Some Cardinality Estimates are More Equal than Others

by

Parimarjan Negi

B.S. in Computer Science, Stanford University (2018)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2022

Author . . . .
Department of Electrical Engineering and Computer Science
January 20, 2022

Certified by . . . .
Mohammad Alizadeh
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Some Cardinality Estimates are More Equal than Others

by

Parimarjan Negi

Submitted to the Department of Electrical Engineering and Computer Science
on January 20, 2022, in partial fulfillment of the
requirements for the degree of
Master of Science

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.

Acknowledgments

This work was only possible due to the support I received from my amazing co-authors: Mohammad Alizadeh, Tim Kraska, Ryan Marcus, Andreas Kipf, Nesime Tatbul, and Hongzi Mao.

I would like to thank my advisor, Mohammad, for constantly supporting me in whichever way was needed. He came up with the key mathematical formulation presented in this work, and in the process, helped me see how to approach modeling seemingly intractable problems. I would like to thank Tim for introducing me to the problem of understanding how cardinality estimation affects query plans, and guiding the vision presented in this work.

Ryan was instrumental in helping me with the endless low-level DBMS debugging and configuration that made the experiments in this paper possible. Andreas' experience building learned models for cardinality estimation was crucial in shaping our approach to the problem, and presenting it in context of the wider work. Nesime was a constant source of insight and advice about the experiments. Hongzi was a mentor to me in our lab, and I am very thankful for how he shared his experience in training neural network models to visualizing the results. And I would thank all of them for helping me refine my presentation, and giving me invaluable advice on improving it.

I would like to thank my lab mate, Lei Yang, who built an incredible system for managing AWS experiments, and I would never have finished the runtime experiments before deadlines if it was not for the ability to scale to several dozen EC2 machines using his scripts.

I would also like to thank Vikram Nathan, Jialin Ding, Prafull Sharma, and Kapil Vaidya for reading various drafts of this work, and providing extremely useful feedback.

Finally, I would thank all my other lab mates in CSAIL: Prateesh, Akshay, Frank, Venkat, Vibhaa, Arjun, and Inho for bearing with me complaining about this project.

Contents

1 Introduction **9**

2 Related Work **13**

3 Overview **15**

4 Flow-Loss **19**

4.1 Definitions 19

4.2 From Shortest Path to Electrical Flows 22

4.3 Bounding Plan-Cost in terms of Flow-Loss 25

4.4 Discussion 26

5 Flow-Loss Analysis **29**

5.1 Cost Model 29

5.2 Shape of Loss Functions 31

5.3 Benefits of Flow-Loss 33

6 Cardinality Estimation Benchmark (CEB) **37**

7 Experiments **41**

7.1 Setup 41

7.2 PostgreSQL Results 44

7.2.1 Testing on seen templates 44

7.2.2 Testing on unseen templates 46

7.2.3 Restricting RAM 49

7.2.4 Join Order Benchmark 50
7.2.5 StackExchange workload 51
7.3 MySQL DBMS 52
7.4 Analysis 53
7.4.1 Learning curves 53
7.4.2 Domain specific regularization effect 54
7.4.3 Ablation Study 55
7.4.4 Training with AQP estimates 56
7.4.5 Training overhead 57

8 Conclusions **59**

List of Figures

1-1 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. 10

3-1 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. 16

4-1 Join graph and optimal plan for sample query Q_1 on the IMDb database. 20

6-1 TOML configuration file for generating queries based on a predefined template and rules. 38

7-9 Ablation study with the FCNN model for seen templates (left), unseen templates (middle), and JOB (right) showing PPC when various components of the featurization scheme are removed. 55

7-10 Mean PPC and runtimes, along with $90p$ to $99p$ error bars for a FCNN model trained with true or wander join cardinalities. 56

List of Tables

6.1 Comparing CEB with JOB. 37

7.1 Models trained and evaluated on the same IMDb templates. We show $\pm 1 stddev$ for Q-Error and PPC from three repeated experiments, and execute plans from one random run. Xp refers to the Xth percentile. 46

1

Introduction

"All animals are equal, but some animals are more equal than others."

George Orwell, Animal Farm

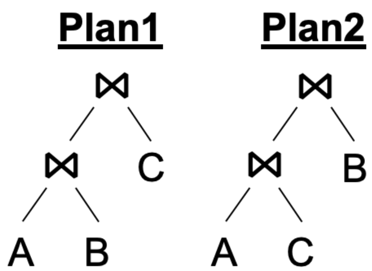
"A man who wants the truth becomes a scientist; a man who wants to give free play to his subjectivity may become a writer; but what should a man do who wants something in between?"
Robert Musil, The Man Without Qualities

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 to cardinality estimation have been successful in improving estimation accuracy [19, 53, 11, 15, 56], 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.

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). Therefore it is crucial to understand which errors are more acceptable for the optimizer. Unsupervised models learn from the data — but they will use model capacity for sub-plans that never occur since they treat every potential query as equally likely. Supervised models require representative

Query

SELECT *
FROM **A**, **B**, **C**
WHERE **A**.b1 = **B**.b1
AND **A**.c1 = **C**.c1



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

$$\text{Cost1} = (|A|+|B|) + (|A \bowtie B|+|C|)$$
$$\text{Cost2} = (|A|+|C|) + (|A \bowtie C|+|B|)$$

Figure 1-1: 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.

workloads, but learn more efficiently by focusing 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.

As a drop-in replacement for the well known Q-Error [35] loss function used to train supervised cardinality estimation models, we propose *Flow-Loss*, a loss function that explicitly emphasizes estimates that matter to query performance for a given workload. Flow-Loss takes the idea of focusing model capacity to its logical extreme — encouraging better estimates only if they improve resulting query plans. For instance, consider Figure 1-1: Estimator2 corrects Estimator1's estimate of $A \bowtie C$, but it actually leads to a worse plan (Plan 2), because the relative cardinalities ($A \bowtie B$ vs. $A \bowtie C$) are incorrect. A loss function using 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 gradient of the cost of a query plan w.r.t. the cardinality estimates used to generate the plan. To do this, we assume a simplified cost model and recast the dynamic programming (DP) algorithm for query optimization as a shortest path problem, which we approximate with a smooth and differentiable analytical objective function. This lets us use gradient descent based techniques to improve the estimates that are most relevant to improving the query plans. We show that improving cardinalities w.r.t. this objective also improves the quality of plans of the complex PostgreSQL cost model and optimizer.

There are two main benefits of training models to minimize Flow-Loss. First, similar to how attention-based models in natural language processing [49] treat certain parts of the input as more important than others, 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 in an ablation study that models trained with Flow-Loss can adapt to removing various components of the featurization scheme, and still do equally well. Meanwhile, ablations cause the Q-Error models to get up to $2\times$ worse w.r.t. PostgreSQL costs. 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, while the Flow-Loss trained models are not much worse at the tail in any of the experiments. These correspond to query runtime improvements of up to $1.5\times$ at the mean, which goes up to over $3\times$ if we restrict PostgreSQL to use $256MB$ RAM.

Our key contributions are:

- **DBMS-based Plan Cost.** Based on Moerkotte et al.'s [35] 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) [24], and has more complex queries.
- **Training with AQP estimates.** A challenge for using supervised cardinality estimation models in practice is that collecting ground truth data is expensive. However, precise estimates are not needed for near-optimal plans. We show that almost equally good query plans can be generated using models trained with Flow-Loss on data collected using approximate query processing (AQP), which is $10-100\times$ faster than computing true values in our implementation. Since Q-Error tends to overfit, it is less robust to noisy training data generated via AQP. In fact, when using AQP training data, there is a clear degradation at the $99th$ percentile on all metrics for models trained with Q-Error, being 2× worse PostgreSQL costs, and 30% slower on runtimes than the models trained with Flow-Loss.

2

Related Work

For cardinality estimation, traditional approaches have used histograms [6], sampling [25], kernel density estimation [17], wavelets [33], or singular value decomposition [42]. Recently, machine learning approaches have shown high estimation accuracy. Many works focus on single-table selectivity estimates [40, 56, 14, 11], 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 [13, 48], Sum-Product Networks [15], or deep autoregressive models [55]. NeuroCard [55] 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. [19], which uses a single deep learning model for the whole workload. While several supervised learning-based works report improved estimation accuracy [38, 19, 53, 54, 11, 10], only a few actually demonstrate improved query performance [16, 39, 37]. 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 [31, 32, 21], learning to use the optimizer's hints [30], learning the cost model [45], re-optimization [41, 47], and bounding worst case cardinalities to avoid bad plans [7].

3

Overview

"The path of least resistance is the path of the loser."

