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

Yuxi Liu
yuxi.liu@duke.edu
Duke University
Durham, NC, USA

Vincent Capol
vincent.capol@duke.edu
Duke University
Durham, NC, USA

Xiao Hu
xiaohu@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Pankaj K. Agarwal
pankaj@cs.duke.edu
Duke University
Durham, NC, USA

Jun Yang
junyang@cs.duke.edu
Duke University
Durham, NC, USA

Abstract

Cardinality estimation (CE) is crucial for database query optimization. Although recent advances in machine-learned CE have improved accuracy, deploying these models in dynamic databa-ses 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.

ACM Reference Format:
Yuxi Liu, Vincent Capol, Xiao Hu, Pankaj K. Agarwal, and Jun Yang. 2018. SWatcher+: A Holistic Approach to Deploying Learned Cardinality Estimators in a Dynamic Setting. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX).* ACM, New York, NY, USA, 24 pages. https://doi.org/XXXXXXX.XXXXXXX

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, 33–35, 39, 40, 55, 56], 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 a dynamically changing database can render existing models inadequate. Most learned CE models cannot be incrementally updated upon every data update or query execution. Retraining these models, the simplest way to stay up-to-date, can be costly — not only in itself but also in the cost of acquiring new training data. When to retrain CE models is far from clear. Most previous work suggests the simple strategy of retraining periodically [5, 20, 21, 24, 56], in time or the number of data updates since the last retraining, often without clear guidance on how to set these periods. Alternative strategies like fine-tuning have been explored [49, 56], but they are model-dependent and require hyperparameter tuning for different workloads. 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 monitors 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 representative of the query workload, and use them to inform retraining decisions, or simply to override estimates without retraining. For example, LEO [45] remembers errors in estimated cardinalities if they differ from the observed truth by at least 5%, and uses them to correct CE for future queries. Although the idea of leveraging feedback from past query executions is not new, making it practical in a dynamic setting — and compatible with modern ML-based estimator — poses many challenges. First, monitoring incurs overhead, but its benefit is not always clear. Many CE errors may not actually impact plan quality [17]. 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 [17] and confirmed in our experiments. [1] Finally, retraining the CE model does not guarantee more accurate estimates where accuracy actually matters.

[1] For example, in Section 5.1, injecting true CEs into queries of query template Q018 in DSB backfires for 23%, resulting in a 13% increase in end-to-end runtime compared to a baseline system using the CE model without retraining.

Conference acronym 'XX, Woodstock, NY

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

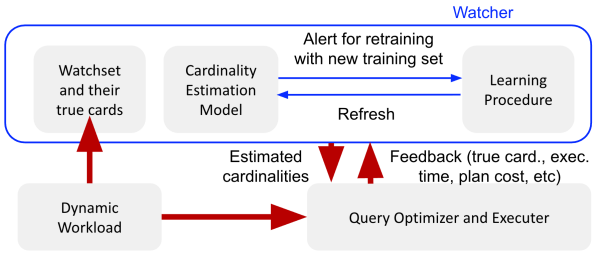


Figure 1: System architecture for watcher-family of methods.

In this paper, we examine the problem of deploying learned CE in a dynamic setting holistically. Instead of focusing narrowly on how a solution improves CE accuracy, we look at its ultimate impact on query performance, as well as its overall cost, including 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.

2 Contributions, Framework, and Preliminaries

2.1 Contributions and Framework

We propose ways to make monitoring more efficient and beneficial beyond retraining — the results are our *watcher*-family of methods. 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. 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 propose two designs, with different trade-offs and levels of sophistication.

- *Watcher1D* **(Section 3)** embodies the basic monitoring idea. It monitors accurate cardinalities for a reservoir sample [47] of single-table selection subqueries, which are straightforward to maintain. We also extend this approach to *Watcher2D*, which maintains the cardinalities of some two-table join subqueries (see details in the full version [25]), but even with the most efficient maintenance methods available, our experiments show that the maintenance cost is too high to provide any practical benefit.
- Going beyond basic monitoring, *SWatcher+* **(Section 4)** 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 *adjust* 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, thereby reducing the monitoring overhead. 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." Compared with cardinalities actively maintained for the watchset, cached information is less accurate, but it is cheaply available even for joins involving more than two tables, and it helps avoid the pitfall of applying only selection corrections. 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.
- Through extensive experiments **(Section 5)** 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, when using MSCN [15] as the black-box CE models, SWatcher+ achieves 44.3% to 101.4% of the end-to-end runtime of periodic retraining, outperforming it in 12 out of 14 query templates, even when retraining is configured with the optimal settings observed post hoc. Using another state-of-the-art CE model JGMP [39], SWatcher+ achieves 56.1% to 99.5% of the end-to-end runtime of periodic retraining and consistently outperforms it for all query templates. SWatcher+ also consistently outperforms the simple monitor-and-correct approach represented by LEO [45], demonstrating the advantage of our smarter, selective monitoring and correction strategy in reducing overhead and avoiding backfiring.

2.2 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.

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.

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 $\text{Opt}(Q, \boldsymbol{\kappa})$ 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, 17, 22, 53], 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 $\text{Cost}(\pi, \boldsymbol{\kappa})$ 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, 53] for cardinality injection, and `pg_hint_plan` [31] for specifying the plan to execute (`EXPLAIN ANALYZE`) and cost (`EXPLAIN`).

3 Watcher1D

This section briefly introduces Watcher1D as a basic monitoring approach. A key observation we shall make is that even with careful data structure design and implementation, its overhead can easily outweighs its potential benefit. This limitation then motivates *selective monitoring* used by our main approach in Section 4.

