Flow-Loss: Learning Cardinality Estimates That Matter

Parimarjan Negi[1], Ryan Marcus[12], Andreas Kipf[1], Hongzi Mao[1], Nesime Tatbul[12], Tim Kraska[1], Mohammad Alizadeh[1]
[1]MIT CSAIL, [2]Intel Labs
{pnegi,rcmarcus,kipf,hongzi,kraska,alizadeh}@mit.edu
{tatbul}@csail.mit.edu

ABSTRACT

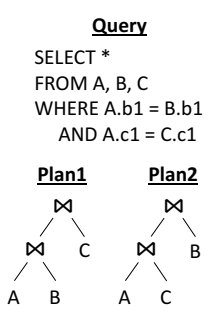
Recently there has been significant interest in using machine learning to improve the accuracy of cardinality estimation. This work has focused on improving average estimation error, but not all estimates matter equally for downstream tasks like query optimization. Since learned models inevitably make mistakes, the goal should be to improve the estimates that make the biggest difference to an optimizer. We introduce a new loss function, Flow-Loss, for learning cardinality estimation models. Flow-Loss approximates the optimizer's cost model and search algorithm with analytical functions, which it uses to optimize explicitly for better query plans. At the heart of Flow-Loss is a reduction of query optimization to a flow routing problem on a certain "plan graph", in which different paths correspond to different query plans. To evaluate our approach, we introduce the Cardinality Estimation Benchmark (CEB) which contains the ground truth cardinalities for sub-plans of over 16K queries from 21 templates with up to 15 joins. We show that across different architectures and databases, a model trained with Flow-Loss improves the plan costs and query runtimes despite having worse estimation accuracy than a model trained with Q-Error. When the test set queries closely match the training queries, models trained with both loss functions perform well. However, the Q-Error-trained model degrades significantly when evaluated on slightly different queries (e.g., similar but unseen query templates), while the Flow-Loss-trained model generalizes better to such situations, achieving $4-8\times$ better 99th percentile runtimes on unseen templates with the same model architecture and training data.

PVLDB Reference Format:
Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, Mohammad Alizadeh. Flow-Loss: Learning Cardinality Estimates That Matter. PVLDB, 14(11): 2019 - 2032, 2021.
doi:10.14778/3476249.3476259

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/learnedsystems/ceb.

1 INTRODUCTION

Cardinality estimation is a core task in query optimization for predicting the sizes of *sub-plans*, which are intermediate operator trees needed during query optimization. Query optimizers use these estimates to compare alternative query plans according to a cost model and find the cheapest plan. Recently, machine learning approaches have been successful in improving cardinality estimation accuracy [9, 14, 18, 54, 57], but they largely neglect the impact of improved estimates on the generated query plans. This is the first work (known to us) that learns cardinality estimates by directly optimizing for the cost of query plans generated by an optimizer.



Cardinality	True	Estimator1	Estimator2
|A|	4	4	4
|B|	2	2	2
|C|	2	2	2
|A ⋈ B|	5	10	10
|A ⋈ C|	8	16	8
Cost1	**13**	**18**	18
Cost2	16	24	**16**

Cost1 = (|A|+|B|) + (|A ⋈ B|+|C|)
Cost2 = (|A|+|C|) + (|A ⋈ C|+|B|)

Figure 1: For this example, we use the sum of the cardinalities as the cost of a plan. With true cardinality values, Plan1 is cheaper than Plan2. This is also the case with Estimator1. Interestingly, however, although Estimator2's cardinality values have smaller error than those of Estimator1, they will mislead the optimizer to choose Plan2.

All learned models will have non-trivial estimation errors due to limitations in model capacity, featurization, training data, and differences between training and testing conditions (e.g., due to changing workloads). We argue that it is therefore crucial to understand which errors are more acceptable for the optimizer. Unsupervised methods learn a model of the data independent of any particular query workload, thereby using model capacity for sub-plans that will never occur. Supervised methods use a representative workload to focus model capacity on likely sub-plans. However, all estimates are not equally important. While an optimizer's decisions may be very sensitive to estimates for some sub-plans (e.g. join of two large tables), other estimates may have no impact on its decisions.

We propose *Flow-Loss*, a loss function for supervised cardinality estimation learning that explicitly emphasizes estimates that matter to query performance for a given workload. Flow-Loss is a drop-in replacement for loss functions like Q-Error [34] that are commonly used to train cardinality estimation models. Flow-Loss takes the idea of focusing model capacity to its logical extreme — encouraging better estimates only if they improve the resulting query plans. For instance, consider Figure 1: Estimator2 corrects Estimator1's estimate of $A \bowtie C$, but it actually leads to a worse plan

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476259

(Plan 2), because the relative cardinalities ($A \bowtie B$ vs. $A \bowtie C$) are wrong. Flow-Loss will show no error for Estimator1, while nudging Estimator2 to correct the relative cardinalities of these two joins.

At its core, Flow-Loss computes the cost of a query plan as a function of the cardinality estimates used to generate the plan. To do this, it approximates the optimizer's cost model and dynamic programming (DP) search algorithm with smooth and differentiable analytical functions. This lets us use standard gradient descent techniques to improve the estimates that are most relevant to improving the query plans. We show that improving cardinality estimates w.r.t. this objective also improves the quality of plans generated by real optimizers like PostgreSQL. A key technical ingredient underlying Flow-Loss is a connection between the optimizer's DP search algorithm and a flow routing problem on a certain "plan graph", in which different paths correspond to different query plans. By exploiting this connection, we derive closed-form expressions relating cardinality estimates to the resulting query plan costs.

There are two main benefits of training models to minimize Flow-Loss. First, Flow-Loss highlights which sub-plans are most relevant to the query optimizer. This helps a model focus its limited capacity on robustly estimating the sizes of such sub-plans. Across various scenarios, we show that Flow-Loss-trained model have worse average estimation accuracy than Q-Error-trained models, but improve the cost of generated plans. For instance, we show that models trained with Flow-Loss can adapt to being provided fewer input features or noisy data collected via approximate query processing (AQP) [24, 25]. It is attractive to use AQP training data because it can be generated significantly faster than the true cardinalities. But, at the $99th$ percentile, Q-Error trained models get significantly worse when using AQP estimates: Q-Error gets 10× worse, PostgreSQL costs get 2× worse, and query runtime gets 30% slower. Meanwhile, Flow-Loss models show no such degradation when switching training data to use AQP estimates.

Second, by having a larger tolerance for errors on less critical sub-plans, training with Flow-Loss can avoid overfitting the model to cardinalities for which precise estimates are not needed, thereby leading to simpler models without sacrificing query performance. Such simpler models typically generalize better. We show that models trained using Q-Error can be brittle, and can lead to significant regressions when the query workload diverges slightly from the training queries; for instance, in the worst cases, models trained with Q-Error are up to $4-8\times$ slower than models trained with Flow-Loss at the $99th$ percentile. These correspond to 1.5-3× better query runtimes at the mean depending on the PostgreSQL configuration.

Our key contributions are:

- **DBMS-based Plan Cost.** Based on Moerkotte et al.'s [34] plan cost, defined using arbitrary cost models, we introduce a cost model-based proxy for the runtime of a query plan in a particular DBMS. We show that it corresponds closely to runtimes, and thus is a useful metric to evaluate the goodness of cardinality estimates in terms of their impact on query optimization. Further, we provide an implementation to easily evaluate the performance of cardinality estimation models on Plan Cost using PostgreSQL or MySQL.
- **Flow-Loss.** We introduce Flow-Loss, a smooth and differentiable approximation of Plan-Cost, which can be optimized by any supervised learning model with gradient descent.
- **Cardinality Estimation Benchmark (CEB).** We create a new tool to generate challenging queries based on templates in a semi-automated way. We use this to create the Cardinality Estimation Benchmark, which is over 100× larger than the Join Order Benchmark (JOB) [23], and has more complex queries.

2 RELATED WORK

For cardinality estimation, traditional approaches have used histograms [3], sampling [24], wavelets [32], kernel density estimation [16]„ or singular value decomposition [42]. Recently, machine learning approaches have shown high estimation accuracy. Many works focus on single-table selectivity estimates [9, 12, 40, 57], but while this is useful in other contexts, such as approximate query processing, it is non-trivial to extend such models to joins using join sampling [59]. Learned cardinality estimation for joins can be categorized into *unsupervised* (data-driven, independent of query workload) and *supervised* (query-driven) approaches. Unsupervised approaches for cardinality estimation include Probabilistic Graphical Models [11, 48], Sum-Product Networks [14], or deep autoregressive models [56]. NeuroCard [56] is the most advanced of these approaches, but it still does not support the complex analytical workloads studied in this work (e.g., queries with self joins). That being said, any unsupervised model can be integrated into our approach by providing their estimates as features.

Supervised approaches use queries with their true cardinalities as training data to build a regression model. Our work builds on the approach pioneered by Kipf et al. [18]. While several such works report improved estimation accuracy [8, 9, 18, 38, 54, 55], only a few actually demonstrate improved query performance [15, 37, 39]. Our approach seeks to learn the cardinalities used by a traditional DBMS optimizer, while using the optimizer's search and cost algorithms for query optimization. Recently, there have been several other learning approaches to improve query performance which are complementary to our methods: learning the complete optimizer [20, 30, 31], learning to use the optimizer's hints [29], learning the cost model [45], re-optimization [41, 47], pessimistic cardinality estimation [4, 5, 13].

3 OVERVIEW

In this section, we will provide the high-level intuition behind our approach, which will be formalized in the next sections. We target supervised learning methods that use a parametric model, such as a neural network, to estimate cardinalities for sub-plans required to optimize a given query. Today, such models are trained using loss functions that compare true and estimated cardinalities for a given sub-plan, such as Q-Error:

Definition 3.1.

$$\text{Q-Error}(\boldsymbol{y}, \hat{y}) = \max(\boldsymbol{y}/\hat{y}, \hat{y}/\boldsymbol{y}). \quad (1)$$

where $\boldsymbol{y}$ and $\hat{y}$ are the true and estimated cardinalities for one sub-plan.

