostgreSQL's optimizer, while relying on both the injected cardinality estimates (CEs) and unrealistic assumptions about join selectivity, sometimes produces even less accurate join cardinality estimates when the injected CEs are more precise. Strategies such as Periodic and Watcher1D face challenges in avoiding these backfire scenarios, as their primary goal is to improve the accuracy of single-selection subquery CEs. However, SWatcher+ effectively mitigates this issue by also correcting the join cardinality estimates through the cache $\mathcal{F}$, underscoring the importance of this design. Notably, the performance of Periodic is evaluated in an "infrequent retraining" setup (i.e., retraining only once), which results in less accurate CEs but fortuitously achieves shorter query execution times. Similarly, Watcher1D benefits from avoiding excessive retraining on the CE model. This explains why both Periodic and Watcher1D achieve lower or comparable end-to-end runtimes for Q018 and Q040 when compared to SWatcher+.

In short, combining join corrections with more accurate selection CEs is crucial. Neither Periodic nor Watcher1D can detect backfire cases in advance and adjust retraining frequency accordingly. Further discussion is presented in Section 6.4.

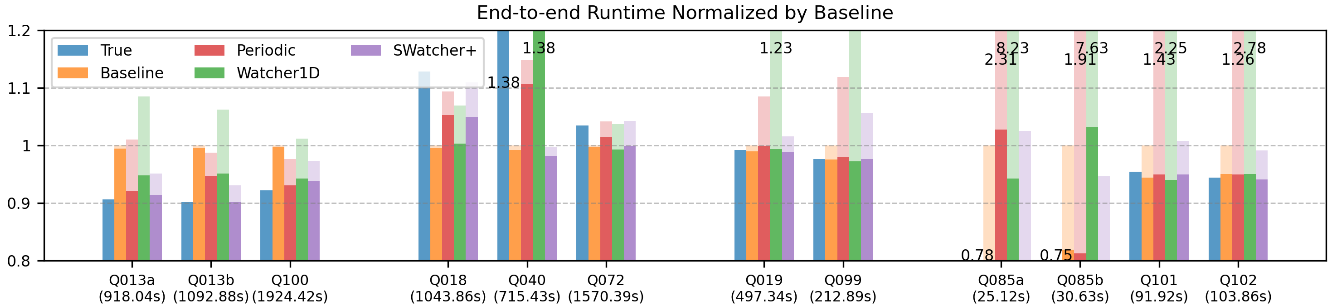


Figure 5: the normalized end-to-end runtime, including both total query execution time (represented by the lower portion of the bar with darker shade) and overhead (represented by the upper portion with lighter shade), across all algorithms and single-selection templates. A number is attached to a bar if its value falls outside the scope. The number below the template name is the end-to-end runtime of Baseline, used for normalization.

Group 3: Q019, Q099. This group reflects cases where query performance is relatively insensitive to the accuracy of CEs, leading to similar total query execution times across all algorithms. Regarding the overhead cost, SWatcher+ achieves a 5% to 20% lower overhead compared to Periodic and Watcher1D. For instance, in Q019, SWatcher+ does not add any items to its $\mathcal{W}$. In Q099, it maintains a small $\mathcal{W}$ with only 2 items by the end of the dynamic workload, applying corrections to 1.17% of queries. However, the overhead from $\mathcal{W}$ admission checks results in SWatcher+ having a slightly larger overhead than Baseline in this case. On the other hand, the retraining strategies in Periodic and Watcher1D are predetermined and inflexible, often leading to unnecessary retraining that adds overhead without noticeable performance gains.

Group 4: Q085a, Q085b, Q099, Q100. This group comprises queries that can finish in a very short time throughout the dynamic workload, with total query execution times of $\leq$ 100 seconds, consistent with the observations from their initial query sets. Additionally, planning time is observed to be 2× to 4× the execution time for each query. In this scenario, SWatcher+ performs equivalently to Baseline as no queries pass the admission checks, avoiding most of the additional overhead. However, both Periodic and Watcher1D incur non-trivial overheads due to unnecessary retraining. For instance, in Q101, Periodic spends 39.18 seconds for a single retraining cycle (including collecting the retraining set). Watcher1D takes 20.51 seconds for a single retraining cycle and 99.78 seconds for maintaining $\mathcal{W}$. These additional costs exceed the total query execution time, which remains under 90 seconds in all cases.

6.3 Group 1: Analysis on Q013b

We extend the observations from **Group 1** to Q013b. First, we perform a componentized runtime analysis, shown in Table 2, across all algorithms with varying parameters. For Periodic, triggering retraining after every batch of DUs ($X = 10$) achieves the lowest total query execution time – over 20 seconds shorter than the other X-values – though it incurs longer retraining time. For Watcher1D, using P95 saves two retraining operations (over 40 seconds) compared to P90 but results in a 17-second slower total execution time. Both configurations exhibit high DU overhead due to maintaining a $\mathcal{W}$ of 1500 subqueries. For SWatcher+, two-table join subqueries corrections yield significantly better savings (25+ seconds) compared to smaller plan cost gaps (<10 seconds). Its query processing overhead is 5× higher than Baseline, Periodic and Watcher1D, owing to the corrections and admission-eviction checks. But, its overhead on processing DUs is low, with $|\mathcal{W}| < 50$ in all cases.

Excluding overhead on DUs, Watcher1D achieves lower runtime than Periodic when both perform the same numbers of retraining, suggesting that Watcher1D identifies better-retraining timings despite high watchset maintenance costs. This highlights the advantage of SWatcher+, which replaces retraining with corrections. Finally, Watcher1D requires less retraining time than Periodic by skipping the collection of new training set through counting queries, but this does not justify its high $\mathcal{W}$ cost.

Next, as discussed previously in Section 1, the performance of certain queries is not necessarily impacted by injecting inaccurate CEs, which is outside the focus of SWatcher+. To analyze how CEs affect the query performance, we evaluate on a per-query basis. Specifically, as shown in Figure 6, the “ideal saving” for each query indicates the potential time saving by injecting true cardinality against injecting the CE from Baseline. We also order these 3000 queries by their “ideal saving” in the figure. The results show that a few queries experience backfire, the leftmost 5%, while the majority remain insensitive to CE accuracy. Only the rightmost 10% of queries, referred to as the *(potentially rewarding)* region, stand to benefit from accurate CEs, with potential savings of at least 49 ms per query. This potent region is exactly the target of SWatcher+. As demonstrated in the last column of Table 2, SWatcher+ achieves the lowest total query execution time for this region.

Third, we present the 50th, 75th, 90th, and 95th percentiles of Q-errors observed from the CEs over the 3000 queries in the dynamic workload across all algorithm setups in Table 3. Watcher1D, with ϵ =P90, achieves the lowest median and P95 Q-errors, showcasing the monitoring effectiveness of $\mathcal{W}$, though relaxing ϵ from P90 to P95 reduces the accuracy of CEs. Meanwhile, Periodic maintains better control over P75 and P90 Q-errors through periodic retraining. Overall, SWatcher+ exhibits less accurate CEs across all percentiles compared to Periodic and Watcher1D. Despite this, the lower Q-errors compared to Baseline demonstrate that SWatcher+'s simple design for correcting single-table selection subqueries is indeed

Table 2: Q013b: the componentized runtimes. †: the parameters, shared within the same algorithm and detailed in Section 6.1, are omitted. The last four columns are presented in seconds. The "Retraining Time" column includes the number of retraining triggered, shown in parentheses. Highlighted rows achieve the lowest end-to-end runtime for each algorithm and serve as the representatives in Figure 5.

Algorithm	Retrain if†	Correct if	Total $T_{\text{Exec}}(Q)$	Retraining Time	Overhead on Qs	Overhead on DUs	Total $T_{\text{Exec}}(Q)$ in potent region
True	**-**	**-**	**985.08**	**0(0)**	**0**	**0**	**75.54**
Baseline	**-**	**-**	**1087.64**	**0(0)**	**5.23**	**0**	**154.53**
Periodic	X=10	-	1014.33	171.68(6)	5.85	-	109.33
Periodic	X=20	-	1042.61	77(3)	5.93	-	118.13
Periodic	**X=30**	**-**	**1035.24**	**38.04(1)**	**5.88**	**-**	**109.33**
Watcher1D	-	ϵ=P90	1022.06	62.98(3)	5.42	92.22	115.40
Watcher1D	**-**	**ϵ=P95**	**1039.64**	**20.7(1)**	**5.53**	**95.37**	**128.30**
SWatcher+	single only	$\theta_{\text{gain}} = 110\%$	1021.89	0(0)	28.18	3.70	110.13
SWatcher+	plus 2-join	$\theta_{\text{gain}} = 110\%$	989.53	0(0)	26.71	3.42	95.32
SWatcher+	single only	$\theta_{\text{gain}} = 101\%$	1011.96	0(0)	27.57	5.31	99.92
SWatcher+	**plus 2-join**	$\theta_{\text{gain}} = 101\%$	**985.67**	**0(0)**	**26.31**	**4.88**	**90.68**

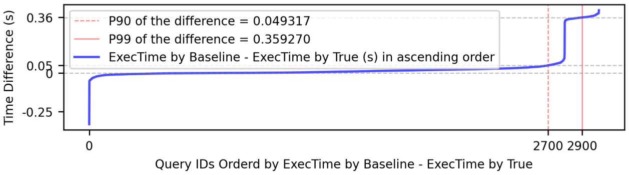


Figure 6: Q013b: "ideal saving" computed and ordered by $T_{\text{Exec}}(Q)$ by Baseline - $T_{\text{Exec}}(Q)$ by True.

