
# Sample-Efficient Cardinality Estimation Using Geometric Deep Learning

Silvan Reiner
University of Konstanz
silvan.reiner@uni.kn

Michael Grossniklaus
University of Konstanz
michael.grossniklaus@uni.kn

## ABSTRACT

In database systems, accurate cardinality estimation is a cornerstone of effective query optimization. In this context, estimators that use machine learning have shown significant promise. Despite their potential, the effectiveness of these learned estimators strongly depends on their ability to learn from small training sets.

This paper presents a novel approach for learned cardinality estimation that addresses this issue by enhancing sample efficiency. We propose a neural network architecture informed by geometric deep learning principles that represents queries as join graphs. Furthermore, we introduce an innovative encoding for complex predicates, treating their encoding as a feature selection problem. Additionally, we devise a regularization term that employs equalities of the relational algebra and three-valued logic, augmenting the training process without requiring additional ground truth cardinalities. We rigorously evaluate our model across multiple benchmarks, examining q-errors, runtimes, and the impact of workload distribution shifts. Our results demonstrate that our model significantly improves the end-to-end runtimes of PostgreSQL, even with cardinalities gathered from as little as 100 query executions.

**PVLDB Reference Format:**
Silvan Reiner and Michael Grossniklaus. Sample-Efficient Cardinality Estimation Using Geometric Deep Learning. PVLDB, 17(4): 740 - 752, 2023.
doi:10.14778/3636218.3636229

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/dbis-ukon/jgmp.

## 1 INTRODUCTION

In recent years, learned query optimization, in general, and learned cardinality estimation, in particular, has become an increasingly active research area. The original vision is simple and appealing: Use meta-data about query executions from log files to train machine learning models and improve the performance of the query optimizer. Such training data comes at no additional cost since it is a byproduct of regular system operation.

Recent methods have strayed from this vision by competing for lower and lower errors or runtimes without accounting for the cost of gathering ground truths. Firstly, models are trained on an abundance of training data: Cardinality estimation models, in particular, are often trained with tens of thousands to a hundred thousand training queries. Secondly, some models not only require lots of training queries but also very particular information about each training query: Query optimizers using reinforcement learning often demand the same query to be executed multiple times with different plans to calculate relative costs, while recent advances in cardinality estimation were made possible by collecting the true cardinality of all of potentially thousands of subplan queries of each query in the training set. By demanding these large amounts of specific training data, existing methods are not fit to function from query logs alone. Instead, they demand the computation of massive amounts of supplemental ground truths, which must be accounted for when analyzing the training costs. These demands are a major contributor to the slow adoption of learned query optimization in practice contrasting with the active research area.

In this paper, we return to the original vision for learned query optimization and build a cardinality estimation model that can realistically be trained with cardinalities from query logs alone. We must address two major challenges to achieve this objective.

Firstly, it is crucial to improve the sample efficiency of our models in order to learn from the limited training data available in query logs. By improving sample efficiency, we can make progress on three major issues in learned cardinality estimation: the cold start problem, query distribution shifts, and data distribution shifts. A more sample-efficient model will outperform classical estimators more quickly on new databases, adapt to new query types with fewer examples, and make do with only cardinalities from the most recent queries.

Secondly, we have to take into account that query logs do not provide an unbiased sample of subplan query cardinalities. Generally, query optimizers choose subplans with low expected cardinalities, leading to a bias that can exclude subplans with very high cardinalities that are part of catastrophic plans from the training set. Figure 1 visualizes this bias for the Join Order Benchmark by comparing the cumulative fraction as a function of the true cardinality of all subplan queries with the ones selected by the PostgreSQL optimizer (PG-selected) and that are, therefore, contained in the query log. The mean cardinality for all subplan queries is 14,000,000, as opposed to only 330,000 for PG-selected subplan queries. It is essential to evaluate the impact of this bias on the models since it potentially limits their practical applicability.

Geometric deep learning can improve the sample efficiency by exploiting the inherent structure of the problem and incorporate invariances into the architectures. We examine the strengths and weaknesses of current state-of-the-art learned cardinality estimators in this light and propose a novel architecture to address them.


Proceedings of the VLDB Endowment, Vol. 17, No. 4 ISSN 2150-8097.
doi:10.14778/3636218.3636229

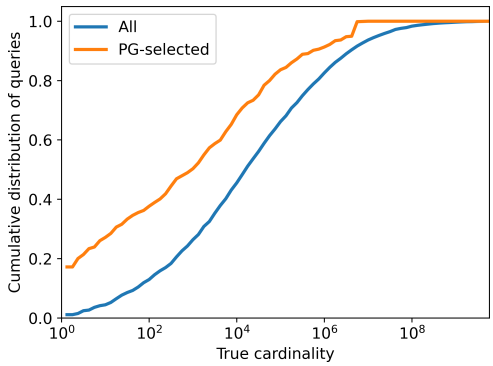



**Figure 1: Cumulative distribution of all JOB subplan queries or only subplan queries part of plans chosen by PostgreSQL.**

Specifically, we present the following contributions in this paper:

- A novel neural network architecture for cardinality estimation that represents queries as join graphs, using geometric deep learning principles to enhance learning effectiveness (Section 3.4).
- A unique approach to the encoding of complex predicates as a feature selection problem, leading to a new encoding method that reduces inference time and improves estimation accuracy (Section 4.1).
- A regularization term that leverages equalities of the relational algebra to enhance training, all without necessitating additional ground truth cardinalities (Section 5).

Furthermore, we conduct a comprehensive evaluation of our novel approach (Section 6), featuring

- an exhaustive evaluation on multiple benchmarks, comparing the q-errors and execution times of various models,
- an in-depth analysis exploring how biases of query logs and workload distribution shifts affect model performance, and
- an ablation study of the effects of various design choices on performance of our neural network architecture.

## 2 RELATED WORK

Learned methods for query optimization can be divided into two broad categories: The first category consists of query-driven methods that need to be trained with a workload of queries. The second category consists of cardinality estimators that use unsupervised learning to summarize the data distribution in the database. These models are trained without a training workload.

### 2.1 Query-Driven Methods

Query-driven methods can further be divided into regression models for query cardinalities or plan costs trained with supervised learning and fully learned query optimizers that model query optimization as a Markov decision process.

Three main types of representation structures are inputs to these models: single fixed-length vectors, tree-structured representations, and multi-sets. The simplest structure represents a query as a single fixed-length vector. This vector is typically generated as engineered encodings [4, 25, 32] or with the use of embedding techniques [18]. While this approach is simple and enables the application of a wide range of machine learning techniques, the constant size of the vector limits its expressivity as an infinite space of possible queries has to be reduced to a finite vector. The other two representations solve this problem using variable-size structures (like trees or sets) over fixed-length encodings of the query elements. Thus, the representations can grow with the queries.

Tree-structured representations are based on logical or physical algebra expressions. These representations can then be processed with, among others, tree convolutional, recurrent, or transformer neural networks. Naturally, this representation is especially suited for learned cost models [11, 43] since the execution cost is a function of a physical plan. Such representations [28, 35] have also been evaluated as both cost models and cardinality estimators. By restricting the representation to left-deep plans, we get a special case where a query is represented as a series of joins. This representation is suitable for models using recurrent neural networks [25]. In addition to cost models, a tree-structured representation is also a natural choice for reinforcement-learned query optimizers [19, 36, 40, 44] since they create increasingly complex subplans with each step. In this framework, the tree-structured representation for the subplan is often combined with a representation for the whole query to describe the current state in the Markov decision process [2, 20].

The last type of representation was introduced with the learned cardinality estimator MSCN [16]. It consists of three multi-sets of encoding vectors. Each element in the first multi-set represents a table occurrence, each element in the second multi-set represents a join, and finally, each element in the third multi-set represents a predicate. In the MSCN architecture, these multi-sets of encoding vectors are aggregated to one single-dimensional vector using deep sets [41]. These aggregated embeddings for each set are then concatenated into a single embedding vector. Fully-connected neural network layers can then process them to estimate the cardinality of the query. The MSCN architecture has repeatedly proven itself to be the current state of the art for learned query-driven cardinality estimation [9, 14]. Various improvements for learned query-driven cardinality estimators have been proposed for or tested as an extension of MSCN, such as loss functions [23] and encodings [22, 24].

### 2.2 Unsupervised Data-Driven Methods

Unsupervised data-driven methods have also been proposed for cardinality estimation. Currently, there are multiple main families of methods. The first family consists of autoregressive models with NeuroCard [38] and its predecessor for single tables Naru [37]. The second family consisting of DeepDB [12] and FLAT [45] utilize sum-product networks [26]. Outside of these two families of methods, there is BayesCard [34] using Bayesian networks and the FactorJoin framework [33] for combining single-table cardinality estimates into estimates for multi-table queries.

## 3 DESIGN PRINCIPLES

This section discusses basic design principles for encodings and architectures for artificial neural networks as well as how these principles apply to cardinality estimation. Then we analyze if and when the most commonly used query representations for learned

cardinality estimation satisfy these principles. Finally, we propose our query representation that is informed by these principles.