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 5*

```
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

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

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

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. Intuitively, $\mathcal{W}$ is comparable to query-driven histograms that track accurate counts for regions frequently accessed by queries. 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 linear-time approach has low overhead because of its simple implementation, which works well in practice since $\mathcal{W}_R$ is usually not large. If $\mathcal{W}_R$ is large, more sophisticated data structures can achieve logarithmic query and update costs; see [25] for details.

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 both factors can affect model performance. 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 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 an advanced data structure). Second, for each query Q, besides the overhead of producing $\hat{\boldsymbol{\kappa}}$ using the model, obtaining accurate cardinalities from the execution incurs essentially no overhead, except for any single-table selection subquery 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 incurs a cost; computing and checking the aggregate error measure has negligible cost, and as discussed, generating the training dataset requires no extra query cost.

As observed in Section 5, for a setup with one query for every 100 updated rows and a watchset of size $n_{\text{watch}} = 1500$, 62.7% of Watcher1D's overhead goes to processing DUs; 4.3% goes to processing Qs (entirely due to CE model inference); the remaining 33.0% goes to retraining. Compared to a baseline system deployed with a CE model with no retraining, Watcher1D on average reduces the querying cost by about 5%, but this gain does not offset the maintenance overhead of $\mathcal{W}$. Using more sample-efficient CE models lowers the maintenance overhead of $\mathcal{W}$ by 56% , since n_{watch} decreases accordingly. It also cuts total query execution time by additional 2% (relative to the baseline), though it does not lower the number of retrainings and even increases (re-)training time due to higher model complexity. Totally, the execution time saving (66.13 seconds) still cannot justify the incurred overhead (116.03 seconds).

Strengths and Limitations. 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 [20, 21, 34] 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 [20, 50] incur high costs in running additional queries to acquire up-to-date training data, which pose a signficant overhead that must be accounted for.

However, Watcher1D has two main limitations that impact its effectiveness. First, Watcher1D focuses solely on monitoring CE errors for single-table selection subqueries, while the CE errors for join queries are more challenging to CE models and more influential on plan quality [17, 18, 53]. We have also extended the monitoring idea to track accurate cardinalities for a sample of two-table join subqueries: the technique, called Watcher2D, is detailed in [25], but our experimental results indicate that its monitoring overhead is prohibitive. As a compromise, for SWatcher+, we propose maintaining lightweight, less accurate information for join subqueries that are cheap to track. Second, Watcher1D maintains accurate cardinalities of a sample of single-table selection subqueries without considering their importance. Even though monitoring single-table selection cardinalities is inexpensive, this cost is incurred on every data update for every watchset query involving the table being updated — similar to the overhead of actively maintaining histograms per update. Hence, for SWatcher+, we propose to reduce the overhead by pruning watchset subqueries based on how much benefit they bring to query optimization.

4 SWatcher+

We now describe SWatcher+, our main watcher algorithm that addresses the limitations of Watcher1D. 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 do not maintain accurate cardinalities for join subqueries because of the high overhead. But to avoid the pitfall of correcting selection CE errors alone, we cache "compensating correction factors" for joins and apply them together with selection cardinality corrections.

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 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 $\mathcal{W}$. 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. We detail the steps in Section 4.1.

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

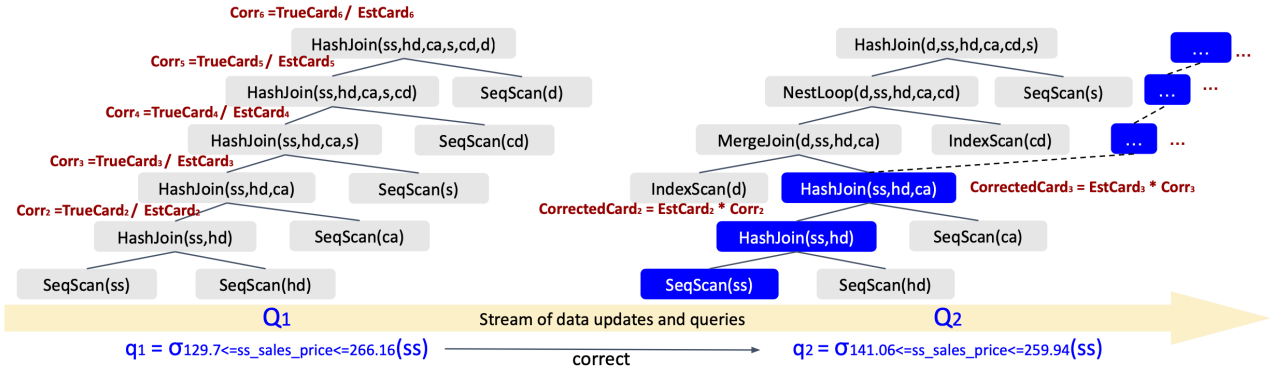


Figure 2: SWatcher+: example of correcting both the single-table selection and the join cardinalities.

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 4.2. Finally, Section 4.3 concludes with cost analysis and discussion.

4.1 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 $\mathsf{sim}(p, p')$, their Jaccard similarity [13] (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 $\mathsf{sim}(p, p')$ as $\frac{\lceil \min(u,u') - \max(l,l') \rceil}{\lceil \max(u,u') - \min(l,l') \rceil}$ 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 $\widehat{\mathsf{card}}(q)$ as $\widehat{\mathsf{card}}(p \wedge \neg\tilde{p}) + \mathsf{sim}(p \wedge \tilde{p}, \tilde{p}) \cdot \mathsf{card}(\tilde{q})$, 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., $\mathsf{sim}(p, \tilde{p}) = 1$), this correction amounts to replacing $\widehat{\mathsf{card}}(q)$ with $\mathsf{card}(\tilde{q})$.

Example 4.1. As shown in Figure 2, Q_2 arrives after Q_1 in the stream of queries and updates. Suppose that Q_1's single-selection subquery q_1 is admitted to $\mathcal{W}$ and not evicted yet. If q_1's condition is the most similar to q_2's — with similarity $\frac{259.94-141.06}{266.16-129.7} = 87.12\%$ exceeding the threshold $\theta_{\mathsf{sim}} = 80\%$ — then $\mathsf{card}(q_1)$ is used to correct $\widehat{\mathsf{card}}(q_2)$ and $\hat{\boldsymbol{\kappa}}[\mathtt{ss}]$. The corrected CE is entered into $\tilde{\boldsymbol{\kappa}}[\mathtt{ss}]$.

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] We illustrate this process in [25] . 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.

Example 4.2. Continuing with Example 4.1, $\hat{\boldsymbol{\kappa}}[\mathtt{ss} \bowtie \mathtt{hd}]$ is computed from $\tilde{\boldsymbol{\kappa}}[\mathtt{ss}]$ and the join selectivity between ss and hd, estimated by the optimizer. It is refined by the compensating corrections factors in $\mathcal{F}_{\{\mathtt{ss,hd,ca,s,cd,d}\},\{q_1\}}(\mathtt{ss} \bowtie \mathtt{hd})$, which was cached after executing Q_1 using the plan illustrated as the tree on the left in Figure 2. Then, we update $\tilde{\boldsymbol{\kappa}}[\mathtt{ss} \bowtie \mathtt{hd}]$. This process is repeated for $\mathtt{ss} \bowtie \mathtt{hd} \bowtie \mathtt{ca}$, $\mathtt{ss} \bowtie \mathtt{hd} \bowtie \mathtt{ca} \bowtie \mathtt{s}$, and beyond, using the compensating corrections factor stored in the index $(J_i = \{\mathtt{ss, hd, ca, s, cd, d}\}, C_i = \{q_1\})$. At the end, $\tilde{\boldsymbol{\kappa}}$, corresponding to the blue bottom-up path in the tree on the right, is injected to get the optimal plan $\mathsf{Opt}(Q_2, \tilde{\boldsymbol{\kappa}})$. Notably, as shown in Figure 2, it is not necessarily the same as the plan for Q_1.

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 $\mathsf{Opt}(Q, \tilde{\boldsymbol{\kappa}})$ to obtain the execution plan π, and records its estimated cost, $\mathsf{Cost}(\pi, \tilde{\boldsymbol{\kappa}})$, and planning time, $\mathsf{T}_{\mathsf{Opt}}(Q)$.

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 with the same signature. For each 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 result

[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.

cardinality of the join subplan divided by the estimate with selection corrections in C_Q but no join corrections otherwise. We also record some basic information for the entry, including the creation timestamp, 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: $\theta_{\mathsf{T_{Exec}}}$, minimum execution time for Q, and $\theta_{\mathsf{T_{Opt}}}$, 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 $\mathsf{T_{Exec}}(\pi) < \theta_{\mathsf{T_{Exec}}}$ or $\mathsf{T_{Opt}}(Q) > \theta_{\mathsf{T_{Opt}}}$, do not admit q. 2) Speedup check: Let $\hat{\boldsymbol{\kappa}}$ the set of model-estimated cardinalities. Let $\boldsymbol{\kappa}'$ 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 $\hat{\pi} = \mathsf{Opt}(Q, \hat{\boldsymbol{\kappa}})$, as well as the plan $\pi' = \mathsf{Opt}(Q, \boldsymbol{\kappa}')$, which would be obtained by admitting q (alone). Let $\boldsymbol{\kappa}^\star$ the set of true cardinalities observed freely from the execution of π. Comparing their costs under $\boldsymbol{\kappa}^\star$ yields the estimated speedup: $\mathsf{gain}(q) = \mathsf{Cost}(\hat{\pi}, \boldsymbol{\kappa}^\star)/\mathsf{Cost}(\pi', \boldsymbol{\kappa}^\star)$. 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 $\mathsf{T_{Exec}}(\pi) \cdot \mathsf{gain}(q) < \theta_{\mathsf{T_{Exec}}}$ or $\mathsf{T_{Opt}}(Q) > \theta_{\mathsf{T_{Opt}}}$, evict q. Note that $\mathsf{T_{Exec}}(\pi) \cdot \mathsf{gain}(q)$ 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 $\mathsf{gain}(q)$.

4.2 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 retraining 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 presented in Section 4.1 — specifically the case when q is not yet a watchset query. Ideally, for the quick checks, we would like to know $\mathsf{T_{Exec}}(\pi)$ and $\mathsf{T_{Opt}}(Q)$, 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 $\mathsf{T_{Exec}}(\pi)$ and $\mathsf{T_{Opt}}(Q)$. For a query Q added to the training set by workload sampling, we use the timings when running Q earlier. Finally, we clear $\mathcal{F}$.