Such a loss function treats every estimate as equally important. Instead, we want a loss function that will focus model capacity on

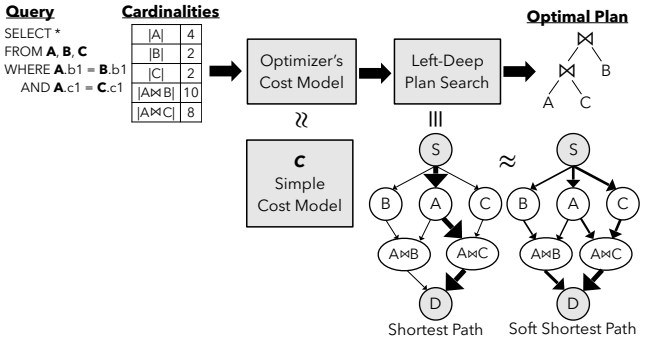


Figure 2: The query optimization process has two non-differentiable components: the cost model and the plan search algorithm. We develop differentiable approximations for these so we can understand how sensitive query plans are to changes in cardinality estimates.

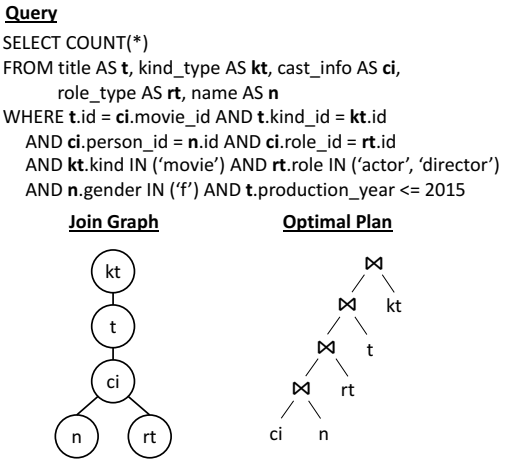


Figure 3: Join graph and optimal plan for sample query Q_1 on the IMDb database.

improving accuracy of estimates that matter most to the quality of the plans produced by the optimizer, while tolerating larger errors for other estimates. This loss function will need to be differentiable so we can optimize it using standard gradient descent methods.

To understand how cardinality estimates impact the resulting query plan, let us consider the basic structure of a query optimizer. There are two independent components, as highlighted in Figure 2: *(i)* a *cost model*, which outputs a cost for every join given the cardinality estimates for all sub-plans. *(ii)* a *DP search algorithm*, which finds the cheapest query plan. Our goal is to approximate both components using analytical functions that can be combined into a single, differentiable loss function:

$$\hat{Y} \xrightarrow{C(\cdot)} \text{Join-Cost} \xrightarrow{S(\cdot)} \text{Plan}. \tag{2}$$

Here $C(\cdot)$ maps the cardinality estimates, $\hat{Y}$, to the cost of each join, and $S(\cdot)$ maps the join costs to the optimal plan. Approximating the cost model as an analytical function is conceptually straightforward since it is already represented using analytical expressions. In principle, we can make this function as precise as we want, but we found that a simple approximation with terms to cost joins with or without indexes works well in our workloads (Definition 4.5).

However, the DP search algorithm is non-trivial to model analytically. Our key contribution is in developing a differentiable analytical function to approximate *left-deep* plan search. Left-deep plans join a single table to a sub-plan at each step. Our construction exploits a connection between left-deep plan search and the shortest path problem on a certain "plan graph". While we focus on left-deep search for tractability, the resulting loss function improves the performance for all query plans, as the sub-plans required for costing left-deep plans are the same as required for all plans.

Figure 2 shows the plan graph corresponding to a simple query that joins three tables A, B, and C. Every edge in the plan graph represents a join and has a cost, and every path between S and D represents a left-deep plan. The DP search algorithm outputs the cheapest plan, i.e. the shortest path. When cardinality estimates change, they change the cost of the edges in the plan graph, possibly changing the shortest path. Therefore, to capture the influence of cardinality estimates on the plan analytically, we need an expression to relate edge costs to the shortest path in the plan graph.

But this alone is not enough. The shortest path is insensitive to small changes to most edge costs (and hence, small changes to most cardinality estimates). For instance, consider any edge not on the shortest path; slightly increasing or decreasing the cost of that edge would not change the shortest path. Therefore an analytical function based on the shortest path would not have a *gradient* with respect to the cost of such edges. This would make it impossible for gradient-descent-based learning approaches to improve.

We tackle these challenges by using a soft approximation to the shortest path problem. In this formulation, the plan graph is viewed as an electrical circuit, with each edge having a resistance equal to its cost. One unit of current is sent from S to D, split across paths in a way that minimizes the total energy consumed.[1] This formulation has two advantages over shortest path. First, it provides an explicit, closed-form expression relating the edge resistances (costs) to the amount of current on every path. Second, it does not suffer from the non-existent gradient problem described above. In an electrical circuit, the current is not exclusively sent on the path with the least resistance (i.e., the path corresponding to the cheapest plan). Instead, all low-resistance paths carry a non-negligible amount of current. Therefore, changing the resistance (cost) of an edge on any of these paths will affect the distribution of current across the entire circuit. The implication in our context is that all joins involved in low-cost query plans matter (even if they do not appear in the cheapest plan). This aligns with the intuition that the optimizer is sensitive to precisely these joins: changing their cost could easily change the plan it picks.

4 DEFINITIONS

This section formally defines the plan graph and the concepts we use to develop our new loss function, Flow-Loss. As a running

[1]Electrical flows have been used for graph algorithms in various fields: modeling random walks [7], developing more efficient algorithms for approximating the maximum flow problem [6, 22, 28], modeling landscape connectivity in ecology [33], and inferring relatedness in evolutionary graphs in biology [27].

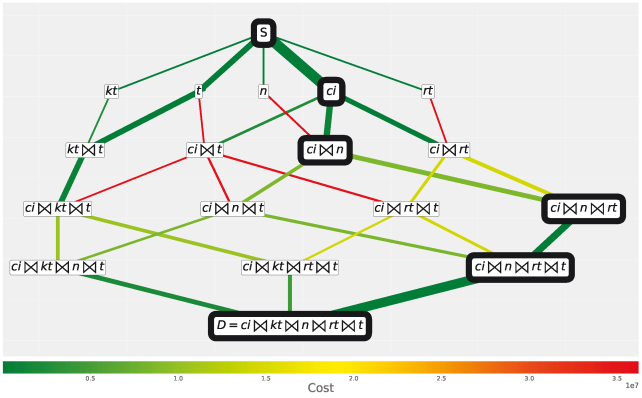


Figure 4: Plan graph (Definition 4.2) for query Q_1. The cheapest path, $P^*(\mathbf{Y})$, is highlighted. The edges are colored according to $C(e, \mathbf{Y})$. The relative thickness of the edges represent the flows computed by Equation 6, $F^*(\mathbf{Y})$.

example, we will consider the query Q_1 (Figure 3) on the Internet Movie Database (IMDb). Throughout this work, joins refer to inner joins, and we ignore cross-joins. For simplicity, we assume all joined columns have an index.

Definition 4.1. *Sub-plan.* Given query Q, a sub-plan is a subset of tables in Q that can be joined using inner joins. In query Q_1 (cf. Figure 3), $kt \bowtie t$ is a sub-plan but $kt \bowtie ci$ is not.

Definition 4.2. *Plan graph.* Given query Q, the plan graph is a directed acyclic graph (V,E) where V is the set of all sub-plans, and there is an edge corresponding to every join in Q between a sub-plan and a base table, i.e. $(u, v) \in E$ if and only if $v = u \bowtie b$ for a base table b. For convenience, we add a node S for the empty set, which has an edge to all nodes containing exactly one table. We use D to denote the node consisting of all tables. Figure 4 shows the plan graph for query Q_1.

Definition 4.3. *Path / Plan, P.* A path (sequence of edges) from S to D in the plan graph. Any left-deep plan corresponds to a path from S to D. For instance, the plan $(((t \bowtie kt) \bowtie ci) \bowtie n) \bowtie rt$ for query Q_1 corresponds to: $S \rightarrow t \rightarrow t \bowtie kt \rightarrow t \bowtie kt \bowtie ci \rightarrow t \bowtie kt \bowtie ci \bowtie n \rightarrow D$ in Figure 4.

Definition 4.4. *Cardinality vector Y.* The cardinalities for each node (sub-plan) in the plan graph. We use $\mathbf{Y}$ and $\hat{Y}$ to refer to true and estimated cardinalities.

Definition 4.5. $C(e, Y)$. A cost model which takes as input an edge (join) e in the plan graph and assigns it a cost given the cardinality vector Y. The cost model can take any functional form. In this paper, to approximate PostgreSQL, we use the following simple cost model:

$$C((u, v), Y) = \min(|u| + \lambda|b|, |u| \cdot |b|) \tag{3}$$

where b is a base table s.t. $u \bowtie b = v$ and $|u|, |b|$ are cardinalities of u and b given by Y. The term $|u| \cdot |b|$ models nested loop joins without an index, and $\lambda = 0.001$ is used to model an index on b. Figure 4 shows the cost of each edge in query Q_1. Flow-Loss can use a more precise cost model (e.g., with terms for other join operators such as hash join), but we found this simple model suffices for our workloads on PostgreSQL. We analyze how well it approximates the PostgreSQL cost model in §6.1 and discuss another model tailored to MySQL in an online appendix [35].