### 3.1 Invariances

Generally, a neural network approximates a function $f : X \rightarrow Y$ from an input $x \in X$ to an output $y \in Y$. In specific cases, there exist transformations of the inputs $t : X \rightarrow X$ such that the output is invariant with respect to $f$: $\forall x \in X : f(x) = f(t(x))$.

Ideally, invariances specific to the problem domain are incorporated into the design of neural network architectures [1]. The invariances should be guaranteed to be satisfied regardless of the values of the learnable parameters. The main benefit is reducing the search space size of possible functions. Instead of searching the space of all possible functions from $X$ to $Y$, we only want to search the space of functions from $X$ to $Y$ that satisfy invariance constraints. This constraint improves the model's sample efficiency.

For the problem of cardinality estimation, several query transformations are invariant with respect to the cardinality. The formulation of these invariances depends on the representation of the queries. For SQL, such transformations are changes that only affect the syntax but not the semantics of the query, such as reordering and renaming tables. For relational algebra, they include equivalences such as commutativity and associativity of join operators. When two relational expressions are logically equivalent, their cardinalities are, of course, equal. For cardinality estimation, this equivalence is prior knowledge that relies only on the relational schema and holds independently of the content of the database.

### 3.2 Uniqueness

Incorporating invariances should not come at the cost of the expressivity of the model. Given $x_1, x_2 \in X$ with differing target values $f(x_1) \neq f(x_2)$ the encodings of $x_1$ and $x_2$ should differ. Otherwise, it will be impossible for the neural network able to approximate the function $f$ since, with the same input encodings, the network outputs are also the same. Applied to the problem of cardinality estimation in relational databases, this means that queries should have unique representations as long as they are not guaranteed to have the same cardinality.

### 3.3 Analysis of the Current State of the Art

Section 2.1 introduced the currently most popular query-based learning representations: algebraic expressions and multi-sets. Now, we evaluate their suitability to the cardinality estimation problem based on invariance and uniqueness principles.

First, representations based on algebraic expressions will naturally satisfy the uniqueness property since two queries that share a representation in relational algebra are necessarily equivalent. However, they are not invariant to cardinality-preserving transformations like commutativity and associativity of the join operators. One query can be represented by many equivalent algebraic expressions. Therefore, the representation based on algebraic expressions is at a disadvantage since this estimator additionally has to learn that equivalent algebraic expressions have the same cardinality. This lack of invariance significantly increases the model's domain and the required training data.

Second, MSCN's multi-set representation satisfies the invariance property since it is not dependent on algebraic transformations or purely syntactic changes to a query. However, the multi-set representation violates the uniqueness property (Figure 2). In this example, the references between the members of the different sets are lost. Joins and predicates cannot be associated with table occurrences unambiguously if there are multiple occurrences of the same table. The MSCN was initially evaluated on the JOB-light benchmark, which does not include queries with multiple occurrences of the same table. The multi-set representation is sufficient for this benchmark, but the expressivity of the model is too restricted for more complex workloads.

In summary, we have shown that representations based on algebraic expressions satisfy the uniqueness property but violate cardinality invariances, whereas, MSCN's multi-set representation satisfies the cardinality invariances but violates the uniqueness property. The former gives the same query different representations, while the latter gives different queries the same representation.

### 3.4 A Join-Graph-Based Query Representation

We propose a join-graph-based query representation that is both unique and respects cardinality invariances. Compared to the multi-set representation, it adds the missing associations between table occurrences, joins, and predicates. It respects cardinality invariances because the representation depends neither on syntactic changes to the query nor on some algebraic expression that implements it. Figure 2 shows how this representation is unique even if one table has multiple *occurrences* with different aliases in a query.

We define the join graph of a query as a labeled directed graph $(V, E, \tau, \pi)$ with the following components.

- A set of nodes $V$ with one node for each table occurrence.
- A set of labeled directed edges $E \subseteq V \times V \times F$ with one edge for each join.
- A node labeling function $\tau : V \rightarrow T$.
- A predicate function $\pi : V \rightarrow P$.

$F$ is the set of foreign keys in the relational schema. An edge $(u, v, f)$ represents a join between tables occurrences $u$ and $v$ with the fk-pk join condition on $f \in F$. The edge is directed from foreign key (fk) to primary key (pk). $T$ is the set tables in the relational schema. The function $\tau$ assigns each table occurrence to its table. $P$ is the set of three-valued logic expressions using columns in the relational schema. The function $\pi$ assigns each table occurrence to the predicates referencing it.

### 3.5 Scope

Our model requires a list of supported joins to label and later encode the edges meaningfully. The foreign keys in the schema are a natural choice since they typically account for a large number of joins. With this choice, we restrict each edge in our join graphs to a fk-pk join.

A query containing fk-fk joins can easily be transformed into a query that only contains fk-pk joins. In the case that a fk-fk join condition is already implied by two fk-pk join conditions, it can be removed from the query. Otherwise, we add a table occurrence to which both foreign keys point. Then we add the two corresponding fk-pk join conditions such that our fk-fk join condition becomes

```
SELECT * FROM movies m
JOIN movies s ON s.sequel_of_id = m.id
JOIN roles r ON r.movie_id = s.id
WHERE m.year = 2000;
```

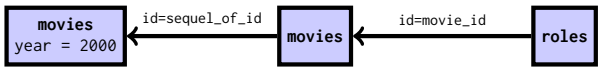
(a) Query a

```
SELECT * FROM movies m
JOIN movies s ON s.sequel_of_id = m.id
JOIN roles r ON r.movie_id = m.id
WHERE s.year = 2000;
```

(b) Query b

```
({movies, movies, roles},
 {sequel_of_id = id, movie_id = id},
 {year = 2000})
```

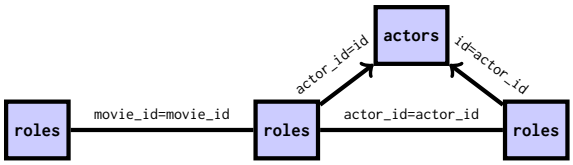
(c) MSCN's multi-set representation of Query a and Query b



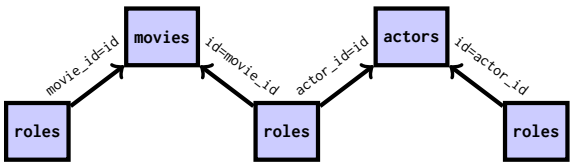

(d) Join graph for Query a


(e) Join graph for Query b

**Figure 2: Two queries which are indistinguishable with MSCN's multi-set representation but have unique join graphs.**




(a) Initial join graph




(b) Preprocessed join graph

**Figure 3: Join graph preprocessing: By adding an occurrence of table `movies`, the fk-fk join `r2.movie_id = r1.movie_id` is transformed into two fk-pk joins. The fk-fk join `r3.actor_id = r2.actor_id` is redundant.**

redundant and can be removed. Figure 3 shows an example for both cases. Since this transformation does not change the cardinality of the query, we can use it to directly represent fk-fk joins.

Still, queries may contain elements not directly supported by our representation, such as non-fk joins, anti and outer joins, aggregations, and others. We deliberately do not to support these operations directly since they would drastically expand the size of our model's input space, increasing the required training data. Instead, we focus on the most common query elements. Using traditional cardinality estimation formulas, the cardinality of not directly supported expressions can be estimated based on our cardinality estimations of their child expressions. Our experimental evaluation demonstrates that this approach is effective.

# 4 ENCODING AND MODEL ARCHITECTURE

We define a novel deep learning architecture called Join Graph Message Passing (JGMP) that incorporates the invariance and uniqueness properties specific to cardinality estimation. First, we describe how the components of the join graph are encoded and then how a cardinality is estimated based on those encodings.

## 4.1 Table and Predicate Encodings

In Section 3.1, we examined the importance of invariance to cardinality-preserving transformations for an estimator, focusing on transformations that preserve the cardinality based on a schema without considering the database state. In this section, we extend our discussion on invariance by incorporating the database state, providing a more comprehensive understanding of invariance in cardinality estimation. We then use this understanding to analyze different ways of encoding predicates and design a novel method.

As an example, consider a state of table `movies` (see Table 1) containing *only* six rows. Table `movies` can occur in different queries filtered by different predicates. Table 2 gives an example of six predicates of occurrences of table `movies`. We observe that the predicates a, b, c, and d all filter the table `movies` similarly. They all only select Movie 1. Predicate a and b represent the simplest case: They are syntactically equivalent. The predicate a and c can be identified as semantically equivalent since the predicate `year <= 1973` can be transformed into `NOT year > 1973` using logical equivalence rules. The predicates a, b, and c are guaranteed to yield the same result without considering the state of table `movies`. However, predicate d is neither syntactically nor semantically equivalent to predicates a, b, and c. To see that this predicate yields the same result as the others, we have to compute the effect of the predicate on the table `movies`. For predicate d, the equivalence to the other predicates depends on the current state of table `movies`.