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.

4.3 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 5. Second, for the cost 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 speed-up check in deciding whether these selection subqueries are worth watching can result in up to $k + 1$ $\mathsf{Opt}(\cdot)$ calls (one to get the baseline plan $\hat{\pi}$ and one for correcting each subquery) plus $k + 1$ $\mathsf{Cost}(\cdot)$ calls (to estimate their costs under true cardinalities). Luckily, in practice, quick checks — i.e., assessing whether a query will run sufficiently long without corrections and whether optimization time is short — eliminate 40%–80% of these calls, as observed in our experiments (Section 5). Note that an extra counting query may be needed if the plan executed did not use a selection subquery, but such cases are rare, as discussed in 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 5 will demonstrate the overall effectiveness of SWatcher+ in comparison with other approaches.

Although SWatcher+ has been presented with several parameters, including θ_{sim}, $\theta_{\mathsf{T_{Exec}}}$, $\theta_{\mathsf{T_{Opt}}}$, θ_{gain}, and $\theta_{n_{\mathsf{watch}}}$, we note that setting them is straightforward and does not require extensive tuning. Specifically, users can leave θ_{sim}, $\theta_{\mathsf{T_{Exec}}}$, $\theta_{\mathsf{T_{Opt}}}$, and θ_{gain} at their

[3] In the worst case, SWatcher+ separately maintains a reservoir sample of past queries, and since the cardinalities are not tracked (unlike Section 3), the overhead is negligible.

default settings, discussed in detail in Section 5, 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.

5 Experiment

We implemented the Watcher algorithms in Python3. PostgreSQL V16.2 is used as the default cost-based query optimizer, with cardinality injection implemented using the methods in [8, 53], and plan hinting facilitated by `pg_hint_plan` [31]. 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-EDBT26.

To demonstrate Watcher's ability to work with different learned cardinality estimation models and to assess its effectiveness in improving their effectiveness, we choose two models MSCN [15] and JGMP [39]. MSCN is known for its low model complexity and low training and inference overhead [8, 48]; further, we disable its data-driven bitmaps so MSCN operates a purely query-driven model that learns only from cardinality feedback from queries. JGMP is a more advanced model that incorporates additional information including data samples from each table as well as PostgresSQL's up-to-date single-table selectivity estimates. It is shown to achieve an accuracy comparable to or better than MSCN with even fewer training samples, albeit with a higher training and inference overhead. The main goal of our experimental evaluation is not a comparison of these models per se, but rather how the additional application of Watcher can improve the deployment of both of these models.

Data and updates. The evaluation is on the DSB benchmark [3]. We choose DSB because it contains a large amount of timestamped data, making it easy to construct an update workload affecting many tables. To study the impact of data distribution shifts, we first generate all benchmark tables with a scaling factor of 2 under default settings. In the DSB schema, we identify the following fact tables, each associated with a timestamp column, as "dynamic": tables `ss`, `cs`, `ws` with columns `*_sold_date_sk`; tables `sr`, `cr`, `wr` with columns `*_returned_date_sk`, and table `inv` with column `inv_date_sk`. To simulate a dynamic workload, we use a 60-day sliding window: as the window moves forward, tuples corresponding to the oldest date are deleted, and tuples for the next date are inserted. The initial database instance contains data with `*_date_sk` between 2450815 and 2450874. After applying 60 days of updates, the final database instance reflects a 60-day forward shift with `*_date_sk` between 2450875 and 2450934, representing a completely different database state. The dimension tables remain unchanged throughout. The third column of Table 1 lists, for some of the query templates we evaluated, the total number of data updates in the workload relevant to each query, ranging from 0.08 to 1.55 million.

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.

Template	# Table	# 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	2
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

Queries. DSB [4] features many complex query templates. To ensure that we can run all approaches and underlying CE models being compared and to make the results possible to interpret, we generate our queries from DSB as follows: i) we filter out DSB query templates that did not involve any dynamic tables or lacked selection conditions on them; ii) we remove templates with complex predicates such as `LIKE` or disjunctions; and iii) we further restrict the number of single-table selection conditions in each template to one or two (without changing join conditions). The last simplification step allows us to study the impact of selectivity estimates in a more controlled setting. Table 1 summarizes the query templates we use; the full queries are in [25]. The query templates involve joining between 5 to 9 tables: 12 of them with prefix "Q" contain one additional selection condition each, while 2 with prefix "TJQ" contain two additional selection conditions each. The numerical IDs of the templates are retained from their numbers in DSB.

Next, we generate the query workload by instantiating query constants in the templates by uniformly sampling them from the appropriate domain. For example, a query generated from Q101 may include the selection condition `ss_sales_price / ss_list_price BETWEEN 0.49 AND 0.89`, where 0.49 and 0.89 are randomly generated.

Combined workloads & evaluation metrics. To obtain a combined query/update workload for testing, we generate 3000 queries (Qs for short) and mix them at different rates with data updates (DUs):

- 6-batch: each contains 10 days of DUs followed by 500 Qs;
- 3-batch: each contains 20 days of DUs followed by 1000 Qs.

Before evaluation starts on the testing workload, all methods are given a number of training queries, randomly generated from the same query distribution as the testing workload, along with information obtained by executing them on the initial database instance. Such information includes single-table true cardinalities (required

by Watcher1D, SWatcher+), two-table join true cardinalities (required by Watcher2D), execution time, planning time, and plan cost (required by SWatcher+), as well as any additional information required for training the underlying CE models.

Since the overall workload performance can be difficult to interpret for a mixture of different query templates, we choose a simpler experimental design where we evaluate each query template in a separate workload. We refer to a 6-batch workload testing a query template by the template itself (e.g., Q013a, TJQ013b), and we refer to a 3-batch workload for a query template by appending "-3B" to the template (e.g., Q013a-3B, TJQ013b-3B).

We note that across templates, the above workloads cover a wide range of characteristics from OLAP to OLTP. Since DSB is OLAP-centric, many templates are complex and expensive (e.g., Q013a, Q013b, and Q100) and hence stand to benefit more from accurate CEs. On the other hand, the way we extracted templates from DSB led to a few (e.g., Q040, Q101) whose execution times are lower than their optimization times, resulting in workloads resembling OLTP. Intuitively, a smart system should avoid paying maintenance and optimization costs for such templates. We still keep such templates in our workloads, to study how different approaches adapt to them. The full version [25] contain results on additional templates that are even simpler (i.e., single-table point queries).

For metrics, we use Q-error [29] to measure CE (in-)accuracy and define *overhead* as the end-to-end running time excluding query execution, including CE model inference time and the cost of maintaining required information for each method.

Methods & parameters setup. The training set size is 1500 for MSCN and 128 for JGMP (128 is the smallest number tested in [39] in which JGMP still outperforms MSCN) with following methods:

- **True**: Always inject true cardinalities for monitored subqueries, with no overhead accounting. While unrealstic, it allows us to evaluate the benefit and limit of knowing true cardinalities.
- **Baseline**: Always use the same CE model with no updates.
- **Periodic**: Periodically retrain the model after every X days of data updates, where $X \in \{10, 20, 30\}$ reflects varying degrees of timeliness. For instance, in a 6-batch setting, $X = 10$ is ideal because it triggers retraining immediately after every batch of DUs, while $X = 20$ means that the 1st, 3rd, and 5th batches are handled by models that lag behind by 10 days of DUs.
- **LEO**: We implemented the monitor-and-correct approach from LEO [45], adapting it to our setting while remaining as faithful as we could. Specifically, LEO records adjustments needed for a subquery whenever the estimate deviates from the truth by at least 5%. For an incoming query, LEO applies an adjustment if it matches a tracked subquery with at least 80% similarity. LEO also updates the adjustments according to data updates.
- **Watcher1D**: Deploy the model with Watcher1D, using $n_{\text{watch}} = n_{\text{test}} = n_{\text{train}} = 1500$ for MSCN and 128 for JGMP; ϵ is 90th (P90) or 95th (P95) percentile of the Q-errors observed during training.