H.G. Wells, "The New Machiavelli"

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 3.0.1. Q-Error.

$$\text{Q-Error}(y^{true}, y^{est}) = \max(\frac{y^{true}}{y^{est}}, \frac{y^{est}}{y^{true}}). \tag{3.1}$$

Such a loss function treats every estimate as equally important. Instead, we want a loss function that will focus model capacity on 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 3-1: *(i)* a *cost model*, which outputs a cost for

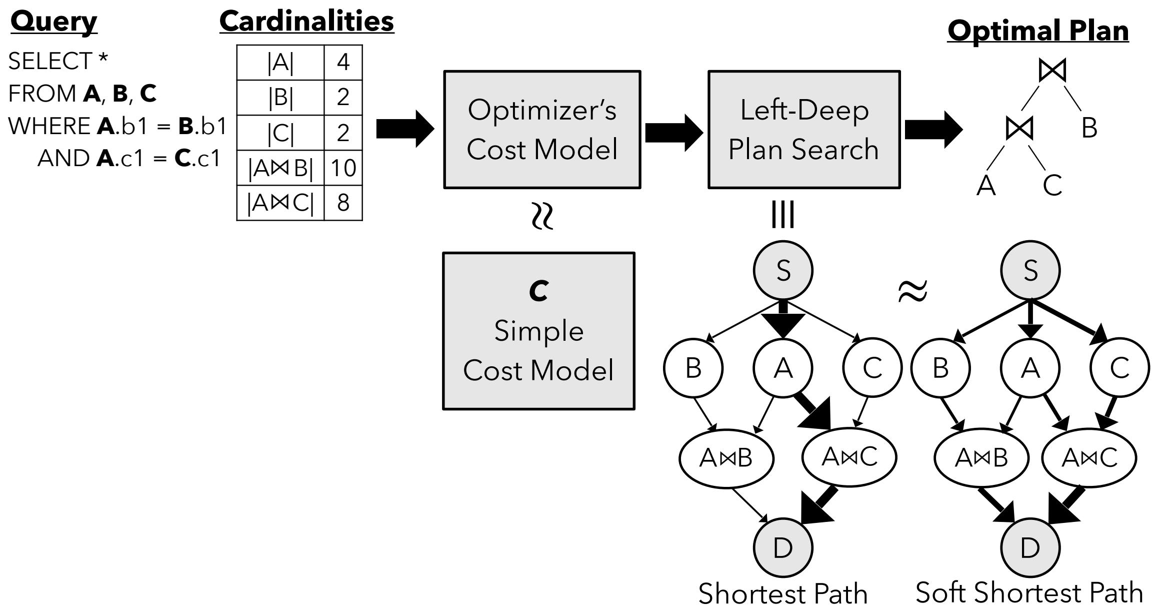


Figure 3-1: The query optimization process has two non-differentiable components: the cost model and the plan search algorithm. We develop differentiable approximations for these so we can understand how sensitive query plans are to changes in cardinality estimates.

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:

$$\hat{Y} \xrightarrow{C(\cdot)} \text{Join-Cost} \xrightarrow{S(\cdot)} \text{Plan}. \tag{3.2}$$

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 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 very simple approximation with terms to cost joins with or without indexes works well in our workloads (Definition 4.1.5).

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 3-1 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 two special nodes, 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

[1]Electrical flows have been used for graph algorithms in various fields: modeling random walks [9], developing more efficient algorithms for approximating maximum flow problem [8, 23, 29], modeling landscape connectivity in ecology [34], and inferring relatedness in evolutionary graphs in biology [28].

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

Flow-Loss

"Could fulfillment ever be felt as deeply as loss?"

Kiran Desai, The Inheritance of Loss

"Lost in the solitude of his immense power, he began to lose direction."

Gabriel García Márquez, One Hundred Years of Solitude

4.1 Definitions

This section formally defines the plan graph and the concepts we use to develop our new loss function, Flow-Loss. As a running example, we will consider the query Q_1 (Figure 4-1) 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 4.1.1. *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 4-1), $kt \bowtie t$ is a sub-plan but $kt \bowtie ci$ is not.

Definition 4.1.2. *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,

Query

```
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 kt.kind IN ('movie') AND rt.role IN ('actor', 'director')
    AND n.gender IN ('f') AND t.production_year <= 2015
```

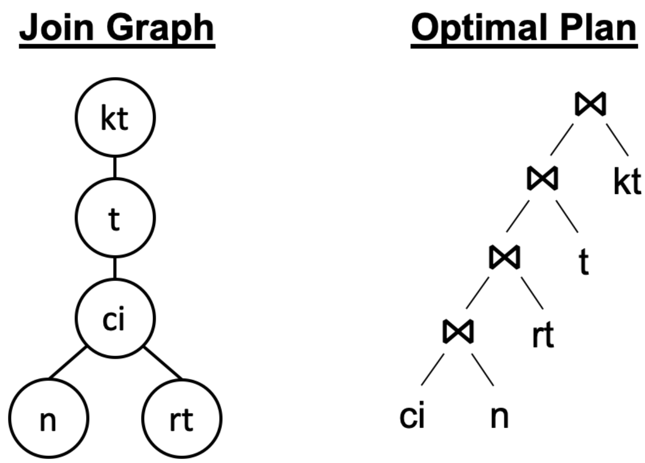


Figure 4-1: Join graph and optimal plan for sample query Q_1 on the IMDb database.

which has an edge to all nodes containing exactly one table. We use D to denote the node consisting of all tables. Figure 4-2 shows the plan graph for query Q_1.

Definition 4.1.3. *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 4-2.

Definition 4.1.4. *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 4.1.5. $C(e, Y)$. 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. In this paper , to

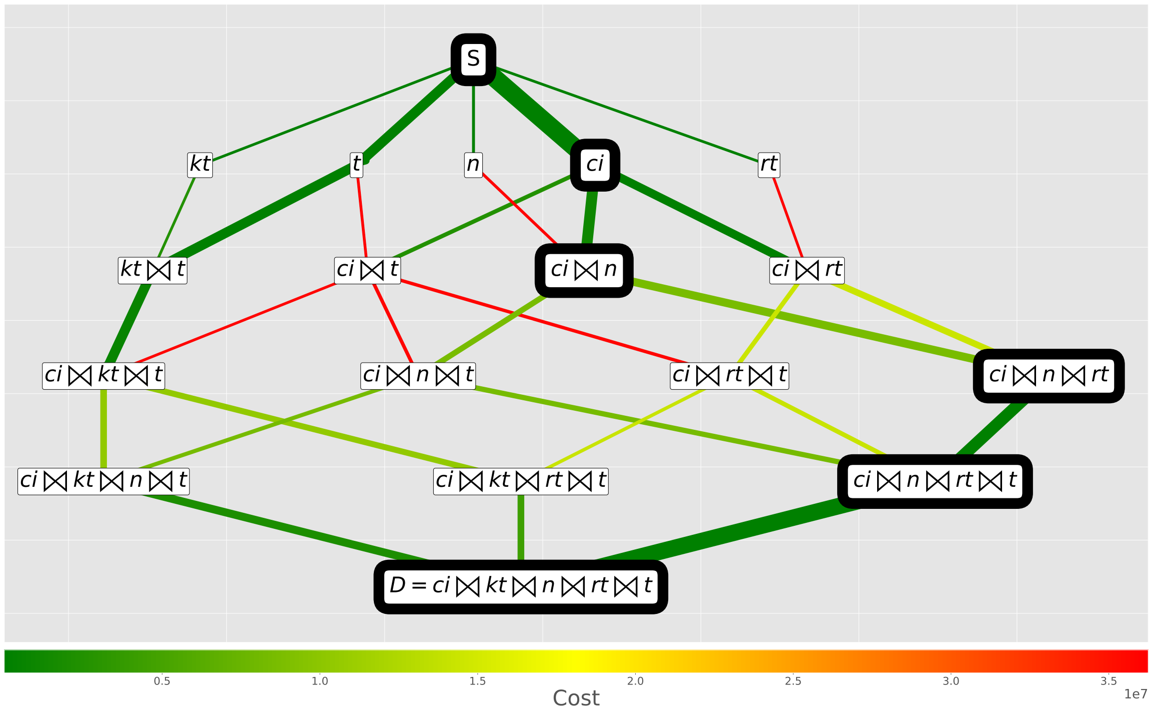


Figure 4-2: Plan graph (Definition 4.1.2) for query Q_1. The cheapest path, P-Opt($\mathbf{Y}$), is highlighted. The edges are colored according to $C(e, \mathbf{Y})$. The relative thickness of the edges represent the flows computed by Equation 4.4, F-Opt($\mathbf{Y}$).

approximate PostgreSQL, we use the following simple cost model:

$$C((u, v), Y) = \min(|u| + \lambda|b|, |u| \cdot |b|) \tag{4.1}$$

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 4-2 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 is effective in our workloads. §5.1 analyzes how well it approximates the PostgreSQL cost model.

Definition 4.1.6. P^* (Y). The cheapest path (plan) in the plan graph with edge costs given by $C(e, Y)$:

$$P^*(Y) = \arg\min_P \sum_{e \in P} C(e, Y). \tag{4.2}$$

For example, given $\mathbf{Y}$, the cheapest path $P^*(\mathbf{Y})$ is highlighted in Figure 4-2. We will use the terms "cheapest" and "shortest" path interchangeably.

Definition 4.1.7. P-Cost, $PC(\hat{Y}, \mathbf{Y})$. The true cost of the optimal path (plan) chosen based on cardinality vector $\hat{Y}$:

$$PC(\hat{Y}, \mathbf{Y}) = \sum_{e \in P^*(\hat{Y})} C(e, \mathbf{Y}). \tag{4.3}$$

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 $\mathbf{Y}$. Note that for a fixed $\mathbf{Y}$, P-Cost takes its lowest value when $\hat{Y} = \mathbf{Y}$.

Remark. As defined, P-Cost is not a distance metric [1] (e.g., it does not satisfy the symmetry property). However, this does not affect its use in our loss function. In an online appendix [3], we use PC to construct a pseudometric [2] that computes a distance between two cardinality vectors.

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 $\hat{Y}$ does unlikely 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 be affected by a small perturbation to $\hat{Y}$. 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 our loss function, Flow-Loss.

4.2 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[1] (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 precise way in which current flows in the circuit can be obtained by solving the following *energy minimization*[2] problem:

$$F^*(Y) = \arg\min_F \sum_{e \in E} C(e, Y) \cdot F_e^2 \tag{4.4}$$

$$\text{s.t} \sum_{e \in Out(S)} F_e = \sum_{e \in In(D)} F_e = 1 \tag{4.5}$$

$$\sum_{e \in Out(V)} F_e = \sum_{e \in In(V)} F_e \tag{4.6}$$

Here the optimization variable F assigns a flow of current to each edge. Equation (4.5) enforces that one unit of flow is sent from S to D. Equation (4.6) 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 4-2 show the flows assigned to each edge by $F^*(\mathbf{Y})$.

Computing F^* is a classical problem in circuit design [4, 8], 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:

$$F^*(Y) = AL^{-1}i, \tag{4.7}$$

where $i \in R^N$ is the constant vector of $[1, 0, ..., -1]$; $A \in R^{M,N}$ is a weighted

[1] 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.

[2] Recall that the energy dissipated when current I flows through a resistor with resistance R is RI^2 [4].