This example illustrates three different types of equivalence between predicates. Predicates a and b are an example of the strictest type, namely syntactic equivalence. Predicates a and c are an example of the second strictest type, namely semantic equivalence. Finally, predicates a and d are an example of the least strict type, which depends on the current state of the table. We will call this type table-state equivalence. In this hierarchy of equivalence types, syntactic equivalence implies semantic equivalence, and semantic equivalence implies table-state equivalence.

The three types of equivalences are invariances of the cardinality of queries. All of them should ideally be incorporated into the encodings. Syntactic equivalence is incorporated trivially. Only a non-deterministic function could violate this invariance. Semantic and table-state equivalence are more challenging to incorporate. Various predicate encodings have been proposed as parts of learned cardinality estimators, cost models, and query optimizers. Most of

**Table 1: Table `movies`.**

| id | name | year |
|---|---|---|
| 1 | The Godfather | 1972 |
| 2 | The Sting | 1973 |
| 3 | The Godfather Part II | 1974 |
| 4 | One Flew Over The Cuckoo's Nest | 1975 |
| 5 | Rocky | 1976 |
| 6 | Annie Hall | 1977 |

**Table 2: Example predicates for occurrences of table `movies`.**

| | Predicate | Table Bitmap |
|---|---|---|
| a | `name LIKE '%Godfather%' AND year <= 1973` | 1 0 0 0 0 0 |
| b | `name LIKE '%Godfather%' AND year <= 1973` | 1 0 0 0 0 0 |
| c | `name LIKE '%Godfather%' AND NOT year > 1973` | 1 0 0 0 0 0 |
| d | `id = 1` | 1 0 0 0 0 0 |
| e | `year BETWEEN 1973 AND 1974` | 0 1 1 0 0 0 |
| f | `NOT name IN ('Rocky', 'The Sting')` | 1 0 1 1 0 1 |

these techniques are either predicate-based (generating one encoding vector per comparison predicate) or column-based (generating one encoding vector per column).

In their most common form, predicate-based techniques represent each comparison predicate with a vector of multiple components: A one-hot encoding for the column, a one-hot encoding for the comparison operator and some encoding for the value. To represent negations, a fourth component can be added. Naively, predicate-based techniques do not incorporate semantic equivalences. The predicate `year <= 1973` will be encoded differently from `NOT year > 1973`. This problem can only be addressed by transforming predicates into a canonical form. However, canonizing complex predicates involving disjunctions is problematic since slight changes to a predicate can lead to completely different canonical forms. Again, it is challenging to learn from such an unstable representation.

Müller et al. [22] propose a column-based technique that partitions the domain of each column and featurizes complex predicates with a vector indicating which partitions are fully selected, partially selected, or not selected. This technique incorporates semantic equivalence. Whether a partition is selected or not does not depend on arbitrary syntactic details.

Both predicate-based and column-based techniques do not incorporate table-state equivalence. These encodings do not depend on the data in the table, except for minimum and maximum values of numeric columns used in normalization and partitioning. Thus, the encoding cannot represent correlations between the table columns. As a consequence, these encodings reduce the sample efficiency because the model has to learn how the table columns are correlated. Additionally, the number of dimensions of the neural net's input increases with the number of table columns or with the number of comparison operator types.

Sample bitmaps are a technique first proposed by Kipf et al. [16] in addition to their predicate encoding. Before training a model, an ordered sample of $n$ random rows is extracted from each table. Then the predicate is evaluated for each row in the sample. The $i^{\text{th}}$ bit of the sample bitmap of length $n$ is given by 1 if the $i^{\text{th}}$ sample row satisfies the predicate and 0 if it does not. The last column of Table 2 shows the bitmaps for a sample of all rows of table `movies` ordered ascendingly by their `id`. A sample bitmap is an encoding with the desired invariance to table-state equivalences. In our example, the table-state equivalent predicates a, b, c, and d have the same sample bitmap $(1, 0, 0, 0, 0, 0)$.

However, sample bitmaps also have major disadvantages that have limited their adoption in learned cardinality estimators [23]. Very selective predicates might not satisfy any sample rows. In this case, the sample bitmap is all zeros and not an informative predicate description. Therefore, large sample sizes are often chosen to reduce the likelihood of zero-valued bitmaps. However, such long bitmaps are expensive to compute and increase the model's size. As a result, model training and inference are slower, and the model is more prone to overfitting.

We aim to shorten the sample bitmaps while keeping the information they carry. This compression is a feature selection problem [13, 27], where the sample rows are the features to be selected to describe table occurrences and their predicates. Supervised feature selection algorithms typically rely on repeated training and evaluation of models, making them computationally expensive. Instead, we apply the greedy unsupervised feature selection algorithm by Farahat et al. [5]. This algorithm selects a diverse set of features and removes the redundant ones. A feature $f$ is considered redundant with respect to a set of other features $S$ if it is a linear combination of the features in $S$. In our example, Movie 5 is considered redundant with respect to the empty set $\emptyset$ since its vector is all zeros. Movie 6 is redundant with respect to the set {Movie 4} since its vector is a duplicate of Movie 4's vector. Movie 3 is redundant with respect to the set {Movie 2, Movie 4} since its vector is the sum of Movie 2 and Movie 4's vectors.

To apply the algorithm, we randomly select up to $10^5$ sample rows of a table and evaluate them on the set of $m \leq 1000$ occurrences of the table in our training data to get a vector of length $m$ for each row. We eliminate sample rows with duplicate vectors to eliminate trivially redundant sample rows. All-one vectors are also removed despite not necessarily being redundant since they do not help to distinguish between the table occurrences. The vectors for the remaining sample rows form a matrix, where each row represents a table occurrence, and each column represents a sample. Then the greedy unsupervised feature selection algorithm is applied to this matrix. It iteratively selects diverse features and stops when the desired number of features is reached or when all remaining features are redundant with respect to the already selected features.

For our example, the algorithm greedily selects sample rows in the order: Movie 1, Movie 3, Movie 2. As the remaining rows are redundant with respect to the sample, the algorithm terminates. Based on this sample, Predicates a, b, c, and d are encoded as $(1, 0, 0)$, Predicate e as $(0, 1, 1)$, and Predicate f as $(1, 1, 0)$.

This process results in a predicate featurization invariant to table-state equivalences. Additionally, the encodings are short and quickly computed during inference.

The feature selection algorithm introduces a bias to the samples that could negatively impact the quality of the estimations of a traditional sampling-based estimator. This bias does not affect our learned estimator since it does not explicitly compute selectivities

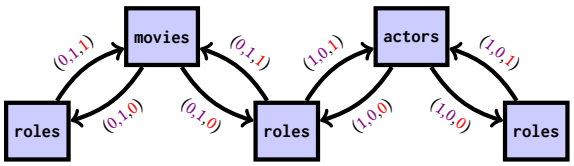



**Figure 4: Join encodings for the query in Figure 3: Foreign key one-hot encodings are colored in violet and the direction component in red.**

as equally weighted averages of the sample bitmaps. Instead, the purpose of the sample bitmaps is to similarly encode predicates that select similar sets of rows, regardless of how the predicates are expressed. The feature-selection algorithm ensures that the encoding has a higher resolution in areas with a lot of training data and lower resolution in areas with less data.

In addition to these bitmaps, we use PostgreSQL's cardinality estimates for table occurrences as input to JGMP, which is a common practice [16, 23] that empirically improves estimation quality.

## 4.2 Join Encodings

In Section 3.4, we have represented queries as labeled join graphs with one directed edge for each join. A message passing neural network always passes messages in the direction of the edges. For our application, passing messages in both directions of a join increases the model's ability to use the join graph's structure. For this reason, we add a reversed edge for each edge in the join graph. Two concatenated components comprise the edge encoding: The first component is a one-hot encoding of the length of all foreign keys in the schema. The second component is a single number 1 if the edge is in the natural direction (fk-pk) or 0 if it is reversed (pk-fk). Figure 4 shows an example of this encoding.

## 4.3 Model Architecture

The JGMP model has three main parts. In the first part, the sample bitmaps are used to compute embeddings for each table occurrence. In the second part, messages are passed over the edges of the join graph to learn about the structure of the query. In the third part, the table occurrence embeddings are aggregated to an embedding of the whole query, which is fed into fully connected layers to produce the final cardinality estimate. Only one model has to be trained for each database since message-passing neural networks can generalize over graphs of varying structures.

In the following equations, we represent multi-layer perceptrons with LeakyReLU activation functions as **MLP**, aggregation functions as **agg**, and concatenations as $\|$. Table 3 shows our choices for the hyperparameters of all modules.