- **Watcher1D+Watcher2D**: Deploy the model with Watcher1D and Watcher2D, with the same settings as above. Recall that Watcher2D is an extension of the Watcher1D idea to joins involving two local selection conditions; more details are available in [25]. This method is not applicable to workloads that test only queries with one selection conditions.
- **SWatcher+**: Deploy the model with SWatcher+. We set $\theta_{\text{sim}} = 0.8$, $\theta_{n_{\text{watch}}} = 600$, and $\theta_{\text{T}_{\text{Opt}}} = 10\% \cdot \theta_{\text{T}_{\text{Exec}}}$ (sensitivity analysis in [25] shows that changes to these settings within sensible ranges have little impact on results). For $\theta_{\text{T}_{\text{Exec}}}$, we choose a value to separate slow and fast queries in the training set when such a separation is evident (see example in [25]) ; otherwise, we set $\theta_{\text{T}_{\text{Exec}}}$ to around the 50th or 75th percentile. These $\theta_{\text{T}_{\text{Exec}}}$ settings are summarized in Table 1. Next, the choice of θ_{gain}, as with the training data size, depends on the characteristic of the underlying model. We set $\theta_{\text{gain}} = 101\%$ for MSCN and 100.01% for JGMP: since JGMP generally produces better estimates at the expense of higher overhead, it makes sense for SWatcher+ to be more aggressive in monitoring and correcting with a lower θ_{gain}. Finally, as an ablation study, we also test a version of SWatcher+ without correcting two-table join subqueries, though the default (with such corrections) performs better overall.

5.1 MSCN with/without Watcher

We start with results evaluating the benefits of applying Watcher-family of methods with MSCN as the underlying CE model; results for JGMP are discussed in Section 5.5. We also focus on workloads for query templates with a single selection condition here (i.e., those starting with "Q"); results for two selection conditions (i.e., those starting with "TJQ") are discussed in Section 5.4.

For each single-selection query template, we measure the end-to-end running time for each method and report the results using their configuration optimized for the lowest running time. For these workloads, we let True acquire and inject true cardinalities for single-table selections for free. We report the Periodic setting with the best performing X, observed post hoc, which would be impractical in a real deployment. Indeed, 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 3, and those for the 3-batch setting in [25]. In Figure 3 (and later in Figure 6 for JGMP), the end-to-end running times are all normalized against True's, which are presented in the X-axis labels. We divide the query templates into four groups below based on key observations about their results; we will dive deeper into Groups 1 and 2 later.

Group 1 (True-improves): Q013a, Q013b, Q100. These templates exhibit a noticeable performance gap of approximately 10% between True and Baseline, indicating that injecting true selection cardinalities into the monitored subqueries of these templates significantly improves query execution times. SWatcher+ achieves the lowest end-to-end running time, outperforming Baseline, Periodic, Watcher1D, and LEO. By applying corrections to model 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 models from Periodic. Conversely, Watcher1D (resp. LEO) incurs much higher overhead due to the cost of maintaining a large $\mathcal{W}$ (resp. set of adjustments), which outweighs execution time savings. A detailed analysis is provided in Section 5.2 through an ablation study and a breakdown of the overhead for each method.

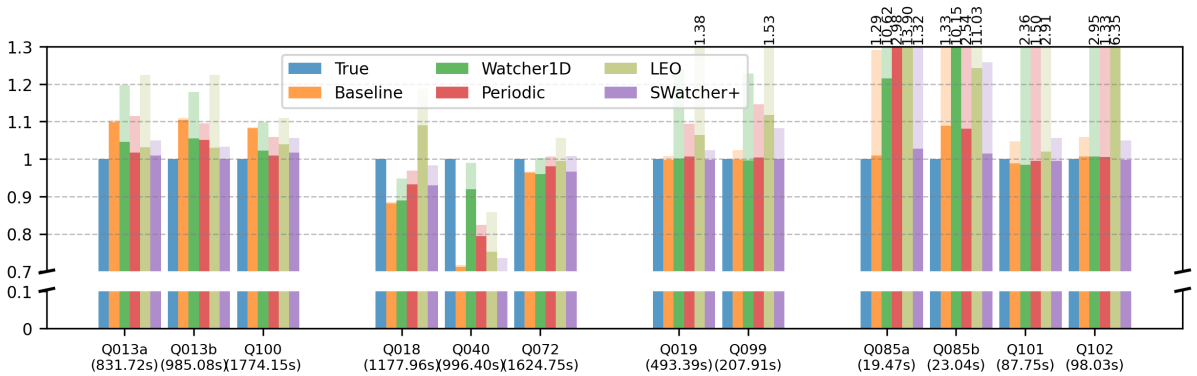


Figure 3: Normalized end-to-end running time, including both total query execution time (represented by the lower portion of the bar with darker shade) and overhead (represented by the upper portion with lighter shade), using MSCN as the CE model, across all methods and single-selection templates. A number is above the bar if its height falls outside the scope. The number below the template name is the end-to-end running time of True, used for normalization.

Group 2 (True-backfires): Q018, Q040, Q072. For this group, injecting true selection cardinalities can backfire, causing a query to execute slower. This is because PostgreSQL's optimizer, while given the true selection cardinalities, still relies on unrealistic assumptions about join selectivity; therefore, it can produce worse estimates for joins and hence worse final plans even if the injected selection cardinalities are true. Strategies such as Periodic, Watcher1D, and LEO face challenges in avoiding these 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 running times 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. Adapting to these cases is even harder for LEO, since it corrects single-selection subquery CEs whenever possible; maintaining accurate adjustments for joins under data changes makes it even more costly. Further discussion is presented in Section 5.3.

Group 3 (True-oblivious): 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 methods. 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 $\mathcal{W}$. For 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. LEO incurs significant overhead in maintaining and applying adjustments, because its actions ignore the impact on performance.

Group 4 (opt-dominated): 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 ≤ 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 overhead. However, both Periodic and Watcher1D incur non-trivial overheads due to unnecessary retraining, and LEO wastes resources on unnecessary adjustments. For instance, in Q101, Periodic spends 39.18s for a single retraining cycle (including collecting the retraining set). Watcher1D takes 20.51s for a single retraining cycle and 99.78s for maintaining $\mathcal{W}$. LEO takes 165.47s for the adjustments. These additional costs exceed the total query execution time, which remains under 90s in all cases.

5.2 Analysis of *True-improves* (Q013b)

We take a closer look at Q013b, from **Group 1** (*True-improves*). First, we perform a componentized running time analysis, shown in Table 2, across all methods with varying parameters. For Periodic, triggering retraining after every batch of DUs ($X = 10$) achieves the lowest total query execution time—over 20s shorter than the other X values—though it incurs longer retraining time. For Watcher1D, using ϵ of P95 saves two retraining operations (over 40s) compared to P90 but results in additional 17s of total execution time. Both configurations exhibit high DU overhead for maintaining a $\mathcal{W}$ of 1500 subqueries. For LEO, applying adjustments whenever possible leads to a mix of savings (e.g., see total $\mathsf{T}_{\mathsf{Exec}}(Q)$ in potent region) and regressions (e.g., see total $\mathsf{T}_{\mathsf{Exec}}(Q)$). For SWatcher+, correcting two-table join subqueries yields significantly better savings (>25s) compared with lowering θ_{gain} from 110% to 101% (<10s). SWatcher+'s query processing overhead is 5× higher than Baseline, Periodic, Watcher1D, and slightly higher than LEO, owing to its corrections and admission-eviction checks. However, SWatcher+'s DU processing overhead is low, with $|\mathcal{W}| < 50$ in all cases.

Table 2: Q013b: componentized running times. †: the parameters, shared within the same method and detailed previously, are omitted. The last five 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 running time for each method and serve as the representatives in Figure 3.

Method	Settings†	Total $T_{Exec}(Q)$	Retraining Time	Overhead on Qs	Overhead on DUs	Total $T_{Exec}(Q)$ 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**
LEO	-	**1014.58**	**0(0)**	**19.03**	**172.67**	**82.31**
SWatcher+	no join correction, θ_{gain} = 110%	1021.89	0(0)	28.18	3.70	110.13
SWatcher+	θ_{gain} = 110%	989.53	0(0)	26.71	3.42	95.32
SWatcher+	no join correction, θ_{gain} = 101%	1011.96	0(0)	27.57	5.31	99.92
SWatcher+	**θ_{gain} = 101%**	**985.67**	**0(0)**	**26.31**	**4.88**	**90.68**

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

Method	Settings†	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
LEO	-	**1.02**	**1.06**	**1.20**	**1.90**
SWatcher+	θ_{gain} = 110% no join correction	1.39	1.72	2.16	2.78
SWatcher+	θ_{gain} = 110%	1.40	1.75	2.20	2.83
SWatcher+	θ_{gain} = 101% no join correction	1.37	1.70	2.15	2.72
SWatcher+	θ_{gain} = 101%	1.38	1.72	2.15	2.74

Excluding overhead on DUs, Watcher1D achieves lower running time than Periodic when both perform the same numbers of retraining, suggesting that Watcher1D identifies better timing for retraining 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 saving does not justify its high $\mathcal{W}$ cost.

Next, as discussed in Section 1, sometimes inaccurate CEs have little impact on query performance; ideally, we would like to avoid the overhead of correcting these CEs. For Q013b, Figure 4 shows, for each of its 3000 queries, the "ideal saving" achievable by True over Baseline through injecting true selection cardinalities. We order the queries in the figure by the amount of ideal saving. The results show that for the leftmost 5% queries, injection backfires; the majority of the queries (in the middle) are insensitive to CE accuracy; the rightmost 10% of queries, located in what we call the "potent" region, stand to benefit from accurate CEs, with potential savings of at least 49ms per query. This region is exactly the target of SWatcher+. As demonstrated in the last column of Table 2, SWatcher+ achieves low total query execution time in the potent region and is outperformed only by LEO. LEO's consistent adjustments strategy yields both benefits and drawbacks – for instance, it

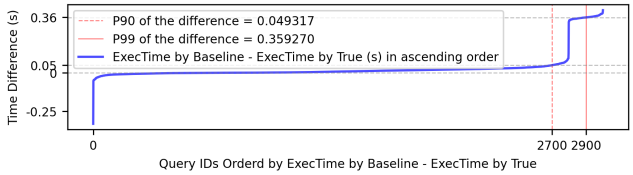


Figure 4: Q013b: "ideal saving" computed and ordered by the difference in execution time between Baseline's and True's.