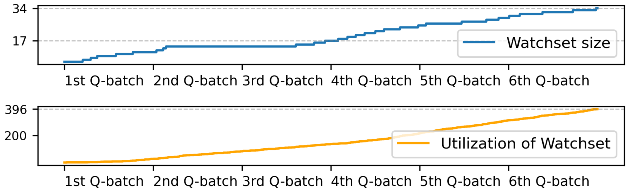


Figure 7: Q013b: growth (upper sub-figure) and utilization (lower sub-figure) of $\mathcal{W}$.

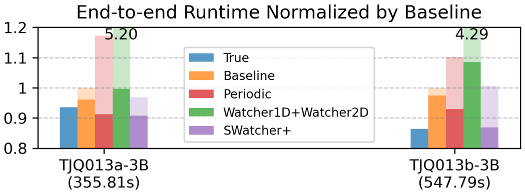


Figure 8: the normalized end-to-end runtime, including both total query execution time (lower/darker portion of the bar) and overhead (upper/lighter portion), across all algorithms and two-table join subqueries.

effective. Notably, retraining does not always yield a more accurate CE model. For instance, in Periodic, the difference between $X = 20$ and $X = 10$ setups lies in the latter having three additional retraining after the DUs in the 1st, 3rd, and 5th batches, whereas the former continues using potentially outdated CE models, trained and deployed one batch earlier, for answering the queries in those batches. This further underscores the importance of SWatcher+'s approach – delivering more accurate CEs precisely where they are needed while avoiding unnecessary retraining.

Table 3: Q013b: percentiles of observed Q-errors on CEs. We embolden the lowest error for each column.

Algorithm	Retrain if†	Correct if	P50	P75	P90	P95
Baseline	-	-	1.46	1.91	2.67	3.35
Periodic	X=10	-	1.27	1.52	**1.87**	2.47
Periodic	X=20	-	1.25	**1.51**	2.01	2.41
Periodic	X=30	-	1.31	1.65	2.19	2.67
Watcher1D	-	ϵ=P90	**1.21**	1.53	1.96	**2.31**
Watcher1D	-	ϵ=P95	1.25	1.65	2.19	2.65
SWatcher+	single only	$\theta_{\text{gain}} = 110\%$	1.39	1.72	2.16	2.78
SWatcher+	plus 2-join	$\theta_{\text{gain}} = 110\%$	1.40	1.75	2.20	2.83
SWatcher+	single only	$\theta_{\text{gain}} = 101\%$	1.37	1.70	2.15	2.72
SWatcher+	plus 2-join	$\theta_{\text{gain}} = 101\%$	1.38	1.72	2.15	2.74

Forth, Figure 7 presents the growth and utilization of SWatcher+ under the specific setup "plus 2-join" and "$\theta_{\text{gain}} = 101\%$". Overall, SWatcher+ selects 1% of single-table selection subqueries, leading to corrections in 13% of queries within the dynamic workload. Notably, SWatcher+ corrects 58% of all queries in the potent region and 89% of those are among the top 1% most potent.

6.4 Group 2: Backfire Templates

In Q018, Q040, and Q072, respectively, 16.2%, 33%, and 19.87% of queries experience a slowdown of at least 50ms per query due to injecting true CEs. Meanwhile, 83%, 66.9%, and 75% queries remain insensitive to the injected CEs, while only a small fraction of queries speed up when true CEs are injected. The primary reason for this backfire effect is the lack of accurate estimation for other subqueries without injections. More concisely, injecting true CEs into single-selection subqueries sometimes worsens join CEs, leading to inaccurate plan cost estimation and suboptimal plan selection. SWatcher+ mitigates the issue in two ways. First, as observed in Q040, the admission-eviction checks effectively determine that injecting more accurate CEs would not improve performance, avoiding caching ineffective subqueries in $\mathcal{W}$ and preventing unnecessary corrections. Second, as demonstrated in Q018 and Q072, SWatcher+ further applies corrections to join CEs, counteracting the inaccuracies propagated from injecting single-selection subqueries solely and guiding plan selection towards the right track. Overall, within the backfire region, SWatcher+ achieves end-to-end runtime savings compared to True: 53 seconds (22.3%) in Q018, 283 seconds (98.2%) in Q040, and 67 seconds (23.5%) in Q072. More concretely, in Q018, about 89.1% of queries receive corrections in both single-selection and two-table join subqueries, contributing 73.6% of the total savings.

6.5 Results on TJQ

TJQ013a-3B and TJQ013b-3B involve selections on tables ss and hd, which are joined via ss_hdemo_sk = hd_demo_sk. We inject CEs to both single-selection subqueries, $\sigma(\texttt{ss})$ and $\sigma(\texttt{hd})$, as well as the two-table join subquery, $\sigma(\texttt{ss}) \bowtie \sigma(\texttt{hd})$. True injects true cardinalities into all three places. In contrast, Baseline, Periodic, and Watcher2D+Watcher1D train separate CE models for each subquery, while SWatcher+ only trains CE models for $\sigma(\texttt{ss})$ and $\sigma(\texttt{hd})$. This separation prevents interference between query fragments. For all three instances of Periodic, we set $X = 30$, while Watcher1D+Watcher2D is configured with ϵ =P90. SWatcher+ runs in two instances: one with "plus 2-join" and the other with $\theta_{\texttt{gain}} = 101\%$, collaborating to correct $\sigma(\texttt{ss}) \bowtie \sigma(\texttt{hd})$ using $\mathcal{F}$. End-to-end runtime results in Figure 8 show that SWatcher+ outperforms Periodic and Watcher1D+Watcher2D, achieving a total query execution time close to or even better than True's. This suggests that even injecting true CEs for the two-table join subquery does not always yield the optimal plan selection, whereas SWatcher+ benefits from additional join corrections beyond $\sigma(\texttt{ss}) \bowtie \sigma(\texttt{hd})$. Maintaining $\mathcal{W}$ in Watcher2D accounts for 86% of the overhead of Watcher1D and Watcher2D, exceeding 60% of the total end-to-end runtime for both query templates. This underscores the impracticality of maintaining highly accurate join cardinalities during query optimization, even for simple two-way joins.

7 RELATED WORK

CE methods are categorized into data-driven and query-driven. Data-driven methods like sampling and histograms [18, 23, 35, 36] struggle with high-dimensional data and joins. ML-based methods [9, 10, 45, 48–50] improve accuracy but are costly. Query-driven methods focus on frequently queried regions, with non-ML methods like Isomer [2, 40] assuming uniform distributions, and ML methods [9, 13, 33, 41, 44] modeling query-cardinality relationships, such as QuickSel [34] refining estimates using probability density functions. Hybrid approaches, like UAE [45], combine data and query-based training. Theoretical work shows that selectivity functions with bounded VC dimensions are learnable [11].

Dynamic workloads with tuple updates and query drift, require frequent retraining [5, 24, 43, 49]. DeepDB [10] absorbs data updates, while QuickSel [34] optimizes models based on new query feedback. UAE [45] fine-tunes the model incrementally. Recent approaches improve robustness but introduce trade-offs. Warper [19] detects drift using thresholds and employs GANs to generate new training queries, though at the cost of execution overhead. RobustMSCN [32] periodically retrains with lower frequency, but this increases storage overhead. ALECE [22] adapts to updates without retraining, although its effectiveness depends on minimal data drift. DDUp [15] focuses solely on detecting and mitigating data drift for neural network-based CE models. Wu et al. [46] address workload drift using a replay buffer. While these methods primarily focus on maintaining CE accuracy, retraining remains necessary for significant data drift. Watcher, in contrast, dynamically adapts black-box estimators to balance query performance and overhead.

8 CONCLUSION AND FUTURE WORK

In this paper, we studied the problem of deploying learned CE models in practical, dynamic settings. As discussed in the introduction, we use a holistic approach for both design and evaluation. This approach has revealed numerous challenges. For instance, as demonstrated in our experiments, even for a synthetic workload like DSB with predictable updates and query patterns, determining the optimal time for retraining the model remains difficult. A common practice is periodic retraining, but its effectiveness heavily depends on the chosen interval. Setting this interval optimally is challenging without hindsight, and periodic retraining often fails to respond in real-time to evolving data and workload characteristics.

Our results further highlight a fundamental challenge in CE: single-table selection CE is generally easier, and maintaining true cardinalities for single tables is more straightforward. However, focusing on single-table estimates while neglecting joins can be counterproductive and, in some cases, lead to significant errors. On the other hand, we show that SWatcher+ effectively reduces overhead by judiciously deciding what to maintain through a benefit-based analysis. Simultaneously, it enhances query optimization by leveraging both maintained cardinalities and cached correction factors. By design, SWatcher+ automatically identifies hard or high-impact queries that warrant dedicated monitoring efforts, better handles backfiring queries caused by over-reliance on one-dimensional corrections, and avoids unnecessary retraining. Our experiments demonstrate that SWatcher+ performs well even without the benefit of hindsight—it outperforms periodic retraining, even when the latter is given an optimal post hoc interval. Furthermore, it delivers reasonable performance even in challenging cases where the benefit margin is razor-thin—a scenario commonly encountered in our experiment based on DSB.