*4.3.1 Table Occurrence Embedding.* Table occurrences $i$ are characterized by their sample bitmaps $s_i$ and a PostgreSQL cardinality estimate $c_i$ (see Section 4.1). Sample bitmaps for occurrences of different tables are semantically unrelated, meaning that any similarity between two sample bitmaps from different tables is coincidental. Additionally, sample bitmap lengths can differ. Due to these factors, we employ separate weights $W_{\tau(i)}$ and biases $b_{\tau(i)}$ when calculating embeddings for occurrences $i$ of tables $\tau(i)$. The

**Table 3: JGMP hyperparameters.**

| Neural Network Architecture | |
|---|---|
| **Hyperparameter** | **Value** |
| sample bitmap $s_i$ length for occurrences $i$ of all tables | $\leq 16$ |
| length of $h_i^0$ for all table occurrences $i$ | 33 |
| layer sizes of $\mathbf{MLP_{edges}}$ | (8) |
| # of message passing layers $N$ | 3 |
| layer sizes of $\mathbf{MLP^n_{mp\_in}}$ for $n \in \{1, 2, ..., N\}$ | (32) |
| # of attention heads of $\mathbf{agg^n_{mp}}$ for $n \in \{1, 2, ..., N\}$ | 4 |
| output size of $\mathbf{agg^n_{mp}}$ for $n \in \{1, 2, ..., N\}$ | 32 |
| layer sizes of $\mathbf{MLP^n_{mp\_out}}$ for $n \in \{1, 2, ..., N\}$ | (32) |
| # of attention heads of $\mathbf{agg_{query}}$ | 8 |
| output size of $\mathbf{agg_{query}}$ | 1024 |
| layer sizes of $\mathbf{MLP_{query}}$ | (256) |
| total # of parameters | $\approx$ 420 000 |

| Training | |
|---|---|
| **Hyperparameter** | **Value** |
| optimizer | Adam [15] |
| learning rate $\lambda$ | 0.0005 |
| # of epochs | 300 |
| batch size | 32 |

PostgreSQL cardinality estimates $c_i$ are then concatenated with these embeddings:

$$h_i^0 = \left(W_{\tau(i)} s_i + b_{\tau(i)}\right) \| \, c_i. \tag{1}$$

*4.3.2 Message Passing.* We obtain the embedding $g_k$ of each edge $k$ in the join graph using a fully connected neural network based on the join encoding $e_k$ (see Section 4.2):

$$g_k = \mathbf{MLP_{edges}}\left(e_k\right). \tag{2}$$

Next, we update node embeddings by applying $N$ custom message passing neural network layers [7, 8, 30]. This approach allows the neural network to leverage the join graph's structure, introducing a useful inductive bias for learning local correlations between neighboring nodes. For $n \in \{1, 2, ..., N\}$ the following equations define the message passing layers:

$$m^n_{j,k} = \mathbf{MLP^n_{mp\_in}}\left(h_j^{n-1} \| \, g_k\right), \tag{3}$$

$$h_i^n = \mathbf{MLP^n_{mp\_out}}\left(h_i^{n-1} \| \underset{(j,k)\in\mathcal{N}(i)}{\mathbf{agg^n_{mp}}}\left(m^n_{j,k}\right)\right). \tag{4}$$

Here, we compute the message $m^n_{j,k}$ from a node $j$ via an edge $k$ to a node $i$ in the $n^{\text{th}}$ message passing layer. The updated node embedding $h_i^n$ is computed based on the old node embedding $h_i^{n-1}$ and the aggregation of messages over the neighborhood $\mathcal{N}(i)$.

*4.3.3 Query Embedding and Final Estimate.* We use skip connections, which have been shown to improve the performance of graph neural networks for a wide range of applications [39], to compute the final node embeddings $h_i$. The final node embeddings are then aggregated to an embedding $q$ for the whole query. The final layer with weights $W_{\text{final}}$ and bias $b_{\text{final}}$ has only one neuron and uses an exponential activation function. This choice is suitable for the last

layer since the true cardinalities are always positive and can differ by many orders of magnitude.

$$h_i = h_i^0 \parallel h_i^1 \parallel \ldots \parallel h_i^N \tag{5}$$

$$q = \mathbf{MLP_{query}}\left(\mathop{\mathbf{agg_{query}}}_{i \in V}(h_i)\right) \tag{6}$$

$$\text{card. est.} = \mathrm{e}^{W_{\text{final}} q + b_{\text{final}}} \tag{7}$$

*4.3.4 Aggregation Function.* For the aggregations in the message passing layers $\mathbf{agg_{mp}}$ and the aggregation of table occurrence embeddings $\mathbf{agg_{query}}$, we use a modified multi-head attention aggregation function [29]. The multi-head attention aggregation function includes learnable parameters and calculates weighted averages of the elements. With these learned weights, it can attend to impactful query features and is thus more expressive than more straightforward aggregation functions such as mean, sum, minimum, and maximum. Given a set $S$ with elements $i$ and their embeddings $x_i$, our aggregation function **agg** is given by

$$\mathop{\mathbf{agg}}_{i \in S}(x_i) = \mathop{\mathbf{MultiHead}}_{i \in S}(x_i) \parallel |S|. \tag{8}$$

Our modification is the concatenation of the set cardinality $|S|$, i.e., for our two uses of aggregations, the degree of nodes in the join graph or the total number of nodes in the join graph. In some cases, the unmodified attention-based aggregation function cannot distinguish between sets of different sizes [42]. Thus, we implement this modification to improve the model's ability to learn from the structure of the join graph.

## 5 SELF-SUPERVISED REGULARIZATION

In this section, we first describe how we train cardinality estimation models in a supervised way. Then, we introduce a new set of constraints on the cardinality estimations and incorporate them into our models by adding a regularization term to our loss function. This additional term can be computed without using true cardinalities.

### 5.1 Supervised Training

We use the mean squared logarithmic error as a loss function to train our models. For a batch containing $n$ subplan queries $q_i$ with a true cardinality $C(q_i)$ and an estimated cardinality $\hat{C}(q_i)$, it is given by

$$\text{loss}_{\text{supervised}} = \frac{1}{n}\sum_{i=1}^{n}\left(\log \hat{C}(q_i) - \log C(q_i)\right)^2. \tag{9}$$

Calculating the loss in a logarithmic space is helpful since cardinalities vary by orders of magnitudes. Without the logarithm, the loss would be dominated by a few subplan queries with the largest cardinalities. With the use of the mean squared logarithmic error, we ensure that the source of the loss is distributed more evenly among the subplan queries.

### 5.2 Cardinality Constraints

In Section 3.1 and Section 4.1, we presented invariances of the cardinality of queries and subsequently incorporated these invariances into our model as hard constraints. In addition, cardinalities of queries have further constraints that go beyond the invariances incorporated into the architecture.

In the following, we will introduce a list of such constraints. The list is not exhaustive and can be extended with additional constraints that cover, for example, different comparison operators. We express the selected constraints using relational algebra.

- **Foreign Key Constraint**
  $C(x \ltimes_{\text{fk}=t.\text{pk}} t) \leq C(x)$, where fk is a foreign key attribute in the relation $x$, $t$ is a base table and there exists a foreign key constraint from attribute fk to attribute pk. If there is a `NOT NULL` constraint on column fk, we get the stronger cardinality constraint of $C(x \ltimes_{\text{fk}=t.\text{pk}} t) = C(x)$.
- **Monotonicity of Inequality Comparisons**
  $C(\sigma_{t.c<v}(x)) \leq C(\sigma_{t.c<u}(x))$, where $c$ is an attribute in relation $x$, and $v$ and $u$ are values in the domain of $c$ with $v < u$. Note that predicates with the comparison operators $>$, $\leq$, and $\geq$ have similar constraints.
- **Law of the Excluded Fourth**
  $C(x) = C(\sigma_a(x)) + C(\sigma_{\neg a}(x)) + C(\sigma_{a \texttt{ IS NULL}}(x))$, where $a$ is a three-valued condition on the relation $x$. If there is a `NOT NULL` constraint on $a$, the cardinality constraint simplifies to $C(x) = C(\sigma_a(x)) + C(\sigma_{\neg a}(x))$, which is equivalent to the law of the excluded third in Boolean logic.
- **Inclusion-Exclusion Principle**
  $C(\sigma_{a \vee b}(x)) + C(\sigma_{a \wedge b}(x)) = C(\sigma_a(x)) + C(\sigma_b(x))$, where $a$ and $b$ are conditions on the relation $x$. If $a$ and $b$ are mutually exclusive, the cardinality constraint simplifies to $C(\sigma_{a \vee b}(x)) = C(\sigma_a(x)) + C(\sigma_b(x))$.

These constraints are well known, but they are, to the best of our knowledge, not explicitly enforced by any existing query-driven learned cardinality estimator. Although, this weakness has been documented with different constraints in the past [10, 31], no solution to enforce them has been proposed. When such constraints are not enforced, they have to be learned by example, increasing the required training data.

