Kepler: Robust Learning for Faster Parametric Query Optimization

LYRIC DOSHI*, Google, USA
VINCENT ZHUANG*, Google, USA
GAURAV JAIN, Google, USA
RYAN MARCUS, University of Pennsylvania, USA
HAOYU HUANG, Google, USA
DENIZ ALTINBÜKEN, Google, USA
EUGENE BREVDO, Google, USA
CAMPBELL FRASER, Google, USA

Most existing parametric query optimization (PQO) techniques rely on traditional query optimizer cost models, which are often inaccurate and result in suboptimal query performance. We propose Kepler, an end-to-end learning-based approach to PQO that demonstrates significant speedups in query latency over a traditional query optimizer. Central to our method is Row Count Evolution (RCE), a novel plan generation algorithm based on perturbations in the sub-plan cardinality space. While previous approaches require accurate cost models, we bypass this requirement by evaluating candidate plans via actual execution data and training an ML model to predict the fastest plan given parameter binding values. Our models leverage recent advances in neural network uncertainty in order to robustly predict faster plans while avoiding regressions in query performance. Experimentally, we show that Kepler achieves significant improvements in query runtime on multiple datasets on PostgreSQL.

CCS Concepts: • **Information systems** → **Query optimization**.

Additional Key Words and Phrases: databases, query optimization, machine learning

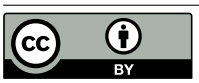
ACM Reference Format:
Lyric Doshi, Vincent Zhuang, Gaurav Jain, Ryan Marcus, Haoyu Huang, Deniz Altınbüken, Eugene Brevdo, and Campbell Fraser. 2023. Kepler: Robust Learning for Faster Parametric Query Optimization. *Proc. ACM Manag. Data* 1, 1, Article 109 (May 2023), 25 pages. https://doi.org/10.1145/3588963

1 INTRODUCTION

Parametric query optimization (PQO) aims to optimize *parameterized queries*, i.e. queries that have identical SQL structure and only differ in the value of bound parameters. Such parameterized queries are ubiquitous in modern database usage and present a significant opportunity for improving query performance because they are executed repeatedly.

*Equal contribution.

Authors' addresses: Lyric Doshi, lyric@google.com, Google, Mountain View, CA, USA; Vincent Zhuang, vincentzhuang@google.com, Google, Mountain View, CA, USA; Gaurav Jain, gaurav@gauravjain.org, Google, Mountain View, CA, USA; Ryan Marcus, rcmarcus@seas.upenn.edu, University of Pennsylvania, Philadelphia, PA, USA; Haoyu Huang, haoyuhuang@google.com, Google, Mountain View, CA, USA; Deniz Altınbüken, denizalti@google.com, Google, Mountain View, CA, USA; Eugene Brevdo, ebrevdo@google.com, Google, Mountain View, CA, USA; Campbell Fraser, campbellf@google.com, Google, Mountain View, CA, USA.



2836-6573/2023/5-ART109
https://doi.org/10.1145/3588963

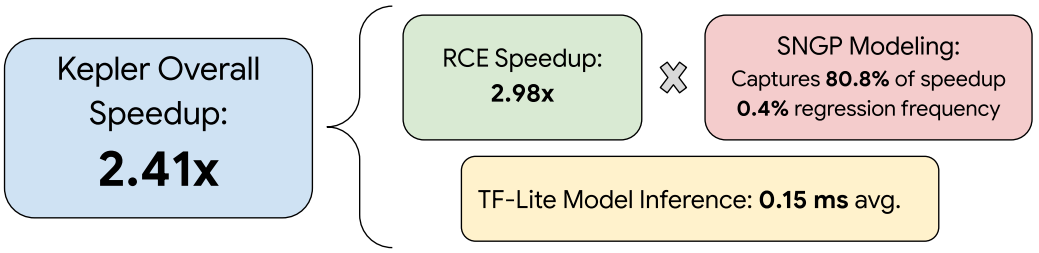


Fig. 1. Kepler achieves an overall 2.41x speedup on Stack by 1. discovering better plans via RCE, 2. capturing the majority of the speedup with SNGP models while minimizing regressions, and 3. having fast model inference time.

However, PQO has primarily been studied from the perspective of reducing query planning time by avoiding re-optimization when possible [7, 9, 13, 17, 18, 34]. Such approaches are implicitly constrained by the performance of the system's query optimizer, and therefore inherit all of the well-studied sub-optimalities of traditional query optimizers [22]. Thus, an ideal system for parameterized queries should not only seek to minimize planning time via PQO, but also optimize query execution performance via query optimization (QO).

A variety of approaches have attempted to improve query optimization by applying machine learning [20, 25, 29, 38, 39]. Unfortunately, most learned query optimization techniques suffer from at least four drawbacks: (1) they require *inference times* higher than traditional methods [19, 24], (2) they have *inconsistent performance* across dataset sizes and distributions [19, 26, 31], and (3) they often have *unclear query performance improvements* [19]. Worse yet, many of these learned systems lack (4) *robustness*: regressions in query performance are unacceptable in most production scenarios [12]. This poses an especially large challenge for learning-based approaches, since they typically cannot guarantee that all of their predictions result in improved execution time [35].

We propose that restricting the query optimization problem to the parameterized query setting poses a more tractable learning problem and hence can be more robustly solved. To this end, we present Kepler (**K**-plan **E**volution for **P**arametric Query Optimization: **L**earned, **E**mpirical, **R**obust), an end-to-end learning-based approach for parameterized queries. Building on prior work in PQO [34], Kepler leverages a novel plan generation strategy, a training query execution phase, and a robust neural network model design. Combined, we show that these techniques provide significant improvements in both planning time and query execution performance, satisfying both the PQO and QO objectives. Best of all, Kepler's use of robust neural network techniques drastically reduces the frequency and magnitude of performance regressions. Figure 1 highlights how each of Kepler's components contribute to a 2.41x geometric mean speedup across the entire Stack benchmark [25].

Kepler follows a decoupled plan generation and learning-based plan prediction architecture similar to the approach of [34] with three key differences. First, Kepler provides the key insight that designing better candidate plan generation algorithms can lead to substantially faster plans than the built-in optimizer's. We propose Row Count Evolution (RCE), a method that efficiently generates candidate plans by perturbing the optimizer's cardinality estimates. RCE only requires a simple interface to any standard cost-based optimizer, making it compatible with most database systems.

Second, Kepler leverages actual query execution data to build a training dataset for best-plan prediction, avoiding the well-studied mismatch between cost models and execution latency [22].

While Kepler's collection of execution data may be costly if the parameterized query is run infrequently, we argue that the additional execution data in our setting is justified by (1) the scale of parameterized queries in production and (2) the query execution speedups afforded by RCE.

Third, Kepler uses robust neural network prediction techniques to decrease tail latency and reduce query regressions (i.e. worse performance than the existing query optimizer). Specifically, Kepler uses Spectral-normalized Neural Gaussian Processes (SNGPs) [23] to accurately quantify how confident it is about a prediction, and falls back to the database's query optimizer when it is uncertain.

Our contributions.

- We identify a novel and practical formulation of query optimization for parameterized query templates in which *speedups* against a classical query optimizer can be robustly achieved.
- We propose a novel candidate plan generation algorithm, Row Count Evolution (RCE), that produces significant speedup compared to classical query optimizers on real-world and synthetic datasets.
- We demonstrate that incorporating robust ML techniques allows models to capture large portions of the speedups while greatly reducing the risk of regressions.
- We demonstrate that our model inference costs are negligible via an end-to-end PostgreSQL integration for the query path.
- We open-source both our system implementation for PostgreSQL[1] as well as our query execution datasets, which we believe is the first dataset tailored towards parameterized query optimization. The datasets collectively represent ~14.2 CPU years of query execution time. They serve as a benchmark for further work on best-plan prediction as well as simulating more efficient techniques for training data collection.

2 RELATED WORK

Parametric query optimization. PQO has been extensively studied in a variety of works [7, 9, 13, 17, 18, 34]. The goal of the standard PQO formulation is to reduce the amount of times the query optimizer is invoked while minimizing the corresponding regression in query latency [7, 13, 34]. Although Kepler also focuses on parametric queries, its primary objective is closer to that of standard query optimization, which seeks to improve query latencies. Kepler also simultaneously improves on the PQO objective by leveraging fast-inference ML models.

Prior PQO approaches typically make simplifying assumptions such as heavily relying on the optimizer cost model or using base table selectivities as input features [13, 34]. This may be feasible for some advanced commercial systems; however, this over-reliance on the existing optimizer is particularly dangerous given the well-studied deficiencies of optimizers such as PostgreSQL [22].

Our approach follows a similar structure as [34], which also decouples the populateCache (candidate generation) and getPlan stages (ML-based prediction). However, since they focus on the standard PQO objective of attempting to match the existing optimizer, they require using a bandit algorithm to reduce their training data cost. By contrast, the primary objectives of Kepler are query performance and robustness, leading to a lower emphasis on training query efficiency.

Several popular database systems have implemented PQO features, including Oracle Adaptive Cursor Sharing, Aurora Managed Plans, and SQL Server Parameter Sensitivity Plan optimization [1–3]. These features all heavily rely on their cost models (based on traditional statistics and heuristics), and do not utilize machine learning models.

[1] https://github.com/google/kepler

Query plan generation. Several prior works suggest methods for candidate generation, which we divide into four main categories.