may backfire (see explanation in Section 5.3) or suffer from inaccurate interpolation, as seen in total $T_{Exec}(Q)$. Because this strategy is agnostic to performance impact, and because maintaining adjustments under DUs incurs high overhead, LEO falls short overall, compared with SWatcher+'s performance-aware design.

Furthermore, in Table 3, we present the 50th, 75th, 90th, and 95th percentiles of Q-errors observed for single-selection CEs over the 3000 queries in the testing workload across all methods. LEO achieves most accurate CEs across all percentiles compared to the others, by maintaining and applying adjustments. Relaxing ϵ from P90 to P95 in Watcher1D reduces accuracy. Notably, retraining does not always yield a more accurate model. For instance, for Periodic, the setting of $X = 10$ retrains strictly more than the setting of $X = 20$, yet its errors are comparable and sometimes worse. Finally, SWatcher+'s Q-errors overall are better than Baseline but worse than LEO, Periodic, and Watcher1D. At a first glance, this observation may seem surprising since SWatcher+ delivers better end-to-end performance. However, it in fact highlights the effectiveness of SWatcher+'s approach — focusing only on reducing errors that matter to query performance, e.g., selected subqueries including joins, instead of fixating on improving selection CE errors.

Finally, when join correction is on and θ_{gain} = 101%, SWatcher+ selects only 1% of single-table selection subqueries and corrects 13% of queries only in this workload. The growth and utilization of $\mathcal{W}$ are illustrated in [25]. Among queries in the potent region (Figure 4), however, SWatcher+ corrects 58% of them, and 89% of those corrected are among the top 1% most potent. In short, SWatcher+ effectively focuses its efforts on queries with high payoff.

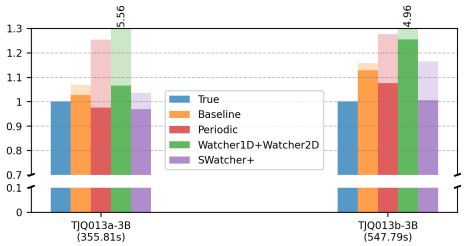


Figure 5: Normalized end-to-end running time for TJQ013a and TJQ013b; same format as Figure 3.

5.3 Analysis of *True-backfires*

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 injection of true CEs. Meanwhile, 83%, 66.9%, and 75% queries remain insensitive to the injected CEs, and only a small fraction of queries run faster when true CEs are injected. The primary reason for this backfiring is the lack of accurate estimation for other subqueries. Injecting true CEs into single-selection subqueries sometimes worsens join CEs, leading to overall inaccurate cost estimation and suboptimal plan. This leads to serious backfiring for LEO, which applies adjustments to over 90% of all queries. On the other hand, the high overhead of maintaining accurate join adjustments under data changes (as Watcher2D does) is unaffordable, especially since LEO is not selective about what to maintain. SWatcher+ mitigates the issue in two ways. First, as observed in Q040, the admission-eviction checks effectively determine cases when injecting more accurate CEs would not improve performance, thereby avoiding caching ineffective subqueries in the watchset and preventing undesirable corrections. Second, as demonstrated in Q018 and Q072, SWatcher+ further applies corrections to join CEs, counteracting the inaccuracies stemmed from injecting single-selection subqueries solely, thereby guiding plan selection back onto the right track. Overall, among queries for which True backfires, SWatcher+ achieves the following improvements in total query execution time compared with True: 53s (22.3%) in Q018, 283s (98.2%) in Q040, and 67s (23.5%) in Q072. As an example, for Q018, about 89.1% of the queries receive corrections in both single-selection and two-table join subqueries, contributing 73.6% of the total savings for this query template.

5.4 Results for Query Templates with Two Selection Conditions (TJQ)

We briefly discuss results on query templates TJQ013a and TJQ013b, which involve local selections on both tables ss and hd, followed by a join on ss_hdemo_sk = hd_demo_sk. The three subqueries of interest are hence $\sigma(\texttt{ss})$, $\sigma(\texttt{hd})$, and $\sigma(\texttt{ss}) \bowtie \sigma(\texttt{hd})$. For these workloads, we let True acquire and inject true cardinalities for all three subqueries of interest for free. Baseline, Periodic, and Watcher1D+Watcher2D each train models for all three subqueries of interest. We omit experiments for LEO due to poor scalability: the adjustment updates under data changes incur unaffordable overhead, the same issue faced by Watcher2D. For all three Periodic instances, we set $X = 30$, while in Watcher1D+Watcher2D, ϵ = P90. Finally, SWatcher+ trains models for $\sigma(\texttt{ss})$ and $\sigma(\texttt{hd})$, but relies on its $\mathcal{F}$ for join CE correction. For ablation study, we test SWatcher+ without correcting join CEs, and with an alternative θ_{gain} setting.

End-to-end running time results in Figure 5 show that SWatcher+ outperforms Periodic and Watcher1D+Watcher2D, achieving a total query execution time close to or 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 running time for both query templates. This underscores the impracticality of maintaining highly accurate join cardinalities during query optimization, even for simple two-way joins.

5.5 JGMP with/without Watcher

We repeat our experiments with JGMP (in Figure 6) as the underlying CE model instead of MSCN,[4] mirroring those in Figure 3. On the same workloads, JGMP achieves better cardinality estimation accuracy but incurs higher costs for both (re-)training (even with fewer training queries) and inference, compared with MSCN—consistent with the findings in [39]. The reason is that JGMP requires fresh samples from the tables for each retraining, and uses PostgreSQL's up-to-date single-table selection estimates for inference, thereby making the model more aware of the current database state. Despite the different design trade-offs made by JGMP vs. MSCN, applying SWatcher+ to JGMP requires only straightforward changes to the training data size and θ_{gain}, as explained in the setup.

Comparing Figure 3 and Figure 6, we observe a shift: more templates, such as Q100 (*True-improves* for MSCN), Q018 and Q072 (*True-backfires* for MSCN), now fall into Group 3 (*True-oblivious*), indicating that JGMP's improved accuracy makes the query performance less sensitive to CE errors, confirming the advantage of incorporating data summaries or real-time statistics in improving CE models. It also generally reduces the number of adjustments LEO requires across all workloads. Moreover, since JGMP's CE models are more sample-efficient, they lead to a significantly smaller n_{watch} and lower overhead in Watcher1D compared to using MSCN as the black box. As a result, Watcher1D outperforms Periodic in two workloads—Q013b and Q085b—despite the latter's post hoc configuration with the best-performing retraining interval.

Next, we see that SWatcher+ outperforms Baseline, Watcher1D and Periodic across all workloads in Groups 1 (*True-improves*), 2 (*True-backfires*), and 4 (*opt-dominated*). This consistent improvement demonstrates that the corrections applied by SWatcher+ effectively reduce the total query execution time and justify the overhead of maintaining $\mathcal{W}$. Even in Group 3 (*True-oblivious*), which consists of workloads less sensitive to CE errors, SWatcher+'s performance-aware selection strategy correctly identifies the limited benefits of corrections and instead relies primarily on the initial CE model without retraining—still outperforming for 4 out of 5 templates.

Finally, there are a few templates (3 out of 11) where LEO outperforms SWatcher+: Q019 and Q099 in Group 3 (*True-oblivious*), and Q102 in Group 4 (*opt-dominated*). In these cases, thanks to JGMP's strength, LEO requires fewer adjustments (though still more than SWatcher+). However, SWatcher+ incurs some overhead in assessing potential performance gains on these queries, before realizing

[4]The current JGMP release [38] does not support Q101, where the selection condition is on an expression ss_sales_price/ss_list_price rather than a single column.

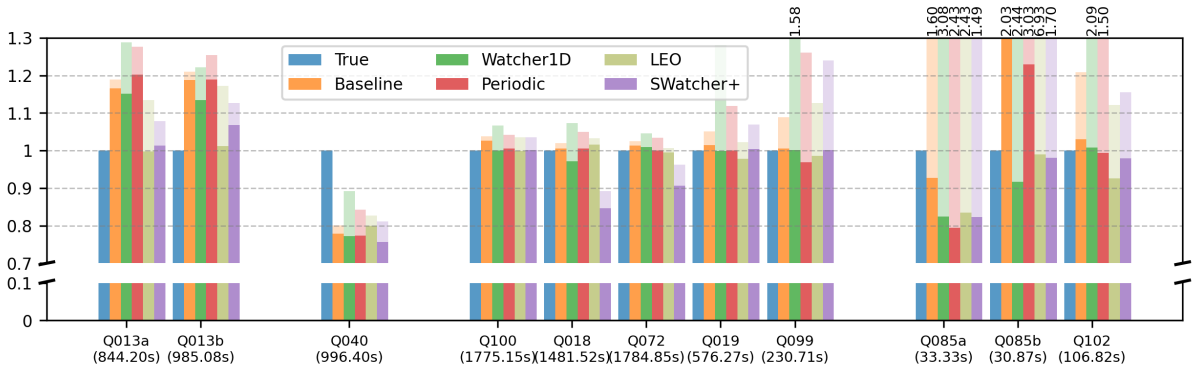


Figure 6: Normalized end-to-end running time using JGMP as the black-box; same format as Figure 3.

that maintaining information for these queries is not worthwhile. Nonetheless, the difference in end-to-end performance is not big, as SWatcher+'s query execution times remain competitive.

6 Related Work

Watcher is a great fit for *query-driven* CE methods, due to their ability to leverage query workloads. Recent studies on Amazon Redshift [52] and on learned query optimization [46, 57] found that many real-world queries repeatedly access similar regions. Query-driven CE models span from traditional statistical summaries [2, 44] to ML-based approaches [5, 12, 15, 24, 30, 32, 35, 39, 49]. Among the above, we have tested Watcher with: MSCN [15], which uses a multi-set query representation, remains a leading query-driven cardinality estimator [14, 48], and JGMP [39] has demonstrated better sample efficiency, which reduces (re-)training overhead.

Recently, more works focus on adapting learning-based CE models on evolving databases. Unlike Watcher, which is model-agnostic, these works build dedicated models that (re-)train or fine-tune upon detecting distribution shifts [5, 24, 35, 48, 49, 56]. DDUp [16] targets data drift on CE models based on neural networks. Warper [20] detects drift statistically and retrains, using GANs to generate new samples that are labeled by extra executions. Wu et al. [50] mitigates workload drift with a replay buffer. Another line of work improves model robustness: RobustMSCN [34] reduces retraining frequency at the cost of higher model complexity, and NeuroCDF [51] enhances generalization to OOD queries. Hybrid approaches ALECE [22] and [10] can directly incorporate data updates. Despite these advances, how to deploy CE models in dynamic environments remains unsolved. In practice, significant data changes still necessitate frequent retraining/refreshing.