### 5.3 Self-Supervised Regularization

We propose a self-supervised regularization term to incorporate these cardinality constraints into the training of models. In addition to the supervised loss (see Equation 9), we calculate a loss term based on a set of constraint examples $E$. A constraint example is an instance of the constraints given in Section 5.2 using specific queries. Examples $e_i = (L_i, \star_i, R_i) \in E$ generally have the form $\left(\sum_{q \in L_i} C(q)\right) \star_i \left(\sum_{q \in R_i} C(q)\right)$, where the $\star_i$-operator is either $=$ or $\leq$, and $L_i$ and $R_i$ are the sets of queries with cardinalities on the left-hand and right-hand sides of the equality or inequality. The loss for such a set of examples $E$ is given by

$$\text{loss}_{\text{constraints}} = \frac{1}{|E|}\sum_{e_i \in E} f(L_i, \star_i, R_i)^2 \tag{10}$$

$$f(L_i, \star_i, R_i) = \begin{cases} g(L_i, R_i) & \text{if } \star_i \text{ is } = \\ \text{ReLU}\left(g(L_i, R_i)\right) & \text{if } \star_i \text{ is } \leq \end{cases} \tag{11}$$

$$g(L_i, R_i) = \log\left(\sum_{q \in L_i} \hat{C}(q)\right) - \log\left(\sum_{q \in R_i} \hat{C}(q)\right). \tag{12}$$

```
SELECT * FROM movies m
WHERE m.year < 1973 OR m.year > 1975;
```

**(a) Query $q_1$**

```
SELECT * FROM movies m
WHERE m.year < 1973;
```

**(b) Query $q_2$**

```
SELECT * FROM movies m
WHERE m.year > 1975;
```

**(c) Query $q_3$**

**Figure 5: A constraint example with $C(q_1) = C(q_2) + C(q_3)$.**

Similar to the supervised loss, we use the logarithm to avoid too much emphasis on queries with large estimated cardinalities. For constraint examples that are equalities, the loss function is the mean squared logarithmic error of the left-hand and the right-hand side of the equation. For constraint examples that are inequalities, we use the ReLU-function, which will be 0 as long as the inequality is satisfied. With the normalizing factor $\frac{1}{|E|}$, the loss is independent of the amount of generated examples.

To illustrate the regularization term, consider the constraint example of the inclusion-exclusion principle in Figure 5. The contribution of this specific example to the loss is the square of

$$f\Big(\{q_1\}, =, \{q_2, q_3\}\Big) = g\left(\{q_1\}, \{q_2, q_3\}\right) \tag{13}$$

$$= \log\left(\hat{C}(q_1)\right) - \log\left(\hat{C}(q_2) + \hat{C}(q_3)\right). \tag{14}$$

While the supervised loss is computed by comparing each estimation of the model $\hat{C}$ to a true cardinality $C$, the regularization term is calculated by comparing estimations of the model with each other. Thus, it can be calculated without gathering ground truths.

Finally, the total loss for training the model is given as the sum of both losses:

$$\text{loss}_{\text{total}} = \text{loss}_{\text{supervised}} + \text{loss}_{\text{constraints}}. \tag{15}$$

The resulting semi-supervised total loss complements the information given by the supervised loss for queries with known cardinalities with the information about the constraints given by the self-supervised loss.

The main advantage of this approach is its flexibility. This loss component can be added to improve the training of *any* neural network architecture for cardinality estimation. Additionally, it is also flexible w.r.t. the constraints. Depending on the query language and the expected workload, further constraints can be added.

*5.3.1 Generating Constraint Examples.* To calculate our loss term, we first have to generate sets of constraint examples. Generally, we work with an infinite space of possible queries whereas the actual workloads only cover a small subset of this space. Thus, while we will not be able to enforce the constraints over the whole space, we can focus on enforcing them for queries in our workload and similar queries.

To generate an example, we first select a random subplan query of a query in our training set. This subplan query can be unlabeled since we do not necessarily know the cardinalities of all subplan queries. Then, we use this query as a starting point to generate a constraint example using pattern matching.

Consider the example shown in Figure 5. With the starting query $q_1$, we generate an example of the inclusion-exclusion principle. The disjunction `m.year < 1973 OR m.year > 1975` matches the pattern $a \vee b$ and thus the queries $q_2$ and $q_3$ are generated. Note, that in this case $a$ and $b$ are mutually exclusive. Therefore, we can omit the query that corresponds to the term $\sigma_{a \wedge b}(x)$. Whenever possible, we choose the stricter versions of the constraints since they carry more information.

We generated two new queries that are similar to our initial training queries for this constraint example. For the next example, we can then randomly choose between using another subplan query in the training set with probability $p$ or one of the two newly generated queries with probability $1 - p$. Setting $p = 1$ restricts the generation to examples involving subplan queries of queries in our training set. Setting $p = 0$ results in a random walk through the space of possible queries. For our experiments we choose $p = \frac{1}{2}$. This choice is a trade-off between the diversity of examples and focusing on examples with queries similar to those in our workload.

Generating these constraint examples is computationally cheap compared to generating data for fully supervised training since calculating their ground truths is not necessary. However, when we only consider the pure training time, excluding the time for generating ground truths, it is a significant cost. Ideally, we would generate a completely new and large set of constraint examples for each epoch of training to minimize the potential for overfitting, but we make trade-offs to keep the training times low. We reuse those 90% of constraint examples from the previous epoch with the largest contributions to our loss. The other 10% are discarded and replaced by newly generated examples. Additionally, in every mini-batch, we use as many constraint examples to calculate $\text{loss}_{\text{constraints}}$ as there are labeled subplan queries used to calculate $\text{loss}_{\text{supervised}}$. In some training scenarios, there can be thousands of labeled subplan queries for a query. In these cases, scaling the number of cardinality examples up accordingly is too expensive so we limit it to 1000 per batch.

## 6 EXPERIMENTAL EVALUATION

We systematically evaluate the performance of the JGMP architecture and the self-supervised regularization using cardinality constraints. This section describes our experimental setup and our results for different experiments, including q-errors, and runtimes.

### 6.1 Experimental Setup

In the following, we describe the execution environment, workloads, training sets, and competitors used in our evaluation.

*6.1.1 Environment.* All experiments were run on a Linux Server with a 4.5 GHz AMD Ryzen Threadripper 3970X 32-Core Processor, 256 GB RAM, and a GeForce RTX 3090 24 GB GPU. We use the GPU for training, but for inference, we only use the CPU to get a fair comparison between PostgreSQL and the machine learning methods. The runtime experiments are executed on PostgreSQL 14.5. We inject the cardinalities using an updated version of Han et al.'s PostgreSQL patch [9].

We use PostgreSQL's default configuration with the following exceptions: We increase the size of the shared buffers to 32 GB and

**Table 4: Query set statistics.**

| Name | Database | Tables | Foreign Keys | Queries | Table Occurrences per Query | String Predicates | Disjunctions | Multiple Occurrences of the Same Table | Non-FK Joins | Non-Inner Joins |
|---|---|---|---|---|---|---|---|---|---|---|
| STATS-CEB | STATS-CEB | 8 | 11 | 146 | 2 - 7 | No | No | No | No | No |
| JOB-light | IMDb | 6 | 5 | 70 | 2 - 5 | No | No | No | No | No |
| JOB | IMDb | 21 | 24 | 113 | 4 - 17 | Yes | Yes | Yes | No | No |
| IMDb-CEB | IMDb | 15 | 16 | 3123 | 6 - 16 | Yes | Yes | Yes | No | No |
| DSB | DSB | 23 | 54 | 360 | 5 - 24 | Yes | Yes | Yes | Yes | Yes |

the working memory to 4 GB since the default values are not appropriate for current hardware. We further disable the GEQO optimizer and join collapsing to search the whole plan space deterministically even for large queries.

*6.1.2 Data and Query Sets.* Table 4 shows an overview of the query sets used in our evaluation. We use the STATS-CEB [9], the DSB [3] and three query sets for the IMDb database: JOB-light [16], JOB [17] and IMDb-CEB [23]. JOB-light and STATS-CEB queries cover smaller schemas and contain simpler predicates and query structures. JOB, IMDb-CEB and DSB contain queries with large join graphs covering more tables. In addition, DSB contains many query elements not directly supported by our methods, which we handle as explained in Section 3.5. For DSB we use a scale factor of 1 GB and generate 10 instances each for the query templates in the "Agg" and "MultiBlock"[1] query sets.

*6.1.3 Competitors.* Our primary focus in this evaluation is on query-driven learned cardinality estimators. Our experiments test three different architectures: MSCN, MSCN-Hybrid, and JGMP. We include MSCN [16] as the current state of the art for query-driven learned cardinality estimation [9, 14] and use the extensions to the encoding described in Flow-Loss [23] to support string predicates and disjunctions. JGMP is our architecture as described in Section 4. MSCN-Hybrid is our adaption of the MSCN using elements from JGMP. In MSCN-Hybrid, our bitmaps from feature-selected rows (see Section 4.1) replace the predicate-based encoding, the LeakyReLU activation function replaces the ReLU activation function, and multi-head attention aggregation (see Equation 8) replaces sum aggregation. We include MSCN-Hybrid to better differentiate between the impact of the high-level neural network architecture and query representation on the one hand and the impact of our encoding and less prominent design choices on the other hand.

The use of self-supervised regularization (see Section 5) is orthogonal to the choice of architecture. For each of the three architectures, we include one version without the regularization (MSCN, MSCN-Hybrid, JGMP) and one with the regularization (MSCN+Reg., MSCN-Hybrid+Reg., JGMP+Reg.).