Definition 4.6. P^* (Y). The cheapest path (plan) in the plan graph with edge costs given by $C(e, Y)$:

$$P^*(Y) = \arg\min_P \sum_{e \in P} C(e, Y). \tag{4}$$

For example, given $\mathbf{Y}$, the cheapest path $P^*(\mathbf{Y})$ is highlighted in Figure 4. We will use the terms "cheapest" and "shortest" path interchangeably.

Definition 4.7. *Plan Cost (P-Cost), $PC(\hat{Y}, \mathbf{Y})$.* The true cost of the optimal path (plan) chosen based on cardinality vector $\hat{Y}$:

$$PC(\hat{Y}, \mathbf{Y}) = \sum_{e \in P^*(\hat{Y})} C(e, \mathbf{Y}). \tag{5}$$

P-Cost can be viewed as an alternative to loss functions like Q-Error to compare estimated and true cardinalities $\hat{Y}$ and $\mathbf{Y}$. It finds the cheapest path using $\hat{Y}$, i.e. $P^*(\hat{Y})$, and then sums the *true* costs of the edges in this path using $\mathbf{Y}$.

Remark. As defined, P-Cost is not a distance metric [52] (e.g., it does not satisfy the symmetry property). However, this does not affect its use in our loss function. In the online appendix [35], we use P-Cost to construct a pseudometric [53].

5 FLOW-LOSS

While P-Cost captures the impact of cardinalities on query plans, it has an important drawback as a loss function: It cannot be minimized using gradient-based methods. In fact, the gradient of P-Cost with respect to $\hat{Y}$ is zero at almost all values of $\hat{Y}$. To see why, notice that a small perturbation to $\hat{Y}$ is unlikely to change the path chosen by $P^*(\hat{Y})$; the path would only change if there were multiple cheapest paths. Therefore P-Cost will also not change. In this section we define an alternative to P^* that has a gradient w.r.t. any cardinality in the plan graph, and use it to construct Flow-Loss.

5.1 From Shortest Path to Electrical Flows

The problem with P^* is that it strictly selects the shortest (cheapest) path in the plan graph. Consider, instead, the following alternative that can be thought of as a "soft" variant of shortest path. Assume the plan graph is an electrical circuit, with edge e containing a resistor with resistance $C(e, Y)$. Now suppose we send one unit of current from S to D. How will the current be split between the different paths from S and D?

In an electric circuit, paths with lower resistance[2] (shorter paths) carry more current, but the current does not flow exclusively on the path with least resistance: assuming all paths have a non-zero resistance, they will all carry some current. Importantly, every edge's resistance affects how current is split across paths. The

[2]For the purpose of this discussion, we view the resistance of a path as the sum of the resistances along its edges, which corresponds to the path's *length* when the resistance is viewed as a distance, or the path's *cost* when the resistance is viewed as the cost of an edge.

precise way in which current flows in the circuit can be obtained by solving the following *energy minimization*[3] problem:

$$F^*(Y) = \arg\min_F \sum_{e \in E} C(e, Y) \cdot F_e^2 \tag{6}$$

$$\text{s.t} \sum_{e \in Out(S)} F_e = \sum_{e \in In(D)} F_e = 1 \tag{7}$$

$$\sum_{e \in Out(V)} F_e = \sum_{e \in In(V)} F_e \tag{8}$$

Here the optimization variable F assigns a flow of current to each edge. Equation (7) enforces that one unit of flow is sent from S to D. Equation (8) is the conservation constraint for all nodes except S and D — it enforces that the amount of flow going in and out of a node should be the same. The thickness of edges in Figure 4 show the flows assigned to each edge by $F^*(\mathbf{Y})$.

Computing F^* is a basic problem in circuit design [1, 6], and it has a simple closed form expression as a function of the resistances $C(e, Y)$. For a plan graph with M edges and N nodes, we can compute the flows by:

$$F^*(Y) = AB^{-1}i, \tag{9}$$

where $i \in R^N$ is the constant vector of $[1, 0, ..., -1]$; $A \in R^{M,N}$ is a weighted adjacency matrix. Each entry is defined by:

$$A_{(u,v),w} = \begin{cases} \frac{1}{C(e,Y)} & \text{if } u = w \\ -\frac{1}{C(e,Y)} & \text{if } v = w \\ 0 & \text{otherwise.} \end{cases}$$

and entries for $B \in R^{N,N}$ are given by:

$$B_{u,w} = \begin{cases} \sum\limits_{e \in In(u) \cup Out(u)} \frac{1}{C(e,Y)} & \text{if } u = w \\ -\frac{1}{C((u,w),Y)} & \text{if } (u, w) \text{ is an edge} \\ 0 & \text{otherwise.} \end{cases}$$

F^* just multiplies two matrices, thus is clearly differentiable. We also provide an explicit closed form expression for the gradient of F^* online [35]. We are now ready to define our final loss function.

Definition 5.1. Flow-Loss.

$$\text{Flow-Loss}(\hat{Y}, \mathbf{Y}) = \sum_{e \in E} C(e, \mathbf{Y}) \cdot F^*(\hat{Y})_e^2 \tag{10}$$

Notice the similarity to P-Cost (Equation 5). P-Cost computed the sum of the true edge costs of the path chosen by $P^*(\hat{Y})$, whereas Flow-Loss is a weighted sum of the true edge costs, where the weight of an edge is the square of the flow assigned to that edge, i.e., $F^*(\hat{Y})_e^2$. An intuitive interpretation of Flow-Loss is the energy dissipated in a circuit with currents $F^*(\hat{Y})$ passing resistances C(e, **Y**). Since $F^*(\cdot)$ and $C(\cdot)$ (Definition 4.5) are both differentiable, so is Flow-Loss, and we can use the chain rule to get the gradients of Flow-Loss w.r.t $\hat{Y}$.

Corollary 5.1. *Flow-Loss is minimized when $\hat{Y} = \mathbf{Y}$.*

[3]Recall that the energy dissipated when current I flows through a resistor with resistance R is RI^2 [1].

Proof. Note that $F^*(\mathbf{Y})$ assigns each edge, F_e s.t. $\sum_{e \in E} C(e, \mathbf{Y}) \cdot F_e^2$ is minimized (Equation 6). This is precisely the equation for Flow-Loss (Equation 10), since the costs in Flow-Loss, $C(e, \mathbf{Y})$, are computed using true cardinalities as well. Thus, setting $\hat{Y} = \mathbf{Y}$, is a (not unique) minimizer of Flow-Loss.

Moerkotte et al. [34] showed $\text{PC}(\hat{Y}, \mathbf{Y}) \leq q^4 \text{PC}(\mathbf{Y}, \mathbf{Y})$, where q is the largest Q-Error over all sub-plans. This loosely bounds how much worse can the plan using $\hat{Y}$ be than the plan using $\mathbf{Y}$ in terms of Q-Error. We prove a similar result for Flow-Loss.

Theorem 5.2.

$$PC(\hat{Y}, \mathbf{Y}) \leq k^2 \textit{Flow-Loss}(\hat{Y}, \mathbf{Y}) \tag{11}$$

$$\leq k^2 \frac{\textit{Flow-Loss}(\hat{Y}, \mathbf{Y})}{\textit{Flow-Loss}(\mathbf{Y}, \mathbf{Y})} PC(\mathbf{Y}, \mathbf{Y}) \tag{12}$$

where $k = \frac{1}{\min_{e \in P^*(\hat{Y})} F^*(\hat{Y})_e}$, i.e., inverse of the minimum flow on the path $P^*(\hat{Y})$.

Proof. Flow-Loss (Equation 10) sums over all edges; Consider only the terms summing over $P^*(\hat{Y})$, i.e.,

$$\sum_{e \in P^*(\hat{Y})} C(e, \mathbf{Y}) \cdot F^*(\hat{Y})_e^2. \tag{13}$$

This is a weighted version of $\text{PC}(\hat{Y}, \mathbf{Y})$. We defined k, such that the smallest weight is $\frac{1}{k^2}$. Thus multiplying Equation 13 by k^2 ensures that the coefficients of $C(e, Y)$ would be greater than 1, and Equation 11 follows. Equation 12 follows because we multiplied Equation 11 with a term greater than 1, since $\text{Flow-Loss}(\mathbf{Y}, \mathbf{Y}) \leq \text{PC}(\mathbf{Y}, \mathbf{Y})$ (to see this, notice that a potential solution for $F^*(\mathbf{Y})$ sets the flow of each edge in $P^*(\mathbf{Y})$ to 1, and rest to 0. This would make Flow-Loss $(\mathbf{Y}, \mathbf{Y}) = PC(\mathbf{Y}, \mathbf{Y})$. But, $F^*(\mathbf{Y})$ chooses the flow values to minimize Flow-Loss $(\mathbf{Y}, \mathbf{Y})$, thus it will be at least as small as $PC(\mathbf{Y}, \mathbf{Y})$) □

$\frac{\text{Flow-Loss}(\hat{Y}, \mathbf{Y})}{\text{Flow-Loss}(\mathbf{Y}, \mathbf{Y})}$ is typically much smaller than k. But k is hard to bound — and gets larger as the set of interesting paths increase. Empirically this seems to be at least as good as the Q-Error bound. But mostly, both these bounds provide intuition for why these are sensible loss functions, since other loss functions, such as mean squared error, provide no worst case guarantees whatsoever.

5.2 Discussion

Beyond left-deep plans. P-Cost, and therefore Flow-Loss, are defined over left-deep plans. Extending Flow-Loss to bushy plans is more challenging: we will need a graph similar to the plan graph, where every valid bushy plan is a path, but this will lead to an exponential increase in the number of paths. But it does not seem required to consider bushy plans explicitly when optimizing for cardinality estimates. First, the best left-deep plan often has reasonable performance compared to the best overall plan [23]. Second, every sub-plan in the query is required to find the best left-deep plan, therefore, the the same set of cardinality estimates are required to optimize both bushy plans and left-deep plans. Moreover, when indices are used, left-deep sub-plans are a prominent part of bushy plans. Hence, estimates that are important for choosing good left-deep plans are also important for bushy plans.