Another class of CE models is *data-driven*, with underlying techniques again spanning from traditional data summaries [10, 19, 23, 36, 37]) to ML [9, 11, 49, 55, 56, 59]. They are generally more robust to workload drift, and process data updates more readily. However, modeling joint data distributions is hard and less effective for complex queries, and maintaining such models is expensive, often demanding frequent retrainings [34]. Therefore, Watcher embraces the data-driven approach when maintaining $\mathcal{W}$, but selectively chooses what to maintain.

The idea of guiding future optimiaztion with query feedback has been explored for decades, beginning with LEO [45] and later productized in Microsoft SQL Server as CE feedback [43]. The latter focuses on en/dis-abling certain optimizer options – such as assuming predicate correlation or independence – rather than deploying learned CE models. LEO [45] targets correcting optimizer estimation errors, but as discussed earlier in Sections 1 and 5, its criterion for what to maintain and correct is based purely on the amount of CE error, whereas Watcher's decision is holistic. Additionally, Watcher considers practical retraining for learned CE models, whereas LEO assumes more traditional CE models. Finally, other lines of work [26, 54, 58] explore replacing the entire optimizer with ML-based modules that directly output query plans.

7 Conclusion and Future Work

In this paper, we have studied the problem of deploying learned CE models in practical dynamic settings. We take a holistic approach in both design and evaluation, considering the ultimate impact of CE on query performance, as well as the overhead in maintaining, retraining, and applying CE models. Our results highlight the pitfall of either fixating on improving single-table CEs (simply because it is easy to do) or overcommitting on maintaining join CEs (simply because it may yield greater improvements in query performance). Instead, our SWatcher+ effectively reduces overhead by judiciously deciding what to maintain through a benefit-based analysis. By design, it automatically identifies high-impact CEs 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, outperforming periodic retraining even when the latter is given an optimal post hoc interval. Furthermore, SWatcher+ delivers reasonable performance even in challenging cases where the benefit margin is razor-thin.

There are several directions for further improving SWatcher+, e.g., monitoring update costs, batch-based update processing, using sketching to maintain CEs under updates, supporting other model refresh strategies beyond retraining (e.g., fine-tuning), etc., which we further detail in [25]. While there is room for improvement, we believe that SWatcher+, along with its key principles — selective monitoring, active CE correction, and a focus on end-to-end running time rather than raw CE error metrics — offers a practical solution to the deployment of learned CE models, as well as valuable insights that can inform future advancements in this area.

8 Artifacts

The full version, codes, detailed instructions and datasets are available at https://github.com/louisja1/Watcher-EDBT26.

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. doi: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] Mike Heddes, Igor Nunes, Tony Givargis, and Alex Nicolau. 2024. Convolution and Cross-Correlation of Count Sketches Enables Fast Cardinality Estimation of Multi-Join Queries. *Proc. ACM Manag. Data* 2, 3, Article 129 (May 2024), 26 pages. doi:10.1145/3654932

[11] 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.

[12] 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.

[13] 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. doi:10.5169/seals-266450

[14] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. 2022. Learned Cardinality Estimation: An In-depth Study. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1214–1227. doi:10.1145/3514221.3526154

[15] 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.

[16] 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. doi:10.1145/3588713

[17] 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. doi:10.14778/3611479.3611494

[18] 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. doi:10.14778/2850583.2850594

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

[20] 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*.

[21] 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/

[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 (Oct. 2023), 197–210. doi:10.14778/3626292.3626302

[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. doi:10.14778/3476249.3476254

[25] Yuxi Liu, Vincent Capol, Xiao Hu, Pankaj Agarwal, and Jun Yang. 2025. [Full Version] SWatcher+: A Holistic Approach to Deploying Learned Cardinality Estimators in a Dynamic Setting. https://github.com/louisja1/Watcher-EDBT26/blob/main/fullversion.pdf

[26] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1275–1288. doi:10.1145/3448016.3452838

[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] Magnus Müller, Lucas Woltmann, and Wolfgang Lehner. 2023. Enhanced Featurization of Queries with Mixed Combinations of Predicates for ML-based Cardinality Estimation. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*, Julia Stoyanovich, Jens Teubner, Nikos Mamoulis, Evaggelia Pitoura, Jan Mühlig, Katja Hose, Sourav S. Bhowmick, and Matteo Lissandrini (Eds.). OpenProceedings.org, 273–284. doi:10.48786/EDBT.2023.22

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

[32] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-loss: learning cardinality estimates that matter. *Proc. VLDB Endow.* 14, 11 (July 2021), 2019–2032. doi:10.14778/3476249.3476259

[33] 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.

[34] 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. doi:10.14778/3583140.3583164

[35] 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.

[36] 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.

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

[38] Silvan Reiner and Michael Grossniklaus. 2023. Github repository for JGMP. https://github.com/dbis-ukon/jgmp. Accessed: March 27, 2025.

[39] Silvan Reiner and Michael Grossniklaus. 2023. Sample-Efficient Cardinality Estimation Using Geometric Deep Learning. *Proc. VLDB Endow.* 17, 4 (Dec. 2023), 740–752. doi:10.14778/3636218.3636229

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

[41] 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.

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

[43] Microsoft SQL Server. 2025. Cardinality estimation (CE) feedback. https://learn.microsoft.com/en-us/sql/relational-databases/performance/intelligent-query-processing-cardinality-estimation-feedback?view=sql-server-ver16. Accessed: Sep 18, 2025.

[44] 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.

[45] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 19–28.

[46] Jeffrey Tao, Natalie Maus, Haydn Jones, Yimeng Zeng, Jacob R. Gardner, and Ryan Marcus. 2025. Learned Offline Query Planning via Bayesian Optimization. *Proc.*

ACM Manag. Data 3, 3, Article 179 (June 2025), 29 pages. doi:10.1145/3725316

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

[48] 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. doi:10.14778/3461535.3461552

[49] 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.

[50] 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. doi:10.1145/3639293

[51] Peizhi Wu, Haoshu Xu, Ryan Marcus, and Zachary G. Ives. 2025. A Practical Theory of Generalization in Selectivity Learning. *Proc. VLDB Endow.* 18, 6 (Aug. 2025), 1811–1824. doi:10.14778/3725688.3725708

[52] Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Stage: Query Execution Time Prediction in Amazon Redshift. arXiv:2403.02286 [cs.DB] https://arxiv.org/abs/2403.02286

[53] 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).

[54] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. 931–944. doi:10.1145/3514221.3517885

[55] 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.

[56] 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.

[57] Zixuan Yi, Yao Tian, Zachary G. Ives, and Ryan Marcus. 2024. Low Rank Approximation for Learned Query Optimization. In *Proceedings of the Seventh International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Santiago, AA, Chile) *(aiDM '24)*. Association for Computing Machinery, New York, NY, USA, Article 4, 5 pages. doi:10.1145/3663742.3663974

[58] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1466–1479. doi:10.14778/3583140.3583160

[59] 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 ϵ for triggering retraining, the dynamic workload L, the watchset size n_{watch}, the trainingset size n_{train} (<= n_{watch}), and the test set size n_{test} (<= n_{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 π and obtain (or compute) the true cardinality $\text{card}(q)$;
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_{test} from $\mathfrak{m}$ of size n_{test}; ▷ get a test set D_{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_{train} from $\mathcal{W}$ of size n_{train};
26: Alert for retraining the CE model with the new training set D_{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 $\text{card}(q)$ 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.

More Complex Data Structure for $\mathcal{W}$. 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, 42] 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.

B 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, 18, 48]. 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.

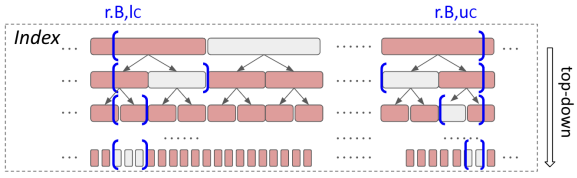