Additionally, we use the cardinality estimates of the unsupervised data-driven methods BayesCard [34], DeepDB [12], FLAT [45], and NeuroCard [38] provided in the Git repository[2] for the STATS benchmark [9]. These unsupervised methods only apply to the STATS-CEB and the JOB-light workloads since they do not support the string predicates in the other workloads.

[1]We exclude Query Template 81 from our experiments as most of its instances time out even for true cardinalities due to inconsistent behavior of the PostgreSQL optimizer.
[2]https://github.com/Nathaniel-Han/End-to-End-CardEst-Benchmark
(last accessed: 12/12/2023)

*6.1.4 Training with Subplan Queries.* Training sets for query-driven models do not only contain cardinalities for the whole queries but also for their subplan queries. The number of subplan queries for each query varies greatly with the number of table occurrences. For the JOB, the number ranges from 12 for the simplest queries to over 14,000 for the most complex query. Therefore, we test two options for training the query-driven models with subplan queries.

In the first option, we add *all subplan queries* to the training set. This option is in line with evaluations of query-driven models on the JOB in other publications [23, 24]. However, this option makes highly unrealistic assumptions about the availability of training data in real-world scenarios. It requires the execution of potentially large amounts of subplan queries for each query in the training set, as each execution only provides cardinalities for a few subplan queries. In addition, we also have to gather cardinalities for costly subplan queries with high cardinalities that the query optimizer otherwise would never need to execute.

In the second option, we only train the model with *PG-selected subplan queries*. PG-selected subplan queries are defined as those subplan queries that are implemented by a subplan of PostgreSQL's execution plan. The cardinalities of PG-selected subplan queries can be gathered with a single execution of PostgreSQL's plan. However, our definition of PG-selected subplan queries excludes subplans containing pushed-down filters such as a table indexed by a nested loop join since the cardinality of the subplan without the pushed down filter cannot be determined from the execution.

A query with $n$ table occurrences can have up to $n$ single-table PG-selected subplan queries and $n-1$ PG-selected subplan queries with joins, totaling a maximum number of $2n-1$ PG-selected subplan queries. This option is much more realistic because cardinalities for these subplan queries are accumulated incidentally within regular system operation. However, it provides only a small and biased sample of all subplan queries since query optimizers favor subplans with low estimated cardinalities.

*6.1.5 Advanced Mini-Batching.* Our models are implemented using PyTorch Geometric [6]. We use the package's functionality for advanced mini-batching to ensure fast training and inference times. Advanced mini-batching is a way to handle varying tensor sizes. In our case, the tensor sizes vary because we have different numbers of table occurrences and joins in each query.

Naively, each tensor can be padded to a maximum size, while masking ensures that the padding does not affect the estimates. However, this approach is computationally inefficient because a cardinality estimation for a small join graph is as expensive as the estimation for the largest join graph.

Advanced mini-batching is a more efficient alternative. By concatenating the tensors of the join graphs in a batch and incrementing edge index tensors and batch index tensors appropriately, the set of join graphs in a batch is processed as one large disconnected join graph. This transformation enables fast parallel computations with little overhead.

For the most critical applications, we can get additional speedups exploiting shared computations: In practice, we rarely want to estimate only one cardinality per query. Instead, cost models require cardinalities for a potentially large set of subplan queries for each query. Similarly, in the training set, we have access to multiple subplan queries for each query. Table occurrences and joins are shared between these subplan queries. Thus, we can accelerate inference and training and reduce memory consumption by sharing computations up to a certain point. The shared computations include the calculation of the encodings, the computation of the edge embeddings $g_k$, and the first table occurrence embeddings $h_i^0$. From this point on, the computations cannot be shared anymore since they depend on the join graph structure, which varies between the subplan queries. The shared embeddings are then replicated for each subplan query that needs them using an index-select operation.

While this implementation provides significant speedups, it is restrictive during training since the subplan queries of a query cannot be spread across mini-batches. Thus, our batch size hyperparameter signifies the number of queries in a mini-batch, while the number of contained subplan queries varies between mini-batches.

For a fair comparison, we also implemented advanced mini-batching and shared computations for MSCN and MSCN-Hybrid.

## 6.2 Cross-Validation Experiments

We perform five-fold cross-validation experiments on the STATS-CEB, JOB-light, JOB and DSB. For the benchmarks containing few queries (STATS-CEB, JOB-light, JOB), we repeat this ten times to reduce the variance in our evaluation, giving 50 trained models per method, training set, and benchmark. The training sets either contain all or only PG-selected subplan queries. For JOB, we also distinguish between assigning queries one-by-one into training and test sets and assigning groups of queries with the same template together into training and test sets. This grouped assignment is more challenging than the query-by-query assignment as the test set does not contain any queries derived from templates in the training set. Table 5 shows the results of the cross-validation experiments in terms of q-error [21] and execution time of the plan selected by the PostgreSQL query optimizer using the cardinality estimates.

The best query-driven method, JGMP+Reg., is on par in terms of q-errors and runtimes with the best unsupervised methods on benchmarks where those are applicable, regardless of whether JGMP+Reg. is trained with all or only PG-selected subplan queries. In addition to their limited scope, Table 6 shows that unsupervised models are generally larger and scale poorly with schema size.

Comparing the different benchmarks, we notice that the benchmarks are not equally difficult. The simplest benchmark is JOB-light with small q-errors and similar runtimes for all methods. STATS-CEB is a more challenging benchmark with higher q-errors and a 28% execution time increase from true cardinalities to PostgreSQL