Anchoring. An unusual property of Flow-Loss compared to loss functions such as Q-Error is that it is not very sensitive to the

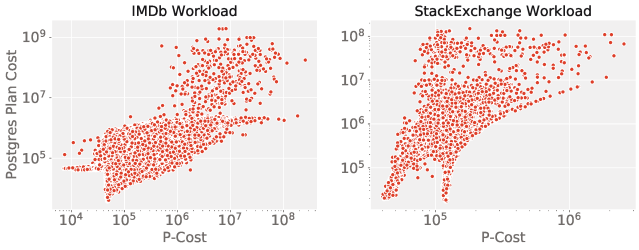


Figure 5: P-Cost versus PPC given true cardinalities for the two workloads we used.

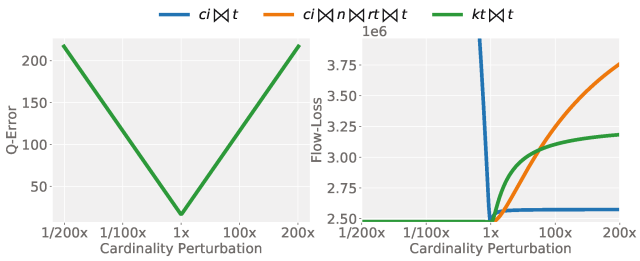


Figure 6: Comparing Q-Error (left) or Flow-Loss (right) as we vary the cardinality estimates of different sub-plans. For each data point we multiply or divide the true value (center) by 2.

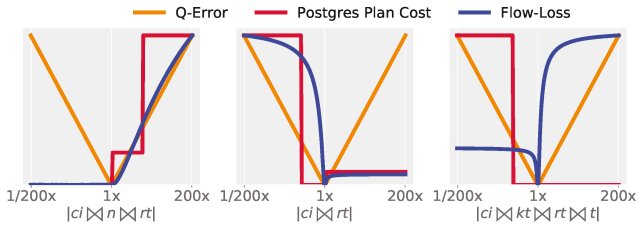


Figure 7: Comparing the shapes of Q-Error, PPC, and Flow-Loss as we vary estimate of one sub-plan, while keeping others fixed at their true values. Each loss curve is plotted with its own scale (not shown). For each data point we multiply or divide the true value (center) by 2.

absolute value of the cardinality estimates. Like an optimizer, Flow-Loss is affected more by the *relative* value of estimates for competing sub-plans. In particular, multiplying the cardinality estimates of all sub-plans of a query by a constant will often not change the cheapest path in the plan graph, because the costs computed using C (Definition 4.5) are linear in the cardinality estimates for most edges. The implication is that training a cardinality estimation model using Flow-Loss does not "anchor" the learned model's outputs to the true values (e.g., it may learn to estimate cardinalities that are all roughly 5× larger than the true values). It is possible to add explicit terms to the loss function that penalize large deviations from true values, or use a more precise cost model that is sensitive to absolute cardinalities.[4] Flow-Loss will optimize for whichever cost model we use. In our workloads we found that Flow-Loss performed well enough without explicit anchoring.

6 FLOW-LOSS ANALYSIS

In this section, we analyze the behavior of Flow-Loss using an example query on PostgreSQL to understand its potential benefits.

6.1 Cost Model

P-Cost and Flow-Loss were defined using the simple cost model C (Definition 4.5). However, our ultimate goal is to improve query performance of a DBMS.

Definition 6.1. Postgres Plan Cost (PPC). PPC is the same as P-Cost (Definition 4.7), but uses the PostgreSQL cost model and dynamic programming based exhaustive search over all plans — not only left-deep plans. To compute PPC, we inject $\hat{Y}$ into the PostgreSQL optimizer to get the cheapest plan (join order and physical operators) for $\hat{Y}$. Then we cost this plan using $\mathbf{Y}$. We implement it using a modified version of the plugin pg_hint_plan[5] [46].

Flow-Loss is an approximation to P-Cost, which in turn is an approximation to PPC. For Flow-Loss to be useful, its cost model C must broadly reflect the behavior of the PostgreSQL cost model. Figure 5 shows a scatter plot of P-Cost versus PPC given true cardinalities for two workloads described in Section 8. The PostgreSQL cost model includes many terms that we do not model, thus we would not expect the scale of P-Cost and PPC to match precisely. Nonetheless, we observe that PPC and P-Cost mostly follow the same trends. It matters less that P-Cost is not very precise, since we are merely using it as a signal to improve the cardinality estimates that lead to high costs. To optimize queries, these cardinality estimates will be provided to the PostgreSQL optimizer.

[4]For example, a cost model that accounts for spilling.
[5]https://github.com/parimarjan/pg_hint_plan

6.2 Shape of Loss Functions

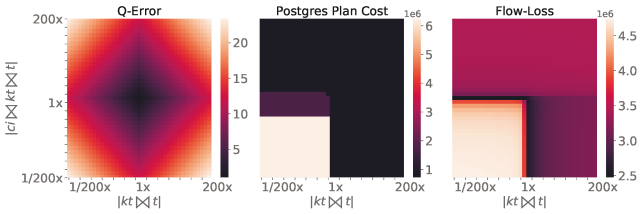


Figure 8: Comparing Q-Error, PPC, and Flow-Loss when we vary estimates of two sub-plans at the same time. The colors go from dark (low errors) to light (high errors).

Next, we will compare the behavior of Q-Error (Equation 1), PPC (Definition 6.1), and Flow-Loss using our running example, query Q_1 (Figure 3). Recall that Figure 4 shows the true cost of each edge, $C(e, \mathbf{Y})$. As we change the cardinality of one node (sub-plan), u, the estimated costs of outgoing edges from u will change, affecting the overall cost of any path (plan) that passes through u.

Flow-Loss is sensitive to underestimates of nodes on bad paths, and overestimates of nodes on good paths. Figure 6

shows three representative examples of how Q-Error and Flow-Loss change as we multiply or divide the cardinality of one node by increasing amounts while keeping the others fixed at their true values. Q-Error changes identically for all nodes (the lines overlap), but the behavior of Flow-Loss differs depending on the node. Node $ci \bowtie t$ has multiple expensive paths that go through it (note the red edges in Figure 4). As we underestimate its cardinality, Flow-Loss shoots up (blue line). This aligns with the intuition that underestimating this node makes bad paths appear cheaper, which may cause the optimizer to choose one of them instead of the actual cheapest path. Overestimating its cardinality, on the other hand, make bad paths appear even more expensive, which is good as we want the optimizer to avoid these paths. Thus, it is sensible that Flow-Loss stays near its minimum in this case. The node $ci \bowtie n \bowtie rt \bowtie t$ is on the cheapest path, while the node $kt \bowtie t$ has two relatively good paths passing through it (c.f. Figure 4). For these nodes, Flow-Loss remains at its minimum for underestimates (since it makes good paths appear cheaper), and shoots up for overestimates (since it makes good paths appear more expensive). Recall that Flow-Loss uses all relatively good paths, not just the cheapest, and therefore, it is impacted by both nodes.

Flow-Loss roughly tracks PPC decision boundaries. Figure 7 compares the shapes of Q-Error, PPC, and Flow-Loss as we vary the cardinality of a single node. Each curve is plotted on its own scale as we only want to compare their trends. Node $ci \bowtie n \bowtie rt$ is already on the cheapest path (cf. Figure 4), so Flow-Loss is only sensitive to overestimating its cardinality, like PPC. Node $ci \bowtie rt$ is not on the cheapest path, and like PPC, Flow-Loss is a lot more sensitive to underestimates as it causes flow to be diverted to the paths containing this node from potentially cheaper paths. Node $ci \bowtie kt \bowtie rt \bowtie t$ is an example of a case where Flow-Loss leads to a different behavior from PPC. For overestimates, PPC is flat at its minimum while Flow-Loss blows up. $ci \bowtie kt \bowtie rt \bowtie t$ is not on the cheapest path, but there are multiple nearly optimal paths using this node (cf. Figure 4). Since Flow-Loss routes a non-trivial amount of flow on such paths, it is sensitive to making them more expensive, even though the optimizer does not switch from the cheapest path (thus, PPC remains flat). This is a desirable property from the standpoint of *robustness*. It reflects the fact that any of the nearly optimal paths could become the cheapest path and get chosen by the optimizer if the cardinalities change slightly. For instance, although node $ci \bowtie kt \bowtie rt \bowtie t$ is not on the cheapest path when all edges are costed using true cardinalities, it would be on the cheapest path if we underestimate the cost of the $ci \bowtie kt \bowtie t \rightarrow ci \bowtie kt \bowtie rt \bowtie t$ edge (or overestimate the cost of the actual cheapest path). In that case, PPC would have been sensitive to increasing the cardinality of this node. By considering all good paths simultaneously, Flow-Loss robustly captures the behavior of the optimizer in response to such variations in cardinalities. As a further example, in Figure 8, we vary cardinalities of two sub-plans simultaneously. Once again we observe that Flow-Loss roughly reflects the behavior of PPC — it is highest when cardinalities for both the nodes are underestimated (lower left quadrant).

6.3 Benefits of Flow-Loss

In practice, cardinality estimation models face several challenges: limited model capacity (making it impossible to learn all the intricacies of the data distribution), limited training data (since collecting ground truth data is expensive), insufficient features (e.g., it may be hard to represent predicates on columns with a large number of categorical values), noisy training data, changing data (e.g., Wang et al. [50] show that learned models can have a steep drop in performance after data is updated), and changing query workloads. Thus, it is inevitable that such models will make mistakes. As the examples in §6.2 suggest, Flow-Loss guides the learning to focus on estimates that matter, and to improve their accuracy only to the extent necessary for improving query performance. This has several positive consequences as we highlight below.