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 $L \in R^{N,N}$ is known as the weighted *Laplacian* of a graph [9, 29], with its entries given by:

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

F^* just multiplies two matrices, thus is clearly differentiable. We also provide an explicit closed form expression for the gradient of F^* online [3]. We are now ready to define our final loss function.

Definition 4.2.1. Flow-Loss.

$$\text{Flow-Loss}(\hat{Y}, \mathbf{Y}) = \sum_{e \in E} C(e, \mathbf{Y}) \cdot F^*(\hat{Y})_e^2 \tag{4.8}$$

Notice the similarity to P-Cost (Equation 4.3). 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 by Equation 4.4, i.e., $F^*(\hat{Y})_e$. An alternative, intuitive interpretation of Flow-Loss is the true "energy dissipation" of the flows $F^*(\hat{Y})$. Since $F^*(\cdot)$ and $C(\cdot)$ (Definition 4.1.5) are both differentiable, so is Flow-Loss, and we can use the chain rule to get the gradients of Flow-Loss w.r.t $\hat{Y}$.

Claim: Flow-Loss is minimized when $\hat{Y} = \mathbf{Y}$.

Proof: Note that $F^*(\mathbf{Y})$ assigns each flow edge, F_e s.t. $\sum_{e \in E} C(e, \mathbf{Y}) \cdot F_e^2$ is minimized (Equation 4.4). This is precisely the equation for Flow-Loss (Equation 4.8), since the costs, $C(e, \mathbf{Y})$, are computed using true cardinalities as well. Thus, setting $\hat{Y} = \mathbf{Y}$, i.e., choosing flows according to the true cardinalities, is one (not unique) minimizer of Flow-Loss.

Similarly, F^* $(\hat{Y})$ will be choosing flows to minimize $\sum_{e \in E} C(e, \hat{Y}) \cdot F_e^2$; since the costs, $C(e, \hat{Y})$ may be arbitrarily different from the true costs, $C(e, \mathbf{Y})$ used in Flow-Loss, the generated flows may have much higher Flow-Loss.

4.3 Bounding Plan-Cost in terms of Flow-Loss

Moerkotte et al. [35] 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 1.

$$\text{PC}(\hat{Y}, \mathbf{Y}) \leq k^2 \text{Flow-Loss}(\hat{Y}, \mathbf{Y}) \tag{4.9}$$

$$\leq k^2 \frac{\text{Flow-Loss}(\hat{Y}, \mathbf{Y})}{\text{Flow-Loss}(\mathbf{Y}, \mathbf{Y})} \text{PC}(\mathbf{Y}, \mathbf{Y}) \tag{4.10}$$

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 4.8) sums over all edges; Consider only the terms summing over $P^*(\hat{Y})$, i.e.,

$$\sum_{e \in P^*(\hat{Y})} C(e, \mathbf{Y}) \cdot F^*(\hat{Y})_e^2. \tag{4.11}$$

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 4.11 by k^2 ensures that the coefficients of $C(e, Y)$ would be greater than 1, and Equation 4.9 follows. Equation 4.10 follows because we multiplied Equation 4.9 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 $PC(\mathbf{Y}, \mathbf{Y}))$ $\square$

$\frac{\text{Flow-Loss}(\hat{Y}, \mathbf{Y})}{\text{Flow-Loss}(\mathbf{Y}, \mathbf{Y})}$ 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.

4.4 Discussion

Beyond left-deep plans. P-Cost, and by extension, Flow-Loss are defined over left-deep plans. Extending Flow-Loss to bushy plans is more challenging: we will need to define 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. Fortunately, 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 [24]. Second, every sub-plan in the query is required to find the best left-deep plan, therefore, the set of cardinality estimates required to optimize bushy plans are also needed for left-deep plans. In particular, 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 *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 4.1.5) are linear in the cardinality estimates for most edges (specifically, the edges corresponding to joins that are cheaper with an index). 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 $5\times$ 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.[3] Flow-Loss will optimize for whichever cost model we use. However, in our workloads we found that even without explicit anchoring, Flow-Loss learns cardinalities that perform well with PostgreSQL.

[3] For example, a cost model that accounts for spilling.

5

Flow-Loss Analysis

"I cannot make you understand. I cannot make anyone understand what is happening inside me. I cannot even explain it to myself."
Franz Kafka, The Metamorphosis

The goal of Flow-Loss is to learn cardinality estimation models that improve query performance of a DBMS. As a concrete example, we focus specifically on improving PostgreSQL. In this section, we analyze the behavior of Flow-Loss using examples on PostgreSQL to understand how it improves on traditional loss functions like Q-Error.

5.1 Cost Model

P-Cost and Flow-Loss were defined using the simple cost model C (Definition 4.1.5). However, our ultimate goal is to improve query performance of PostgreSQL, which we quantify using the actual PostgreSQL cost model.

Definition 5.1.1. Postgres Plan Cost (PPC). PPC is the same as P-Cost (Definition 4.1.7), but uses the PostgreSQL cost model and PostgreSQL's dynamic programming implementation of 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

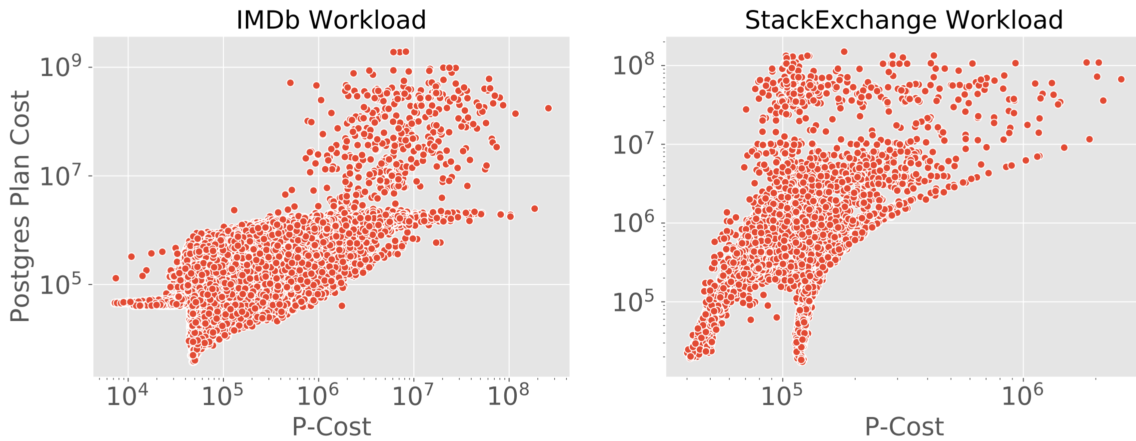


Figure 5-1: P-Cost versus PPC given true cardinalities for the two workloads we used.

implement it using a modified version of the plugin pg_hint_plan[1] [46]. We disable materialization and parallelism in PostgreSQL as they add complexity which makes it harder to analyze. Similarly, we can use the DBMS MySQL to define MySQL Plan Cost. We modify the open source MySQL code to implement the necessary functionality of injecting cardinality estimates for query optimization[2].

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 5-1 shows a scatter plot of P-Cost versus PPC given true cardinalities for two workloads described in Section 6. 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 with its full cost model.

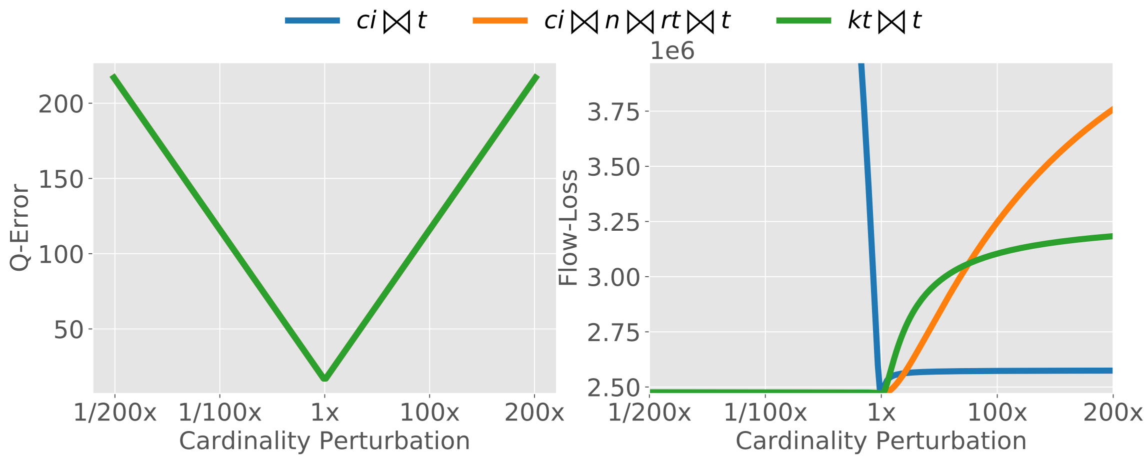


Figure 5-2: Comparing Q-Error (left) or Flow-Loss (right) as we vary the cardinality estimates of different sub-plans. For each data point we multiply or divide the true value (center) by 2.