(1) **Default optimizer plans.** The simplest method combines the optimizer's selected plan for each query instance. This approach is frequently found in PQO algorithms since they seek to cache the optimizer's plans [34]. This strategy is also employed in [30] to estimate the empirical suboptimality of existing query optimizers. The quality of the resulting candidate plan set is predicated upon the optimizer's ability to either generate optimal plans for each query instance or a sufficient variety of good plans across the workload to benefit from plan sharing. However, we empirically observed that the optimizer fails to do so on real-world datasets. (Table 10b).
(2) **Cost-based plan pruning.** populateCache algorithm [34] extends the default optimizer candidate generation method with cost-based K-set identification to prune the candidate set to size K. However, this pruning method may mistakenly prune good plans if the correlation between the cost estimates and actual execution times are poor.
(3) **Optimizer configuration parameters.** Query optimizers typically expose a variety of configuration parameters that can be used to alter their query planning behavior. In particular, PostgreSQL has configuration parameters that allow one to disable entire classes of join and scan operators from being used in query plans. Bao selectively applies subsets of these parameters in order to generate new query plans [25]. Although simple, disabling operator types is a heavy-handed and indirect approach to generating new plans.
(4) **Exact cardinalities.** Exact cardinality query optimization (ECQO) attempts to construct the optimal plan by computing the plan induced by the exact cardinality values of all possible sub-plans [10]. However, for sufficiently complex queries, evaluating these exponentially-many sub-plans is prohibitively slow even with optimizations [32]. The selected plans are also not always the fastest, as observed by [30].

In summary, these methods are all unsatisfactory for a variety of reasons: failure to generate faster plans (1, 2, 4), ineffectively exploring the plan space (3), or are computationally intractable (4).

Machine learning for query optimization. A wide range of techniques apply ML on QO, most notably for predicting cardinality estimates (CE) [20, 38, 39]. Recent work show cardinality estimation may be brittle in practice, and that even small Q-errors can lead to noticeably worse plans [22, 35]. In general, these work do not measure the actual end-to-end execution latency of selected plans after integrating their models into an optimizer [24].

Several approaches have demonstrated improved query performance, but typically do not consider the issue of robustness. Neo [26] and Bao [25] leverage tree convolutional neural networks to adaptively optimize plans using reinforcement learning and contextual bandits respectively. These online algorithms offer no guarantees on stability or regression avoidance, and hence cannot reliably be deployed in production. Similarly, techniques applying deep reinforcement learning to QO have not demonstrated consistently better performance and suffer from robustness issues [21, 28, 37]. For example, Figure 9 in [37] indicates a significant amount of regressions both at train and test time.

3 OVERVIEW

In this section, we describe our problem setting (Section 3.1), give an overview of our approach (Section 3.2), and further discuss specific design choices that are made in Kepler (Section 3.3).

3.1 Problem Setting

As in prior work [34], we consider parameterized queries that are repeatedly invoked with different parameter bindings. Such queries are specified by a template Q with m parameterized predicates[2] $x_0, \ldots, x_{m-1}$ of varying data types. We let q denote a specific query instance, i.e. Q with a fixed set of parameter binding values. A query plan p associated with a template Q specifies how to execute any query instance $q \sim Q$. A *sub-plan query* of Q is Q restricted to only a subset of its tables [14], and its output cardinality is referred to as its *sub-plan cardinality*. We assume a fixed database system with a built-in query optimizer, and denote the default plan $p_{\texttt{default}}(q)$ to be the plan selected by the query optimizer for q. Finally, a workload $W \subset \mathcal{W}$ consists of a set of query instances $\{q_0, \ldots, q_{n-1}\}$ for a single template Q, where $\mathcal{W}$ denotes the space of all possible query instances.

3.2 Kepler Overview

Our approach at a high level follows that of [34]: we consider a single, isolated query template Q, and decouple the problems of generating a set of possible plans and deciding which plan to use for each query instance. More formally, these problems can be described as:

(1) **Candidate generation.** Generate a candidate set of k plans $\{p_0, \ldots, p_{k-1}\}$ for Q, out of the exponentially-large set of all possible plans $\mathcal{P}$ (corresponding to `populateCache` in [34]).
(2) **Best-plan prediction.** Learn a mapping $M : \mathcal{W} \rightarrow P$ that minimizes some objective, e.g. some measure of execution latency over the workload (corresponding to `getPlan` in [34]).

Unlike [34], who attempt to match the performance of the built-in optimizer, our goal is to improve upon the built-in optimizer as much as possible. To achieve this, Kepler includes a sophisticated candidate generation algorithm, described in Section 4, that empirically generates better plans than the built-in optimizer. The afforded speedups allow Kepler to avoid relying on potentially-brittle online learning approaches (e.g. contextual bandits) during the training data collection phase.

Objective. We first define several key metrics and terms in our problem setting. For a given query instance, we denote the optimal plan over some plan set P as $p^P_{\texttt{opt}} = \min_{p \in P} ExecTime(p, q)$, where $ExecTime$ refers to the actual execution time. We define $p^*_{\texttt{opt}}(q)$ as the optimal plan over all possible plans, i.e. when $P = \mathcal{P}$. Typically, the optimal plan refers to $p^*_{\texttt{opt}}$ for candidate generation and $p^P_{\texttt{opt}}$ for modeling. We also refer to near-optimal plans as plans that have similar execution time to $p^P_{\texttt{opt}}$ or $p^*_{\texttt{opt}}$.

For some fixed candidate set P, we define the (oracle) speedup ratio relative to the default plan as:

$$S^{\texttt{opt}}(P, W) = \frac{\sum_{q \in W} ExecTime(p_{\texttt{default}}, q)}{\sum_{q \in W} ExecTime(p^P_{\texttt{opt}}, q)} \tag{1}$$

This quantity is the factor by which we can improve the total execution time of the workload if we had oracle access to the optimal plan in P for each query instance. We note that this ratio corresponds exactly with the definition of execution cost sub-optimality in [34]; the re-naming to speedup emphasizes the differences in our system objectives. Since we can union P with the set of all default plans over W, this speedup ratio is always lower bounded by 1.

Similarly, for some model $M : \mathcal{W} \rightarrow P$, we define its model speedup as:

$$S^{\texttt{model}}(W) = \frac{\sum_{q \in W} ExecTime(p_{\texttt{default}}, q)}{\sum_{q \in W} ExecTime(p_{\texttt{model}}, q)} \tag{2}$$

[2]The parameters do not necessarily have to be in predicates, e.g. they may appear in a LIMIT clause. However, our experiments only include the parameterized predicate case.

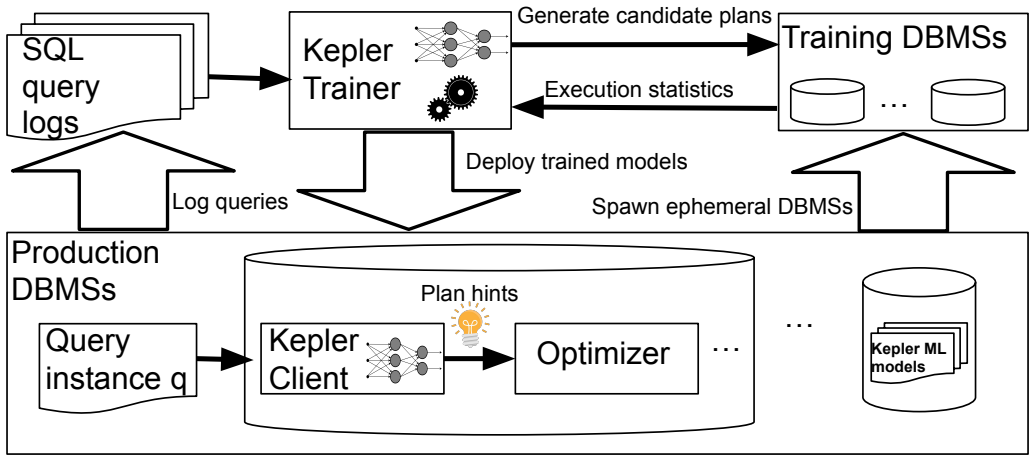


Fig. 2. Kepler architecture.

This quantity corresponds to how much faster the model is at executing a workload than the default optimizer. Although $S^{\texttt{model}}$ is by definition upper bounded by $S^{\texttt{opt}}$, it is not necessarily lower bounded by 1, i.e. if the model selects plans worse than the default plan.

An auxiliary objective of Kepler is reducing workload tail latency. Several work have identified that database optimizers may perform significantly worse in the tail of the query latency distribution, which poses a significant obstacle for use cases that require a more uniform runtime [25].

Kepler architecture. Figure 2 shows the architecture of Kepler, consisting of a Kepler trainer and Kepler client. The trainer ingests query instances from the query logs produced by production DBMSs and aggregates them into query templates. For each query template Q_i, the Kepler trainer aims to find the near-optimal plans for all its query instances q_j. It uses Row Count Evolution (RCE) to generate candidate plans p_k and executes the queries with these plans to collect execution statistics. To minimize impact on production DBMSs, the trainer may optionally request a production DBMS to spawn ephemeral instances to execute these queries. The Kepler trainer trains an ML model to predict the best plan for q_j based on these execution statistics and deploys the trained models into the production DBMSs.

A Kepler client maintains a mapping from a query template to an ML model. When a production DBMS receives a query instance q, the client first checks if an ML model is available for q. If available, it performs model inference to predict the best plan hints and provides the hints to the optimizer only if the associated confidence score is higher than a threshold. Otherwise, it falls back to the built-in optimizer to produce a plan.

Changing environments and workloads. In our current implementation, Kepler assumes a fixed system state, including database configuration, optimizer implementation, and data distribution. If any of these aspects changes relatively slowly or infrequently, Kepler can periodically collect new execution data and retrain purely on data from the new system state. We posit that in the majority of production parameterized query use cases, (1) the database is reconfigured infrequently, and (2) the data distribution drifts slowly, e.g. in scenarios in which a relatively small amount of similarly-distributed data is added each day. Additionally, Kepler is designed to be robust to dynamic workloads in which query parameter binding values change by detecting when inputs are out of its training distribution (see Section 7.4).