There are several directions for further improving SWatcher+. As noted in Section 5, monitoring update costs more effectively could help refine its efficiency. Additionally, implementing more efficient batch-based processing (rather than row-at-a-time processing) may further reduce update overhead. Evaluating SWatcher+ on more challenging workloads - where both data and query patterns evolve in less predictable ways than in standard benchmarks such as DSB - could better showcase its adaptability. While there is still much room for improvement, we believe our holistic approach is crucial for the practical deployment of learned CE models. The key principles behind SWatcher+ - selective monitoring, active CE correction, and a focus on real query performance rather than raw CE error metrics - offer valuable insights that can inform future advancements in this area.

REFERENCES

[1] Cecilia R Aragon and Raimund Seidel. 1989. Randomized search trees. In *FOCS*, Vol. 30. 540–545.

[2] N. Bruno, S. Chaudhuri, and L. Gravano. 2001. STHoles: A multidimensional workload-aware histogram. In *Proc. 20th ACM SIGMOD Int. Conf. Management Data*. 211–222.

[3] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: A decision support benchmark for workload-driven and traditional database systems. *Proceedings of the VLDB Endowment* 14, 13 (2021), 3376–3388.

[4] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB Github Repository: SPJ templates. https://github.com/microsoft/dsb/tree/main/query_templates_pg/spj_queries. Accessed: September 27, 2023.

[5] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. *Proc. VLDB Endow.* 12, 9 (May 2019), 1044–1057. https://doi.org/10.14778/3329772.3329780

[6] e-maxx aka maxdiver. 2013. e-maxx.ru. http://e-maxx.ru/algo/treap. Accessed: January 21, 2025.

[7] Google. 2011. LevelDB: A fast key-value storage library. https://github.com/google/leveldb. Accessed: 2024-04-23.

[8] Han, Yuxing and Wu, Ziniu and Wu, Peizhi and Zhu, Rong and Yang, Jingyi and Liang, Tan Wei and Zeng, Kai and Cong, Gao and Qin, Yanzhao and Pfadler, Andreas and Qian, Zhengping and Zhou, Jingren and Li, Jiangneng, and Cui, Bin. 2022. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *VLDB* 15, 4 (2022).

[9] S. Hasan, S. Thirumuruganathan, J. Augustine, N. Koudas, and G. Das. 2020. Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries. In *Proc. 39th ACM SIGMOD Int. Conf. Management Data*. 1035–1050.

[10] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig. 2019. DeepDB: learn from data, not from queries! *Proc. VLDB Endow.* 13, 7 (2019), 992–1005.

[11] Xiao Hu, Yuxi Liu, Haibo Xiu, Pankaj K. Agarwal, Debmalya Panigrahi, Sudeepa Roy, and Jun Yang. 2022. Selectivity Functions of Range Queries are Learnable. In *SIGMOD* (Philadelphia, PA, USA) *(SIGMOD '22)*. 959–972.

[12] Paul Jaccard. 1901. Etude de la distribution florale dans une portion des Alpes et du Jura. *Bulletin de la Societe Vaudoise des Sciences Naturelles* 37 (01 1901), 547–579. https://doi.org/10.5169/seals-266450

[13] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper. 2019. Learned cardinalities: Estimating correlated joins with deep learning. (2019).

[14] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*. www.cidrdb.org.

[15] Meghdad Kurmanji and Peter Triantafillou. 2023. Detect, Distill and Update: Learned DB Systems Facing Out of Distribution Data. *Proc. ACM Manag. Data* 1, 1, Article 33 (may 2023), 27 pages. https://doi.org/10.1145/3588713

[16] Kukjin Lee, Anshuman Dutt, Vivek Narasayya, and Surajit Chaudhuri. 2023. Analyzing the Impact of Cardinality Estimation on Execution Plans in Microsoft SQL Server. *Proc. VLDB Endow.* 16, 11 (July 2023), 2871–2883. https://doi.org/10.14778/3611479.3611494

[17] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[18] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling.. In *Cidr*.

[19] Beibin Li, Yao Lu, and Srikanth Kandula. 2022. Warper: Efficiently Adapting Learned Cardinality Estimators to Data and Workload Drifts. In *Proceedings of the 2022 International Conference on Management of Data*.

[20] Beibin Li, Yao Lu, Chi Wang, and Srikanth Kandula. 2021. Cardinality Estimation: Is Machine Learning a Silver Bullet?. In *AIDB*. https://www.microsoft.com/en-us/research/publication/cardinality-estimation-is-machine-learning-a-silver-bullet/

[21] Pengfei Li, Wenqing Wei, Rong Zhu, Bolin Ding, Jingren Zhou, and Hua Lu. 2023. ALECE: An Attention-based Learned Cardinality Estimator for SPJ Queries on Dynamic Workloads. *Proc. VLDB Endow.* 17, 2 (Oct. 2023), 197–210. https://doi.org/10.14778/3626292.3626302

[22] Pengfei Li, Wenqing Wei, Rong Zhu, Bolin Ding, Jingren Zhou, and Hua Lu. 2023. ALECE: An Attention-based Learned Cardinality Estimator for SPJ Queries on Dynamic Workloads. *Proc. VLDB Endow.* 17, 2 (2023), 197–210. https://www.vldb.org/pvldb/vol17/p197-li.pdf

[23] R. J. Lipton, J. F. Naughton, and D. A. Schneider. 1990. Practical selectivity estimation through adaptive sampling. In *Proc. 9th ACM SIGMOD Int. Conf. Management Data*. 1–11.

[24] Jie Liu, Wenqian Dong, Qingqing Zhou, and Dong Li. 2021. Fauce: fast and accurate deep ensembles with uncertainty for cardinality estimation. *Proc. VLDB Endow.* 14, 11 (July 2021), 1950–1963. https://doi.org/10.14778/3476249.3476254

[25] Yuxi Liu, Vincent Capol, Xiao Hu, Pankaj K. Agarwal, and Jun Yang. 2024. SWatcher+: A Holistic Approach to Deploying Learned Cardinality Estimators in a Dynamic Setting. https://github.com/louisja1/Watcher-VLDB25/blob/main/fullversion.pdf

[26] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. 2020. Bao: Learning to Steer Query Optimizers. *arXiv preprint arXiv:2004.03814* (2020).

[27] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. 2019. Neo: A learned query optimizer. 12, 11 (2019), 1705–1718.

[28] R. Marcus and O. Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. In *aiDM*. 1–4.

[29] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proceedings of the VLDB Endowment* 2, 1 (2009), 982–993.

[30] Satoshi Nagayasu. 2023. pg_hint_plan. https://github.com/ossc-db/pg_hint_plan. Accessed: September 27, 2023.

[31] P. Negi, R. Marcus, H. Mao, N. Tatbul, T. Kraska, and M. Alizadeh. 2020. Cost-Guided Cardinality Estimation: Focus Where it Matters. In *Proc. 36th Annu. IEEE Int. Conf. Data Eng.* IEEE, 154–157.

[32] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust Query Driven Cardinality Estimation under Changing Workloads. *Proc. VLDB Endow.* 16, 6 (feb 2023), 1520–1533. https://doi.org/10.14778/3583140.3583164

[33] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2019. An empirical analysis of deep learning for cardinality estimation. *arXiv preprint arXiv:1905.06425* (2019).

[34] Y. Park, S. Zhong, and B. Mozafari. 2020. Quicksel: Quick selectivity learning with mixture models. In *Proc. 39th ACM SIGMOD Int. Conf. Management Data,*. 1017–1033.

[35] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita. 1996. Improved histograms for selectivity estimation of range predicates. *ACM Sigmod Record* 25, 2 (1996), 294–305.

[36] V. Poosala and Y. E. Ioannidis. 1997. Selectivity estimation without the attribute value independence assumption. In *VLDB*, Vol. 97. Citeseer, 486–495.

[37] Diogo Repas, Zhicheng Luo, Maxime Schoemans, and Mahmoud Sakr. 2023. Selectivity Estimation of Inequality Joins in Databases. *Mathematics* 11, 6 (2023), 1383.

[38] Jens M. Schmidt. 2009. Interval Stabbing Problems in Small Integer Ranges. In *Algorithms and Computation*, Yingfei Dong, Ding-Zhu Du, and Oscar Ibarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 163–172.

[39] Raimund Seidel and Cecilia R Aragon. 1996. Randomized search trees. *Algorithmica* 16, 4 (1996), 464–497.

[40] U. Srivastava, P. J. Haas, V. Markl, M. Kutsch, and T. M. Tran. 2006. Isomer: Consistent histogram construction using query feedback. In *Proc. 22th Annu. IEEE Int. Conf. Data Eng.* 39–39.

[41] Ji Sun and Guoliang Li. 2019. An end-to-end learning-based cost estimator. *arXiv preprint arXiv:1906.02560* (2019).

[42] Jeffrey S. Vitter. 1985. Random sampling with a reservoir. *ACM Trans. Math. Softw.* 11, 1 (March 1985), 37–57. https://doi.org/10.1145/3147.3165

[43] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are we ready for learned cardinality estimation? *Proc. VLDB Endow.* 14, 9 (May 2021), 1640–1654. https://doi.org/10.14778/3461535.3461552

[44] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a learning optimizer for shared clouds. *Proceedings of the VLDB Endowment* 12, 3 (2018), 210–222.