Model capacity. Lower capacity models, or less expressive features, make it harder for learned models to achieve high accuracy. Flow-Loss helps use the limited model capacity in a way that maximizes the model's impact on query performance.

Domain-specific regularization. A model trying to minimize Q-Error treats each estimate as equally important, which makes it easy to overfit to the training data. Regularization is a general approach to mitigate overfitting and improve generalization, but generic regularization techniques such as weight decay [2] simply bias towards learning simpler models (e.g., smoother functions) without taking advantage of the problem structure. Flow-Loss provides a stronger, guided regularization by utilizing domain-specific knowledge about query optimization.[6] The key information is to know which details of the training data can be ignored without impacting query performance. If estimation errors on a subset of sub-plans do not typically cause worse plans, then there is no need to learn a more complex model to correct them. This is precisely what Flow-Loss does by allowing a high tolerance to cardinality estimation errors for noncritical sub-plans.

Tolerance to noisy training data. As a direct consequence of the previous point, by ignoring accuracy on less important subsets of the data, Flow-Loss can better handle noisy, or missing training data, which can let us avoid the expensive process of executing all sub-plans to generate the true cardinalities. Instead, we can train models using approximate cardinalities obtained via sampling [25].

7 FEATURIZATION AND MODELS

This section introduces the model architectures and featurization that we use to evaluate Flow-Loss.

Featurization. As described by Kipf et al. [18], a sub-plan q is mapped to three sets of input vectors: T_q, J_q, and P_q for the tables, joins, and predicates in the sub-plan. We augment these with a vector G_q that captures the properties of the sub-plan in the context of the plan graph. A one-hot vector encodes each table in the sub-plan (T_q), and a second one-hot vector encodes each join (J_q). For RANGE predicates, we use min-max normalization [18, 39]. For IN predicates we use feature hashing [43], in which categorical features with large alphabet sizes are hashed to N bins. Even if

[6] There are similar examples in other ML applications, e.g., Li et al. show domain-specific loss functions for physics applications lead to improved generalization via implicit regularization [26].

N is much smaller than the alphabet size, it still provides a signal for the learned models. For LIKE predicates feature hashing with character n-grams [51], and use additional features such as the number of characters and the presence of a digit. We find that $N = 10$ bins each for every column-operator pair works well on our workloads. As proposed by Dutt et al. [9], we add the cardinality estimate for each table (after applying its predicates) from PostgreSQL to that table's vector in T_q, which we found to be sufficient for our workload. For a stronger runtime signal, we could add *sample bitmaps* [18, 19] (i.e., bitmaps indicating qualifying sample tuples), however, as this would significantly increase the model's parameters, we omit this optimization. Similarly, we do not explicitly encode GROUP BY columns like earlier work does [17] and rely on PostgreSQL's estimates instead. G_q is a vector for the plan graph-based properties of a sub-plan. This includes information about the immediate children of the sub-plan node in the plan graph (i.e., the nodes obtained by joining the sub-plan with a base table). Specifically: the number of children, the cost using PostgreSQL's estimated cardinalities of the join producing that child, and the relative estimated cardinality of that child compared to the sub-plan. Intuitively, such information about neighboring plan graph nodes could be useful to generalize to new queries. For all cardinalities, we apply log transformation for training the models [9].

Models. To compare Q-Error and Flow-Loss, we train two representative neural network architectures with both loss functions. Fully-Connected Neural Network (FCNN) was used by Ortiz et al. [39] and Dutt et al. [9]. It takes as input a 1-D feature vector that concatenates the vectors in T_q, J_q, P_q, and G_q. Multi-Set Convolutional Network (MSCN) was proposed by Kipf et al. [18] based on the DeepSets architecture [58], and we extend it to include the G_q features as well. These are very different architectures, and represent important trade-offs — FCNN is a lightweight model that trains efficiently, but does not scale to increasing database sizes (number of parameters grow with the number of columns), while MSCN's set-based formulation is scalable but less efficient to train.

8 CARDINALITY ESTIMATION BENCHMARK (CEB)

Table 1: Comparing CEB with JOB.

Dataset	JOB (IMDb)	CEB (IMDb)	CEB (SE)
# Queries	113	13,644	3435
# Sub-plans	70K	3.5M	500K
# Templates	31	15	6
# Joins	5 – 16	5 – 15	5 – 8
# Optimal plans	88	2200	113

Benchmark. We create a tool to generate a large number of challenging queries based on predefined templates and rules. Using this tool, we generate the Cardinality Estimation Benchmark (CEB) [36], a workload on two different databases (IMDb [23] and StackExchange (SE) [44]) containing over $16K$ unique queries and true cardinalities for over $4M$ sub-plans including COUNT and GROUP BY aggregates, and RANGE, IN, and LIKE predicates. Table 1 summarizes the key properties of CEB, and contrasts them with Join Order Benchmark (JOB) [23]. Notice that for the $13K$ IMDb queries in CEB, there are over $2K$ unique plans generated by PostgreSQL with true cardinalities — showing that different predicates lead to a diverse collection of optimal query plans. CEB addresses the two major limitations of queries used in previous works [8, 18, 39]: First, past work on supervised cardinality estimation [8, 18, 39] evaluate on workloads with only up to six joins per query. CEB has much more complex queries ranging from five to sixteen joins. Second, while JOB [23] contains challenging queries with up to 16 joins, they only have two to five queries per template. This is insufficient training data for supervised learning methods. CEB contains hundreds of queries per hand-crafted template with real-world interpretations.

```
Example Template

[base sql]
SELECT COUNT(*)
FROM title AS t, kind_type AS kt, cast_info AS ci,
     role_type AS rt, name AS n
WHERE t.id = ci.movie_id AND t.kind_id = kt.id
  AND ci.person_id = n.id AND ci.role_id = rt.id
  AND t.production_year <= <YEAR>
  AND kt.kind IN <KIND>
  AND rt.role IN <ROLE>
  AND n.gender IN <GENDER>

[predicates]
name = year
dependencies = [ ]
keys = [YEAR]
columns = [t.production_year]
pred_type = <=
sampling_method = uniform
type = list
options = [1920, 1946, 1975, 2000, 2015]

[predicates]
name = kind_role_gender
dependencies = [year]
keys = [KIND, ROLE, GENDER]
columns = [kt.kind, rt.role, n.gender]
pred_type = IN
sampling_method = quantile
min_samples = 2
max_samples = 7
type = sql

SELECT
kt.kind, rt.role, n.gender, COUNT(*)
FROM title AS t, kind_type AS kt, cast_info AS ci,
     role_type AS rt, name AS n
WHERE t.id = ci.movie_id AND t.kind_id = kt.id
  AND ci.person_id = n.id AND ci.role_id = rt.id
  AND t.production_year <= <YEAR>
  GROUP BY kt.kind, rt.role, n.gender
  ORDER BY COUNT(*)
```

Figure 9: TOML configuration file for generating queries based on a predefined template and rules.

Query generator. Generating predicate values for query templates is challenging because predicates interact in complex ways, and sampling them independently would often lead to queries with zero or very few results. Our key insight is to generate interesting predicate values for multiple columns together, using predefined SQL queries that take into account correlations and other user specified conditions. Figure 9 shows a complete template which generates queries with the same structure as our running example, Q_1. We will walk through the process of generating a sample query following the rules specified in this template. [base sql] is the SQL query to be generated, with a few unspecified predicates to be filled in. [predicates] are rules to choose the predicates for groups columns. For the predicate YEAR we choose a value uniformly from the given list. We sample predicate values for the remaining three IN predicates together because KIND, ROLE, and GENDER are highly correlated columns. For these, we also add YEAR as a dependency — as the year chosen would influence predicate selectivities for all these columns. We generate a list of candidate triples using a GROUP BY query, and sample 2 to 7 values for each IN predicate.

Timeouts. Some sub-plans in the StackExchange queries time out when collecting the true values. This is due to unusual join graphs which make certain sub-plans behave like cross-joins (see online appendix [35]). In such cases, we use a large constant value in

place of the true cardinalities as the label for the timed out sub-plans in the training data. We verified that the plans generated by injecting all known true cardinalities and this constant value into PostgreSQL leads to almost 10× faster runtimes than using the default PostgreSQL estimates.

Approximate training data. Intuitively, we may not need precise cardinality estimates to get the best plans — thus, approximate query processing (AQP) techniques, such as *wander join* [25] or IBJS [24], should provide sufficient accuracy. However, we cannot use these techniques for query optimization because they are too slow to provide estimates for all sub-plans at runtime. But these techniques are much faster than generating the ground truth cardinality estimates for all sub-plans, which is the most expensive step in building a cardinality estimation model. We modify the wander join algorithm to efficiently generate all the cardinality estimates in a given workload (excluding LIKE / regex queries), with precise implementation details given in the online appendix [35]. We use this only as a proof of concept; our implementation is not optimized, and uses a mix of Python and SQL calls. Despite this, we generate the wander join estimates with speedups over generating ground truth data that range from 10× to 100× for different templates. For instance, for the largest template with around $3K$ sub-plans, generating all the ground truth data on a single core takes about 5 *hours*, while wander join estimates take less than 5 *minutes*. In Section 9.5, we explore if the wander join estimates are as good as true cardinalities to train learned models.

9 EXPERIMENTS

Setup. We use PostgreSQL 12 and MySQL 8 (with the MyISAM storage backend). We tune the configurations to reasonable settings, while disabling some optimizations like parallelism and materialization in both the DBMSs. The precise configurations, and code to reproduce the execution environment is provided online [36]. For the runtime experiments, we use Amazon EC2 instances with a NVMe SSD device, and 8GB RAM.

Loss functions. Our main focus is to compare the Q-Error and Flow-Loss loss functions to train the neural network models. We use the true cardinalities and estimates from PostgreSQL as baselines to compare against the learned models.