Figure 7: Watcher2D: top-down computation of one counting query. Blue parentheses is the condition $B = r.B \wedge C \geq l_C \wedge C < u_C$ or the sub-conditions generated and pushed down to the lower layers. Gray (resp. red) squares are ranges that are (resp. are not) fully covered by the conditions.

First, we determine the subset $\mathcal{W}'_{RS}$ of $\mathcal{W}_{RS}$ for which $r.A$ passes the local selection condition $l_A \leq r.A < u_A$. Only queries in $\mathcal{W}'_{RS}$ 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* [41], 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 7. 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 5, 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 5, 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.

Details of Implementation. 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 above, 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 $\{|\sigma_{B=r.B \wedge C \geq l_C \wedge C < u_C} S| \mid \forall \sigma_C = (l_C, u_C) \in \mathcal{W}\}$.

To facilitate this, Watcher2D constructs a hierarchical index over columns B and C. Each node in this tree[5] is a key-value pair, where the key ecnodes a triplet $(\overline{b}, \overline{c_0}, \overline{c_1})$, representing a predicate $B = \overline{b} \wedge C \geq \overline{c_0} \wedge C < \overline{c_1}$. The value stores the count of tuples in S satisfying this predicate, i.e., $|\sigma_{B=\overline{b} \wedge C \geq \overline{c_0} \wedge C < \overline{c_1}} S|$. 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 $|\sigma_{B=\overline{b}} S| \geq \theta\}$, with information related to infrequent B-values stored only in the bottom layer. Additionally, a metadata store tracks column domains and

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

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[6].

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 $|\sigma_{B=r.B \wedge C \geq l_C \wedge C<u_C} S|$ via a top-down traversal of the hierarchical index. At the topmost layer, if a contiguous sequence of key-value pairs $(r.B, c_x) \rightarrow count_x, (r.B, c_{x+1}) \rightarrow count_{x+1}, \ldots, (r.B, c_y) \rightarrow count_y$ are fully covered by $r.B$ and range $[l_C, u_C)$, i.e., $c_x \geq l_C \wedge c_y < u_C$, part of the answer is obtained as $\sum_{i=x}^{y} count_i$. Then, it pushes down 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.

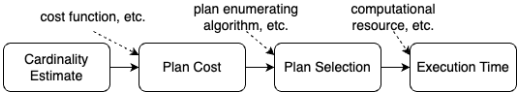


Figure 8: From cardinality estimate to query execution time

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 8 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 $\widehat{\text{card}}(q)$ 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[7]. 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 $\text{card}(q)$, as well as the speedup $\text{gain}(q)$, 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 $\widehat{\text{card}}(q)$ by the currect $\mathfrak{m}$. Then, in lines 4-5, we collect information needed by Detect from

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

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

$\mathcal{W}_{\text{old}}$ 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 4. 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}.

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 already 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, θ_{sim} for applying correction, a triple $(\theta_{\mathsf{T}_{\text{Exec}}}, \theta_{\mathsf{T}_{\text{Opt}}}, \theta_{\text{gain}})$ parameters for the detection, a size budget $\theta_{n_{\text{watch}}}$ for triggering retraining, and the trainingset size n_{train}.

1: $\mathfrak{m} \leftarrow \mathfrak{m}_0$; ▷ deploy the estimator
2: $\mathcal{F} \leftarrow \emptyset$;
3: $\mathcal{W} \leftarrow \text{Refill}(D_0, \mathfrak{m}, \mathcal{F}, (\theta_{\mathsf{T}_{\text{Exec}}}, \theta_{\mathsf{T}_{\text{Opt}}}, \theta_{\text{gain}}))$ ▷ initialize watchset by Algorithm 3
4: add all q in D_0 to the query pool; ▷ we maintain a n_{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: $\mathcal{W} \leftarrow \text{Refill}(\mathcal{W}, \mathfrak{m}, \mathcal{F}, (\theta_{\mathsf{T}_{\text{Exec}}}, \theta_{\mathsf{T}_{\text{Opt}}}, \theta_{\text{gain}}))$; ▷ 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: $\tilde{\boldsymbol{\kappa}} \leftarrow \text{Correct}(Q, q, \hat{\boldsymbol{\kappa}}, \mathcal{W}, \mathcal{F}, \theta_{\text{sim}})$; ▷ correct the cardinality estimate by $\mathcal{W}$ and $\mathcal{F}$ if possible (Algorithm 5)
19: $\pi \leftarrow \text{Opt}(Q, \tilde{\boldsymbol{\kappa}})$; ▷ get the plan by injecting $\tilde{\boldsymbol{\kappa}}$
20: execute Q by π, record $\mathsf{T}_{\text{Exec}}(\pi)$, $\text{Cost}(Q, \tilde{\boldsymbol{\kappa}})$, $\mathsf{T}_{\text{Opt}}(Q)$ and a set C of true join cardinalities observed for π, and get $\text{card}(q)$ from the execution or compute it through a report query;
21: update $\mathcal{F}$ by a new mapping $\mathcal{F}_{J_Q, C_Q} \coloneqq \mathcal{F}_{J_Q, \{q\}} \coloneqq \{J' \rightarrow \frac{\text{card}(J')}{\widehat{\text{card}}(J')} \mid \forall J' \in C\}$ as well as the creation timestamp, the query Q, and $\mathsf{T}_{\text{Exec}}(\pi)$, $\mathsf{T}_{\text{Opt}}(Q)$; ▷ $\text{card}(J')$ is observed from executing π, and $\widehat{\text{card}}(J')$ is in $\tilde{\boldsymbol{\kappa}}$
22: $\phi, \text{gain}(q) \leftarrow \text{Detect}(\theta_{\mathsf{T}_{\text{Exec}}}, \theta_{\mathsf{T}_{\text{Opt}}}, \theta_{\text{gain}}, Q, \text{card}(q), \widehat{\text{card}}(q), C, \mathsf{T}_{\text{Exec}}(\pi), \mathsf{T}_{\text{Opt}}(Q), \mathcal{W})$ ▷ detect if to admit/evict q to/from $\mathcal{W}$
23: **if** ϕ **then** ▷ passes the checks
24: **if** $q \in \mathcal{W}$ **then**
25: update $\text{gain}(q)$ cached in $\mathcal{W}$;
26: **else**
27: $\mathcal{W}[q] \leftarrow (\text{card}(q), \text{gain}(q))$ ▷ 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: $\text{Retrain}(n_{\text{train}}, \mathcal{W}, \text{the access to the query pool})$;
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 $\text{card}(q)$ 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: $\widehat{\text{card}}(q) \leftarrow \mathfrak{m}(q)$; ▷ get the cardinality estimate by $\mathfrak{m}$
6: $\phi, \text{gain}(q) \leftarrow \text{Detect}(\theta_{\mathsf{T}_{\text{Exec}}}, \theta_{\mathsf{T}_{\text{Opt}}}, \theta_{\text{gain}}, Q, \text{card}(q), \widehat{\text{card}}(q), \emptyset, \mathsf{T}_{\text{Exec}}(\pi), \mathsf{T}_{\text{Opt}}(Q), \mathcal{W})$; ▷ detect if to admit/evict q to/from $\mathcal{W}$
7: **if** ϕ **then** ▷ passes the checks
8: $\mathcal{W}_{\text{new}}[q] \leftarrow (\text{card}(q), \text{gain}(q))$ ▷ 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 $\text{card}(q)$ of the watchset query, the estimated cardinality $\widehat{\text{card}}(q)$ 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 ϕ representing if q passes the check and the speedup $\text{gain}(q)$

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 $\text{card}(q)$ and C; ▷ $\boldsymbol{\kappa}^{\star}$ contains all the known true cardinalities at that point
5: set $\hat{\boldsymbol{\kappa}}$ with the cardinality estimate $\widehat{\text{card}}(q)$ 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 $\text{card}(q)$ 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 $\text{Opt}(Q, \hat{\boldsymbol{\kappa}})$; ▷ representing the plan by relying on $\mathfrak{m}$ only
8: get the plan $\pi' \leftarrow \text{Opt}(Q, \boldsymbol{\kappa}')$, 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: $\text{gain}(q) = \frac{\text{Cost}(\hat{\pi}, \boldsymbol{\kappa}^{\star})}{\text{Cost}(\pi', \boldsymbol{\kappa}^{\star})}$; ▷ 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** $\text{gain}(q) \geq \theta_{\text{gain}}$ **then** ▷ speedup check
11: **return** true, $\text{gain}(q)$
12: **end if**
13: **end if**
14: **return** false, none;
15: **else**
16: **if** $\mathsf{T}_{\text{Exec}}(\pi) \cdot \text{gain}(q) < \theta_{\mathsf{T}_{\text{Exec}}}$ 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 $\text{gain}(q)$;
20: **if** $\text{gain}(q) \geq \theta_{\text{gain}}$ **then** ▷ speedup check
21: **return** true, $\text{gain}(q)$;
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 θ_{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 $\tilde{\boldsymbol{\kappa}}$
  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_{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_{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_{train};

Illustration of Usage and Maintenance of $\mathcal{F}$. We illustrate the usage and maintenance of $\mathcal{F}$ by an example in Figure 9.

D More Details of Experiment

We first shown the examples of the distribution of query execution time in the training sets in Appendix D.1. Next, in Appendix D.2, we present all the templates used in Section 5. Then, we present the specific design of the external index of Watcher2D in Appendix B. In Appendix D.3, we show more details for the ablation studies of SWatcher+'s parameters. In Appendix D.4, we show an example of the growth and utilization of SWatcher+'s watchset. In Appendix D.5, we show the results on end-to-end runtime for three-batch setting.

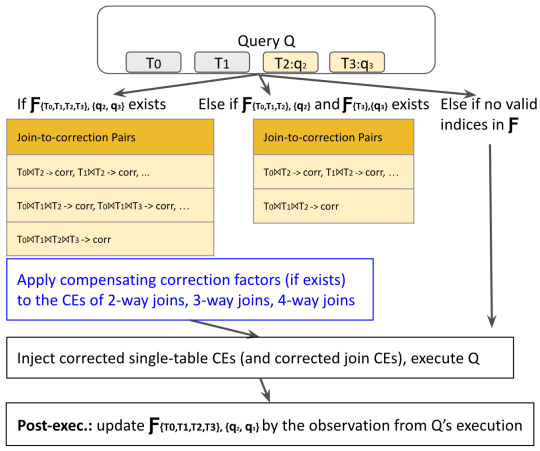
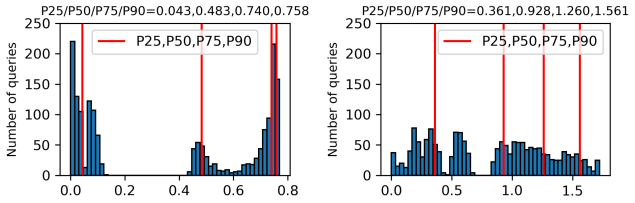


Figure 9: Usage and maintenance of $\mathcal{F}$ on Q joining T_1, T_2, T_3, T_4, assuming the subqueries of T_2, T_3 were corrected by watchset queries q_2, q_3, i.e., $J_Q = \{T_0, T_1, T_2, T_3\}, C_Q = \{q_2, q_3\}$.



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

Figure 10: Query latency (sec.) in the initial query set for Q013b and Q100 (only the fast 95% is displayed in the histograms)

D.1 Examples of Query Execution Time Distribution in Training Sets

We present the distributions of the query execution times in the training set of Q013b and Q100 in Figure 10, where the separations between slow and fast queries are evident.

D.2 Templates Used in Section 5