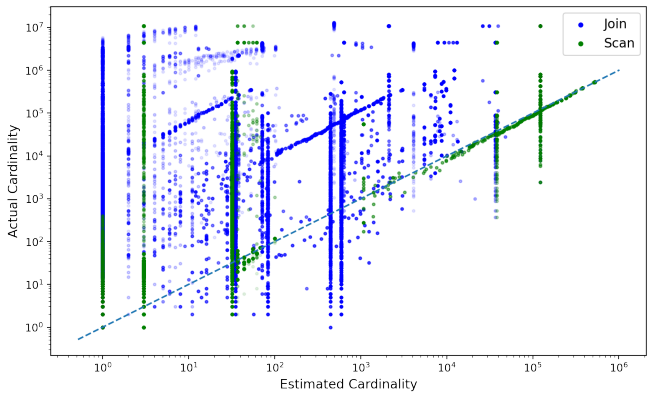


Fig. 3. Predicted vs. exact cardinalities on instances of Stack q12_2. Dashed y=x line is ideal (i.e. actual = estimated).

Limitations. The target usage of Kepler is for parameterized queries that are executed frequently enough to justify the training data collection cost. As discussed in Section 8, the exact training data collection regime in this paper serves the dual purposes of definitively demonstrating the speedups available and enabling further research in efficiency. We anticipate a final production system will use an iteration of this research with leaner training data collection. The cost-benefit analysis of using Kepler is situation-dependent; ultimately the user must weigh the potential query performance gains against the cost. If ephemeral instances are used for training data collection, Kepler assumes they are representative of production query performance.

3.3 Kepler Design Choices

In this section, we further discuss the specific design choices made to ensure that Kepler can be reliably deployed with minimal production overhead.

Using actual execution latencies. Since the objective of Kepler is to reduce actual end-to-end query latencies, it necessitates executing queries on a real database to provide ground-truth signal. To minimize the training collection time and avoid load on the production system, the DBMS may spawn ephemeral instances to speed up and isolate the training execution process.

Limiting reliance on cost models. By collecting actual execution latencies, Kepler eschews explicitly relying on optimizer cost estimates for determining the quality of a plan. Figure 3 shows the estimated vs. exact cardinalities of all joins on a sample of query instances from Stack [25]. In particular, 64% of points have estimated cardinality = 1, likely due to the independence assumption of the PostgreSQL optimizer.

Falling back to the built-in optimizer. Kepler avoids regressions by falling back to the existing query optimizer when it is not confident in identifying the optimal plan. Given the low overhead of model inference, the overall Kepler inference cost is nearly always lower than that of Opt-Always. For cases where planning time is a concern due to high fallback frequency, one can incorporate an additional model designed to predict a safe plan without re-invoking the optimizer.

Independence of query templates. Kepler handles query templates independently, i.e. each query template will generate its own candidate plans, collect its own training data, and train a model specific to that template. Though potentially more expensive than a procedure that generalizes over multiple query templates, this design has the advantages of 1) providing a more tractable

Table 1. RCE hyperparameters.

Notation	Definition
G	Number of generations.
b	Exponent base for row count perturbation.
m	Exponent range for row count perturbation.
S	Number of plans sampled from the previous generation.
N	Number of perturbation for each candidate plan.

ML problem, and 2) isolating each query from regressions caused by changes pertaining to other queries as models iterate over time and new query templates are on-boarded. Leveraging shared information between query templates while not increasing the risk of regressions is an interesting direction for future work.

4 ROW COUNT EVOLUTION

The goal of the candidate generation stage is to construct a set of plans P such that it contains a near-optimal plan for every query instance q in the workload distribution $\mathcal{W}$. Additionally, P should be sufficiently small such that it is feasible to execute each plan for training dataset collection. Balancing these two competing objectives is the main challenge for any candidate generation algorithm.

In this work, we only consider generating fully-specified plans, i.e. the join order and every join/scan method are defined. Alternatively, a candidate generation algorithm could specify a subset of the plan decisions and allow the query optimizer to determine the remainder.

Workload candidate generation. Given an algorithm A for generating a candidate plan set over a single query instance q, we define the corresponding plan set over a workload W as the union of the per-instance plan sets $A(W) := \bigcup_{q \in W} A(q)$ (see lines 1-7, Algorithm 1). We also define *plan sharing* to describe the case where $p^P_{\texttt{opt}}(q)$ is generated from some other query instance q' (i.e. $p^P_{\texttt{opt}}(q) \notin A(q), p^P_{\texttt{opt}}(q) \in A(q')$).

Our approach. We propose Row Count Evolution (RCE)[3], a computationally-efficient algorithm that generates new plans by randomly perturbing the optimizer's cardinality estimates. RCE is predicated on the idea that cardinality misestimates are the primary underlying reason for optimizer suboptimality. RCE exploits the fact that our candidate generation stage only needs to generate a set of plans that contains a (near-)optimal plan instead of directly identifying a single performant plan. Like Bao [25], RCE leverages the built-in query optimizer to generate candidate plans, but does so in a more fine-grained and efficient way.

We instantiate the idea of applying random perturbations as an evolutionary-style algorithm, described in Algorithm 1. RCE maintains a sequence of generations of plans, with the initial generation consisting solely of the query optimizer's plan. To construct subsequent generations, RCE first uniformly samples parent plans from the previous generation. For each of these base plans, RCE perturbs the join cardinalities of *only the sub-plans that appear in the parent plan* by multiplicative factors sampled from an exponentially-spaced range (lines 27-39). By repeating this process multiple times and feeding in the resulting perturbations into the query optimizer, RCE generates a set of children plans (lines 15-21). Out of these, only unseen plans (i.e. those that did not appear in any prior generation) are kept for the next generation (lines 18-19).

[3]The name "Row Count" is inspired by the PostgreSQL extension pg_hint_plan's row count hints, which we use to modify the PostgreSQL optimizer's cardinality estimates.

Algorithm 1 Row Count Evolution.

```
1:  function WorkloadCandidateGeneration(workload W)
2:      P ← {}
3:      for query instance q ∈ W do
4:          P ← P ∪ RowCountEvolution(q)
5:      return P
6:
7:  function RowCountEvolution(query instance q)
8:      p_0 = the base plan for q
9:      C_0 = {(p_0, {}, {s → 0 ∀ sub-plans s})}
10:     for generations g = 1, 2, ... G do
11:         Sample up to S base plans B_g uniformly from C_{g−1}
12:         C_g ← {}
13:         for (base plan p, row count map r) ∈ B_g do
14:             for i = 1, 2, ..., N do
15:                 r' ← SamplePerturbations(p, r)
16:                 p' ← GetOptimizerPlan(r')
17:                 if p'! = p then
18:                     C_g.add((p', r'))
19:     return C_0 ∪ C_1 ∪ ... ∪ C_G
20:
21: function SamplePerturbations(plan p, row count map r)
22:     for sub-plan s ∈ p do
23:         w ← p.getEstimatedCardinality(s)
24:         e_l ← − min(log_b(w), m)
25:         e_u ← e_l + 2m
26:         Sample f uniformly from [b^{e_l}, ..., b^{e_u}]
27:         r[s] ← w · f
28:     return r
```

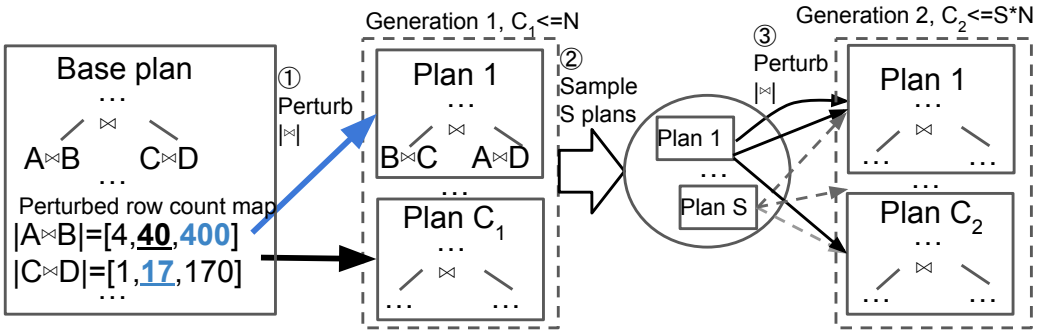


Fig. 4. An example RCE process.

Example. Figure 4 shows an example of RCE generating candidate plans for a query instance with two generations. The base plan joins the result of $A \bowtie B$ and $C \bowtie D$ with estimated $|A \bowtie B| = 40$, $|C \bowtie D| = 17$. RCE first constructs a set of candidate row counts for each sub-plan by perturbing their cardinalities by multiplicative factors. These candidate row counts for $A \bowtie B$ and $C \bowtie D$ are $[4, 40, 400]$ and $[1, 17, 170]$, respectively, using a base of 10 and a range of 1. RCE then uniformly samples new join cardinalities from these sets; one sample of 400 and 17 influences the optimizer to produce a new plan Plan-1 in generation 1 ①. It repeats the same process N times to produce C_1 plans in generation 1. Next, RCE samples S from a deduplicated set of plans from generation 1 ② and randomly perturbs the row counts on each sampled plan N times to generate C_2 plans in generation 2 ③.

RCE as exact cardinality matching. One interpretation of RCE is that it efficiently builds a covering set of exact-cardinality plans. The RCE-generated candidate set contains plans generated from a diverse range of perturbed sub-plan cardinalities. If there are sufficiently many perturbations, likely at least one will be reasonably close to the exact cardinalities for any particular query instance and their respective induced plans will also likely be similar.