[45] Peizhi Wu and Gao Cong. 2021. A unified deep model of learning from both data and queries for cardinality estimation. In *Proceedings of the 2021 International Conference on Management of Data*. 2009–2022.

[46] Peizhi Wu and Zachary G. Ives. 2024. Modeling Shifting Workloads for Learned Database Systems. *Proc. ACM Manag. Data* 2, 1, Article 38 (mar 2024), 27 pages. https://doi.org/10.1145/3639293

[47] Haibo Xiu, Pankaj K Agarwal, and Jun Yang. 2024. PARQO: Penalty-aware robust plan selection in query optimization. *Proceedings of the VLDB Endowment* 17, 13 (2024).

[48] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: one cardinality estimator for all tables. *Proceedings of the VLDB Endowment* 14, 1 (2020), 61–73.

[49] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. 2019. Deep unsupervised cardinality estimation. *Proc. VLDB Endow.* 13, 3 (2019), 279–292.

[50] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2020. FLAT: fast, lightweight and accurate method for cardinality estimation. *arXiv preprint arXiv:2011.09022* (2020).

A DETAILS OF Watcher1D

We add the details of Watcher1D by a walk-through of Algorithm 1.

Algorithm 1 Watcher1D

```
Input: the initial query set $D_0$, the initial CE model $\mathfrak{m}_0$ to be deployed, an
    error metric $E$ and a corresponding threshold $\epsilon$ for triggering retraining,
    the dynamic workload $L$, the watchset size $n_{\text{watch}}$, the trainingset size
    $n_{\text{train}}$ ($<= n_{\text{watch}}$), and the test set size $n_{\text{test}}$ ($<= n_{\text{watch}}$).
 1: $\mathfrak{m} \leftarrow \mathfrak{m}_0$;  ▷ deploy the estimator
 2: $\mathcal{W} \leftarrow \{(q, \widehat{\text{card}}(q), \text{card}(q)) \mid q$ is extracted from $Q$ and $Q$ is randomly sampled from $D_0$ with size budget $n_{\text{watch}}\}$;  ▷ initialize watchset
 3: for each operation $o_i$ in the workload $L$ do
 4:     if receive a new CE model $\mathfrak{m}'$ then
 5:         $\mathfrak{m} \leftarrow \mathfrak{m}'$;  ▷ deploy new estimator
 6:         update $\widehat{\text{card}}(q) \in W$ by the new $\mathfrak{m}$;
 7:     end if
 8:     if $o_i$ is a data update then  ▷ assume that $o_i$ is a tuple $r$ inserted to/deleted from the table $R$
 9:         perform the data update to the database;
10:         for each $(q, \text{card}(q))$ in $W_R$ do
11:             if $r$ passes the selection condition in $q$ then
12:                 $\text{card}(q) \leftarrow \text{card}(q) \pm 1$;  ▷ the true cardinality $\text{card}(q)$ increases (resp. decreases) by 1 if $o_i$ is an insert (resp. a delete).
13:             end if
14:         end for
15:     else if $o_i$ is a query $Q$ then
16:         extract $q$ from $Q$;
17:         enter $\widehat{\text{card}}(q)$ to $\hat{\boldsymbol{\kappa}}$;  ▷ estimated by $\mathfrak{m}$
18:         $\pi \leftarrow \text{Opt}(Q, \hat{\boldsymbol{\kappa}})$;
19:         execute $Q$ using $\pi$ and obtain (or compute) the true cardinality card$(q)$;
20:         update $\mathcal{W}$ by $(q, \widehat{\text{card}}(q), \text{card}(q))$ if selected by reservoir sampling;
21:     end if
22:     Get a random uniform sample $D_{\text{test}}$ from $\mathfrak{m}$ of size $n_{\text{test}}$;  ▷ get a test set $D_{\text{test}}$
23:     $e \leftarrow E(\forall \text{card}(q) \in D_{\text{test}}, \forall \widehat{\text{card}}(q) \in D_{\text{test}})$;  ▷ get an aggregated error by metric $E$
24:     if $e > \epsilon$ then  ▷ trigger a retraining
25:         Get a random uniform sample $D_{\text{train}}$ from $\mathcal{W}$ of size $n_{\text{train}}$;
26:         Alert for retraining the CE model with the new training set $D_{\text{train}}$;
27:         if $e > \epsilon$ then
28:             Recursively apply $\epsilon \leftarrow 1.5\epsilon$ until $e \le \epsilon$;
29:         else if $e < \frac{\epsilon}{1.5}$ then
30:             $\epsilon \leftarrow \frac{\epsilon}{1.5}$;
31:         end if
32:     end if
33: end for
```

Lines 1-2 represent the initialization of Watcher1D, including $\mathfrak{m}$ and $\mathcal{W}$. Note that data updates in the dynamic workload only change the true cardinalities of the selected subqueries, but not the cardinality estimates because the input for $\mathfrak{m}$, i.e., the query, stays the same. Lines 3-33 show how Watcher1D performs the dynamic workload. First, in lines 4 to 7, whenever a new estimator is received, Watcher1D replaces the current estimator with the new one. This is the only case where the cardinality estimate for a query might change. In lines 10-14, Watcher1D processes a data update by scanning each subquery cached in $\mathcal{W}$. Otherwise, in lines 15-21, if it is a query Q, Watcher1D collects the estimates $\hat{\boldsymbol{\kappa}}$ from the current estimator $\mathfrak{m}$, and Q is executed using the plan $\text{Opt}(Q, \hat{\boldsymbol{\kappa}})$. Watcher1D then updates watchset accordingly by reservoir sampling, which may require computing card(q) by a report query with an extra overhead if q cannot be retrieved from the execution feedback. Finally, in lines 22-32, Watcher1D computes an aggregated error by the given metric E against a test set D_{test} generated from $\mathcal{W}$. If it exceeds the threshold ϵ, Watcher1D triggers a retraining. ϵ is adjusted according to the post-retraining error.

B DETAILS OF WATCHER2D

Instead of extending the B+-tree of the database system, Watcher2D employs external indices using LevelDB [7], a high-performance on-disk key-value store library, to efficiently answer counting queries on a single table. As discussed in Section 4, given an update r to table R, computing $\{\Delta\text{card}(q) \mid \forall q \in \mathcal{W}\}$, where Δ denotes the change, is equivalent to evaluating $\{|\sigma_{B=r.B \,\wedge\, C \ge l_C \,\wedge\, C < u_C} S| \mid \forall \sigma_C = (l_C, u_C) \in \mathcal{W}\}$.

To facilitate this, Watcher2D constructs a hierarchical index over columns B and C. Each node in this tree[4] is a key-value pair, where the key ecnodes a triplet $(\overline{b}, \overline{c_0}, \overline{c_1})$, representing a predicate $B = \overline{b} \wedge C \ge \overline{c_0} \wedge C < \overline{c_1}$. The value stores the count of tuples in S satisfying this predicate, i.e., $|\sigma_{B=\overline{b} \,\wedge\, C \ge \overline{c_0} \,\wedge\, C < \overline{c_1}} S|$. This hierarchical index has a fixed fan-out β. Higher layers contains fewer nodes covering broader ranges, while lower layers have more nodes covering fine-grained ranges. Within each layer, keys are sorted in ascending order of B-values and then the C-ranges. To simplify the storage, the explicit key representation in our implementation is actually $(\overline{b}, \overline{c_0})$, with $\overline{c_1}$ hidden but can be inferred from the next key within the same layer. The index is built bottom-up: the lowest layer consists of nodes where $\overline{c_0} = \overline{c_1} - 1$ (i.e., unit-sized C ranges). and each higher layer merges β consecutive nodes from the layer below.

To optimizer efficiency, Watcher2D constructs the hierarchical structure only for frequently occurring B-values. Specifically, the hierarchy is built only for values $\overline{b}$ where $|\sigma_{B=\overline{b}} S| \ge \theta\}$, with information related to infrequent B-values stored only in the bottom layer. Additionally, a metadata store tracks column domains and tuple counts for each B-value is employed by Watcher2D. Our default parameters, obtained via ablation studies, are $\theta = 10, \beta = 5$, and the maximum of $\ell = 6$ layers[5].

When a data update occurs (e.g., a tuple r is inserted or deleted in table R), Watcher2D processes it in two steps. First, it updates the affected key-value pairs in the external index for table R. Specifically, if $r.B$ is infrequent, only the bottom-layer key-value pair is updated; otherwise, updates propagate to one key-value pair per layer. Second, Watcher2D computes the impact of r on every subquery in $\mathcal{W}_{RS'}$ using the indices on table S. For every q, it evaluates $|\sigma_{B=r.B \,\wedge\, C \ge l_C \,\wedge\, C < u_C} S|$ via a top-down traversal of the hierarchical index. As shown in Section 4, at the topmost layer, if a contiguous sequence of key-value pairs $(r.B, c_x) \to count_x, (r.B, c_{x+1}) \to count_{x+1}, \ldots, (r.B, c_y) \to count_y$ are fully covered by $r.B$ and range $[l_C, u_C)$, i.e., $c_x \ge l_C \wedge c_y < u_C$, part of the answer is obtained as $\sum_{i=x}^{y} count_i$. Then, it pushes down

[4] the tree can degenerate into a forest with multiple roots in the uppermost layers.