**Table 5: Results of cross-validation experiments.**

| Train. Set | Method | q-Error | | | Execution Time [s] | | |
|---|---|---|---|---|---|---|---|
| | | Med. | G.M. | Mean | Med. | G.M. | Mean |
| STATS-CEB | | | | | | | |
| | TrueCard | 1 | 1 | 1 | 0.18 | 0.36 | 17.11 |
| | PostgreSQL | 1.64 | 3.27 | 584.9 | 0.24 | 0.48 | 21.87 |
| data-driven | BayesCard | **1.18** | **2.31** | **18.6** | 0.20 | **0.40** | 17.56 |
| | DeepDB | 1.98 | 3.26 | 133.4 | 0.22 | 0.44 | 19.44 |
| | FLAT | 1.68 | 2.62 | 112.0 | **0.19** | 0.42 | **17.44** |
| | NeuroCard | 951.23 | $2 \cdot 10^3$ | $2 \cdot 10^9$ | 0.35 | 0.71 | 43.43 |
| all subplan queries | MSCN | 2.76 | 4.44 | 735.6 | 0.19 | 0.43 | 20.93 |
| | MSCN+Reg. | 2.20 | 3.36 | 286.0 | 0.19 | 0.41 | 21.76 |
| | MSCN-Hyb. | 2.17 | 3.32 | 181.0 | 0.20 | 0.42 | 19.68 |
| | MSCN-Hyb.+Reg. | 2.13 | 3.12 | 95.5 | 0.20 | 0.41 | 19.21 |
| | JGMP | **1.34** | **1.71** | **3.3** | **0.18** | **0.37** | **17.45** |
| | JGMP+Reg. | 1.37 | 1.77 | 4.1 | **0.18** | **0.37** | 17.59 |
| PG-selected subplan queries | MSCN | 2.77 | 5.30 | 3437.6 | 0.20 | 0.49 | 21.54 |
| | MSCN+Reg. | 2.47 | 4.42 | 1808.6 | 0.2 | 0.47 | 28.56 |
| | MSCN-Hyb. | 2.10 | 3.76 | 3110.3 | 0.21 | 0.49 | 29.06 |
| | MSCN-Hyb.+Reg. | 2.38 | 3.87 | 1105.8 | 0.21 | 0.46 | 20.88 |
| | JGMP | **1.58** | **2.38** | 85.0 | **0.19** | **0.40** | 19.07 |
| | JGMP+Reg. | 1.60 | 2.43 | **52.9** | 0.20 | **0.40** | **18.64** |
| JOB-light | | | | | | | |
| | TrueCard | 1 | 1 | 1 | 0.97 | 0.57 | 7.60 |
| | PostgreSQL | 1.27 | 1.70 | 2.9 | 1.10 | 0.59 | 7.74 |
| data-driven | BayesCard | **1.14** | 1.40 | 1.9 | **0.91** | **0.57** | **7.64** |
| | DeepDB | 1.66 | 1.66 | 2.2 | 1.06 | 0.62 | 7.67 |
| | FLAT | 1.23 | **1.39** | **1.7** | 0.93 | 0.61 | **7.64** |
| | NeuroCard | 2.15 | 2.74 | 4.7 | 1.11 | 0.59 | 7.79 |
| all subplan queries | MSCN | 2.25 | 3.00 | 9.2 | 1.06 | 0.68 | 7.88 |
| | MSCN+Reg. | 2.22 | 3.16 | 14.7 | 1.03 | 0.67 | 7.84 |
| | MSCN-Hyb. | 1.92 | 2.31 | 4.9 | 1.02 | 0.63 | 8.05 |
| | MSCN-Hyb.+Reg. | 1.75 | 2.22 | 4.8 | 1.05 | 0.63 | 7.96 |
| | JGMP | **1.12** | **1.27** | **1.6** | **0.98** | **0.58** | **7.63** |
| | JGMP+Reg. | 1.13 | 1.30 | 1.9 | **0.98** | 0.59 | 7.66 |
| PG-selected subplan queries | MSCN | 2.59 | 3.69 | 12.6 | 1.23 | 0.75 | 10.09 |
| | MSCN+Reg. | 2.25 | 3.23 | 13.3 | 1.21 | 0.68 | 7.95 |
| | MSCN-Hyb. | 1.89 | 2.61 | 6.7 | 1.15 | 0.62 | **7.74** |
| | MSCN-Hyb.+Reg. | 1.94 | 2.54 | 5.9 | 1.10 | 0.62 | 7.85 |
| | JGMP | 1.25 | 1.52 | **2.1** | **1.04** | **0.61** | 7.75 |
| | JGMP+Reg. | **1.24** | **1.47** | **2.1** | **1.04** | **0.61** | 7.77 |
| JOB | | | | | | | |
| | TrueCard | 1 | 1 | 1 | 0.17 | 0.18 | 0.45 |
| | PostgreSQL | 108.00 | 149.09 | 46991.9 | 0.23 | 0.30 | 1.03 |
| all subplan queries | MSCN | 3.40 | 4.87 | 325.4 | 0.28 | 0.30 | 0.98 |
| | MSCN+Reg. | 3.62 | 5.17 | 51.3 | 0.25 | 0.26 | 0.86 |
| | MSCN-Hyb. | 2.69 | 3.89 | 31.8 | 0.25 | 0.23 | 0.80 |
| | MSCN-Hyb.+Reg. | 2.70 | 3.83 | 34.3 | 0.22 | **0.21** | 0.79 |
| | JGMP | 2.18 | 3.01 | 33.1 | **0.21** | 0.22 | 0.67 |
| | JGMP+Reg. | **2.15** | **2.88** | **19.1** | **0.21** | **0.21** | **0.65** |
| PG-selected subplan queries | MSCN | 10.18 | 16.86 | 1534.2 | 0.29 | 0.38 | 2.51 |
| | MSCN+Reg. | 8.98 | 13.70 | 294.0 | 0.23 | 0.27 | 1.16 |
| | MSCN-Hyb. | 6.80 | 11.08 | 356.1 | 0.21 | 0.23 | 0.67 |
| | MSCN-Hyb.+Reg. | 5.82 | 8.94 | 249.1 | **0.20** | 0.22 | 0.65 |
| | JGMP | 4.84 | 7.46 | 126.9 | 0.21 | 0.23 | 0.70 |
| | JGMP+Reg. | **4.35** | **6.48** | **79.0** | **0.20** | **0.21** | **0.57** |
| grouped all subplan queries | MSCN | 5.68 | 8.35 | 1127.7 | 0.28 | 0.32 | 1.18 |
| | MSCN+Reg. | 5.53 | 7.98 | 994.0 | 0.25 | 0.27 | 0.88 |
| | MSCN-Hyb. | 5.52 | 11.88 | 684.7 | 0.24 | 0.25 | 0.81 |
| | MSCN-Hyb.+Reg. | 5.93 | 12.75 | 739.2 | 0.24 | 0.23 | 0.88 |
| | JGMP | **4.01** | **6.93** | **254.0** | **0.21** | 0.23 | 0.70 |
| | JGMP+Reg. | 4.57 | 9.84 | 735.1 | **0.21** | **0.22** | **0.66** |
| grouped PG-sel. subplan queries | MSCN | 10.78 | 17.59 | 16476.7 | 0.32 | 0.45 | 7.02 |
| | MSCN+Reg. | 10.02 | 15.49 | 41778.3 | 0.25 | 0.31 | 1.34 |
| | MSCN-Hyb. | 10.86 | 20.00 | 2764.8 | 0.23 | 0.26 | 0.97 |
| | MSCN-Hyb.+Reg. | 11.90 | 22.35 | 2683.5 | 0.23 | 0.25 | 0.83 |
| | JGMP | **6.93** | **12.09** | **450.3** | 0.22 | 0.24 | 0.67 |
| | JGMP+Reg. | 7.60 | 15.59 | 1614.3 | **0.21** | **0.22** | **0.59** |
| DSB | | | | | | | |
| | TrueCard | 1 | 1 | 1 | 0.82 | 0.65 | 12.24 |
| | PostgreSQL | 2.04 | 3.14 | 184.1 | 0.77 | 0.77 | 12.58 |
| all subplan queries | MSCN | 1.85 | 2.71 | **10.6** | 0.87 | 0.80 | 17.37 |
| | MSCN+Reg. | 1.82 | 11.8 | 2.58 | 0.86 | 0.79 | 13.86 |
| | MSCN-Hyb. | **1.60** | **2.27** | 44.2 | 0.89 | 0.79 | **12.60** |
| | MSCN-Hyb.+Reg. | 1.85 | 2.62 | 72.1 | 0.86 | 0.79 | 12.79 |
| | JGMP | 1.94 | 2.73 | 70.9 | 0.86 | **0.66** | 12.71 |
| | JGMP+Reg. | 1.76 | 2.47 | 63.0 | **0.84** | 0.67 | 12.91 |
| PG-selected subplan queries | MSCN | 17.85 | 28.83 | 2310.4 | 0.95 | 0.93 | 17.97 |
| | MSCN+Reg. | **9.59** | **12.55** | **117.7** | 0.93 | 0.87 | 17.80 |
| | MSCN-Hyb. | 100.31 | 105.29 | 3809.0 | 0.93 | 0.90 | 21.84 |
| | MSCN-Hyb.+Reg. | 26.86 | 32.07 | 620.2 | 0.96 | 0.87 | 17.76 |
| | JGMP | 45.70 | 52.10 | 1587.7 | 0.91 | 0.77 | 17.68 |
| | JGMP+Reg. | 13.36 | 15.03 | 120.3 | **0.82** | **0.67** | **12.50** |

**Table 6: Model sizes [MB].**

| Method | STATS-CEB | JOB-light | JOB | DSB |
|---|---|---|---|---|
| BayesCard | 5.9 | 1.6 | n/a | n/a |
| DeepDB | 162 | 34 | n/a | n/a |
| FLAT | 310 | 3.4 | n/a | n/a |
| NeuroCard | 337 | 6.9 | n/a | n/a |
| MSCN | 0.4 | 0.4 | 0.5 | 0.7 |
| MSCN-Hybrid | 0.5 | 0.4 | 0.5 | 0.5 |
| JGMP | 1.7 | 1.7 | 1.7 | 1.7 |

estimates. In the STATS-CEB benchmark, we observe a high variance in execution times between the queries in the benchmark. For example, the three longest-running queries comprise 72% of the total PostgreSQL execution time of all 146 queries. Since a few queries dominate the mean execution times, we also report the geometric mean (G.M.) as a more robust measure. JOB is a more challenging benchmark with high q-errors and a 129% execution time increase from true cardinalities to PostgreSQL estimates.

The results support the hypothesis that training with only PG-selected subplan queries is more challenging than training with all subplan queries. We generally observe larger q-errors for all methods on all benchmarks and longer execution times, especially on JOB and DSB. Similarly, the workload shift resulting from assigning queries to training and test sets in groups of similar queries also provides an additional challenge on JOB. However, we can observe that different models are affected by these two additional difficulties to a different degree. While the geometric mean execution time of MSCN deteriorates from 0.30 s to 0.45 s, the geomeric mean execution time of JGMP only increases from 0.22 s to 0.24 s.

Using the self-supervised regularization term during training further improves the sample efficiency. While the benefits of the regularization are less obvious for the easier benchmarks, there are clear improvements on JOB, especially when training only with PG-selected subplan queries. This improvement is not surprising since the regularization term will have the most impact when we need more training data. The MSCN-Hybrid architecture generally ranks between the MSCN and JGMP architectures across different benchmarks and training options. This result shows that both the overall architecture with the join graph representation of queries, as well as the combination of our proposed encoding of complex predicates and more minor innovations, contribute to the improved performance of JGMP.

On DSB, we observe only a 3% increase in mean execution times from true cardinalities for directly supported subplan queries to PostgreSQL estimates, indicating a limited potential to improve the plans of the few longest-running DSB queries. However, the respective 18% increase in geometric means shows considerable potential for relative runtime improvements over the whole query set. Using all subplan queries as training data, all models can reduce the q-errors compared to PostgreSQL. However, only JGMP can improve PostgreSQL's geometric mean execution time. Training with only PG-selected queries is especially challenging on this benchmark since expressions that are not directly supported can further reduce the size of the training set, which is reflected in the q-errors of all models. In this scenario, the consistency between the estimates enforced by our regularization proves beneficial again, as JGMP+Reg. is the only model that outperforms PostgreSQL's execution times. Overall, the experiments on DSB show that query-driven learned cardinality estimation can be beneficial even on workloads containing not directly supported expressions.