Multiplicative perturbations. Applying multiplicative perturbations is well-motivated by the standard metric of Q-error in cardinality estimation. RCE further uses an exponentially-spaced perturbation set in order to have a similar support as the optimizer's Q-error distribution.

Perturbing only relevant sub-plans. Instead of perturbing all $2^n - 1$ sub-plans (for a query joining n tables), RCE only perturbs the cardinalities of the $n - 1$ sub-plans that actually appear in the sampled query plans. This significantly increases the efficiency of RCE with only a small loss of generality: since the set of perturbations is inherited between generations, a misestimated sub-plan cardinality will only never be perturbed if its cardinality is significantly overestimated by the query optimizer. However, this is an unlikely scenario since query optimizers tend to underestimate sub-plan cardinalities due to the independence assumption.

This re-optimization of only the sub-plan cardinalities that appear in the optimizer plan bears a strong resemblance to the re-optimization procedure in [36], which iteratively re-optimizes using sampling-based cardinality estimates. The key differences in our setting are (1) we do not have to return a single plan, and (2) we require a fast procedure since we repeat it for each query instance, motivating the use of perturbations over sampling.

RCE as local search. RCE effectively explores the plan space via a random walk in the low-dimensional subspace of sub-plan cardinalities, initialized at the optimizer's cardinality estimates. This formulation implicitly leverages the fact that while these initial estimates are typically incorrect, they are still more informative than random estimates.

RCE hyperparameters. Our implementation of RCE includes a variety of hyperparameters that allow one to flexibly trade off the number of generated plans against the potential total speedup (Table 1).

- **Width and depth of the perturbation tree.** Increasing the number of generations G increases the number of plans, making it more likely a good plan is found. However, plans in later generations are perturbed further from the original plan, and may have a lower likelihood of being relevant. To ensure constant-time processing for each generation, we sample (up to) a fixed number S of base plans in each generation, and perturb each one N times.
- **Perturbation values.** The exponent base b and range m limits the magnitude of a single perturbation. We also introduce a sub-plan perturbation limit that controls the number of

times a specific sub-plan can be perturbed, effectively controlling the total perturbation range of any given sub-plan.

- **Direct limits on number of plans.** We implement limits on the number of plans that can be generated from a single parameter and in total. Once the limit is reached, the evolutionary candidate generation process is terminated and only default plans are kept for remaining parameters. The total plans limit is a soft limit since the final evolutionary iteration may produce up to the single-parameter limit and the remaining parameters may contribute new default plans.

5 TRAINING DATA COLLECTION

After generating candidate plan set P, we execute each plan over a training workload to generate a dataset of execution latencies for supervised best-plan prediction. The training workload may be provided by the user or captured in a DBMS query log [34]. Rather than executing all candidate plans for each query instance, we use adaptive timeouts and construct near-optimal plan covers to prune suboptimal plans.

Execution mechanics. We force the optimizer to produce a candidate plan by providing all join/scan methods and the join order via hints. We parallelize the execution of query instances and their candidate plans in multiple databases. We simulate a warm buffer cache scenario by executing each plan multiple times and taking the minimum as the estimated latency [22]. This repeated execution strategy also reduces the potential noise in our execution time measurements; though we observed the amount of noise to be inconsequential in our experimental setup. We leave a full analysis of query execution time under different caching, concurrency, and resource availability settings to future work.

Adaptive timeouts and plan execution reordering. We use a timeout policy to minimize wasted resources on executing sub-optimal candidate plans. The timeout policy adapts from [37] with two main modifications. First, we always execute the default plan first and adaptively reorder the remaining plans to maximize the impact of the timeout's progressive tightening on a per query-instance basis. We execute plans in ascending order of their historical execution latencies across query instances as a simple heuristic for tightening the timeout as quickly as possible. Second, we do not apply the tightened timeout for the first iteration of each plan in order to ensure that each plan simulates a warm-cache scenario.

Online plan cover pruning. We also use an online plan pruning technique to eliminate plans based on actual execution time. Specifically, we initially execute all plans for the first N query instances of a query template, then use a Set Cover formulation to prune down to a minimal *plan cover* set for the remaining query instances. The pruned set becomes our k candidate plans for the query template, i.e. our models only attempt to predict from those plans.

We consider a plan to be near-optimal for a query instance q if its execution time is within a $1 + \epsilon$ factor of the fastest time for q we have seen so far. Each plan has an associated set of query instances for which it is near-optimal. The *plan cover* is the smallest set of plans such that each query instance has a near-optimal plan in the set. We construct the plan cover using the standard greedy approximation for Set Cover, which iteratively picks the plan that is near-optimal for the most remaining query instances. We additionally relax the problem to require that only $1 - \delta$ of all query instances be covered, allowing us to trade off the plan cover size and the achievable speedup.

Tail latency reordering. For many query templates, the distribution of default execution latencies is heavy-tailed. Parameters in the tail tend to be more sub-optimal, and therefore have an outsized impact on the total speedup. To ensure the plan cover computation includes these parameters, we

evaluate these query instances first. This reordering produces a 7-8x reduction in total execution time and number of plans over the entire Stack dataset.

6 ROBUST BEST-PLAN PREDICTION

After collecting a full training dataset of actual execution latencies over our candidate plan set, we use supervised ML to predict the best plan for any query instance. Kepler trains one model for each query template with the objective to maximize workload speedup while minimizing regressions. Kepler also falls back to the optimizer's plan when its predicted confidence is low. Section 7 shows that the inference time of our model is negligible compared to the typical query planning time of a classical optimizer.

6.1 Features

Given a template Q with m parameters, Kepler uses solely the m parameter values as input features. The supported types include numerics (float/int), strings, and dates/timestamps. We apply standard preprocessing techniques to each type: embeddings for strings/low-dimensional integer features, normalization to $N(0, 1)$ for numerics, and numeric conversion for date/time features.

We do not convert the parameter values to their respective base table selectivities as in [34] for the following reasons. First, selectivity is inherently a lossy representation and may obscure information when two distinct values have the same selectivity. Second, selectivity is inferior when the optimizer's cardinality estimation is sub-optimal, see Figure 3.

String columns and vocabulary selection. For each string-valued column, we construct a fixed-size vocabulary in order to limit model size. String features are one-hot encoded via a lookup table, with buckets for out-of-vocabulary values. An embedding layer is then applied on this one-hot encoding, creating a learnable embedding for each value in the vocabulary.

We choose the vocabulary as the top-k values ordered by the total possible improvement of all query instances with that value. We define the *max marginal improvement* strategy as selecting the top-k column values v in column i under the following objective:

$$m(v, i) = \sum_{q \in W, x_i = v} ExecTime(p_{\texttt{default}}, q) - ExecTime(p^P_{\texttt{opt}}, q) \tag{3}$$

Our evaluation shows that this strategy is effective. For columns with significantly more distinct values, one may factorize embeddings over subcolumns [38].

6.2 Training Objectives

Kepler models maximize the model speedup defined in Equation 2 while minimizing the number of regressions. This objective is not directly differentiable, so we discuss various surrogate learning objectives.

Multi-label classification. We model best-plan prediction as a *multi-label* classification problem in which each near-optimal plan has a positive label (as opposed to just the optimal plan) [33]. The multi-label objective also provides a richer supervised signal, improving the quality of the learned intermediate representations. We use the single-label transformation of multi-label classification loss by training the near-optimal probability of each candidate plan with binary cross-entropy loss.

Although our models only predict plans in the plan cover, which may not necessarily contain every query instances' default plan, our definition of near-optimality does exploit the availability of default plan execution data during training. We define a plan to be *near-optimal* if its estimated latency improvement is within a $1 + \tau$ factor of the optimal improvement latency. Namely, we say a plan p is near-optimal if $(\ell_d - \ell_p)(1 + \tau) \geq (\ell_d - \ell_o)$, where $\tau > 0$, $\ell_d = ExecTime(p_{\texttt{default}}, q)$, $\ell_p =$

$ExecTime(p, q)$, and $\ell_o = \min_{p \in P} ExecTime(p, q)$. Computing near-optimality requires execution times for all query instances for all plans.

Prior work formulate best-plan prediction as a regression [6, 27, 34] and multi-class classification problem [34]. Both formulations are unsatisfactory for a variety of reasons. Regression across a significant range can be unstable, a problem that is exacerbated by our timeout procedure, which obscures the true latency of suboptimal plans. Regression attempts a more challenging problem with finer granularity than required, imposing unnecessary constraints and objectives on the training. We only need to predict the identity of the optimal plan rather than its execution time. Inversions and gross over-estimates of non-optimal plans are acceptable to us but will weigh heavily in regression loss. Meanwhile, classification objectives that predict a single optimal plan perform poorly in scenarios when multiple plans can be near-optimal and empirical execution latencies can be subject to noise. For example, consider a problem where plans p_1, p_2 execution latencies' are both drawn from $C + N(0, 1)$ for some large C. Then a multi-class classifier will have equal predicted likelihood for p_1 and p_2 and thus have low confidence, when in actuality being confident in p_1 and/or p_2 is desirable.

Example-dependent loss. Different query instances may have disproportionate impact on the overall objective Equation 2. We leverage the standard sample-weighting approach example-dependent cross entropy [8, 16] to prioritize those with the largest improvement delta. For plans worse than the default plan, we upweight them by a factor C. For all near-optimal plans, we apply a soft weighting based on their empirical execution improvement, i.e. $1 + D \log(\ell_d - \ell_p)$, where C and D are both tunable hyperparameters.

6.3 Models

We use simple feedforward neural networks as our base models. For inference efficiency, we consider a neural network with one output head per plan on top of a shared representation, which improves inference speed and model size over approaches that have separate models for each plan [34].

We train our neural network models with standard minibatch SGD. In a real-world setting, the model's hyperparameters can be tuned via simple search techniques or more sophisticated algorithms by partitioning the training data into a train and validation set.