5.2 Shape of Loss Functions

Next, we will compare the behavior of Q-Error (Definition 3.0.1), PPC (Definition 5.1.1), and Flow-Loss using our running example, query Q_1 (Figure 4-1). Recall that Figure 4-2 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 under-estimates of nodes on bad paths, and over-estimates of nodes on good paths. Figure $5-2$ 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 4-2). As we under-estimate its cardinality, Flow-Loss shoots up (blue line). This aligns with the intuition that under-estimating this node makes bad paths appear cheaper, which may cause the optimizer to choose one of them instead of the actual cheapest path. Over-estimating

[1]https://github.com/parimarjan/pg_hint_plan
[2]https://github.com/parimarjan/mysql-server

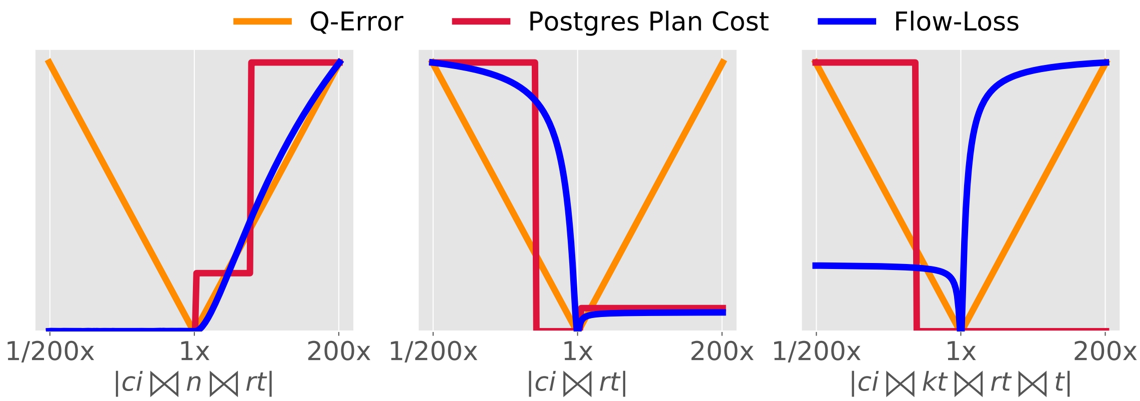


Figure 5-3: Comparing the shapes of Q-Error, PPC, and Flow-Loss as we vary estimate of one sub-plan, while keeping others fixed at their true values. Each loss curve is plotted with its own scale (not shown). For each data point we multiply or divide the true value (center) by 2.

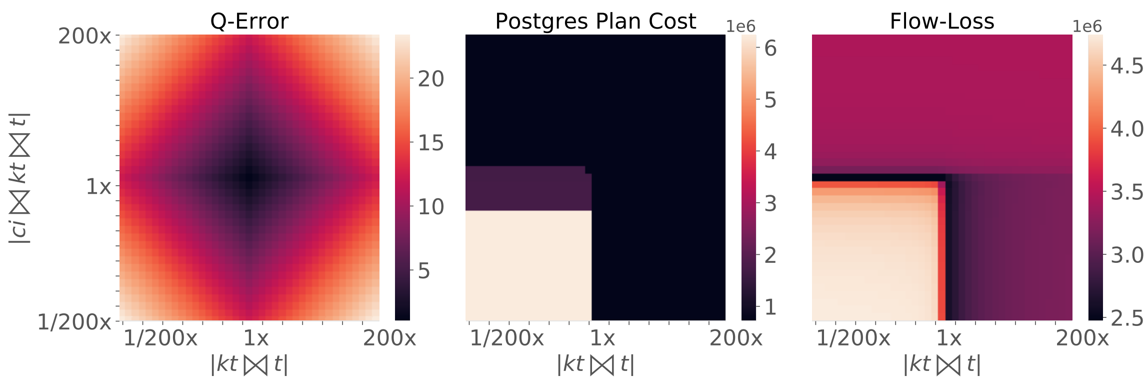


Figure 5-4: Comparing Q-Error, PPC, and Flow-Loss when we vary estimates of two sub-plans at the same time. The colors go from dark (low errors) to light (high errors).

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 4-2). For these nodes, Flow-Loss remains at its minimum for under-estimates (since it makes good paths appear cheaper), and shoots up for over-estimates (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 5-3 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 are only interested in comparing their behavioral trends. Node $ci \bowtie n \bowtie rt$ is already on the cheapest path (cf. Figure 4-2), so Flow-Loss is only sensitive to over-estimating 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 under-estimates 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 4-2). 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 cost 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 more robustly captures the behavior of the optimizer in response to such variations in cardinalities. As a further example, in Figure 5-4, 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 under-estimated (lower left quadrant in the figures).

5.3 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. [51] 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 §5.2 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 utilizing 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 [5] simply bias towards learning simpler models (e.g., smoother functions) without taking advantage of the problem structure. Flow-Loss provides a stronger, guided regularization effect by utilizing domain-specific knowledge about query optimization.[3] 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 many 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 well-performing models using approximate cardinalities obtained via fast sampling

[3] 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 [27].

techniques [26].

6

Cardinality Estimation Benchmark (CEB)

Table 6.1: 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 [24] 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 6.1 summarizes the key properties of CEB, and contrasts them with Join Order Benchmark (JOB) [24]. 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

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 6-1: TOML configuration file for generating queries based on a predefined template and rules.

queries used in previous works [19, 39, 10]: First, past work on supervised cardinality estimation [19, 10, 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 [24] 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.

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 one, or multiple columns together, using predefined SQL queries that take into account correlations and other user specified conditions. Figure 6-1 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 of one or more columns. The predicate YEAR is of type less than or equal to, and we choose a value uniformly from the given list. We sample filter 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. From this list, we 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 [3]). 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 $10\times$ faster runtimes than using the default PostgreSQL estimates. Thus, despite the timeouts, our labels for StackExchange are a good target for training a cardinality estimation model.

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* [26] or IBJS [25], 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 by far the most expensive step in building a cardinality estimation model. Therefore, we propose that training models on data generated by AQP techniques may be an important component of a practical learned cardinality estimation system. 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 [3]. We use this only as a proof of concept; our implementation is not optimized, and uses a mix of Python and SQL calls to do the random walks in wander join. 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* on average, and give almost equally good plans. In Section 7.4, we explore if the wander join estimates are as good as true cardinalities to train learned models.[1]

[1] This learning problem is similar to how obfuscation by adding random noise to training data is used to learn ML models while preserving privacy [58].

7

Experiments

"People almost invariably arrive at their beliefs not on the basis of proof but on the basis of what they find attractive."
Blaise Pascal, De l'art de persuader

"The inability to predict outliers implies the inability to predict the course of history."
Nassim Nicholas Taleb, The Black Swan

"Insanity is doing the same thing over and over again and expecting different results."
Narcotics Anonymous, 1981

7.1 Setup

This section introduces the evaluation setup, including the model architectures, featurization scheme, loss functions, and the DBMS setup. The implementations of these models are available online[1].

Featurization. As described by Kipf et al. [19], 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

[1] github.com/parimarjan/learned-cardinalities

(T_q), and a second one-hot vector encodes each join (J_q). For RANGE predicates, we use min-max normalization [19, 39]. For IN predicates we use feature hashing [43]. This is a standard technique in ML applications where categorical features with large alphabet sizes are hashed to N bins. Even if N is much smaller than the alphabet size, it still provides a signal for the learned models. For LIKE predicates we use feature hashing with character n-grams [52]. This is useful to distinguish between extremely common and uncommon characters. For LIKE, we also include the number of characters and the presence of a digit as additional features. We find that $N = 10$ bins each for every column-operator pair works well on our workloads. As proposed by Dutt et al. [11], 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* [19, 20] (i.e., bitmaps indicating qualifying sample tuples), however, as this would significantly increase the model's parameters and hence increase memory requirements, we omit this optimization in this work. Similarly, we do not explicitly encode GROUP BY columns like earlier work does [18] 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 PostgreSQL's 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. We also add the PostgreSQL cardinality and cost estimate for the sub-plan to G_q. For all cardinalities, we apply log transformation [11].

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. [11]. 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. [19] based on the DeepSets architecture [57], 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 neural network weights grow with the number of columns), while MSCN uses a set-based formulation that is scalable but is less efficient to train.

Setup. We use PostgreSQL 12 for the PostgreSQL experiments, 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 (m5ad.large for IMDb, and m5ad.xlarge for StackExchange).

Evaluation metrics. To evaluate our experiments we use Q-Error (Definition 3.0.1), Postgres Plan Cost, (PPC, Definition 4.1.7), and actual query runtimes. Q-Errors are computed per sub-plan, while PPC and runtimes are computed per query. PPC is in the cost model units - thus, the absolute values are significantly larger. These do not exactly translate into runtimes due to the inconsistencies between the cost model and the reality. But, we find that significantly large differences in Postgres Plan Error are reflected in the runtimes.

Loss functions. Our main focus is to compare the Q-Error and Flow-Loss loss functions to train the neural network models. In the online appendix [3], we also compare with Prioritized Q-Error [37], which was our exploratory earlier work in tweaking the Q-Error loss function to focus on queries which have high PPC. We use the true cardinalities and estimates from PostgreSQL as baselines to compare against the learned models.

Baselines. We use the true cardinalities and estimates from PostgreSQL as baselines to compare the learned cardinality estimation models.

Training and test sets. We consider two scenarios:

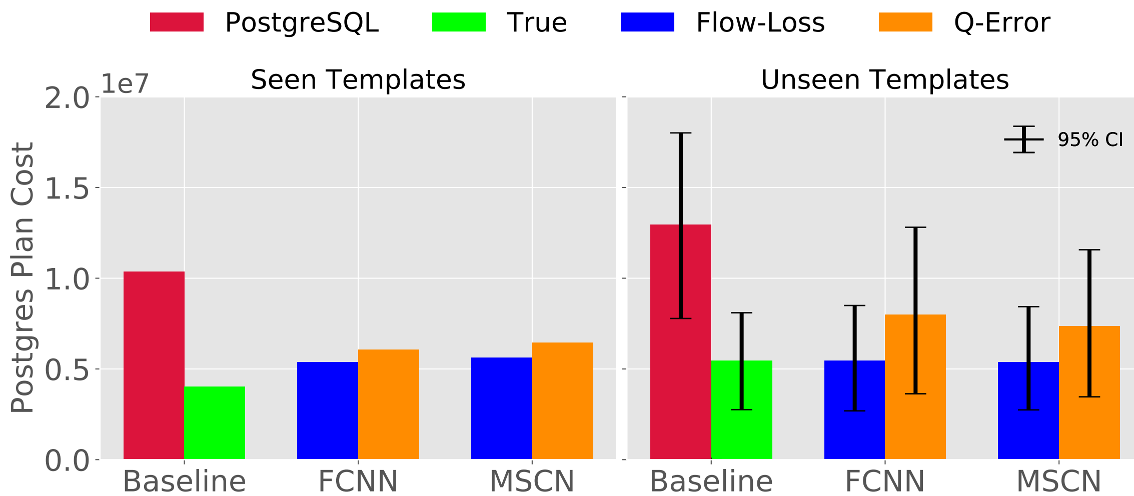
1. **Testing on seen templates.** The model is evaluated on new queries from the same templates that it was trained on. We put 20% of the queries of each template into the validation set, and 40% each into the training and test sets. The hyperparameters are tuned on the validation set, and we report 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 would be a significant overlap with the training set on query *sub-plans*. This tests the robustness of these models to slight shifts in the workload.