Training and test sets. We consider two scenarios:

(1) **Testing on seen templates.** The model is evaluated on new queries from the same templates that it was trained on. We put 20% of the queries of each template into the validation set (used to tune hyperparameters), and 40% each into the training and test sets. We report results from the test set.
(2) **Testing on unseen templates.** The model is evaluated on different templates than the ones it was trained on. We split the templates equally into training and test templates. Since the number of templates is much smaller than the number of queries, we use ten-fold cross-validation for these experiments: the training / test set splits are done randomly using ten different seeds (seeds = $1 - 10$). We use the same hyperparameters as determined in the seen templates scenario. Even though the templates are different in the second scenario, there is significant overlap with the training set on query *sub-plans*. This tests the robustness of these models to slight shifts in the workload.

Key result. Figure 10a shows the results of all approaches w.r.t. PPC on IMDb. All models outperform PostgreSQL's estimator significantly on seen templates. However, only the Flow-Loss trained models do so consistently on unseen templates as well. For seen templates, the models trained using Flow-Loss do better than the models trained using Q-Error on PPC. All models get worse when evaluated on unseen templates - but the Flow-Loss models degrade more gracefully. When the queries are from seen templates, the difference in PPC does not translate into runtime improvements (cf. Figure 10b). However, on unseen templates, we see clear improvements in runtime as well.

9.1 Testing on seen templates

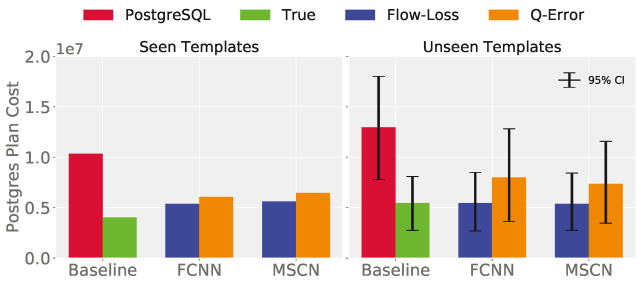
Worse Q-Error, better PPC, similar runtimes. We give detailed evaluations on the seen templates in the online appendix [35], but the key takeaway is that all learned cardinality estimation models do equally well and improve significantly over PostgreSQL estimates. The median Q-Error of the models trained using Flow-Loss was typically 2× worse than models trained using Q-Error, while being up to 10× worse at the $99th$ percentile. But, this is to be expected — our goal was to improve cardinality estimates only when it is important for query optimization. As seen in Figure 10, the Flow-Loss trained models improve mean PPC over the Q-Error models, getting close to the PPC with true cardinalities. This suggests that Flow-Loss models better utilize their model capacity to focus on sub-plans that are more crucial for PPC. It also shows that better Q-Error estimates do not directly translate into improved plans. However, in terms of runtimes, all models do equally well, and are very close to the performance of using true cardinalities.

9.2 Testing on unseen templates

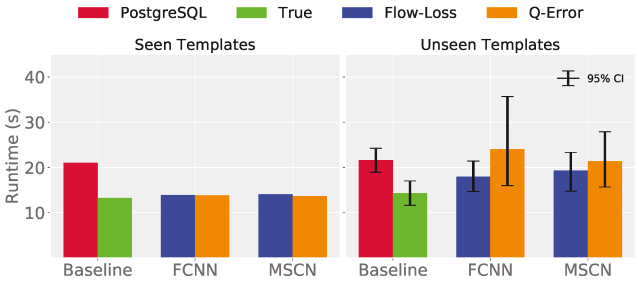
When we split the training set and test set by templates, each partition leads to very different information available to the models — therefore we will analyze the partitions individually.

Flow-Loss generalizes better. In Figure 11a, we look at the performance of a model trained with Flow-Loss compared to one trained with Q-Error w.r.t. query runtime. A single bar represents the same model architecture (FCNN or MSCN) trained and evaluated on one of the ten partitions in the unseen templates scenario. This figure highlights the overall trends across all unseen partition experiments: we see significant improvements on some partitions, relatively smaller regressions on some partitions, and similar performance on many partitions. This behavior is also reflected in the PPC trends.

Zooming in on partitions. For the FCNN and MSCN models, we sort all the partitions by the difference in the mean runtimes between the Flow-Loss and the Q-Error models. We select the best, median, and worst partition for Flow-Loss and show the $50p$, $90p$, and $99p$ for runtimes in Figure 12. For both architectures, the model trained with Flow-Loss significantly improves on the best partition , particularly at the tail — being up to 8×, and hundreds of seconds faster than the Q-Error model at the 99th percentile. On the worst partition, it is about 20 seconds slower than the Q-Error model at the $99th$ percentile. There are an additional six cases where the Flow-Loss models improve over the Q-Error models, with improvements in tens of seconds, which is comparable to the best improvement

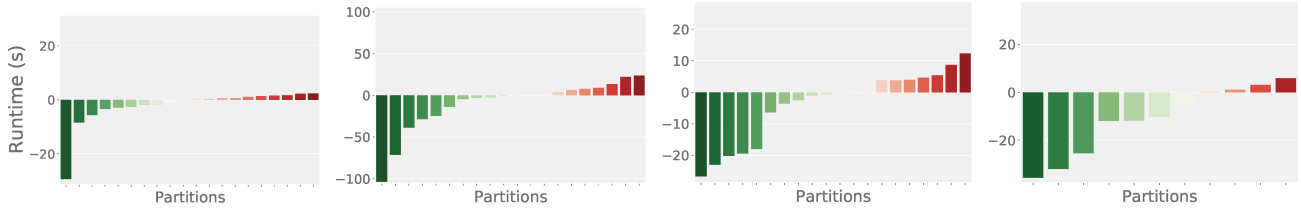


(a) Mean Postgres Plan Cost (PPC).



(b) Mean query runtimes.

Figure 10: Comparing performance of all models on seen versus unseen templates. For unseen templates, we do ten experiments using ten different training/test template splits.



(a) IMDb. Using PostgreSQL. FCNN or MSCN model.

(b) Same as 11a, but restricting PostgreSQL to 256MB RAM.

(c) IMDb. Using MySQL. FCNN or MSCN model.

(d) StackExchange. Using PostgreSQL. MSCN model.

Figure 11: Each bar shows the mean runtime improvement (green) or regression (red) of Flow-Loss over Q-Error on a single unseen partition using the same model architecture. *Lower is better for Flow-Loss models.*

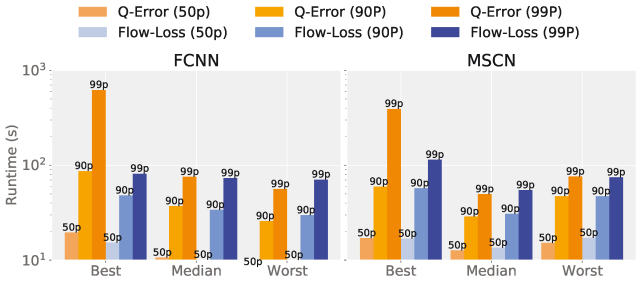


Figure 12: Best, median, and worst partition for Flow-Loss models by runtime difference from its corresponding Q-Error model on unseen template partitions for the FCNN architecture (left) and MSCN architecture (right) on the IMDb workload on PostgreSQL.

of the Q-Error model. As we highlight next, even these smaller improvements suggest more robust and better quality plans.

Restricting RAM. We re-execute the query plans from the unseen templates partitions after restricting PostgreSQL to a docker container with only 256MB RAM. The goal is to simulate a database significantly larger than available RAM, as is common in the real world. (The size of all tables in the IMDb database comes to about 5 GB). This scenario emphasizes the robustness of query plans in more challenging execution environments; bad plans that process a lot of unnecessary intermediate rows may cause more spills to disk, leading to disastrous performance. To reduce overhead due to the slower execution speeds, we re-execute a representative sample of 25% of the queries. Figure 11b plots the difference of the mean query runtime between the Q-Error and Flow-Loss models. Across multiple partitions, the Q-Error model leads to significant degradation of performance, having mean query runtime up to 3× slower in two cases. In one case, the mean query runtime difference between the Q-Error model and the Flow-Loss model goes from 3 seconds (w/o restrictions), to over 70 seconds after restricting RAM to 256MB. In the cases where the Q-Error model had done better, restricting to 256MB RAM, increases its relative improvement over the Flow-Loss model, but it only goes up to being 1.5×, and 20 seconds faster in the best case. Moreover, the Q-Error trained models also lead to a significantly larger number of timeouts. We use a 15 minute query time out (in the experiments using the full, 8GB RAM, no query times out). But in these restricted setting, in the worst case (for Q-Error), the Q-Error model has 59 timeouts vs. 6 for the Flow-Loss model; while in the best case (for Q-Error), it has 4 timeouts vs. 11 for the Flow-Loss model.

Join Order Benchmark. JOB is not suitable for training a supervised learning model as it has too few queries. But, we use it as an evaluation set for a model trained on CEB. This is similar to the unseen templates scenario: The JOB queries are less challenging in terms of PPC (for instance, PostgreSQL estimates have 20× lower mean PPC on JOB than CEB). However, they are more diverse: JOB has 31 templates, and includes predicates on columns not seen in CEB. Figure 13 summarizes the results of the Flow-Loss and Q-Error models for both architectures over three repeated runs.

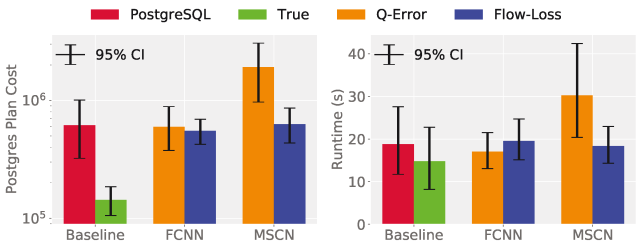


Figure 13: Mean PPC and runtimes for all models trained with Q-Error or Flow-Loss on CEB, and evaluated on JOB.