[5] Further tuning is left for future work, as parameter optimization is not the main focus of our paper

two sub-queries $|\sigma_{B=r.B \wedge C\geq l_C \wedge C<c_x} S|$ (only if $c_x > l_C$) and $|\sigma_{B=r.B \wedge C\geq c_y \wedge C<u_C} S|$ (only if $c_y < u_C$) to the layer below. If such a sequence does not exist, the original counting query, denoted by $r.B$ and range $[l_C, u_C)$, is directly pushed down to the layer below. This process continues until all counting queries are resolved using the bottom layer.

Since the main overhead arises from accessing on-disk key-value pairs, Watcher2D employs batch processing to improvce efficiency. Specifically, instead of processing each subquery independently, it evaluates all subqueries in all the subqueries in $\mathcal{W}_{RS'}$ layer by layer. This avoids redundant key-value lookups when subqueries share overalpping C-ranges, further optimizing performance.

C DETAILS OF SWATCHER+

SWatcher+ is based on an observation: an inaccurate cardinality estimate might not necessarily lead to a longer query execution time compared to using the true cardinality for several possible reasons:

1. injecting the cardinality estimate leads to the same query plan obtained by injecting the true cardinality;
2. otherwise, even though injecting the cardinality estimate leads to a different query plan, this plan might have a cost that is just slightly higher than that of the plan obtained by injecting the true cardinality;
3. otherwise, even though injecting the cardinality estimate leads to a much worse query plan in terms of the plan cost, the query execution can finish in a short time regardless of which plan is used, and the execution time difference might be negligible.

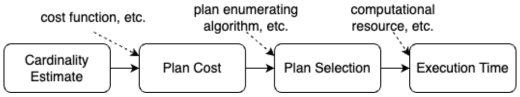


Figure 9: From cardinality estimate to query execution time

All the reasons above reflect that query optimization is a very complex process and its effectiveness depends on many other factors than just the cardinality estimates. Figure 9 shows a simplification of this process and some factors that are involved. It indicates that it is more important to provide accurate cardinality estimates to those queries that are more sensitive to inaccurate cardinality estimates. By this motivation, SWatcher+ has a different design of watchset and how to select subqueries, which focuses on queries that are more sensitive to the error of cardinality estimation.

In the following, we list the pseudocode of SWatcher+ as long as each sub-routine it utilizes, including Refill, Detect, Correct, and Retrain. Notably, since SWatcher+ can be deployed to monitor multiple non-overlapping fragments in the query, e.g., in TJQ013a-3B and TJQ013b-3B, q in the following algorithms can be replaced by a set of qs corresponding to all the monitored fragments. It means that the pre-execution process is applied to each fragment, and after the execution, the post-execution process is also applied to each fragment. To obtain the plan, all the single-selection cardinality estimates by $\mathfrak{m}$ are collected and then the corrections on join cardinality estimates are applied.

Pseudocode of SWatcher+. We walk through SWatcher+ by Algorithm 2. First, SWatcher+ deploys the initial CE model, initialize the set of compensating correction factors $\mathcal{F}$, and (re-)fill the watchset in lines 1-3. This process is repeated after every (re-)training (as shown in lines 7-9), because the cardinality estimates might change and whether a query is (still) considered as a watchset query needs additional checks. We present Refill later in Algorithm 3 associated with a line-by-line walk-through. Second, besides $\mathcal{W}$, SWatcher+ also maintains a query pool, which is a random uniform sample of queries, in order to provide queries from the query distribution whenever it is needed. Third, in lines 11-13, if o_i is a data update, SWatcher+ performs similarly as Watcher1D does. Forth, if o_i is a query, corresponding to lines 14-34, SWatcher+ extracts the single-selectio subquery q from Q, adds it to the query pool, and collect the estimates $\widehat{\mathsf{card}}(q)$ from current $\mathfrak{m}$ and form $\hat{\boldsymbol{\kappa}}$. Instead of directly injecting $\hat{\boldsymbol{\kappa}}$, SWatcher+ tries to apply a correction on $\hat{\boldsymbol{\kappa}}$ through Correct (details in Algorithm 5) as shown in line 18. Then, in lines 19-20, SWatcher+ injects the corrected cardinality estimates $\tilde{\boldsymbol{\kappa}}$ to get the plan π, executes Q by this plan, and collects the associated information[6]. Fifth, $\mathcal{F}$ is updated through the information collected from the feedback of the execution. Sixth, in lines 22, SWatcher+ calls Detect to decide if q will be a watchset query. We leave the discussion of Detect in Algorithm 4. As shown in lines 23-28, if q is a watchset query, i.e., ϕ is true, the true cardinality $\mathsf{card}(q)$, as well as the speedup $\mathsf{gain}(q)$, are appended to or updated in $\mathcal{W}$. Otherwise, as shown in lines 29-34, the corresponding information will be removed from both $\mathcal{W}$ and $\mathcal{F}$. Lastly, SWatcher+ triggers a retraining based on the number of items in the watchset (see more details in Algorithm 6).

Pseudo-code of Refill. SWatcher+ calls Refill to get a new watchset. As shown in Algorithm 3, we check each q in either the initial query set or the old watchset and update $\widehat{\mathsf{card}}(q)$ by the currect $\mathfrak{m}$. Then, in lines 4-5, we collect information needed by Detect from $\mathcal{W}_{\text{old}}$ and $\mathcal{F}$. Next, in line 6-9, we call Detect to check if q is a watch query and update $\mathcal{W}_{\text{new}}$ correspondingly. One difference between Algorithm 3 line 6 and Algorithm 2 line 22 is whether we have a set of true cardinalities observed from the execution feedback. And in Refill, we do not have that information and an empty set is entered instead. We specify the usage of these true join cardinalities for Detect in Algorithm 4.

Pseudocode of Detect. We details Detect in Algorithm 4. First, depending on whether q is already in $\mathcal{W}$, we have two branches lines 2-14 and lines 15-26. Both of them have a block for quick checks and a block for speedup check. We do not repeat the details that have been discussed in Section 5. In general, quick checks are for detecting if a query is running slow enough and having a short planning time. And speedup check is to assess the potential speedup if we cache the true cardinalities for the current subquery in $\mathcal{W}$ and the associated correction factors in $\mathcal{F}$. Specifically, we use the plan cost (from the plan if and the plan if we don't admit q) ratio to denote the potential speedup, and admit a q only if the speedup is larger than θ_{gain}.

[6] Similarly, if $\mathsf{card}(q)$ is not available from the feedback of execution, SWatcher+ issues a report query to the database with an overhead

Pseudocode of Correct. We detail this process in Algorithm 5. After obtaining the watchset query $\tilde{q}$ with the highest similarity, Correct applies the single-selection corrections in line 3 and then collect the join cardinalities estiamted by the optimizer in line 4. Next, in lines 5-10, Correct searches $\mathcal{F}$ for the corrections over the join cardinality estimates. Notably, this step is based on the fact that Correct alreadys applies the set of corrections on the single-selection subqueries $\{q\}$.

Pseudocode of Retrain. SWatcher+ triggers retraining when more than $\theta_{n_{\text{watch}}}$ queries enter $\mathcal{W}$, as shown in line 35 of Algorithm 2. The process of collecting a new (re-)training set is detailed in Algorithm 6. In essence, SWatcher+ includes as many watchset queries and their true cardinalities (used as labels) in the trainingset as possible. Additionally, if $n_{\text{train}} > |\mathcal{W}|$, it randomly samples more queries from the query pool and assigns labels to them by issuing report queries to the database. This new trigger design is different from Watcher1D's.