Uncertainty. Kepler models incorporate calibrated predictions and uncertainty estimates to avoid predicting significantly suboptimal plans. Two state-of-the-art approaches for incorporating uncertainty into neural networks are ensembling and Spectral-normalized Neural Gaussian Processes (SNGPs) [23]. The former trains M distinct models simultaneously and estimates the uncertainty from their joint outputs. The latter applies spectral normalization to all layers, providing a bi-Lipschitz guarantee on all intermediate representations, and uses a Gaussian process output layer to efficiently estimate the uncertainty. Since ensembling increases the training and inference cost by a linear factor M, Kepler uses the SNGP approach due to its lower overhead.

7 EXPERIMENTS

Our evaluation of Kepler seeks to demonstrate that it robustly achieves state-of-the-art execution latency speedups on parameterized query workloads. We summarize our main results as follows:

- An end-to-end implementation of Kepler on PostgreSQL substantially outperforms the built-in optimizer and Bao. Both RCE and ML models play large roles in achieving this speedup. (Section 7.2)
- RCE discovers significantly better plans than existing candidate generation baselines. We also observe that RCE plans are frequently superior to exact-cardinality plans. (Section 7.3)

- Using SNGP models is crucial to capturing speedups generated by RCE while minimizing query regressions. (Section 7.4).
- We release a dataset consisting of ~14.2 years of query executions as a benchmark for future research in modeling approaches (Section 7.5).

Objectives. To evaluate our methods, we use both RCE speedup $S^{\texttt{opt}}(RCE)$ (shortened as S^{RCE}) and model speedup $S^{\texttt{model}}$, defined in Equations 1 and 2 respectively. We note that $S^{\texttt{model}} = p \cdot S^{RCE}$, where $0 \leq p \leq 1$ corresponds to the proportion of the speedup the model captures. Since the model may predict worse plans than the built-in optimizer, we also measure the query regression frequency $P^{\texttt{reg}}$, defined as the proportion of test query instances the model does at least 10% worse than the default optimizer on. The primary metrics for each of our components are:

(1) **End-to-end performance:** $S^{\texttt{model}}$
(2) **Candidate generation performance:** S^{RCE}
(3) **Model performance:** p, $P^{\texttt{reg}}$

Kepler aims to maximize $S^{\texttt{model}}$ by maximizing p and S^{RCE}, while minimizng $P^{\texttt{reg}}$. We also report the 99th-percentile tail latency speedup, which may be relevant in applied scenarios. We define this as $P_{99}^{\texttt{method}}(W) = \frac{p_{99}(\{ExecTime(p_{\texttt{default}}, q) \forall\, q \in W\})}{p_{99}(\{ExecTime(p_{\texttt{method}}, q) \forall\, q \in W\})}$, where $p_{99}(C)$ denotes the 99th percentile highest value in a collection C.

7.1 Setup

Datasets and query extraction. We use two synthetic benchmarks: TPC-H (uniform and skewed with Zipf factor = 1, 10 GB [4]), and Stack, a database consisting of real-world StackExchange data [25]. TPC-H consists of 22 parameterized queries. We use an augmented version of Stack with 87 parameterized queries: 42 from the original benchmark and 45 additional manually-written query templates.

All experiments were run using PostgreSQL 13.5 on Google Cloud Platform (GCP) n1-highmem-16 instances with 16 CPU cores, 108 GB of RAM, and 2 TB of SSD. Following [22], we set shared_buffers to 75 GB, effective_cache_size to 80 GB, and work_mem to 4 GB to ensure that the entire dataset fits in memory. For TPC-H, we use the indexes defined in BenchBase [11]. For Stack, we add indexes on all primary keys, foreign keys, and columns that appear in a predicate of any query.

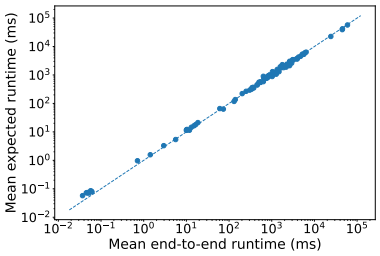
Query instance generation. We follow the official TPC-H specification [5] to generate parameter values of each query template. For Stack, we synthetically generate parameter values so that every query instance returns nonempty results. This is accomplished by uniformly sampling rows from the result set of a derived query that selects column values for which parameterized predicates would produce at least one value. Range predicates are constructed by first sampling a single value in the manner, then sampling lower/upper bounds around this value.

Training query execution. For each query instance and plan hints, we execute the resulting plan three times to simulate a warm-cache scenario, and take the minimum latency as the ground truth. For slow queries, we executed each up to 8 times in parallel on the same machine, and observed negligible differences with the serial execution setting. We leave a full analysis of different execution scenarios to future work.

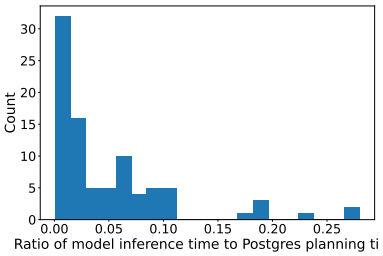
RCE hyperparameters. Unless stated otherwise, we use the same values for all RCE hyperparameters in all of our experiments, demonstrating its efficacy even when untuned for specific benchmarks. We set the number of generations G to 3, the exponent base b to 10, the exponent range m to 2, the number of perturbations per plan N to 20, and the number of samples extracted

S^{model}	% of queries
>1.2	64.4%
>2x	32.2%
>10x	14.9%
>20x	4.6%

(a) Summary of PostgreSQL Kepler speedups on Stack.



(b) Actual end-to-end vs predicted latencies (ms).



(c) Histogram of model inference to PostgreSQL planning time ratios.

Fig. 5. PostgreSQL integration evaluation.

from each generation S to 20. For each query template, we run RCE on the first 50000 query instances for Stack, and all query instances for TPC-H.

Model details. All of our experiments use a fixed base neural network with three layers of 64 hidden units each. We use Adam with learning rate 3e-4, ReLU activation functions, and 10-dimensional string embeddings. For SNGP models, we additionally apply spectral normalization to all dense layers, and replace the output dense layer with a random Fourier feature Gaussian Process with 128 random features. For all models, we fall back to the default plan if the predicted confidence is less than 0.9. For all queries, we use a 80/20 train/test split, and report results (speedups, regressions) on the test workload. We did not attempt to tune our models or perform model selection, although it is straightforward to do so by reserving a validation set from the training dataset.

7.2 Kepler Improves Query Execution Latency

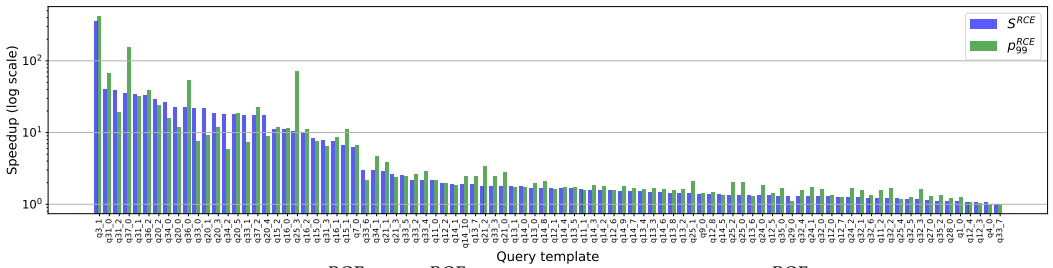
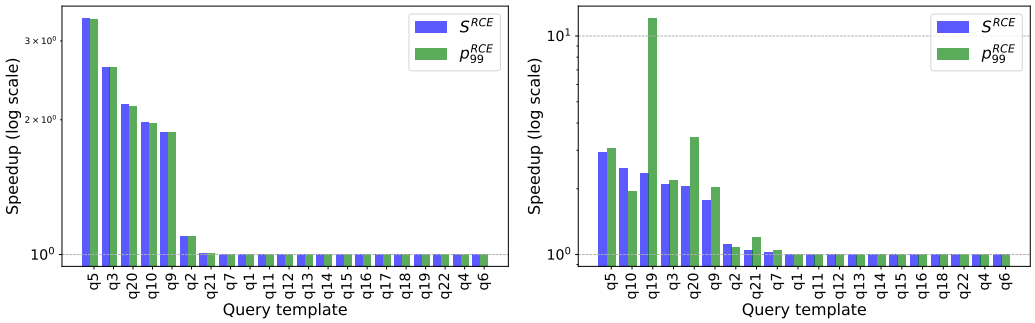


Fig. 6. S^{RCE} and P_{99}^{RCE} on Stack by query, sorted by S^{RCE}

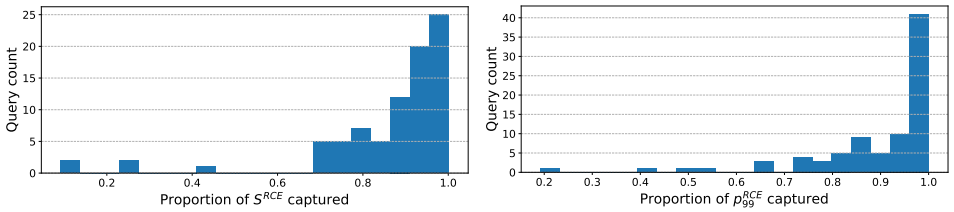
End-to-end performance. We integrate Kepler into the PostgreSQL query optimizer to demonstrate that it delivers large speedups in a real deployment. Our implementation loads Tensorflow Lite models on the database server for fast CPU model inference and uses the `pg_hint_plan` extension to force specific plans via hints. Providing the query id as a comment with the SQL query text from any PostgresSQL client connection triggers Kepler query plan prediction.