Both Flow-Loss models improve slightly on PPC over PostgreSQL while achieving similar runtimes. The FCNN model trained with Q-Error performs similarly, but the MSCN model trained with Q-Error shows much higher variance and does significantly worse. This experiment shows that even when the queries are very different, Flow-Loss-trained models avoid disastrously bad estimates.

9.3 MySQL DBMS

We conduct the same set of experiments seen so far using the MySQL DBMS instead of PostgreSQL to ensure our modeling assumptions, and Flow-Loss is not restricted to PostgreSQL.

New cost model to retrain Flow-Loss models. Flow-Loss relies on a differentiable approximation to the underlying cost model of the DBMS. For approximating Postgres Plan Cost, we had used the cost model in Definition 4.5. As it turns out, using the same cost model was not as good an approximation for MySQL Plan Cost. So instead, for the MySQL evaluations, we developed a cost model approximation tailored to MySQL to use for Flow-Loss.

Similar trends to PostgreSQL experiments. On the seen templates, both the loss functions perform equally well, and significantly improve on heuristic DBMS estimates. In Figure 11c, we show the results on the unseen templates using the MySQL DBMS. In general, the trends of the runtime improvements are comparable to the results from PostgreSQL, which we discussed in the previous section. These results show that using Flow-Loss is valuable across different DBMS', and that it can adapt to different cost models. At the same time, the dependence of Flow-Loss on the quality of the differentiable cost model approximation is a drawback — it requires additional work when using it in a new evaluation scenario.

9.4 StackExchange workload

Finally, we study the performance of the MSCN model on the StackExchange (SE) database using a smaller workload.

Similarities to IMDb results. On seen templates, learned models using both loss functions improve significantly over the PostgreSQL estimates — having query runtime that is over 100 seconds faster at the mean. On unseen templates, the Flow-Loss models improve significantly over the Q-Error models (cf. Figure 11d), with improvements of over 20 seconds at the mean on three of the partitions.

9.5 Analysis

Next, we show experiments on the IMDb workload to better understand when, and why the Flow-Loss models may improve performance. These experiments are guided by the intuition from §6.3.

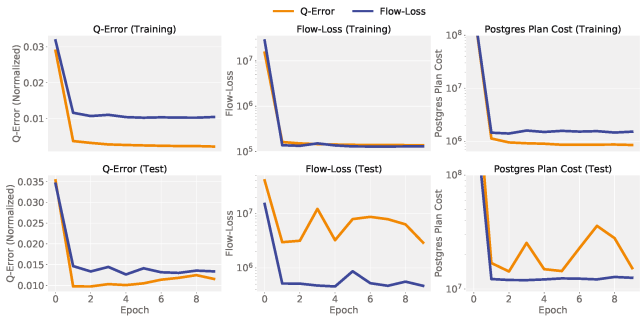


Figure 14: Learning curves for one unseen templates partition showing mean of all metrics for MSCN models.

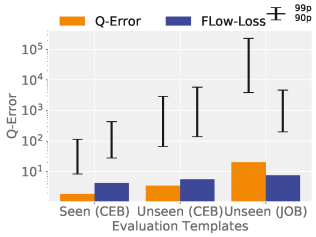


Figure 15: Median, 90p, and 99p Q-Error for FCNN models trained on queries from CEB and evaluated on seen templates, unseen templates, and JOB.

Learning curves. Figure 14 shows the MSCN model's learning curves for (normalized) Q-Error, Flow-Loss, and PPC on one partition (seed = 7) trained using Q-Error or Flow-Loss. We see that the Q-Error model has smooth training set curves for all metrics, but behaves erratically on the test set. This is because it is trying to minimize estimation accuracy, but for queries from unseen templates, it is much more challenging. Note that the Flow-Loss curves closely resemble the PPC curves (e.g., the Q-Error model on the test set). In §6.2, we showed simple examples where the Flow-Loss metric closely tracked the PPC. This shows that Flow-Loss can track the PPC well even in more complex scenarios. Notice that on the test set, in terms of Q-Error, the Flow-Loss model is consistently worse than the Q-Error model, while it is much better on the Flow-Loss and PPC metrics. This suggests the Flow-Loss model is using its capacity better to focus on PPC.

Domain specific regularization effect. Figure 15 shows the median, 90p, and 99p Q-Errors for the three scenarios we have looked at. We show results for the FCNN architecture and omit MSCN here, which shows similar trends. Our hypothesis has been that estimation accuracy on certain sub-plans are more important than others for query optimization. In the seen templates scenario, the Q-Error model has such high estimation accuracy everywhere (a Q-Error of only 100 at the 99p over millions of sub-plans), that it does reasonably well in terms of query optimization without explicitly caring about the important sub-plans. To achieve such low estimation errors, the model needs to get quite complex, and overfit to noisy patterns, like precise estimates for ILIKE predicates or for sub-plans with 10 tables that may anyway get pruned during

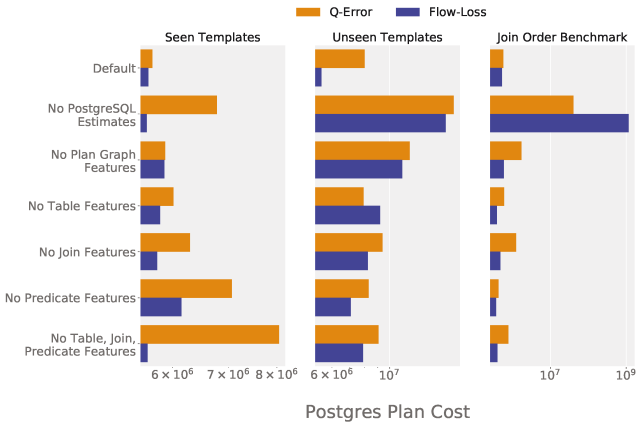


Figure 16: Ablation study with the FCNN model for seen templates (left), unseen templates (middle), and JOB (right) showing mean PPC when various components of the featurization scheme are removed.

the dynamic programming optimizer search. This can lead to more complex models, which may be brittle to workload changes. Often, we do not need such precise estimates everywhere. As we have seen from the results presented earlier, (e.g., Figure 10a), models trained with Flow-Loss achieve lower PPC despite higher Q-Error, which suggests that it learns a simpler model that does not try to improve cardinality estimates when they do not matter for query optimization. More strikingly, as we consider the unseen templates scenario — the models trained with Q-Error get almost 10× worse at the 90p and 99p, compared to the seen templates scenario. With larger errors, the effect of an estimate on query performance can be more important. As we saw in the results in §9.2, this drastic drop sometimes causes much worse query plans being generated. The estimation accuracy of models trained with Flow-Loss only get about 1.5× worse on unseen templates compared to their performance on the seen templates. This suggests that its estimates are less brittle to slight changes in the workload. This pattern continues on to the JOB templates — where the Flow-Loss models even have better estimation accuracy than the Q-Error models. This supports our regularization hypothesis (cf. §6.3), and shows that the Flow-Loss models can avoid overfitting in a way that does not harm its performance on PPC, but the simpler models help it generalize.

Ablation study. Next, we seek to understand the impact of the various components of the featurization (cf. §7) by an ablation study in which we remove key elements of the featurization, and evaluate the PPC on the seen templates, unseen templates, and JOB. We again focus on FCNN and omit MSCN, which follows similar trends. Figure 16 summarizes the results. There are two main highlights. First, on the seen templates, Flow-Loss models can adapt to removing various featurization components, and do as well as with the default features, meanwhile, the Q-Error models suffer significantly with worse featurization. This shows that when constrained with fewer resources, the Flow-Loss model can better use its capacity to minimize PPC. Second, PostgreSQL features are crucial for generalization. These include various cardinality and cost estimates (cf. §7). Both the models get significantly worse on

Figure 17: Mean PPC and runtimes, along with 90p to 99p error bars for a FCNN model trained with true or wander join cardinalities.

unseen templates without these features. This explains how the models do relatively well on different templates — these heuristic features have similar semantics across different kinds of queries.

Training with AQP estimates. Figure 17 shows the PPC and runtimes for models trained with true cardinalities and with wander join estimates for a subset of nine templates (that do not include LIKE predicates) from the CEB workload. The Flow-Loss model is robust when trained using the noisy wander join estimates — meanwhile, the Q-Error model trained using wander join estimates has a clear drop in performance at the tail for both PPC and runtimes. The same trend is observed for Q-Errors as well , which suggests that the model trained with Q-Error is overfitting to the noisy estimates and supports our hypothesis that the models trained using Flow-Loss are able to avoid overfitting to noisy data that may not be as relevant for query optimization (cf. §6.3).

Training Overhead. In terms of size and inference time, both Q-Error and Flow-Loss trained models have the same performance since they share exactly the same architecture. Compared to Q-Error, there is a 3 – 5× overhead for training either architecture with Flow-Loss due to the additional calculations needed for Flow-Loss— the bottleneck is computing B^{-1} in Equation 9.[7] On the CPU, when using Q-Error, the FCNN architecture trains for 10 epochs on the IMDb workload in under 1000 seconds, and the MSCN model takes up to 2500 seconds.

10 CONCLUSIONS

We showed a DBMS-based plan cost is a useful proxy to runtimes, and is an important alternative to Q-Error when evaluating a cardinality estimator. This lets us view cardinality estimation from a new lens — and we developed Flow-Loss as a smooth, differentiable approximation to plan cost that can be used to train models via gradient descent based learning techniques. Using a new Cardinality Estimation Benchmark, we provide evidence that Flow-Loss can guide learned models to better utilize their capacity to learn cardinalities that have the most impact on query performance. Even more importantly, it can help models avoid overfitting to cardinality estimates that are unlikely to improve query performance — leading to more robust generalization when evaluated on queries from templates not seen in the training data, and helping models learn more robustly from training data generated using AQP techniques.