Algorithm 2 SWatcher+

```
Input: the initial query set $D_0$, the initial CE model $\mathfrak{m}_0$ to be deployed, the dynamic workload $L$, $\theta_{\text{sim}}$ for applying correction, a triple $(\theta_{\text{T}_{\text{Exec}}}, \theta_{\text{T}_{\text{Opt}}}, \theta_{\text{gain}})$ parameters for the detection, a size budget $\theta_{n_{\text{watch}}}$ for triggering retraining, and the trainingset size $n_{\text{train}}$.
1: $\mathfrak{m} \leftarrow \mathfrak{m}_0$;  ▷ deploy the estimator
2: $\mathcal{F} \leftarrow \emptyset$;
3: $\mathcal{W} \leftarrow$ Refill$(D_0, \mathfrak{m}, \mathcal{F}, (\theta_{\text{T}_{\text{Exec}}}, \theta_{\text{T}_{\text{Opt}}}, \theta_{\text{gain}}))$  ▷ initialize watchset by Algorithm 3
4: add all $q$ in $D_0$ to the query pool;  ▷ we maintain a $n_{\text{train}}$-size pool of all single-table queries we have seen so far in case we need more queries for retraining  ▷ initialize the set of compensating correction factors as emptyset
5: for each SQL statement $o_i$ in the workload $L$ do
6:     if receive a new model $\mathfrak{m}'$ then
7:         $\mathfrak{m} \leftarrow \mathfrak{m}'$;  ▷ deploy new estimator
8:         $\mathcal{W} \leftarrow$ Refill$(\mathcal{W}, \mathfrak{m}, \mathcal{F}, (\theta_{\text{T}_{\text{Exec}}}, \theta_{\text{T}_{\text{Opt}}}, \theta_{\text{gain}}))$;  ▷ refill watchset by Algorithm 3
9:         $\mathcal{F} \leftarrow \emptyset$
10:    end if
11:    if $o_i$ is a data update then
12:        perform the data update to the database;
13:        update the true cardinalities in $\mathcal{W}$;  ▷ the same as lines 10-14 in Algorithm 1
14:    else if $o_i$ is a query $Q$ then
15:        extract $q$ from $Q$;
16:        enter $\widehat{\text{card}}(q)$ to $\hat{\boldsymbol{\kappa}}$;
17:        add $q$ to the query pool;
18:        $\tilde{\boldsymbol{\kappa}} \leftarrow$ Correct$(Q, q, \hat{\boldsymbol{\kappa}}, \mathcal{W}, \mathcal{F}, \theta_{\text{sim}})$;  ▷ correct the cardinality estimate by $\mathcal{W}$ and $\mathcal{F}$ if possible (Algorithm 5)
19:        $\pi \leftarrow$ Opt$(Q, \tilde{\boldsymbol{\kappa}})$;  ▷ get the plan by injecting $\tilde{\boldsymbol{\kappa}}$
20:        execute $Q$ by $\pi$, record $\text{T}_{\text{Exec}}(\pi)$, Cost$(Q, \tilde{\boldsymbol{\kappa}})$, $\text{T}_{\text{Opt}}(Q)$ and a set $C$ of true join cardinalities observed for $\pi$, and get card$(q)$ from the execution or compute it through a report query;
21:        update $\mathcal{F}$ by a new mapping $\mathcal{F}_{J_Q, C_Q} \coloneqq \mathcal{F}_{J_Q, \{q\}} \coloneqq \{J' \to \frac{\text{card}(J')}{\widehat{\text{card}}(J')} \mid \forall J' \in C\}$ as well as the creation timestamp, the query $Q$, and $\text{T}_{\text{Exec}}(\pi), \text{T}_{\text{Opt}}(Q)$;  ▷ card$(J')$ is observed from executing $\pi$, and $\widehat{\text{card}}(J')$ is in $\tilde{\boldsymbol{\kappa}}$
22:        $\phi$, gain$(q) \leftarrow$ Detect$(\theta_{\text{T}_{\text{Exec}}}, \theta_{\text{T}_{\text{Opt}}}, \theta_{\text{gain}}, Q$, card$(q)$, $\widehat{\text{card}}(q), C, \text{T}_{\text{Exec}}(\pi), \text{T}_{\text{Opt}}(Q), \mathcal{W})$  ▷ detect if to admit/evict $q$ to/from $\mathcal{W}$
23:        if $\phi$ then  ▷ passes the checks
24:            if $q \in \mathcal{W}$ then
25:                update gain$(q)$ cached in $\mathcal{W}$;
26:            else
27:                $\mathcal{W}[q] \leftarrow$ (card$(q)$, gain$(q)$)  ▷ records $q$'s associated information
28:            end if
29:        else  ▷ fails the checks
30:            if $q \in \mathcal{W}$ then
31:                remove $q$ from $\mathcal{W}$ and remove any entries in $\mathcal{F}$ indexed by any $(J, C)$ where $q \in C$;  ▷ remove $q$ and the associated compensating correction factors from $\mathcal{F}$
32:            end if
33:        end if
34:    end if
35:    if $|\mathcal{W}| \geq \theta_{n_{\text{watch}}}$ then  ▷ trigger a retraining (more details in Algorithm 6)
36:        Retrain$(n_{\text{train}}, \mathcal{W}$, the access to the query pool);
37:    end if
38: end for
```

Algorithm 3 Refill

```
Input: the initial query set or an old watchset denoted by $\mathcal{W}_{\text{old}}$, the current CE model $\mathfrak{m}$,, the current compensating correction factors $\mathcal{F}$, the triplet parameters $(\theta_{\mathsf{T}_{\text{Exec}}}, \theta_{\mathsf{T}_{\text{Opt}}}, \theta_{\text{gain}})$
Output: a new watchset $\mathcal{W}_{\text{new}}$
 1: $\mathcal{W}_{\text{new}} \leftarrow \emptyset$;
 2: for every $q \in \mathcal{W}_{\text{old}}$ do
 3:     get the $\mathsf{card}(q)$ in $\mathcal{W}_{\text{old}}[q]$
 4:     find the most recent entry in $\mathcal{F}$ indexed by $(J, C)$ where $q \in C$ and collect the recorded query $Q$, timings $\mathsf{T}_{\text{Exec}}(\pi), \mathsf{T}_{\text{Opt}}(Q)$;
 5:     $\widehat{\mathsf{card}}(q) \leftarrow \mathfrak{m}(q)$;  ▷ get the cardinality estimate by $\mathfrak{m}$
 6:     $\phi, \mathsf{gain}(q) \leftarrow \mathsf{Detect}(\theta_{\mathsf{T}_{\text{Exec}}}, \theta_{\mathsf{T}_{\text{Opt}}}, \theta_{\text{gain}}, Q, \mathsf{card}(q), \widehat{\mathsf{card}}(q), \emptyset, \mathsf{T}_{\text{Exec}}(\pi), \mathsf{T}_{\text{Opt}}(Q), \mathcal{W})$;  ▷ detect if to admit/evict $q$ to/from $\mathcal{W}$
 7:     if $\phi$ then  ▷ passes the checks
 8:         $\mathcal{W}_{\text{new}}[q] \leftarrow (\mathsf{card}(q), \mathsf{gain}(q))$  ▷ records the true cardinality of $q$ and the speedup
 9:     end if
10: end for
11: return $\mathcal{W}_{\text{new}}$;
```

Algorithm 4 Detect

```
Input: the triplet parameters $(\theta_{\mathsf{T}_{\text{Exec}}}, \theta_{\mathsf{T}_{\text{Opt}}}, \theta_{\text{gain}})$, the query $Q$, the true cardinality $\mathsf{card}(q)$ of the watchset query, the estimated cardinality $\widehat{\mathsf{card}}(q)$ by the current $\mathfrak{m}$, a set $C$ of observed true join cardinalities, the most recent execution time $\mathsf{T}_{\text{Exec}}(Q)$, the most recent planning time $\mathsf{T}_{\text{Opt}}(Q)$, the current watchset $\mathcal{W}$
Output: a boolean flag $\phi$ representing if $q$ passes the check and the speedup $\mathsf{gain}(q)$
 1: extract $q$ from $Q$;
 2: if $q \notin \mathcal{W}$ then
 3:     if $\mathsf{T}_{\text{Exec}}(\pi) \geq \theta_{\mathsf{T}_{\text{Exec}}}$ and $\mathsf{T}_{\text{Opt}}(Q) \leq \theta_{\mathsf{T}_{\text{Opt}}}$ then  ▷ quick checks
 4:         set $\boldsymbol{\kappa}^{\star}$ with $\mathsf{card}(q)$ and $C$;  ▷ $\boldsymbol{\kappa}^{\star}$ contains all the known true cardinalities at that point
 5:         set $\hat{\boldsymbol{\kappa}}$ with the cardinality estimate $\widehat{\mathsf{card}}(q)$ and leave others to be estimated by the optimizer;  ▷ $\boldsymbol{\kappa}'$ means that "if we fully rely on $\mathfrak{m}$'s estimates"
 6:         set $\boldsymbol{\kappa}'$ with $\mathsf{card}(q)$ and $C$ and leave others to be either uncorrected estimates by the $\mathfrak{m}$ (if have) or estimated by the optimizer;  ▷ $\boldsymbol{\kappa}^{\star}$ means that "if we know the true cardinalities because we cache them by $\mathcal{W}$ and $\mathcal{F}$"
 7:         get the baseline plan $\hat{\pi}$ by $\mathsf{Opt}(Q, \hat{\boldsymbol{\kappa}})$;  ▷ representing the plan by relying on $\mathfrak{m}$ only
 8:         get the plan $\pi' \leftarrow \mathsf{Opt}(Q, \boldsymbol{\kappa}')$, which would be obtained by admitting $q$ (alone if we also have other fragments to be monitored in the template);  ▷ representing the plan by knowing some true cardinalities thanks to $\mathcal{W}$ and $\mathcal{F}$
 9:         $\mathsf{gain}(q) = \frac{\mathsf{Cost}(\hat{\pi}, \boldsymbol{\kappa}^{\star})}{\mathsf{Cost}(\pi', \boldsymbol{\kappa}^{\star})}$;  ▷ compute the speedup by the ratio between the two plans. Notably, we always use as many true cardinalities as we know, i.e., $\boldsymbol{\kappa}^{\star}$ to cost the plans
10:         if $\mathsf{gain}(q) \geq \theta_{\text{gain}}$ then  ▷ speedup check
11:             return true, $\mathsf{gain}(q)$
12:         end if
13:     end if
14:     return false, none;
15: else
16:     if $\mathsf{T}_{\text{Exec}}(\pi) \cdot \mathsf{gain}(q) < \theta_{\mathsf{T}_{\text{Exec}}}$ or $\mathsf{T}_{\text{Opt}}(Q) > \theta_{\mathsf{T}_{\text{Opt}}}$ then  ▷ quick checks
17:         return false, none;
18:     else
19:         proceed the same as lines 4-9 and get $\mathsf{gain}(q)$;
20:         if $\mathsf{gain}(q) \geq \theta_{\text{gain}}$ then  ▷ speedup check
21:             return true, $\mathsf{gain}(q)$;
22:         else
23:             return false, none;
24:         end if
25:     end if
26: end if
```

```
Algorithm 5 Correct
Input: the current query $Q$, the current subquery $q$, the cardinality estimates $\hat{\boldsymbol{\kappa}}$ generated by $\mathfrak{m}$, the watchset $W$, the similarity threshold $\theta_{\text{sim}}$
Output: the corrected cardinality estimates $\tilde{\boldsymbol{\kappa}}$
$\tilde{q} = \arg\max_{\forall q \in W} \text{sim}(p, \tilde{p})$;   ▷ find the watchset query $\tilde{q}$ with condition $\tilde{p}$ having the highest similarity with $q$'s condition $p$
if $\text{sim}(q, \tilde{q}) \geq \theta_{\text{sim}}$ then
    enter $\widehat{\text{card}}(p \wedge \neg\tilde{p}) + \text{sim}(p \wedge \tilde{p}, \tilde{p}) \cdot \text{card}(\tilde{q})$ to $\hat{\boldsymbol{\kappa}}$
    let the optimizer to estimate the join cardinalities for query $Q$ and record them in $\tilde{\boldsymbol{\kappa}}$;
    denote the set $\{q\}$ of corrections by $C_Q$ and the joins in $Q$ that are only related to tables in $C_Q$ by $J_Q$;
    look for a set of signatures $\{(J_i, C_i)\}$ in $\mathcal{F}$ such that $J_i$'s are disjoint, each $C_i$ is exactly $C_Q$ restricted to $J_i$ and $J_Q$ are maximally covered by $J_i$'s;
    for join $J'$ from small to large that appear in $\tilde{\boldsymbol{\kappa}}$ do   ▷ smaller joins are corrected before larger joins
        find the only signature $(J_i, C_i)$ (if exists) such that $J' \subseteq J_i$;   ▷ there is at most one such signature exists because $J_i$'s are disjoint
        $\tilde{\boldsymbol{\kappa}}[J'] \leftarrow \tilde{\boldsymbol{\kappa}}[J'] \cdot \mathcal{F}_{J_i,C_i}(J')$;   ▷ apply the compensating correction factors to the join cardinalities estimated by the optimizer
    end for
else
    $\tilde{\boldsymbol{\kappa}} \leftarrow \hat{\boldsymbol{\kappa}}$;   ▷ meaning that no corrections can be done
end if
return $\tilde{\boldsymbol{\kappa}}$;
```

```
Algorithm 6 Retrain
Input: the size $n_{\text{train}}$ of (re-)trainingset, the watchset $\mathcal{W}$, the access to the query pool
if $|\mathcal{W}| \geq n_{\text{train}}$ then
    $D_{\text{train}} \leftarrow$ the $(q, \text{card}(q))$-pairs in a random uniform sample from $\mathcal{W}$ of size $n_{\text{train}}$;
else
    $D_{\text{train}} \leftarrow$ all $(q, \text{card}(q))$-pairs from $\mathcal{W}$;
    $D_{\text{train}} \leftarrow D_{\text{train}}$ plus $(n_{\text{train}} - |\mathcal{W}|)$ more $(q, \text{card}(q))$-pairs that are randomly generated from the query pool and labeled by report queries to the database;
end if
alert for updating the model and offer the new trainingset $D_{\text{train}}$;
```