We executed a sample of 1000 evaluation set query instances per query on the integrated Kepler PostgresSQL system. Table 5a summarizes the speedups of Kepler over Stack, demonstrating that our PostgreSQL implementation achieves nontrivial speedups on the majority of queries, with over 2x speedup over the entire workload for 32.2% of queries. These speedups indicate that Kepler is able to bypass inaccuracies PostgreSQL's cardinality estimation and cost model via RCE.



(a) TPC-H. (b) TPC-H skewed (z=1).

Fig. 7. S^{RCE} and P_{99}^{RCE} on TPC-H by query, sorted by S^{RCE}.



(a) Stack, model speedup ratio histogram. (b) Stack, p99 model speedup ratio histogram.

Fig. 8. Model results on Stack.

Figure 5b confirms that the Kepler deployment achieves near-identical speedups to those expected based on the pre-collected execution dataset. This is because the use of lightweight ML models and planning hints incur low planning-time overhead. Figure 5c shows the distribution of the ratio of model inference times to PostgreSQL planning time for all queries in Stack. The model inference time is mostly under 5% of PostgreSQL planning time and at most 30%.

Our total speedup results over entire workloads are quite significant since our workloads – query instances sampled uniformly from the space of non-empty query instances – are not designed to adversarially challenge the optimizer. Next, we summarize the contributions from the two key components: (1) RCE to uncover the potential speedups and (2) the ML models to capture speedups. Finally, we compare the results to Bao as a baseline.

RCE speedups. We illustrate the efficacy of RCE by showing that it achieves large speedups on both Stack and TPC-H. Figure 6 shows the per-template S^{RCE} and P_{99}^{RCE}, with RCE achieving over 2x speedup on 31/87 queries and over 1.2x speedup on 78/87 queries.

Similarly, RCE improves 6/22 queries on TPC-H uniform (Figure 7a) and 9/22 queries on TPC-H skewed (Figure 7b). In particular, RCE finds larger speedups on TPC-H skewed due to the non-uniformity in its data distribution.

ML models predict fastest plans and avoid regressions. Next, we show that our ML models are able to robustly capture the speedup produced by RCE, i.e. they maximize p while minimizing P^{reg}. In Figures 8a and 8b, we plot what proportion of S^{RCE} and P_{99}^{RCE} on Stack we respectively capture per query. These distributions show that our models can reliably predict near-optimal plans: our models capture over 80% of S^{RCE} on over 80% of Stack queries. In Figure 9, we plot the distribution

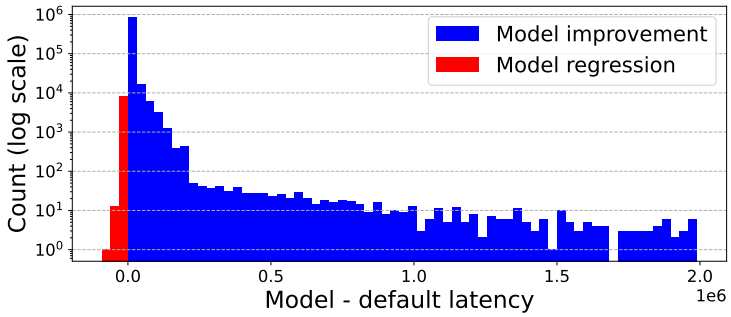


Fig. 9. Query improvements and regressions on all query instances in Stack.

Table 2. Comparison of Bao and Kepler performance.

Query	Bao speedup	Kepler speedup
q11_0	1.201	**1.651**
q12_0	1.238	**1.271**
q13_0	1.408	**1.566**
q14_0	**1.848**	1.619
q15_0	1.157	**7.727**
q16_0	1.168	**9.250**

of the absolute magnitude of model improvements and regressions compared to the default plan, illustrating that the frequency and magnitude of the regressions are minimal compared to those of the improvements.

Bao on parameterized queries. We evaluate Bao, one of the few prior approaches that demonstrates actual improvement in execution latency, on our parameterized version of Stack [25]. For illustrative purposes, we run Bao for 2000 query instances on six representative templates from the original Stack dataset [25]. As shown in Table 2, we observed that Kepler outperforms Bao on 5 out of 6 templates, and in particular is able to find far greater speedups on q15_0 and q16_0. As we later show in Figure 10a, this is because the candidate generation algorithm in Bao is severely suboptimal.

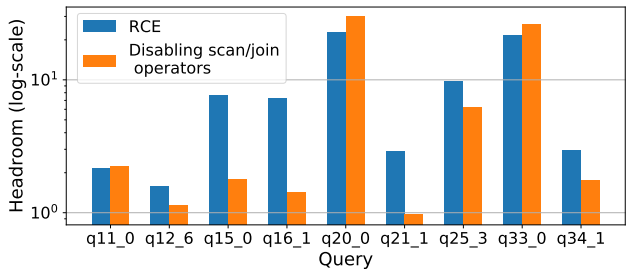
Training data collection cost. The speedups achieved by Kepler come at a nontrivial training query execution cost - on average, we used 39 CPU days worth of query execution time per query template. Hence, Kepler is most applicable to workloads where query templates are executed at high frequency. We discuss directions to significantly reduce this training data collection cost at the end of Section 7.4 as well as Sections 7.5 and 8.

7.3 Analyzing RCE

Having observed the behavior of the overall system, we now discuss characteristics of the first major component of Kepler: RCE. We evaluate RCE's performance against candidate generation and plan pruning baselines before exploring the effects of index configuration and key hyperparameter choices. After discussing RCE's empirical performance against those of exact cardinality (EC) plans, we close the section by offering perspectives and empirical justifications for why RCE works well.

Comparison against baselines. We compare against two main candidate generation baselines:

- PG: the default candidate generation method (Section 2) using the PostgreSQL optimizer.



(a) Comparison between RCE and Bao candidate generation (disabling scan/join methods).

Algorithm	Speedup S	Tail speedup $P99$	% improved
RCE	3.14	3.65	89.0%
RCE + PCP (ours)	**3.11**	**3.63**	**87.9%**
RCE + CBP	1.37	1.77	50.5%
PG	2.0	2.07	49.4%
PG + CBP [34]	1.96	2.02	49.4%

(b) Comparison between PG and RCE with alternative pruning algorithms. We report average and 99th-percentile speedup aggregated using geometric mean, as well as the percentage of queries having total speedup greater than 20%. PG + CBP corresponds to the method in [34].

Fig. 10. Candidate generation baseline comparisons on Stack.

Table 3. Effect of number of generations in RCE in Stack query q11_0.

G	Total num plans	Plan cover size	S^{RCE}
1	9	6	1.369
2	29	7	2.185
3	62	9	2.235
4	99	8	2.166

- Bao: for each query instance, we try every Bao arm, i.e. all 48 valid combinations on disabling various join/scan methods in PostgreSQL [27].

Since [34] consider PG + cost-based pruning (CBP) as their candidate generation method, we simultaneously compare RCE against PG and our plan-cover pruning (PCP) algorithm against CBP in Table 10b. RCE finds significantly more speedups than PG: S^{RCE} is over 1.2 on 89% of Stack queries, as opposed to 49.4% for PG. Although CBP achieves little speedup loss when applied to PG, it incurs far greater loss when applied to RCE, indicating that optimizer cost estimates cannot reliably predict the quality of RCE plans. By contrast, PCP achieves almost no degradation in speedup since it uses actual execution data to evaluate plans.

The Bao candidate generation method produces far more plans than RCE, so for computational reasons we evaluated it on a diverse subset of queries designed to be representative of the entire Stack dataset. Figure 10a shows that RCE achieves similar or better speedup on all queries, and is notably able to find non-trivial speedup on each query.

Table 4. Geometric means of per-query S^{RCE} for various index configurations.

PK	+FK	+Predicates	+DBA
1.888	2.41	5.286	5.284

Table 5. Comparison of average query latencies (in seconds) for RCE best plan, exact cardinality plan, and PostgreSQL selected plan.

Query	RCE	Exact Cardinality	Query optimizer
q11_0	**0.045**	0.167	0.180
q12_2	**0.350**	0.768	0.954
q14_10	2.503	**2.107**	5.732
q16_1	**1.136**	2.174	8.575
q20_0	**0.012**	0.046	0.232
q33_0	**0.519**	2.920	3.377

Number of generations G. The hyperparameter G has a significant impact on the number and quality of RCE-generated plans, which we illustrate by varying G for q11_0 in Stack, with results shown in Table 3. The number of plans and plan cover size steadily grows up to three generations, although most of the speedup is captured in plans found using two generations. Increasing G to four generations does not produce any marginal benefit.

Exponential row count perturbation range: b and m. To justify our choice of perturbation range hyperparameters $b = 10$ and $m = 2$, we analyzed the perturbation factor necessary to induce a change at a single level in the plan. For each of $n - 1$ sub-plans in a query tree (for a query Q with n tables), we use binary search to identify the factor by which the cardinality estimate for that sub-plan must be perturbed in order to induce a change in the optimizer's plan. On Stack, we observed that although some plans needed a 10^6 perturbation factor to induce a plan change, the vast majority of factors were less than 10^2.