7.2 PostgreSQL Results

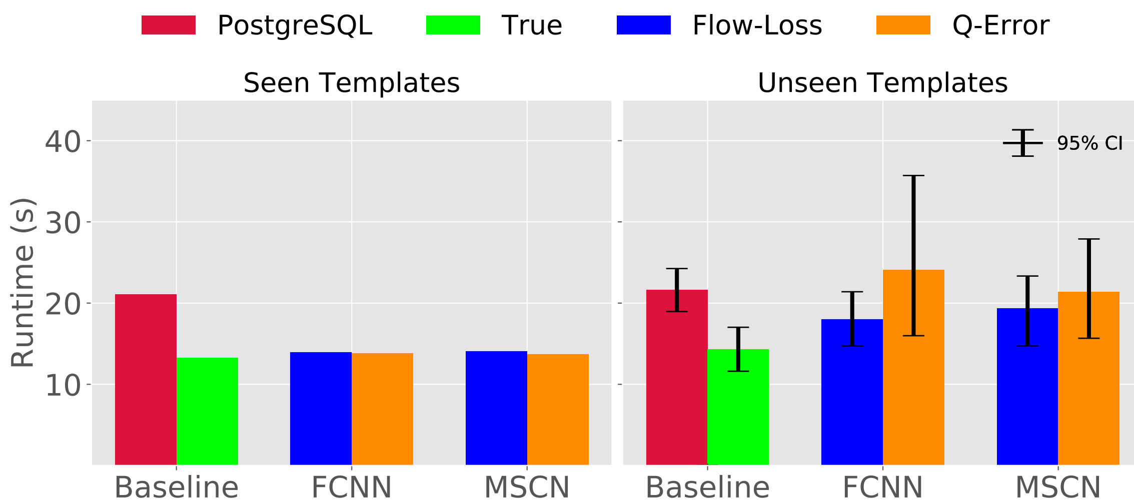
Key results. Figure 7-1a shows the results of all approaches w.r.t. PPC on IMDb. All models outperform PostgreSQL's estimator significantly on seen templates. However, only the Flow-Loss trained models do so consistently on unseen templates as well. For seen templates, the models trained using Flow-Loss 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 7-1b). However, on unseen templates, we see clear improvements in runtime as well.

7.2.1 Testing on seen templates

Table 7.1 summarizes the results for the IMDb workload when trained and tested on all IMDb templates. Each experiment is repeated three times, and we show $\pm 1 stddev$



(a) Mean Postgres Plan Cost (PPC).



(b) Mean query runtimes.

Figure 7-1: Comparing performance of all models on seen versus unseen templates. For unseen templates, we do ten experiments using ten different training/test template splits.

for each statistic. All learned models improve significantly over PostgreSQL on all metrics, and do about equally well. All models do well in this scenario, but there are subtle differences which we highlight below.

Worse Q-Error, better PPC, similar runtimes. The models trained to minimize Q-Error naturally do best on Q-Error - even the 99th percentile Q-Error of millions of sub-plans goes only up to 100. But, this is to be expected — our goal was to improve cardinality estimates only when it is important for query optimization. As

Table 7.1: Models trained and evaluated on the same IMDb templates. We show $\pm 1 stddev$ for Q-Error and PPC from three repeated experiments, and execute plans from one random run. Xp refers to the Xth percentile.

	Q-Error			Postgres Plan Cost (Millions)			Runtime		
	50p	90p	99p	Mean	90p	99p	Mean	90p	99p
Baselines									
True	1	1	1	**4.7**	1.1	131.5	**13.23**	28.96	60.83
PostgreSQL	42.87	48K	2.2M	10.7	9.6	269.7	20.86	46.86	101.57
FCNN									
Q-Error	**2.0 ± 0.1**	10.5 ± 1.6	0.1K ± 32.0	6.4 ± 0.5	1.8 ± 0.1	142.6 ± 11.7	**13.83**	30.82	60.10
Prioritized Q-Error	2.5 ± 0.1	17.1 ± 1.9	0.4K ± 83.9	**5.9 ± 0.4**	1.7 ± 0.0	135.8 ± 18.9	13.91	31.48	61.40
Flow-Loss	4.0 ± 0.9	83.4 ± 36.3	4.1K ± 1.9K	6.0 ± 0.7	1.8 ± 0.1	136.0 ± 23.9	13.93	30.97	60.50
MSCN									
Q-Error	**2.0 ± 0.0**	9.6 ± 0.2	0.1K ± 3.9	6.4 ± 0.3	1.9 ± 0.1	189.1 ± 6.6	**13.69**	30.49	58.49
Prioritized Q-Error	2.8 ± 0.4	25.2 ± 8.3	0.7K ± 0.4K	6.1 ± 0.4	2.4 ± 0.4	163.8 ± 10.1	13.83	31.10	60.00
Flow-Loss	3.0 ± 0.1	44.4 ± 5.6	2.1K ± 0.4K	**5.5 ± 0.6**	2.2 ± 0.1	161.9 ± 16.2	14.08	31.61	59.88

seen in Figure 7-1, the Flow-Loss trained models distinctly 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.

PPC versus runtimes. Figure 7-2 shows the trends for PPC and runtimes are roughly correlated. Notice that the costs (x-axis) are shown on a log scale — thus, *order of magnitude better costs translate to faster runtimes.*

7.2.2 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 7-3a, we look at the performance of a model trained with Flow-Loss compared to one trained with Q-Error w.r.t. PPC and 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, and

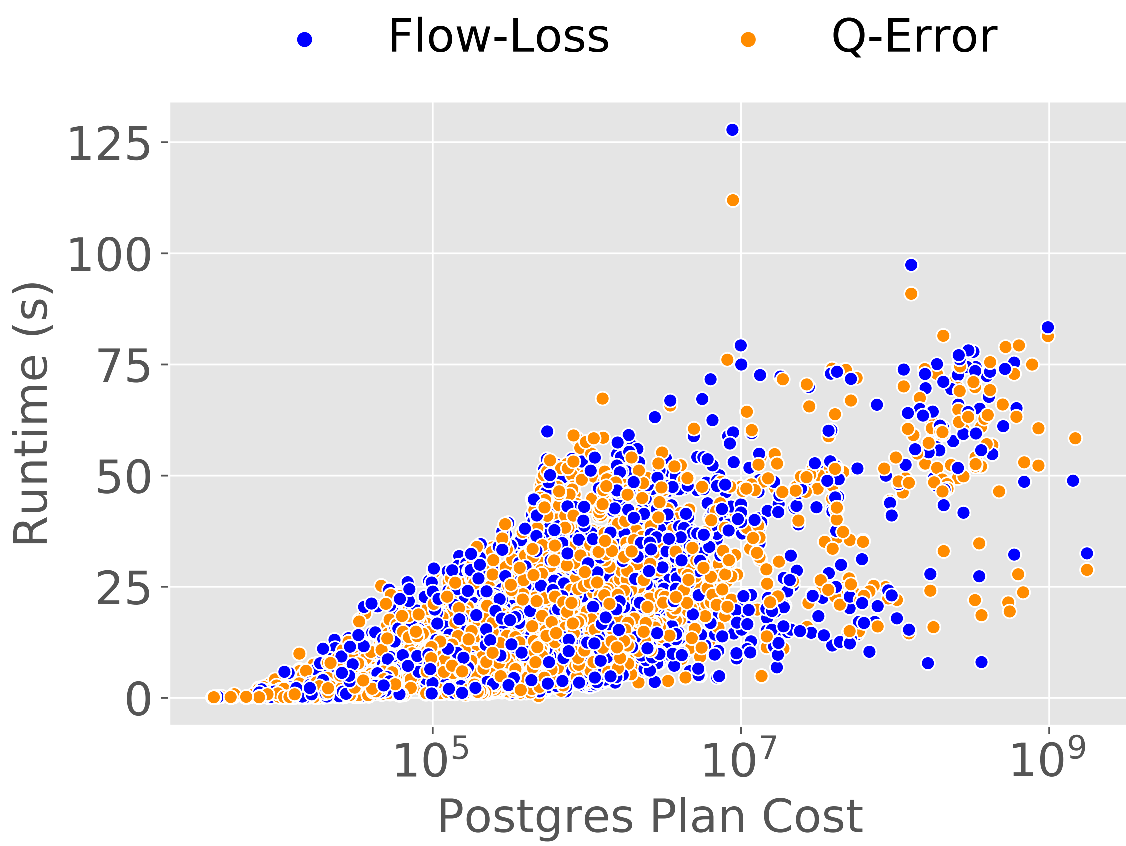
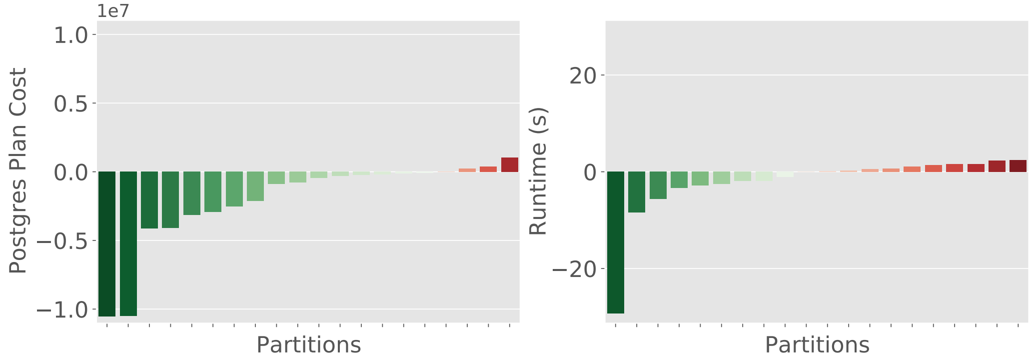