D MORE DETAILS OF EXPERIMENT

In Appendix D.1, we first present all the templates used in Section 6. We present the specific design of the external index of Watcher2D in Appendix B. In Appendix D.2, we show more details for the ablation studies of SWatcher+'s parameters. In Appendix D.3, we show the results on end-to-end runtime for three-batch setting.

D.1 Templates Used in Section 6

```sql
-- Q013a
SELECT *
FROM store_sales, store, customer_demographics,
      household_demographics, customer_address, date_dim
WHERE s_store_sk = ss_store_sk
AND ss_sold_date_sk = d_date_sk
AND ss_hdemo_sk=hd_demo_sk
AND cd_demo_sk = ss_cdemo_sk
AND ss_addr_sk = ca_address_sk
AND ss_sales_price BETWEEN 100.00 AND 150.00;
```

```sql
-- Q013b
SELECT *
FROM store_sales, store, customer_demographics,
      household_demographics, customer_address, date_dim
WHERE s_store_sk = ss_store_sk
AND ss_sold_date_sk = d_date_sk
AND ss_hdemo_sk=hd_demo_sk
AND cd_demo_sk = ss_cdemo_sk
AND ss_addr_sk = ca_address_sk
AND ss_net_profit BETWEEN 100 AND 200;
```

```sql
-- Q018
SELECT *
FROM catalog_sales, customer_demographics, customer,
      customer_address, date_dim, item
WHERE cs_sold_date_sk = d_date_sk
AND cs_item_sk = i_item_sk
AND cs_bill_cdemo_sk = cd_demo_sk
AND cs_bill_customer_sk = c_customer_sk
AND c_current_addr_sk = ca_address_sk
AND cs_wholesale_cost BETWEEN 84 AND 89;
```

```sql
-- Q019
SELECT *
FROM date_dim, store_sales, item, customer, customer_address,
      store
WHERE d_date_sk = ss_sold_date_sk
AND ss_item_sk = i_item_sk
AND ss_customer_sk = c_customer_sk
AND c_current_addr_sk = ca_address_sk
AND ss_store_sk = s_store_sk
AND ss_wholesale_cost BETWEEN 73 AND 93;
```

```sql
-- Q040
SELECT *
FROM warehouse, item, date_dim, catalog_sales, catalog_returns
WHERE cs_order_number = cr_order_number
AND cs_item_sk = cr_item_sk
AND i_item_sk = cs_item_sk
AND cs_warehouse_sk = w_warehouse_sk
AND cs_sold_date_sk = d_date_sk
AND cs_wholesale_cost BETWEEN 81 AND 100;
```

```sql
-- Q072
SELECT *
FROM catalog_sales, inventory, warehouse, item,
      customer_demographics, household_demographics,
date_dim d1, date_dim d2, date_dim d3
WHERE cs_item_sk = inv_item_sk
AND w_warehouse_sk = inv_warehouse_sk
AND i_item_sk = cs_item_sk
AND cs_bill_cdemo_sk = cd_demo_sk
AND cs_bill_hdemo_sk = hd_demo_sk
AND cs_sold_date_sk = d1.d_date_sk
AND inv_date_sk = d2.d_date_sk
AND cs_ship_date_sk = d3.d_date_sk AND d3.d_date > d1.d_date +
      interval '3 day'
AND d1.d_week_seq = d2.d_week_seq
AND inv_quantity_on_hand < cs_quantity
AND cs_wholesale_cost BETWEEN 73 AND 93;
```

```sql
-- Q072
SELECT *
FROM catalog_sales, inventory, warehouse, item,
      customer_demographics, household_demographics,
date_dim d1, date_dim d2, date_dim d3
WHERE cs_item_sk = inv_item_sk
AND w_warehouse_sk = inv_warehouse_sk
AND i_item_sk = cs_item_sk
AND cs_bill_cdemo_sk = cd_demo_sk
AND cs_bill_hdemo_sk = hd_demo_sk
AND cs_sold_date_sk = d1.d_date_sk
AND inv_date_sk = d2.d_date_sk
AND cs_ship_date_sk = d3.d_date_sk AND d3.d_date > d1.d_date +
      interval '3 day'
AND d1.d_week_seq = d2.d_week_seq
AND inv_quantity_on_hand < cs_quantity
AND cs_wholesale_cost BETWEEN 73 AND 93;
```

```sql
-- Q085a
SELECT *
FROM web_sales, web_returns, web_page, customer_demographics cd1,
      customer_demographics cd2, customer_address, date_dim, reason
WHERE ws_web_page_sk = wp_web_page_sk
AND ws_item_sk = wr_item_sk
AND ws_order_number = wr_order_number
AND ws_sold_date_sk = d_date_sk
AND cd1.cd_demo_sk = wr_refunded_cdemo_sk
AND cd2.cd_demo_sk = wr_returning_cdemo_sk
AND ca_address_sk = wr_refunded_addr_sk
AND r_reason_sk = wr_reason_sk
AND cd1.cd_marital_status = cd2.cd_marital_status
AND cd1.cd_education_status = cd2.cd_education_status
AND ws_sales_price BETWEEN 100.00 AND 150.00;
```

```sql
-- Q085b
SELECT *
FROM web_sales, web_returns, web_page, customer_demographics cd1,
      customer_demographics cd2, customer_address, date_dim, reason
WHERE ws_web_page_sk = wp_web_page_sk
AND ws_item_sk = wr_item_sk
AND ws_order_number = wr_order_number
AND ws_sold_date_sk = d_date_sk
AND cd1.cd_demo_sk = wr_refunded_cdemo_sk
AND cd2.cd_demo_sk = wr_returning_cdemo_sk
AND ca_address_sk = wr_refunded_addr_sk
AND r_reason_sk = wr_reason_sk
AND cd1.cd_marital_status = cd2.cd_marital_status
AND cd1.cd_education_status = cd2.cd_education_status
AND ws_net_profit BETWEEN 100 AND 200;
```

```sql
-- Q099
SELECT *
FROM catalog_sales, warehouse, ship_mode, call_center, date_dim
WHERE cs_ship_date_sk = d_date_sk
AND cs_warehouse_sk = w_warehouse_sk
AND cs_ship_mode_sk = sm_ship_mode_sk
AND cs_call_center_sk = cc_call_center_sk
AND cs_list_price BETWEEN 248 AND 277;
```

```sql
-- Q100
SELECT *
FROM item as item1, item as item2, store_sales as s1, store_sales
      as s2, date_dim, customer, customer_address,
      customer_demographics
WHERE item1.i_item_sk < item2.i_item_sk
AND s1.ss_ticket_number = s2.ss_ticket_number
AND s1.ss_item_sk = item1.i_item_sk
AND s2.ss_item_sk = item2.i_item_sk
AND s1.ss_customer_sk = c_customer_sk
AND c_current_addr_sk = ca_address_sk
AND c_current_cdemo_sk = cd_demo_sk
AND d_date_sk = s1.ss_sold_date_sk
AND s1.ss_list_price BETWEEN 80 AND 94;
```

```sql
-- Q101
SELECT *
FROM store_sales, store_returns, web_sales, date_dim d1, date_dim
      d2, item, customer, customer_address, household_demographics
WHERE ss_ticket_number = sr_ticket_number
AND ss_customer_sk = ws_bill_customer_sk
AND ss_customer_sk = c_customer_sk
AND c_current_addr_sk = ca_address_sk
AND c_current_hdemo_sk = hd_demo_sk
AND ss_item_sk = sr_item_sk
AND sr_item_sk = ws_item_sk
AND i_item_sk = ss_item_sk
AND sr_returned_date_sk = d1.d_date_sk
AND ws_sold_date_sk = d2.d_date_sk
AND d2.d_date BETWEEN d1.d_date AND (d1.d_date + interval '90
      day')
AND ss_sales_price / ss_list_price BETWEEN 73 * 0.01 AND 93 *
      0.01;
```

```sql
-- Q102
SELECT *
FROM store_sales, web_sales, date_dim d1, date_dim d2, customer,
      inventory, store, warehouse, item, customer_demographics,
      household_demographics, customer_address
WHERE ss_item_sk = i_item_sk
AND ws_item_sk = ss_item_sk
AND ss_sold_date_sk = d1.d_date_sk
AND ws_sold_date_sk = d2.d_date_sk
AND d2.d_date BETWEEN d1.d_date AND (d1.d_date + interval '30
      day')
AND ss_customer_sk = c_customer_sk
AND ws_bill_customer_sk = c_customer_sk
AND ws_warehouse_sk = inv_warehouse_sk
AND ws_warehouse_sk = w_warehouse_sk
AND inv_item_sk = ss_item_sk
AND inv_date_sk = ss_sold_date_sk
AND inv_quantity_on_hand >= ss_quantity
AND s_state = w_state
AND c_current_cdemo_sk = cd_demo_sk
AND c_current_hdemo_sk = hd_demo_sk
AND c_current_addr_sk = ca_address_sk
AND ws_wholesale_cost BETWEEN 73 AND 93;
```

```sql
-- TJQ013a
SELECT * FROM
    store_sales, household_demographics, store,
          customer_demographics, customer_address, date_dim
WHERE s_store_sk = ss_store_sk
    AND ss_sold_date_sk = d_date_sk
    AND ss_hdemo_sk = hd_demo_sk
    AND cd_demo_sk = ss_cdemo_sk
    AND ss_addr_sk = ca_address_sk
    AND ss_sales_price BETWEEN 100.00 AND 150.00
    AND hd_dep_count BETWEEN 0 AND 9;
```

```sql
-- TJQ013b
SELECT * FROM
    store_sales, household_demographics, store,
          customer_demographics, customer_address, date_dim
WHERE s_store_sk = ss_store_sk
    AND ss_sold_date_sk = d_date_sk
    AND ss_hdemo_sk = hd_demo_sk
    AND cd_demo_sk = ss_cdemo_sk
    AND ss_addr_sk = ca_address_sk
    AND ss_net_profit BETWEEN 100 AND 200
    AND hd_dep_count BETWEEN 0 AND 9;
```