[7] A long series of works [6, 21, 49] develop ways to approximate B^{-1} in linear time, utilizing the structure of the electric flows formulation. We also expect it to be faster on GPUs with fast matrix inverse operations [10].

REFERENCES

[1] Anant Agarwal and Jeffrey Lang. 2005. *Foundations of analog and digital electronic circuits.* Elsevier.

[2] Christopher M Bishop. 1995. Regularization and complexity control in feed-forward networks. (1995).

[3] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. 2001. STHoles: A Multidimensional Workload-Aware Histogram. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001.* 211–222. https://doi.org/10.1145/375663.375686

[4] Walter Cai, Magdalena Balazinska, and Dan Suciu. 2019. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In *Proceedings of the 2019 International Conference on Management of Data.* 18–35.

[5] Jeremy Chen, Yuqing Huang, Mushi Wang, Semih Salihoglu, and Ken Salem. 2021. Accurate Summary-based Cardinality Estimation Through the Lens of Cardinality Estimation Graphs. *arXiv preprint arXiv:2105.08878* (2021).

[6] Paul Christiano, Jonathan A Kelner, Aleksander Madry, Daniel A Spielman, and Shang-Hua Teng. 2011. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proceedings of the forty-third annual ACM symposium on Theory of computing.* 273–282.

[7] Peter G Doyle and J Laurie Snell. 1984. *Random walks and electric networks.* Vol. 22. American Mathematical Soc.

[8] Anshuman Dutt, Chi Wang, Vivek R. Narasayya, and Surajit Chaudhuri. 2020. Efficiently Approximating Selectivity Functions using Low Overhead Regression Models. *Proc. VLDB Endow.* 13, 11 (2020), 2215–2228. http://www.vldb.org/pvldb/vol13/p2215-dutt.pdf

[9] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek R. Narasayya, and Surajit Chaudhuri. 2019. Selectivity Estimation for Range Predicates using Lightweight Models. *PVLDB* 12, 9 (2019), 1044–1057. https://doi.org/10.14778/3329772.3329780

[10] Pablo Ezzatti, Enrique S. Quintana-Ortí, and Alfredo Remón. 2011. Using graphics processors to accelerate the computation of the matrix inverse. *J. Supercomput.* 58, 3 (2011), 429–437. https://doi.org/10.1007/s11227-011-0606-4

[11] Lise Getoor, Benjamin Taskar, and Daphne Koller. 2001. Selectivity Estimation using Probabilistic Models. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001.* 461–472. https://doi.org/10.1145/375663.375727

[12] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020.* 1035–1050. https://doi.org/10.1145/3318464.3389741

[13] Axel Hertzschuch, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2021. Simplicity Done Right for Join Ordering.. In *CIDR.*

[14] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *Proc. VLDB Endow.* 13, 7 (2020), 992–1005. http://www.vldb.org/pvldb/vol13/p992-hilprecht.pdf

[15] Oleg Ivanov and Sergey Bartunov. 2017. Adaptive cardinality estimation. *arXiv preprint arXiv:1711.08330* (2017).

[16] Martin Kiefer, Max Heimel, Sebastian Breß, and Volker Markl. 2017. Estimating Join Selectivities using Bandwidth-Optimized Kernel Density Models. *Proc. VLDB Endow.* 10, 13 (2017), 2085–2096. https://doi.org/10.14778/3151106.3151112

[17] Andreas Kipf, Michael Freitag, Dimitri Vorona, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2019. Estimating Filtered Group-By Queries is Hard: Deep Learning to the Rescue. *1st International Workshop on Applied AI for Database Systems and Applications* (2019).

[18] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. [n.d.]. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings.* http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf

[19] Andreas Kipf, Dimitri Vorona, Jonas Müller, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. [n.d.]. Estimating Cardinalities with Deep Sketches. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* 1937–1940. https://doi.org/10.1145/3299869.3320218

[20] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. *CoRR* abs/1808.03196 (2018). arXiv:1808.03196 http://arxiv.org/abs/1808.03196

[21] Rasmus Kyng, Yin Tat Lee, Richard Peng, Sushant Sachdeva, and Daniel A Spielman. 2016. Sparsified cholesky and multigrid solvers for connection laplacians. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing.* 842–850.

[22] Yin Tat Lee, Satish Rao, and Nikhil Srivastava. 2013. A new approach to computing maximum flows using electrical flows. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing.* 755–764.

[23] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[24] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings.* http://cidrdb.org/cidr2017/papers/p9-leis-cidr17.pdf

[25] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data.* 615–629.

[26] Li Li, Stephan Hoyer, Ryan Pederson, Ruoxi Sun, Ekin D Cubuk, Patrick Riley, and Kieron Burke. 2020. Kohn-Sham equations as regularizer: Building prior knowledge into machine-learned physics. *arXiv preprint arXiv:2009.08551* (2020).

[27] Wes Maciejewski. 2012. Resistance and relatedness on an evolutionary graph. *Journal of The Royal Society Interface* 9, 68 (2012), 511–517.

[28] Aleksander Madry. 2016. Computing maximum flow with augmenting electrical flows. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS).* IEEE, 593–602.

[29] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2020. Bao: Learning to Steer Query Optimizers. *CoRR* abs/2004.03814 (2020). arXiv:2004.03814 https://arxiv.org/abs/2004.03814

[30] Ryan Marcus and Olga Papaemmanouil. 2019. Towards a Hands-Free Query Optimizer through Deep Learning. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings.* http://cidrdb.org/cidr2019/papers/p96-marcus-cidr19.pdf

[31] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *PVLDB* 12, 11 (2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

[32] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. 1998. Wavelet-based histograms for selectivity estimation. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data.* 448–459.

[33] Brad H McRae, Brett G Dickson, Timothy H Keitt, and Viral B Shah. 2008. Using circuit theory to model connectivity in ecology, evolution, and conservation. *Ecology* 89, 10 (2008), 2712–2724.

[34] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *PVLDB* 2, 1 (2009), 982–993. https://doi.org/10.14778/1687627.1687738

[35] Parimarjan Negi. 2021. Flow-Loss Online Appendix. Retrieved July 27, 2021 from https://parimarjan.github.io/flow_loss_appendix [Online;].

[36] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Mao Hongzi, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Cardinality Estimation Benchmark. Retrieved July 27, 2021 from https://github.com/learnedsystems/ceb [Online;].

[37] Parimarjan Negi, Ryan Marcus, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2020. Cost-Guided Cardinality Estimation: Focus Where it Matters. In *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW).* IEEE, 154–157.

[38] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2018. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD 2018, Houston, TX, USA, June 15, 2018.* 4:1–4:4. https://doi.org/10.1145/3209889.3209890

[39] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2019. An Empirical Analysis of Deep Learning for Cardinality Estimation. *CoRR* abs/1905.06425 (2019). arXiv:1905.06425 http://arxiv.org/abs/1905.06425

[40] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. 2018. Quicksel: Quick selectivity learning with mixture models. *arXiv preprint arXiv:1812.10568* (2018).

[41] Matthew Perron, Zeyuan Shang, Tim Kraska, and Michael Stonebraker. 2019. How I Learned to Stop Worrying and Love Re-optimization. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019.* 1758–1761. https://doi.org/10.1109/ICDE.2019.00191

[42] Viswanath Poosala and Yannis E Ioannidis. 1997. Selectivity estimation without the attribute value independence assumption. In *VLDB*, Vol. 97. 486–495.

[43] Dipanjan (DJ) Sarkar. 2019. Categorical Data. https://towardsdatascience.com/understanding-feature-engineering-part-2-categorical-data-f54324193e63

[44] StackExchange. 2020. StackExchange Data Explorer. https://data.stackexchange.com/

[45] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *PVLDB* 13, 3 (2019), 307–319. https://doi.org/10.14778/3368289.3368296

[46] NIPPON TELEGRAPH and TELEPHONE CORPORATION. 2013. PG Hint Plan. https://pghintplan.osdn.jp/pg_hint_plan.html

[47] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* 1153–1170. https://doi.org/10.1145/3299869.3300088

[48] Kostas Tzoumas, Amol Deshpande, and Christian S. Jensen. 2013. Efficiently adapting graphical models for selectivity estimation. *VLDB J.* 22, 1 (2013), 3–27. https://doi.org/10.1007/s00778-012-0293-7

[49] Nisheeth K Vishnoi. 2012. Laplacian solvers and their algorithmic applications. *Theoretical Computer Science* 8, 1-2 (2012), 1–141.

[50] 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).

[51] John Wieting, Mohit Bansal, Kevin Gimpel, and Karen Livescu. 2016. Charagram: Embedding words and sentences via character n-grams. *arXiv preprint arXiv:1607.02789* (2016).

[52] Wikipedia. [n.d.]. Metric (Mathematics). Retrieved July 27, 2021 from https://en.wikipedia.org/wiki/Metric_(mathematics) [Online;].

[53] Wikipedia. [n.d.]. Pseudometric Space. [Online;].

[54] Lucas Woltmann, Claudio Hartmann, Maik Thiele, Dirk Habich, and Wolfgang Lehner. 2019. Cardinality estimation with local deep learning models. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2019, Amsterdam, The Netherlands, July 5, 2019*. 5:1–5:8. https://doi.org/10.1145/3329859.3329875

[55] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a Learning Optimizer for Shared Clouds. *PVLDB* 12, 3 (2018), 210–222. https://doi.org/10.14778/3291264.3291267

[56] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *CoRR* abs/2006.08109 (2020). arXiv:2006.08109 https://arxiv.org/abs/2006.08109

[57] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Peter Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *PVLDB* 13, 3 (2019), 279–292. https://doi.org/10.14778/3368289.3368294

[58] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. 2017. Deep sets. In *Advances in neural information processing systems*. 3391–3401.

[59] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. 1525–1539. https://doi.org/10.1145/3183713.3183739