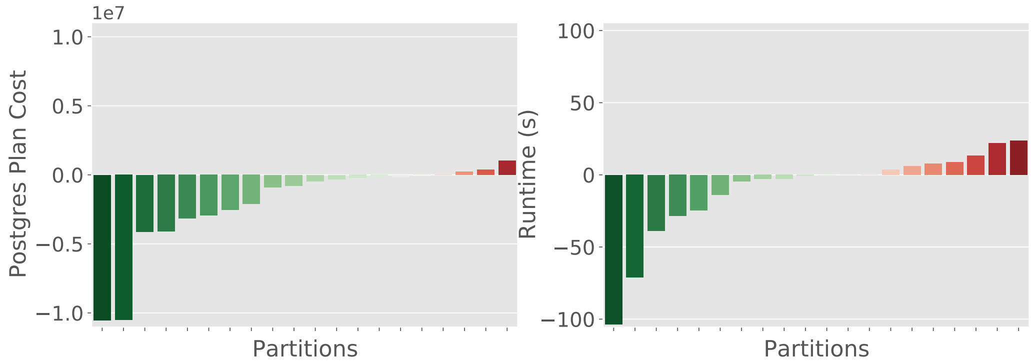
Figure 7-2: PPC versus runtimes for MSCN models trained with Q-Error or Flow-Loss and evaluated on seen templates.

similar performance on many partitions. Comparing the left (PPC), and the right (query runtime) plots, we see that broadly the PPC trends show the same behavior as the runtimes, although it is hard directly translate between them given this data.

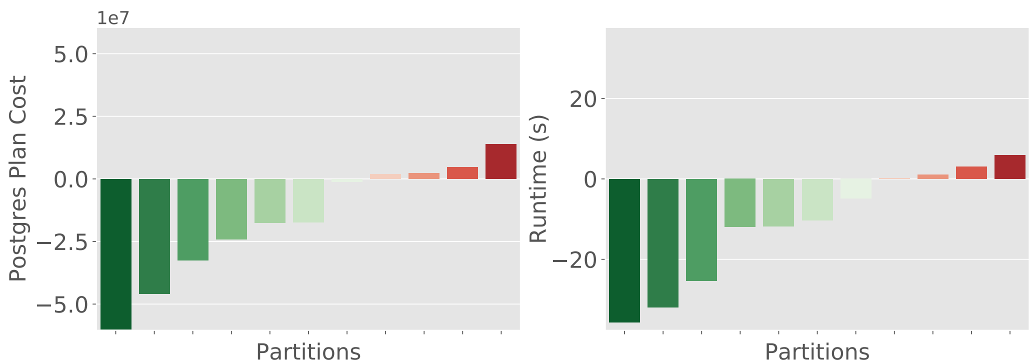
Zooming in on partitions. In Figures 7-4a, 7-4b, we show the 50p, 90p, and 99p for runtimes of FCNN and MSCN models trained with Q-Error or Flow-Loss. For both architectures, the model trained with Flow-Loss significantly improves on all percentiles for the best partition — being up to 8×, and several hundred seconds faster than the Q-Error model. On the worst partition, it is only up to 20 seconds slower than the Q-Error model. Further, there are an additional 6 cases where the Flow-Loss models had improvements in tens of seconds, comparable to the best improvement of the Q-Error model. As we highlight next, even these smaller improvements suggest more robust and better quality plans.



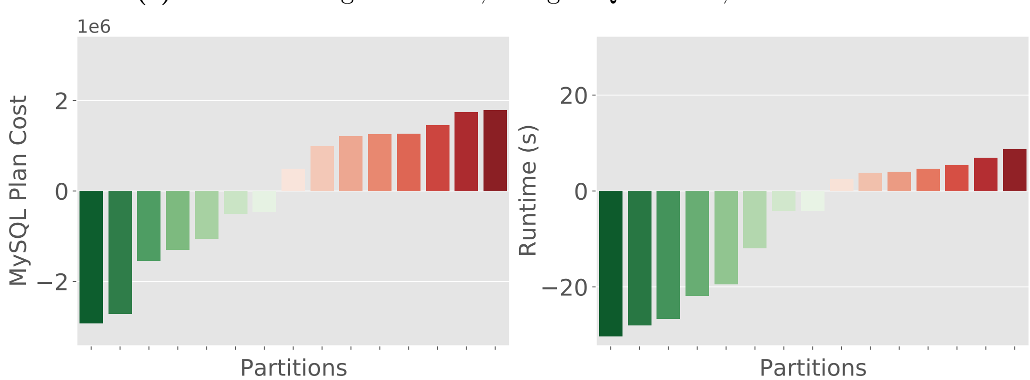
(a) IMDb workload, PostgreSQL DBMS, FCNN or MSCN model.



(b) IMDb workload, PostgreSQL DBMS w/ 256MB RAM. MySQL DBMS, FCNN or MSCN model.

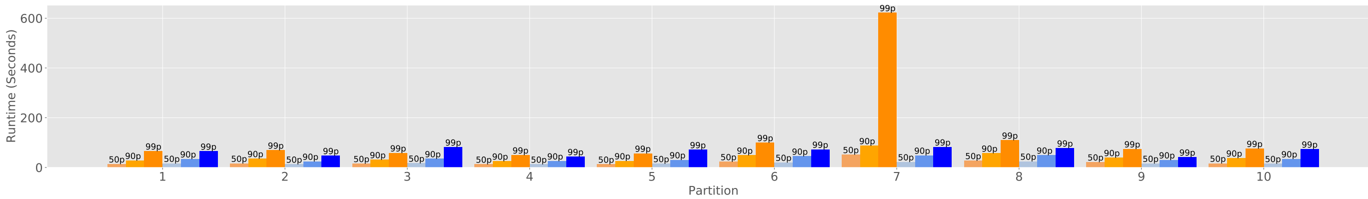


(c) StackExchange workload, PostgreSQL DBMS, MSCN model.

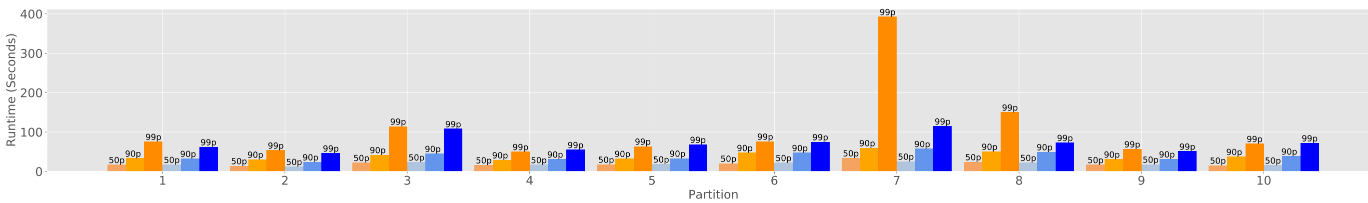


(d) IMDb workload, MySQL DBMS, FCNN or MSCN model.

Figure 7-3: Each bar shows the mean PPC or runtime improvement (green), or regression (red) of Flow-Loss over Q-Error on an unseen partition and the same model architecture (FCNN or MSCN). *Lower is better for Flow-Loss.*



(a) PostgreSQL, FCNN.



(b) PostgreSQL, MSCN.

Figure 7-4: Showing 50p, 90p and 99p runtimes for all partitions for Flow-Loss models on unseen template partitions, trained by either Q-Error or Flow-Loss.

7.2.3 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 7-3b 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 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

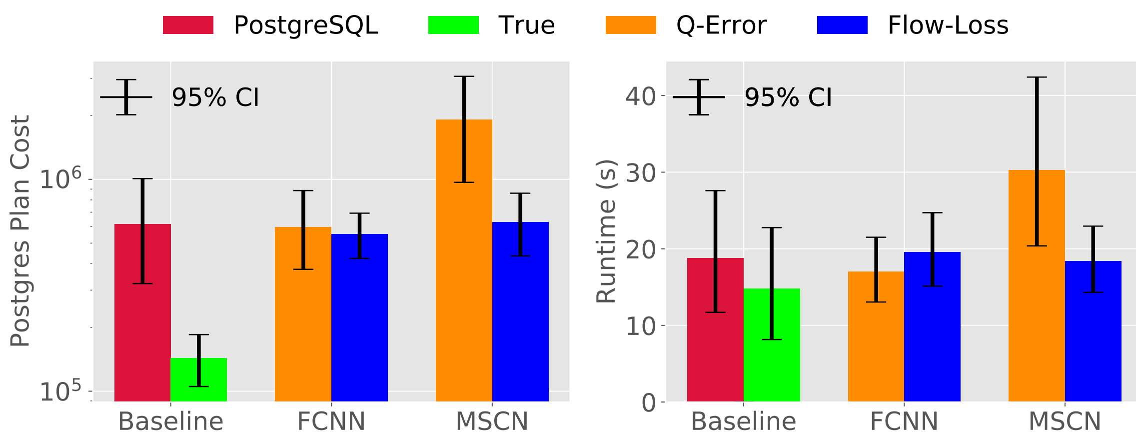


Figure 7-5: Mean PPC and runtimes for all models trained with Q-Error or Flow-Loss on CEB, and evaluated on JOB.

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 v/s 6 for the Flow-Loss model; while in the best case (for Q-Error), it has 4 timeouts v/s 11 for the Flow-Loss model.