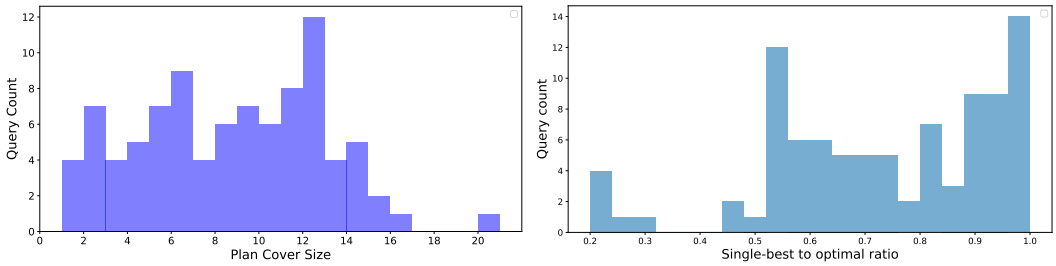
Robustness to index configuration. RCE finds better plans on databases with different index configurations. We ran RCE over the following index configurations for Stack: primary keys only (PK), foreign keys (FK), predicate columns, and database administrator defined additional indexes [25]. Table 4 shows RCE finds faster plans in all configurations. Similar to [22], we find that more indexes leads to a larger speedup.

Can RCE discover optimal plans? Although RCE demonstrably generates faster plans than a variety of baselines, we would also like to know how close are RCE-generated plans to the true optimal plans $p^*_{\texttt{opt}}(q)$. Since it is infeasible to determine $p^*_{\texttt{opt}}(q)$ in practice, we instead compare against the standard benchmark of exact cardinality plans [22, 29].

We executed exact cardinality plans for a subset of the training workload and compared them against their respective best RCE plans in Table 5. We again used a subset of queries from the Stack dataset for computational reasons. Due to the large number of joins in some query templates, we additionally set the exact cardinality for a subset of tables to be a high constant if its corresponding query did not finish within 15 minutes. Notably, RCE plans are substantially faster than exact cardinality plans on 5 out of 6 queries. This illustrates that even accurate cardinality estimation methods can lead to suboptimal plans due to incorrect assumptions and other deficiencies in the cost model.

Table 6. Comparison of average query latencies (in seconds) for RCE-all, RCE-cluster, and RCE-instance.

Query	RCE-all	RCE-cluster	RCE-instance
q11_1	0.024	0.027	0.051
q12_6	0.307	0.307	0.316
q20_0	0.006	0.006	0.010
q21_1	0.129	0.129	0.153
q25_3	2.212	2.212	2.212
q33_0	0.593	0.670	0.810
q34_1	2.482	2.663	2.914



(a) Histogram of plan cover sizes in Stack. (b) Histogram of single-best plan to all-plan S^{RCE} ratio.

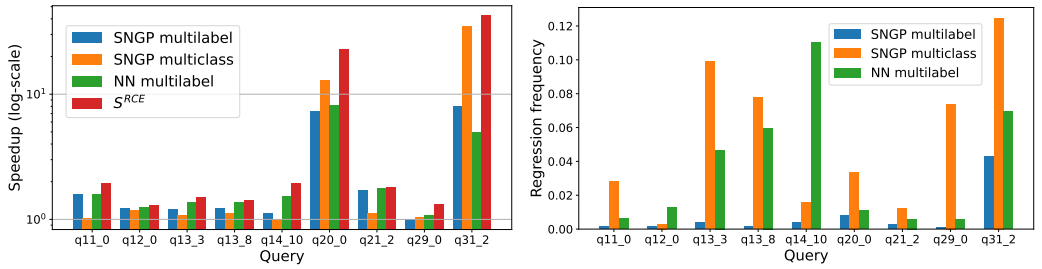
Fig. 11. Parameter sensitivity on all Stack queries.

Query clusters facilitate RCE.. Query instances with similar parameter binding values often have similar query plan behavior, e.g. as visualized by plan diagrams [15]. We hypothesize that these groups of similar instances, or *clusters*, can dramatically increase the efficacy of RCE via plan sharing. Recall that in our candidate generation procedure, we execute RCE for each query instance and take the union over all resulting plan sets. Hence, each cluster only requires a single query instance's RCE process to reach the cluster-wide optimal plan. For a cluster of size N and probability $\mathbb{P}(q_i)$ of query instance q_i discovering the optimal plan, the overall probability of discovering the plan over the cluster is $1 - \prod_{i=1}^{N}(1 - \mathbb{P}(q_i))$, which rapidly approaches 1 for sufficiently large N and a reasonable distribution of $\mathbb{P}(q_i)$.

To demonstrate the existence and impact of clusters, we define a cluster for each plan p as all query instances for which p is the fastest plan. Then, for each query instance q, we compare the execution latencies of the fastest plan from three plan sets: (1) RCE-all, containing all plans from all query instances, (2) RCE-cluster, containing all plans from query instances in the same cluster as q, and (3) RCE-instance, containing only the plans generated from the RCE process for q. As shown in Table 6, RCE-all and RCE-cluster have very similar execution times, while RCE-instance is often slower, indicating that intra-cluster plan sharing plays a large role in the efficacy of RCE.

7.4 ML Models

We first motivate the use of ML models by demonstrating that Stack is highly *parameter sensitive* – i.e. different query instances have different optimal plans. We then justify modeling design choices with an ablation of using SNGP in our models and evaluation of varying confidence thresholds highlight the importance of incorporating robustness as a primary design component in Kepler. We conclude with extensive analyses around feature space selection, embedding vocabularies, and training data size.



(a) $S^{\texttt{model}}$ (b) $P^{\texttt{reg}}$

Fig. 12. Comparison of uncertainty approaches.

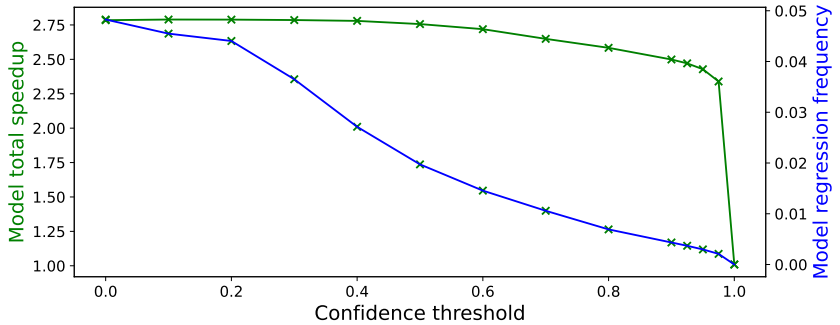


Fig. 13. Tradeoff between $S^{\texttt{model}}$ and $P^{\texttt{reg}}$ (aggregated respectively using geo-mean and mean over all Stack queries.

Parameter sensitivity. We investigate the parameter sensitivity of Stack and TPC-H queries based on our execution data. On Stack, all but four query templates had plan cover size greater than one (Figure 11a). In Figure 11b, we show the distribution of *single-best plan suboptimality ratios*, defined as the ratio of total latency of the oracle best plan against the total latency of the single best plan (i.e., the fixed plan with minimum total execution time). Ratios less than 1 indicate that using only a single plan incurs a loss in speedup, with lower values being more severely suboptimal. Thus, Figure 11b implies that multiple plans are necessary to capture the full speedup.

On the other hand, TPC-H is designed to not be parameter sensitive, which we confirmed by observing a plan cover of size 1 for all queries. Hence, our modelling results focus solely on Stack.

Loss functions/training objectives. Figure 12 compares various loss functions and models: SNGP with multilabel loss, SNGP with multiclass loss, and a vanilla NN with multilabel loss. Multilabel loss + SNGP achieves similar or better speedup to other methods, while having a lower regression frequency for all queries.

Model calibration and uncertainty. We further investigate the ability of our SNGP models in producing calibrated output probabilities. In Figure 13, we plot the test workload model speedup and regression frequency as a function of confidence threshold varying from 0 (no falling back) to 1 (always falling back). The captured speedup and regression frequency both decay smoothly as a function of confidence, which allows the user to specify their tolerance for regressions. This

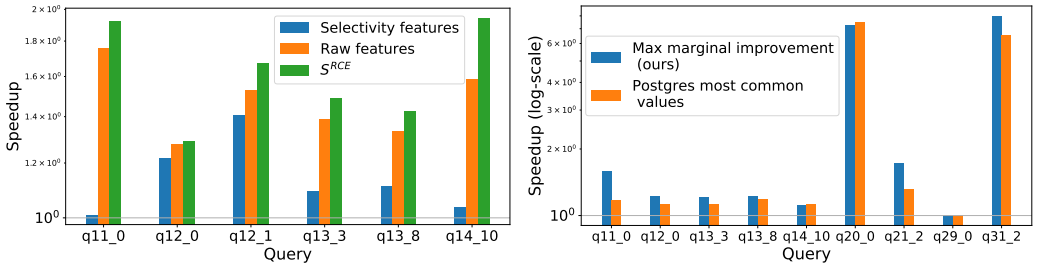
	S^{RCE}	$S^{\texttt{model}}$	$P^{\texttt{reg}}$
No fallback	1.044	0.781	23.8%
Fallback	1.044	0.997	0.2%

(a) Site name.

	S^{RCE}	$S^{\texttt{model}}$	$P^{\texttt{reg}}$
No fallback	2.027	1.866	3.3%
Fallback	2.027	1.822	0.1%

(b) Question last activity date.

Fig. 14. Out-of-distribution experiments.



(a) Selectivity vs raw features.

(b) Embedding vocabulary.

Fig. 15. Speedup ablations on features and vocabulary.

figure also demonstrates the importance of the fallback mechanism: one can dramatically reduce the regression frequency while only sacrificing a small portion of the speedup.

SNGP out-of-distribution detection. Although Kepler is designed for relatively static workloads, it is robust to dynamic workloads by falling back to the default plan for out-of-distribution (OOD) inputs. We evaluate SNGP's OOD detection ability by holding out specific slices of the training distribution for an example query, q21_2 on Stack. We consider two variants: (1) holding out sites totaling up to 20% of the workload, and (2) holding out the last 20% of last_activity_date values on the question table. Figure 14 shows that in both scenarios, Kepler's fallback mechanism allows it accurately detect OOD inputs and drastically reduce $P^{\texttt{reg}}$ while still preserving some speedup.