D.2 Ablation Studies of SWatcher+'s Parameters

Besides Q013b shown in Section 6.3, we also present the compenentized runtimes results for Q013a (in Table 4) and Q018 (in Table 5). In overall, the setup $\theta_{\text{sim}} = 80\%$, $\theta_{n_{\text{watch}}} = 20\%$ the number of queries in the workload, $\theta_{T_{\text{Opt}}} = 10\% \cdot \theta_{T_{\text{Exec}}}$, $\theta_{\text{gain}} = 101\%$ and $\theta_{T_{\text{Exec}}}$ as detailed in Table 1, plus correcting two-table join subqueries, consistently perform the best compared to other setups. Notably, there are some templates, e.g., Q019, Q040, Q099, with consistently empty $\mathcal{W}$ where "plus 2-join" should perform similarly as "single only' does, and some templates, e.g., Q085a, Q085b, Q101, Q102, where SWatcher+

are skipped. To sum up, we stick to this setup for SWatcher+ as it provides the best end-to-end runtimes for most of the cases.

D.3 Results of End-to-End Runtime for Three-Batch Setting

As shown in Figure 10, for each single-selection query template with three-batch setting, we measure the end-to-end runtime for each algorithm and report the results using their configuration optimized for the lowest runtime. The observations are very similar to those for the six-batch setting in Section 6.2. In overall, SWatcher+ achieves the lower end-to-end runtimes compared to Periodic and Watcher1D, even when they are with the optimized parameters observed post hoc. Moreover, SWatcher+ can sometimes achieve a similar total execution time to True's by doing corrections only instead of retraining.

Table 4: Q013a: the componentized runtimes for SWatcher+ with various setups.

Algorithm	Retrain if†	Correct if	Total $\mathrm{T_{Exec}}(Q)$	Retraining Time	Overhead on Qs	Overhead on DUs	Total $\mathrm{T_{Exec}}(Q)$ in potent region
SWatcher+	single only	$\theta_{\text{gain}} = 110\%$	860.32	0(0)	26.62	5.76	36.05
SWatcher+	plus 2-join	$\theta_{\text{gain}} = 110\%$	989.53	0(0)	26.71	3.42	52.72
SWatcher+	single only	$\theta_{\text{gain}} = 101\%$	874.00	0(0)	27.53	8.87	45.81
SWatcher+	**plus 2-join**	$\boldsymbol{\theta_{\text{gain}}} = 101\%$	**839.49**	**0(0)**	**26.25**	**7.69**	**65.17**

Table 5: Q018: the componentized runtimes for SWatcher+ with various setups.

Algorithm	Retrain if†	Correct if	Total $\mathrm{T_{Exec}}(Q)$	Retraining Time	Overhead on Qs	Overhead on DUs	Total $\mathrm{T_{Exec}}(Q)$ in potent region
SWatcher+	single only	$\theta_{\text{gain}} = 110\%$	1153.70	0(0)	31.24	16.65	4.05
SWatcher+	plus 2-join	$\theta_{\text{gain}} = 110\%$	1151.05	0(0)	32.25	17.35	3.86
SWatcher+	single only	$\theta_{\text{gain}} = 101\%$	1191.25	0(0)	32.96	33.79	4.78
SWatcher+	**plus 2-join**	$\boldsymbol{\theta_{\text{gain}}} = 101\%$	**1095.65**	**0(0)**	**31.71**	**30.54**	**15.30**

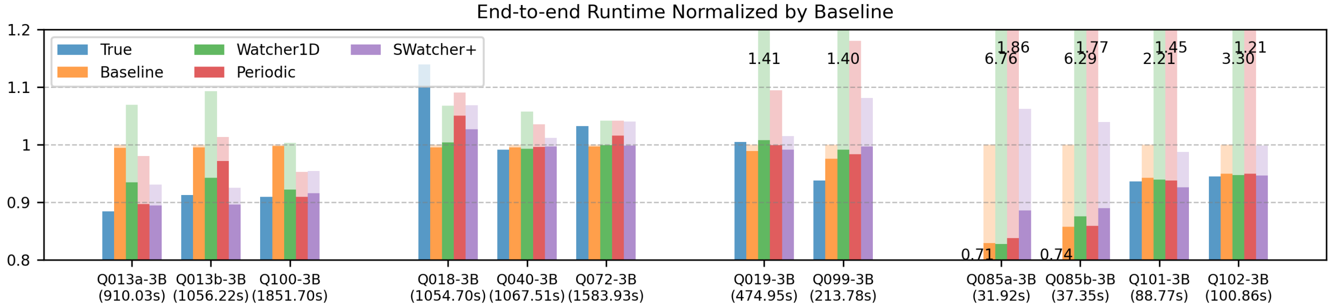


Figure 10: the normalized end-to-end runtime for three-batch setting, including both total query execution time (represented by the lower portion of the bar with darker shade) and overhead (represented by the upper portion with lighter shade), across all algorithms and single-selection templates. A number is attached to a bar if its value falls outside the scope. The number below the template name is the end-to-end runtime of Baseline, used for normalization.