7.2.4 Join Order Benchmark

Join Order Benchmark (JOB) is not suitable for training a supervised learning model as it has too few queries. But, we can use it as an evaluation set for a model trained on all the templates from 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. We train on queries from all the CEB templates and evaluate on the JOB queries. Figure 7-5 summarizes the results of the Flow-Loss and Q-Error models for both architectures over three repeated runs. 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. We use this experiment as a sanity check to show that even when the queries are very different, our models avoid

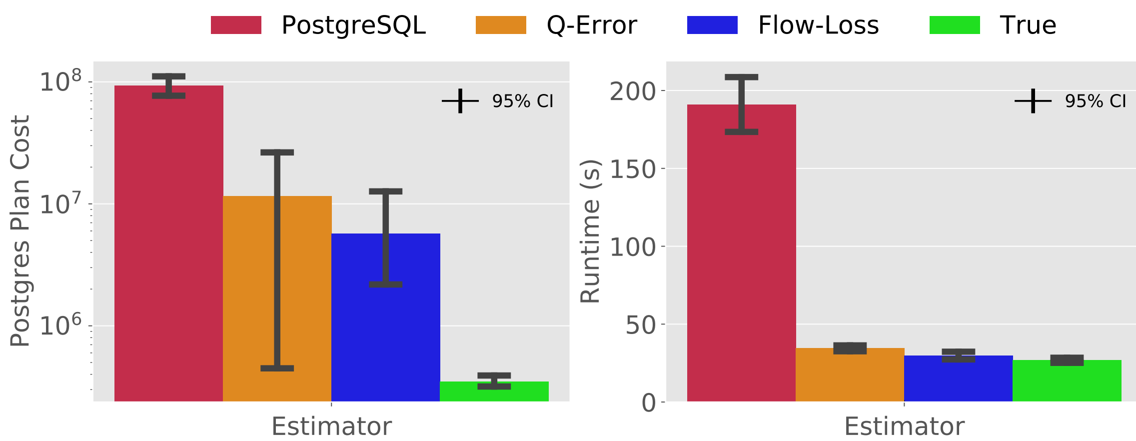


Figure 7-6: Mean PPC (five repeated runs) and runtimes (from one random run) for baselines and MSCN models evaluated on seen templates from the StackExchange workload.

disastrously bad estimates.

7.2.5 StackExchange workload

Finally, we study the performance of the MSCN model on the StackExchange (SE) database using a workload that consists of fewer templates and queries than the IMDb workload. Figure 7-6 summarizes the results for the SE workload when evaluated on the seen templates, and Figure 7-3c shows difference between the mean PPC or runtime achieved by the MSCN model trained with Q-Error or Flow-Loss.

The key difference is that due to timeouts we do not have exhaustive ground truth about all the sub-plan cardinalities (see §6). The timeouts are replaced by a constant value larger than any cardinality in the workload. The plans generated by providing these cardinalities, along with the true cardinalities for known values, (Estimator: True) results in almost 10× faster runtimes than the default PostgreSQL estimates.

Similarities to IMDb results. On seen templates, both loss functions improve significantly over the PostgreSQL estimates, and perform similarly to each other, although the Q-Error models exhibit more variance on PPC over five repeated experiments (cf. Figure 7-6). On unseen templates, the Flow-Loss models improve over the Q-Error models (cf. Figure 7-3c), with improvements of over 20 seconds at the mean on three of the partitions. The magnitude of improvements are larger than on the IMDb workload, with improvements of over 20 seconds at the mean on three partitions. Partially, this is because the database size is also larger than IMDb — thus better plans lead to more substantial improvements — similar to what we see when we restrict the RAM for the IMDb workloads. *This emphasizes that we would expect such behavior to scale to larger databases, where the model capacity and regularization benefits of Flow-Loss can become more critical.*

7.3 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 4.1.5. 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 trained each Flow-Loss model using a cost model approximation tailored to MySQL. The exact cost model, and the trade-offs associated with selecting a cost model, are provided in the online appendix [3].

Similar trends to PostgreSQL experiments. On the seen templates, both the loss functions perform equally well, and significantly improve on heuristic DBMS estimates (presented in the online appendix [3]). In Figure 7-3d, 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 DBMSs, 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.

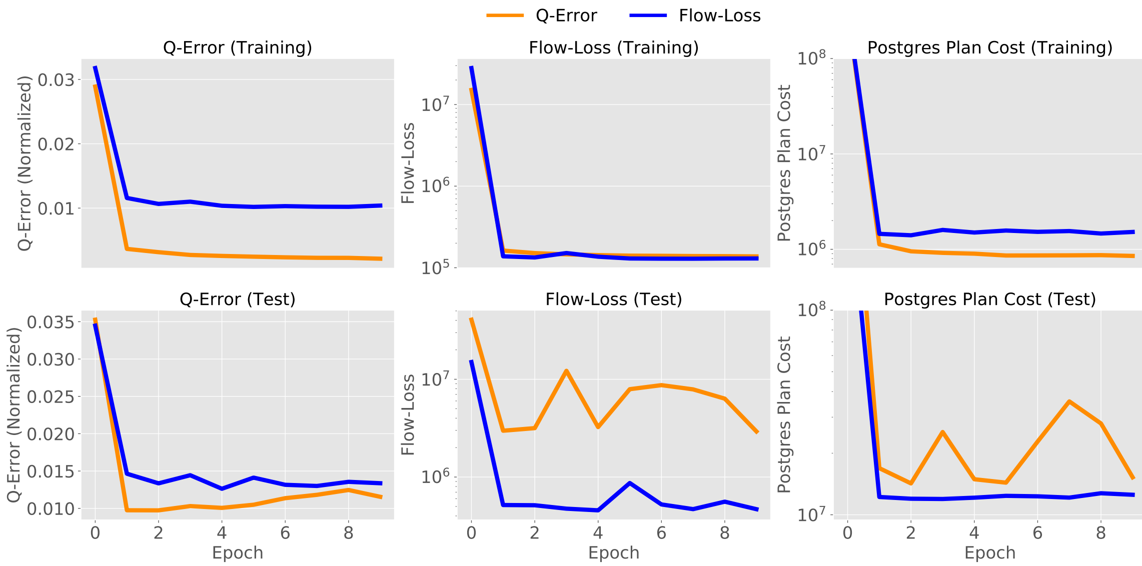


Figure 7-7: Learning curves for one unseen templates partition showing mean of all metrics for MSCN models.

7.4 Analysis

In this section, 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 and hypothesis from §5.

7.4.1 Learning curves

Figure 7-7 shows the MSCN model's learning curves for Q-Error (normalized while training as done by Dutt et al. [11]), 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 it behaves erratically on the test set. This is because it is trying to minimize estimation accuracy, but since the test set contains queries from unseen templates, it is much more challenging. Note that the Flow-Loss curves closely resemble the PPC curves. This similarity is particularly obvious for the Q-Error model on the test set. In §5.2, 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

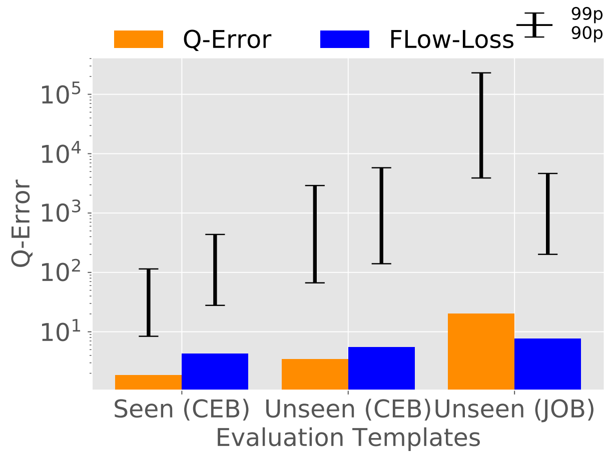


Figure 7-8: Median, 90p, and 99p Q-Error for FCNN models trained on queries from CEB and evaluated on seen templates, unseen templates, and JOB.

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 instead of minimizing estimation accuracy.

7.4.2 Domain specific regularization effect

Figure 7-8 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 performs very similar. For seen templates, the models trained with Q-Error only go up to 100 at the 99p of millions of unseen 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 the dynamic programming optimizer search. Often, we do not need such precise estimates. Models trained with Flow-Loss achieve better PPC despite an order of magnitude higher Q-Errors at the 99p, which suggests that it learns a simpler model that seems more effective for the task of 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, while the models trained with Flow-Loss only get about 1.5× worse. This pattern continues on to the JOB templates — where the

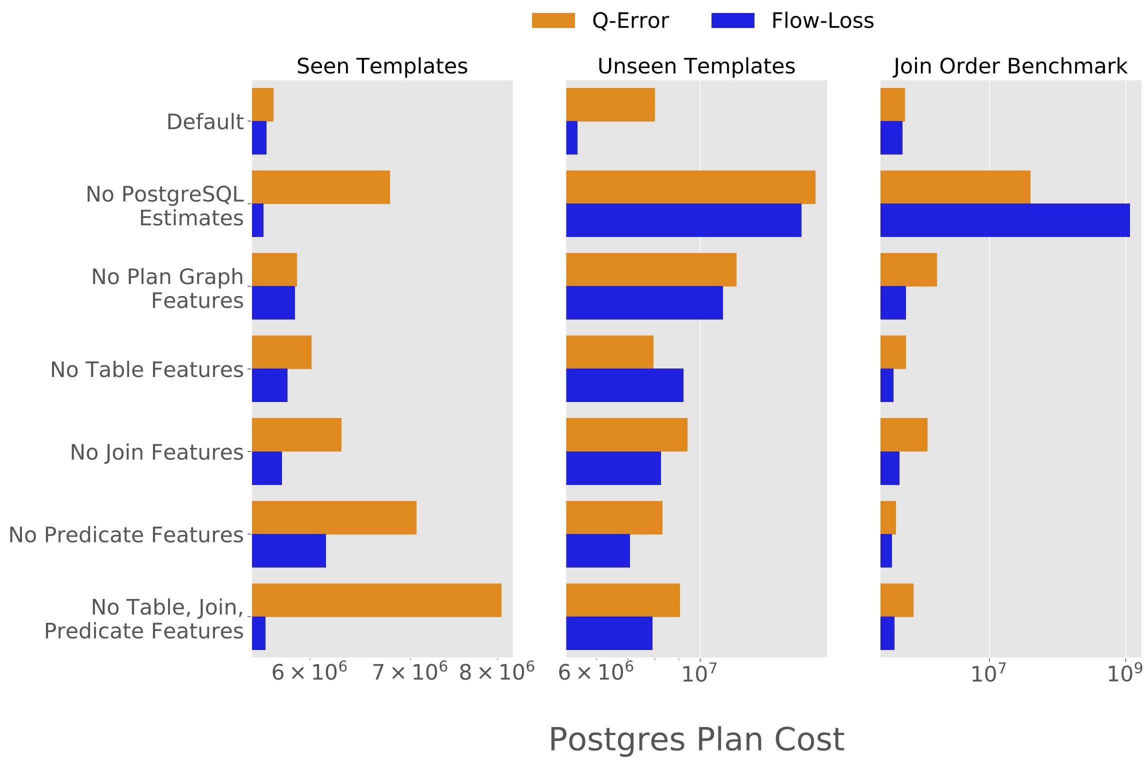


Figure 7-9: Ablation study with the FCNN model for seen templates (left), unseen templates (middle), and JOB (right) showing PPC when various components of the featurization scheme are removed.