```
-- 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;
```

```
-- 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;
```

```
-- 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;

-- 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;

-- 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;

-- 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;

-- 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;

-- 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;

-- 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;

-- 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;

-- 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;

-- 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;

-- Q102
```

```sql
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.3 Ablation Studies of SWatcher+'s Parameters

Besides Q013b shown in Section 5.2, 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, $\theta_{\text{T}_{\text{Opt}}} = 10\% \cdot \theta_{\text{T}_{\text{Exec}}}$, $\theta_{\text{gain}} = 101\%$ and $\theta_{\text{T}_{\text{Exec}}}$ 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.4 The Growth and Utilization of Q013b's $\mathcal{W}$

Figure 11 shows the growth and utilization of SWatcher+'s watchset over the course of the workload.

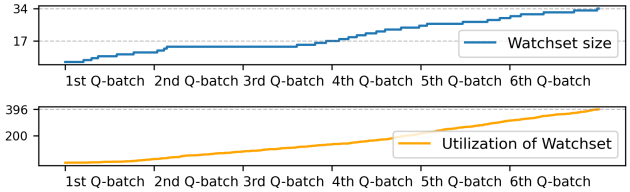


Figure 11: Q013b: growth (upper sub-figure) and utilization (lower sub-figure) of $\mathcal{W}$.

D.5 Results of End-to-End Runtime for Three-Batch Setting

As shown in Figure 12, 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 5.1. 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.

D.6 Results of OLTP-style Workload

We generate queries using the following template:

```sql
-- TJQ013b
SELECT * FROM store_sales WHERE ss_customer_sk = 2;
```

Instead of batching (i.e., executing a batch of Qs followed by a batch of DUs), we uniformly distribute 3000 Qs over the 60-day forward-shift duration according to the `ss_date_sk` attribute. For such simple queries, the total query execution time of all 3000 Qs is under 0.1 seconds across all methods. Thus, our analysis focuses on the overhead incurred by each method, which better reflects their behavior under this workload.

Overall, Baseline performs best, as its only overhead is the per-query model inference time. SWatcher+ incurs slightly higher overhead on Qs due to maintaining the query pool (though not $\mathcal{W}$ or $\mathcal{F}$, as it admit no watch queries) and a small overhead on DUs from retraining checks (i.e., whether $|\mathcal{W}|$ exceeds the threshold). In contrast, Periodic, Watcher1D, and LEO incur substantial unnecessary overhead, as their strategies focus on improving accuracy of CEs without considering performance gains or losses.

As a side note, if the characteristics of such workloads are known in advance or detected early, it may be preferable to switch from cost-based query optimization to a rule-based strategy, such as caching a few common plans. This universal technique could be incorporated into all these methods, including SWatcher+. However, as shown in Section 5.1 and Section 5.5, certain complex query templates either have short execution time (compared to optimization time) or are insensitive to CE accuracy – cases that simple strategies struggle to detect, unlike SWatcher+'s selective monitoring. However, since this paper focuses on deploying ML-based cardinality estimators, our experiments emphasize complex templates where CEs meaningfully influence plan choices and consequently the performance.

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

Algorithm	Retrain if†	Correct if	Total $T_{Exec}(Q)$	Retraining Time	Overhead on Qs	Overhead on DUs	Total $T_{Exec}(Q)$ in potent region
SWatcher+	single only	θ_{gain} = 110%	860.32	0(0)	26.62	5.76	36.05
SWatcher+	plus 2-join	θ_{gain} = 110%	989.53	0(0)	26.71	3.42	52.72
SWatcher+	single only	θ_{gain} = 101%	874.00	0(0)	27.53	8.87	45.81
SWatcher+	**plus 2-join**	θ_{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 $T_{Exec}(Q)$	Retraining Time	Overhead on Qs	Overhead on DUs	Total $T_{Exec}(Q)$ in potent region
SWatcher+	single only	θ_{gain} = 110%	1153.70	0(0)	31.24	16.65	4.05
SWatcher+	plus 2-join	θ_{gain} = 110%	1151.05	0(0)	32.25	17.35	3.86
SWatcher+	single only	θ_{gain} = 101%	1191.25	0(0)	32.96	33.79	4.78
SWatcher+	**plus 2-join**	θ_{gain} = 101%	**1095.65**	**0(0)**	**31.71**	**30.54**	**15.30**

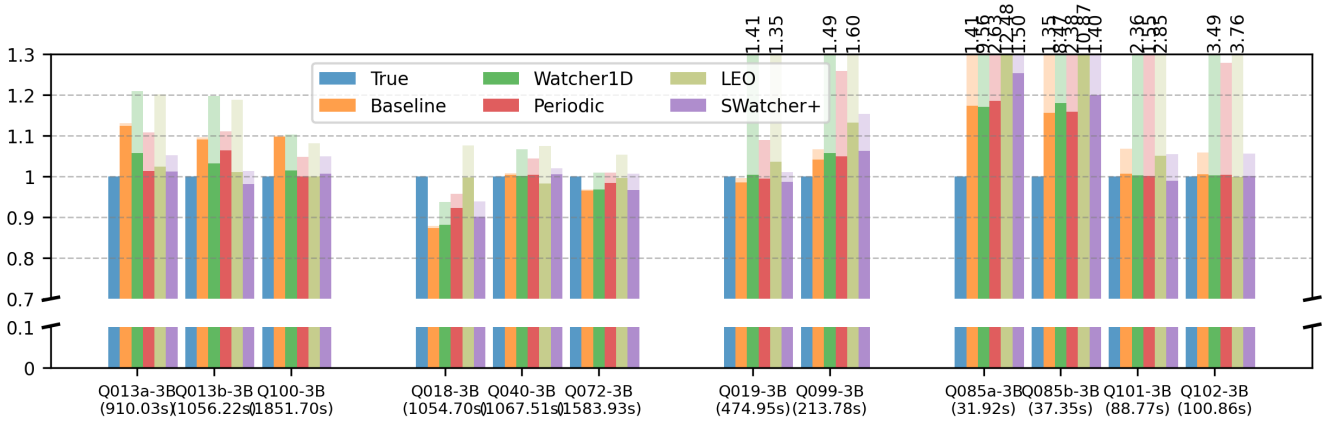


Figure 12: the normalized end-to-end runtime for three-batch setting, including both total query execution time (represented by the lower portion of the bar with darker shade) and overhead (represented by the upper portion with lighter shade), across all algorithms and single-selection templates. A number 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.

Table 6: PQ001: componentized overhead. †: the parameters, shared within the same method and detailed previously, are omitted. The last five 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 running time for each method.

Method	Settings†	Retraining Time	Overhead on Qs	Overhead on DUs
Baseline	-	**0(0)**	**6.86**	**0**
Periodic	X=10	55.98(6)	7.89	-
Periodic	X=20	24.43(3)	8.11	-
Periodic	**X=30**	**12.09(1)**	**7.94**	**-**
Watcher1D	ϵ=P90	101.66(5)	7.31	69.30
Watcher1D	**ϵ=P95**	**99.21(4)**	**7.26**	**69.14**
LEO	-	**0(0)**	**19.71**	**120.60**
SWatcher+	$\theta_{T_{Exec}}$ = 0.03 ms, θ_{gain} = 110%	0(0)	9.07	0.11
SWatcher+	**$\theta_{T_{Exec}}$=0.03 ms, θ_{gain}** = 101%	**0(0)**	**8.68**	**0.11**

E More Details of Future Work

There are several directions for further improving SWatcher+. As noted in Section 4, 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. Using efficient data summaries to approximate cardinality changes rather than maintaining exact values in $\mathcal{W}$ can be a viable alternative. Supporting other model

refresh strategies beyond retraining - such as fine-tuning - would be interesting. 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 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.