Raw parameter values vs selectivity features. In Figure 15a, we ablate our feature choice using raw features and selectivity features. We compare their model speedups to S^{RCE} on a subset of Stack queries. Selectivity features perform far worse due to the poor cardinality estimates in PostgreSQL.

Vocabulary. In Figure 15b, we ablate how we select the embedding vocabulary for string features. In particular, we evaluate our max marginal improvement method (described in Section 6) against choosing the most frequent values based on the PostgreSQL histogram. Our results confirm that the best strategy is choosing the vocabulary to the be the values with the most potential impact on the speedup.

How much training data is required? We evaluate the performance of our models when using less data by subsampling the training data size, as shown in Figure 16. As expected, model speedup improves with more training data while maintaining a regression frequency below 0.2%. The leftmost point uses only 5% of the data, or 200 training query instances, demonstrating that a large amount of speedup can be robustly captured with small amounts of training data.

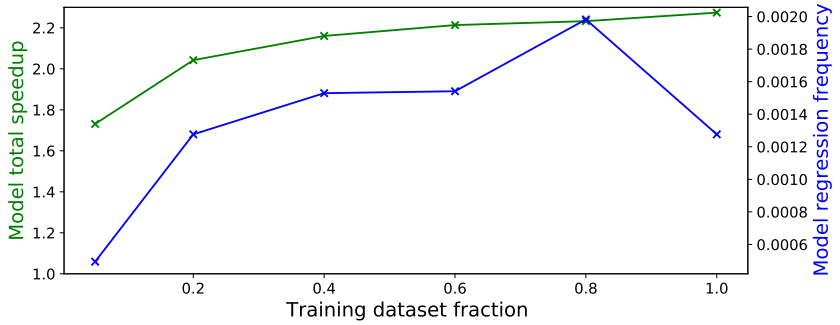


Fig. 16. Model speedup and regression frequency as a function of fraction of training data used.

7.5 Dataset Contribution

To train models and evaluate Kepler, we generated a query execution dataset that comprises ~14.2 years of CPU time across 200 million query executions of 131 query templates. By releasing this dataset online[4], we envision that it can be used to facilitate future research in modeling and efficiency without having to execute any queries. For example, possible use cases include simulating active learning approaches that only selectively execute a subset of queries, or developing better models or loss functions. We believe that the techniques developed using this dataset – and not the specifically trained models – will be directly transferable to other parameterized query settings.

8 CONCLUSION AND FUTURE WORK

We introduced Kepler, a system that can robustly speed up parameterized queries using a learning-based approach. We extensively evaluated Kepler on PostgreSQL and demonstrated that (1) our novel candidate generation algorithm RCE can provide significant speedups in query execution latency, and (2) robust ML models can reliably predict faster plans while avoiding regressions. Interestingly, we observed that RCE-generated plans were often far better than exact cardinality plans, indicating that even a widely-used system as PostgreSQL has significant room for improvement. Evaluating Kepler on database platforms other than PostgreSQL is a natural next step; we believe that the empirical nature of Kepler allows it to discover performance gains regardless of the DBMS.

There are a myriad of future directions for improving the efficiency and performance of Kepler. For example, prior work in cardinality estimation can benefit Kepler in several ways. Instead of perturbing uniformly, RCE can leverage generative cardinality distributions to sample higher likelihood perturbations, e.g. from NeuroCard [38]. Another possibility is to augment the model features with the query plan tree and selectivity estimates, allowing the model to determine when cardinality estimates are accurate, as well as leveraging shared structure between similar query templates. Our models are the first demonstration that speedups can be robustly captured; they can likely be substantially improved via additional modeling techniques and tuning. Similarly, while our end-to-end PostgreSQL integration is sufficient to demonstrate Kepler's performance gains on a real system, every aspect of this implementation can be further tuned.

We utilized an expensive training data collection procedure in order to make more robust claims about our results and produce a complete dataset for further modeling and efficiency research. For practical purposes, our training procedure can likely be made much more efficient, e.g. via active

[4]https://github.com/google/kepler

learning. In conjunction with Figure 16, this implies that similar performance can be achieved with significantly less training cost.

REFERENCES

[1] 2022. Introduction to Aurora PostgreSQL Query Plan Management. https://aws.amazon.com/blogs/database/introduction-to-aurora-postgresql-query-plan-management/

[2] 2022. Oracle: Improving Real-World Performance Through Cursor Sharing. https://docs.oracle.com/en/database/oracle/oracle-database/18/tgsql/improving-rwp-cursor-sharing.html

[3] 2022. Parameter Sensitivity Plan optimization. https://docs.microsoft.com/en-us/sql/relational-databases/performance/parameter-sensitivity-plan-optimization?view=sql-server-ver16

[4] 2022. Skewed Data Generator for TPCH. https://github.com/gunaprsd/SkewedDataGenerator

[5] 2022. TPCH Benchmark. https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp

[6] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. 2012. Learning-based query performance modeling and prediction. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 390–401.

[7] Gunes Aluç, David E DeHaan, and Ivan T Bowman. 2012. Parametric plan caching using density-based clustering. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 402–413.

[8] Alejandro Correa Bahnsen, Djamia Aouada, and Björn Ottersten. 2014. Example-dependent cost-sensitive logistic regression for credit scoring. In *2014 13th International conference on machine learning and applications*. IEEE, 263–269.

[9] Surajit Chaudhuri, Hongrae Lee, and Vivek R Narasayya. 2010. Variance aware optimization of parameterized queries. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 531–542.

[10] Surajit Chaudhuri, Vivek Narasayya, and Ravi Ramamurthy. 2009. Exact cardinality query optimization for optimizer testing. *Proceedings of the VLDB Endowment* 2, 1 (2009), 994–1005.

[11] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288. http://www.vldb.org/pvldb/vol7/p277-difallah.pdf

[12] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R Narasayya. 2019. Ai meets ai: Leveraging query executions to improve index recommendations. In *Proceedings of the 2019 International Conference on Management of Data*. 1241–1258.

[13] Anshuman Dutt, Vivek Narasayya, and Surajit Chaudhuri. 2017. Leveraging re-costing for online optimization of parameterized queries with guarantees. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1539–1554.

[14] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, et al. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *arXiv preprint arXiv:2109.05877* (2021).

[15] Naveen Reddy Jayant R Haritsa. 2005. Analyzing plan diagrams of database query optimizers. In *Proceedings of the 31st international conference on Very large data bases. VLDB Endowment*. 1228–1239.

[16] Alexander Hepburn, Ryan McConville, Raúl Santos-Rodríguezo, Jesús Cid-Sueiro, and Dario García-García. 2018. Proper losses for learning with example-dependent costs. In *Second International Workshop on Learning with Imbalanced Domains: Theory and Applications*. PMLR, 52–66.

[17] Arvind Hulgeri and S Sudarshan. 2002. Parametric query optimization for linear and piecewise linear cost functions. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 167–178.

[18] Yannis E Ioannidis, Raymond T Ng, Kyuseok Shim, and Timos K Sellis. 1997. Parametric query optimization. *The VLDB Journal* 6, 2 (1997), 132–151.

[19] 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*. 1214–1227.

[20] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).

[21] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196* (2018).

[22] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[23] Jeremiah Liu, Zi Lin, Shreyas Padhy, Dustin Tran, Tania Bedrax Weiss, and Balaji Lakshminarayanan. 2020. Simple and principled uncertainty estimation with deterministic deep learning via distance awareness. *Advances in Neural Information Processing Systems* 33 (2020), 7498–7512.

[24] Yao Lu, Srikanth Kandula, Arnd Christian König, and Surajit Chaudhuri. 2021. Pre-training summarization models of structured datasets for cardinality estimation. *Proceedings of the VLDB Endowment* 15, 3 (2021), 414–426.

[25] 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*. 1275–1288.
[26] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711* (2019).
[27] Ryan Marcus and Olga Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–4.
[28] Ryan Marcus and Olga Papaemmanouil. 2018. Towards a hands-free query optimizer through deep learning. *arXiv preprint arXiv:1809.10212* (2018).
[29] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: learning cardinality estimates that matter. *arXiv preprint arXiv:2101.04964* (2021).
[30] Richard T Snodgrass, Sabah Currim, and Young-Kyoon Suh. 2022. Have query optimizers hit the wall? *The VLDB Journal* 31, 1 (2022), 181–200.
[31] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned cardinality estimation: A design space exploration and a comparative evaluation. *Proceedings of the VLDB Endowment* 15, 1 (2021), 85–97.
[32] Immanuel Trummer. 2019. Exact cardinality query optimization with bounded execution cost. In *proceedings of the 2019 international conference on management of data*. 2–17.
[33] Grigorios Tsoumakas and Ioannis Katakis. 2007. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining (IJDWM)* 3, 3 (2007), 1–13.
[34] Kapil Vaidya, Anshuman Dutt, Vivek Narasayya, and Surajit Chaudhuri. 2021. Leveraging query logs and machine learning for parametric query optimization. *Proceedings of the VLDB Endowment* 15, 3 (2021), 401–413.
[35] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2020. Are we ready for learned cardinality estimation? *arXiv preprint arXiv:2012.06743* (2020).
[36] Wentao Wu, Jeffrey F Naughton, and Harneet Singh. 2016. Sampling-based query re-optimization. In *Proceedings of the 2016 International Conference on Management of Data*. 1721–1736.
[37] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. *arXiv preprint arXiv:2201.01441* (2022).
[38] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: one cardinality estimator for all tables. *arXiv preprint arXiv:2006.08109* (2020).
[39] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep unsupervised cardinality estimation. *arXiv preprint arXiv:1905.04278* (2019).

Received July 2022; revised October 2022; accepted November 2022