Flow-Loss models even have better estimation accuracy than the Q-Error models. This supports our regularization hypothesis (cf. §5.3), 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 make it generalize better to changing query patterns.

7.4.3 Ablation Study

Next, we seek to understand the impact of the various components of the featurization (cf. §7.1) 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 7-9 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

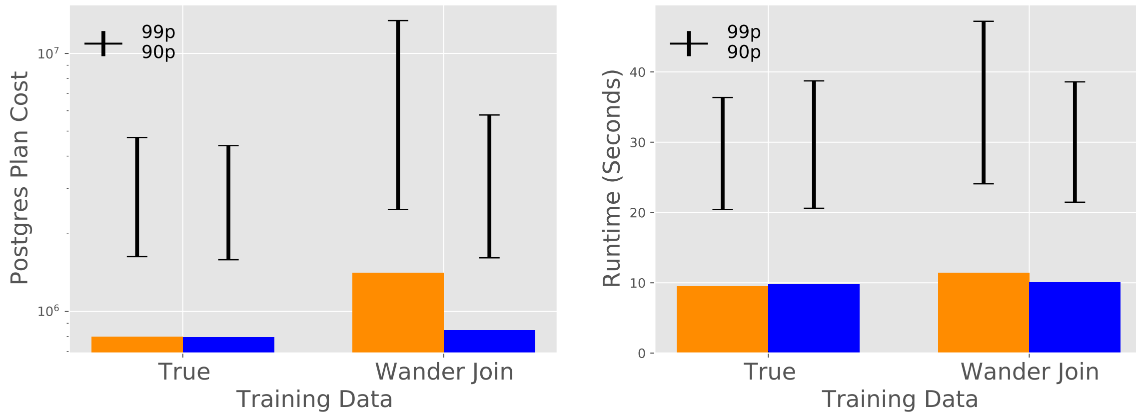


Figure 7-10: Mean PPC and runtimes, along with 90p to 99p error bars for a FCNN model trained with true or wander join cardinalities.

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. §7.1). Both the Flow-Loss and the Q-Error models get significantly worse on 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. Plan graph features also seem to help more than others for generalization to unseen templates.

7.4.4 Training with AQP estimates

Figure 7-10 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. Notice that training with wander join estimates is almost as good as training with true cardinalities. And, 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 tails for both PPC and runtimes. The same trend is observed for Q-Errors as well (see [3]). This 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. §5.3).

7.4.5 Training overhead

Here, we present the overhead of using the Flow-Loss models on the IMDb workload.

Training time. Compared to Q-Error, there is a $3-5\times$ overhead for training either architecture with Flow-Loss due to the additional calculations needed for Flow-Loss—the bottleneck is computing L^{-1} in Equation 4.7.[2] 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.

Inference time. As in [19, 11], the inference times for these neural networks is in the order of a few milliseconds (after featurization) and hence fast enough for query optimization.

Model sizes. The MSCN model is 2.4MB, while the FCNN model is 4.7MB. Sizes are the same for all loss functions.

[2]A long series of works [8, 22, 50] develop ways to approximate L^{-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 [12].

8

Conclusions

"Everything's got a moral, if only you can find it."

Lewis Carrol, Alice's Adventures in Wonderland

"We shall not cease from exploration, and the end of all our exploring will be to arrive where we started and know the place for the first time."
T. S. Eliot, Little Gidding

We showed that Postgres Plan Cost (PPC) 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 PPC 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. Generating ground truth cardinalities in order to train a model is expensive; moreover, updates to the data would quickly make such training data stale. Thus, avoiding the overhead of generating true labeled data can significantly improve adoption of learned cardinality estimation models in practice.

Bibliography

[1] Metric (mathematics). `https://en.wikipedia.org/wiki/Metric_(mathematics)`. [Online;].

[2] Pseudometric space. `https://en.wikipedia.org/wiki/Pseudometric\_space`. [Online;].

[3] Optimizer-cost online appendix. `https://parimarjan.github.io/flow_loss_appendix`, 2020. [Online;].

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

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

[6] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. 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*, pages 211–222, 2001.

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

[8] Paul Christiano, Jonathan A Kelner, Aleksander Madry, Daniel A Spielman, and Shang-Hua Teng. 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*, pages 273–282, 2011.

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

[10] Anshuman Dutt, Chi Wang, Vivek R. Narasayya, and Surajit Chaudhuri. Efficiently approximating selectivity functions using low overhead regression models. *Proc. VLDB Endow.*, 13(11):2215–2228, 2020.

[11] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek R. Narasayya, and Surajit Chaudhuri. Selectivity estimation for range predicates using lightweight models. *PVLDB*, 12(9):1044–1057, 2019.

[12] Pablo Ezzatti, Enrique S. Quintana-Ortí, and Alfredo Remón. Using graphics processors to accelerate the computation of the matrix inverse. *J. Supercomput.*, 58(3):429–437, 2011.

[13] Lise Getoor, Benjamin Taskar, and Daphne Koller. 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*, pages 461–472, 2001.

[14] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 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*, pages 1035–1050, 2020.

[15] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. Deepdb: Learn from data, not from queries! *Proc. VLDB Endow.*, 13(7):992–1005, 2020.

[16] Oleg Ivanov and Sergey Bartunov. Adaptive cardinality estimation. *arXiv preprint arXiv:1711.08330*, 2017.

[17] Martin Kiefer, Max Heimel, Sebastian Breß, and Volker Markl. Estimating join selectivities using bandwidth-optimized kernel density models. *Proc. VLDB Endow.*, 10(13):2085–2096, 2017.

[18] Andreas Kipf, Michael Freitag, Dimitri Vorona, Peter Boncz, Thomas Neumann, and Alfons Kemper. Estimating filtered group-by queries is hard: Deep learning to the rescue. *1st International Workshop on Applied AI for Database Systems and Applications*, 2019.

[19] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 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*.

[20] Andreas Kipf, Dimitri Vorona, Jonas Müller, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 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.*, pages 1937–1940.

[21] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *CoRR*, abs/1808.03196, 2018.

[22] Rasmus Kyng, Yin Tat Lee, Richard Peng, Sushant Sachdeva, and Daniel A Spielman. Sparsified cholesky and multigrid solvers for connection laplacians. In

Proceedings of the forty-eighth annual ACM symposium on Theory of Computing, pages 842–850, 2016.

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

[24] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[25] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 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*, 2017.

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

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

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

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

[30] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Learning to steer query optimizers. *CoRR*, abs/2004.03814, 2020.

[31] Ryan Marcus and Olga Papaemmanouil. 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*, 2019.

[32] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *PVLDB*, 12(11):1705–1718, 2019.

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

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

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

[36] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Mao Hongzi, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. Cardinality estimation dataset. `https://github.com/cardinality-estimation-benchmark`, 2020. [Online;].

[37] Parimarjan Negi, Ryan Marcus, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. Cost-guided cardinality estimation: Focus where it matters. In *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*, pages 154–157. IEEE, 2020.

[38] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 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*, pages 4:1–4:4, 2018.

[39] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. An empirical analysis of deep learning for cardinality estimation. *CoRR*, abs/1905.06425, 2019.

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

[41] Matthew Perron, Zeyuan Shang, Tim Kraska, and Michael Stonebraker. 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*, pages 1758–1761, 2019.

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

[43] Dipanjan (DJ) Sarkar. Categorical data, Mar 2019.

[44] StackExchange. Stackexchange data explorer, 2020.

[45] Ji Sun and Guoliang Li. An end-to-end learning-based cost estimator. *PVLDB*, 13(3):307–319, 2019.

[46] NIPPON TELEGRAPH and TELEPHONE CORPORATION. Pg hint plan, 2013.

[47] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 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*, pages 1153–1170, 2019.

[48] Kostas Tzoumas, Amol Deshpande, and Christian S. Jensen. Efficiently adapting graphical models for selectivity estimation. *VLDB J.*, 22(1):3–27, 2013.

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

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

[51] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. Are we ready for learned cardinality estimation? *arXiv preprint arXiv:2012.06743*, 2020.

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

[53] Lucas Woltmann, Claudio Hartmann, Maik Thiele, Dirk Habich, and Wolfgang Lehner. 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*, pages 5:1–5:8, 2019.

[54] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. Towards a learning optimizer for shared clouds. *PVLDB*, 12(3):210–222, 2018.

[55] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. Neurocard: One cardinality estimator for all tables. *CoRR*, abs/2006.08109, 2020.

[56] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Peter Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. Deep unsupervised cardinality estimation. *PVLDB*, 13(3):279–292, 2019.

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

[58] Tianwei Zhang, Zecheng He, and Ruby B Lee. Privacy-preserving machine learning through data obfuscation. *arXiv preprint arXiv:1807.01860*, 2018.

[59] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 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*, pages 1525–1539, 2018.