**Table 7: Mean cardinality estimation inference time [s] per query including all of its subplan queries.**

| Method | STATS-CEB | JOB-light | JOB | DSB |
|---|---|---|---|---|
| MSCN | 0.0045 | 0.0042 | 0.0493 | 0.0239 |
| MSCN-Hybrid | 0.0069 | 0.0064 | 0.0610 | 0.0281 |
| JGMP | 0.0104 | 0.0099 | 0.0729 | 0.0283 |

**Table 8: Mean training times [min].**

| Method | All Subplan Queries | PG-Selected Subplan Queries |
|---|---|---|
| MSCN | 0.2 | 0.1 |
| MSCN+Reg. | 6.7 | 2.4 |
| MSCN-Hybrid | 1.9 | 1.6 |
| MSCN-Hybrid+Reg. | 8.6 | 4.1 |
| JGMP | 2.4 | 2.0 |
| JGMP+Reg. | 7.3 | 3.9 |

Table 7 shows the inference times per query to gather all subplan query cardinality estimates required by the PostgreSQL query optimizer. The inference times do not depend on the training set or the regularization. We observe that the inference times on JOB and DSB are higher than those on STATS-CEB and JOB-light since the queries in the former workloads are much more complex and, therefore, contain more subplan queries that need to be estimated. JGMP's improved estimates come at the cost of higher inference times compared to MSCN due to the added cost of the more complex computations in the message passing and aggregation layers. Overall, the inference times for all model architectures on all benchmarks are short compared to the differences in execution times, showing that using learned cardinality estimation can provide a net benefit. The efficient implementation of the models using advanced mini-batching and shared computations between subplan queries, as explained in Section 6.1.5, is a major contributor to the low inference times.

Table 8 shows the training times for all methods on JOB. Overall training times for any method do not exceed 9 minutes due to the small training sets and efficient implementation. However, we observe an expected increase in training times in the more complex model architectures and an additional increase when training with regularization. Arguably, the differences in the estimation quality are large enough to justify this additional overhead. The increased training time for the regularization is mainly due to the generation of constraint examples. Notably, more constraint examples are generated for the training with all subplan queries, which is also reflected in the training times. Using the regularization term is a trade-off between training times and quality of estimates that is especially appealing when there is only minimal training data.

## 6.3 Varying Training Set Size & Workload Shift

We further investigate how the amount of available training data influences the performance of the query-driven estimators using the IMDb-CEB containing over 3000 queries. Specifically, we randomly assign subsets of sizes ranging from 128 to 2048 to the training set. We repeat this process with different random subsets resulting in

**Table 9: Results of the workload shift experiment.**

| Train. Set | Method | q-Error | | | Execution Time [s] | | |
|---|---|---|---|---|---|---|---|
| | | Med. | G.M. | Mean | Med. | G.M. | Mean |
| | TrueCard | 1 | 1 | 1 | 0.17 | 0.18 | 0.45 |
| | PostgreSQL | 108.00 | 149.09 | 46991.9 | 0.23 | 0.30 | 1.03 |
| IMDB-CEB pg-selected subplan queries | MSCN | 14.12 | 23.45 | 31606.3 | 0.42 | 0.56 | 4.21 |
| | MSCN+Reg. | 11.28 | 17.70 | 2628.3 | 0.43 | 0.40 | 1.78 |
| | MSCN-Hyb. | 8.89 | 14.33 | **2063.4** | 0.30 | 0.35 | 1.31 |
| | MSCN-Hyb.+Reg. | **8.12** | **14.28** | 3730.0 | **0.26** | **0.31** | 1.01 |
| | JGMP | 10.07 | 17.84 | 34782.5 | 0.31 | 0.35 | 2.49 |
| | JGMP+Reg. | 9.79 | 17.23 | 2255.8 | 0.31 | **0.31** | **0.92** |

10 trained models for each method and training set size. In these experiments, we train the models exclusively with PG-selected subplan queries since it is more realistic and challenging than training with all subplan queries.

Figure 6 shows the results of this experiment on the IMDb-CEB. JGMP is the best-performing architecture regarding q-errors and execution times across all training set sizes. However, it is notable that the differences between the architectures are least pronounced for the largest and increase for smaller training sets. This behavior further supports our claim that the JGMP architecture is more sample-efficient than its competitors. Similarly, the regularization term provides the greatest benefits when training with few queries. Interestingly, we observe that for large training sets, the variants with regularization have larger q-errors but much lower execution times. While a fully supervised loss function tries to minimize the q-error more directly, the self-supervised regularization term ensures that the estimates are mutually consistent. While this consistency between subplan query estimates is not necessarily reflected in lower q-errors, it is crucial for generating good plans. This effect underscores the recent critiques of q-errors as the only metric for cardinality estimation [9, 23] and the need for injecting cardinality estimates into query optimizers to gather execution times.

We use the models trained on IMDb-CEB with the largest training set size of 2048 to conduct a workload shift experiment. These trained models are evaluated without any modifications on the JOB queries, referencing tables and columns that do not appear in the IMDb-CEB. Table 9 shows the results of this experiment. Regularization is valuable in this highly challenging scenario. Models trained with regularization use the constraint examples to gather

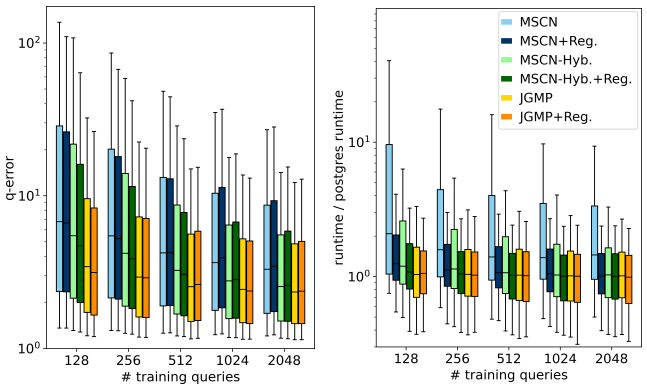



**Figure 6: Performance of different architectures on IMDb-CEB for varying training set sizes.**

**Table 10: Results of the predicate encoding ablation study on JOB using JGMP with regularization.**

| Train. Set | Method | q-Error | | | Execution Time [s] | | |
|---|---|---|---|---|---|---|---|
| | | Med. | G.M. | Mean | Med. | G.M. | Mean |
| | TrueCard | 1 | 1 | 1 | 0.17 | 0.18 | 0.45 |
| | PostgreSQL | 108.00 | 149.09 | 46991.9 | 0.23 | 0.30 | 1.03 |
| PG-sel. subplan queries | 16 FS (default) | **4.35** | **6.48** | **79.0** | **0.20** | **0.21** | **0.57** |
| | 16 FS w/o PG-est. | 5.45 | 8.21 | 649.6 | 0.22 | 0.25 | 0.99 |
| | No samples | 6.15 | 9.47 | 151.9 | 0.24 | 0.27 | 0.81 |
| | 16 random | 5.2 | 7.94 | 154.3 | 0.22 | 0.26 | 0.82 |
| | 1000 random | 5.14 | 7.71 | 27036.5 | 0.21 | 0.25 | 0.86 |

information on queries outside the original training set. Notably, both architectures that use bitmaps from feature-selected samples perform similarly well, indicating that this approach is beneficial for generalization and against overfitting.

## 6.4 Ablation Study

We conduct an ablation study to assess the influence of our predicate encoding in more detail. The results of this ablation study using cross-validation on the JOB trained with PG-selected subplan queries are shown in Table 10. We compare the default JGMP configuration with 16 feature-selected samples (16 FS) to four different variants: "16 FS w/o PG-est." is the only variant omitting the PostgreSQL estimates in the base table encodings, "no samples" omits the samples completely, "16 random" and "1000 random" use random samples of the respective sizes. The feature-selected samples provide a clear benefit compared to all other variants regarding q-errors and execution times. Similarly, the PostgreSQL estimates in the encoding also improve the estimates.

# 7 CONCLUSION

We showed that the JGMP architecture and self-supervised regularization based on cardinality constraints improve the sample efficiency of learned query-driven cardinality estimators. We also showed that learned query-driven estimators can outperform classical cardinality estimators and come close to using true cardinalities in terms of execution times, even with training data from only 100 query executions. These findings underscore the great potential of geometric deep learning in query optimization. In query optimization, we are blessed with highly structured data. Incorporating this structure into our models is critical to the success of this line of research.

While our evaluation shows great promise and the efficient implementation of our models reduces inference times, learned cardinality estimation, in general, still comes at the cost of additional overheads. Our experiments include many examples where these overheads pay off. However, there will always be fast-executing queries where classical cardinality estimators perform well enough, and the benefit of learned cardinality estimation is not worth the cost. For this reason, it is critical future work to design inexpensive heuristics for database systems to determine for which queries, workloads, or databases to use learned cardinality estimation.


# ACKNOWLEDGMENTS

This work is in part supported by Grant No. GR 4497/5 of the Deutsche Forschungsgemeinschaft (DFG).

# REFERENCES


[1] Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. 2021. Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges. https://doi.org/10.48550/ARXIV.2104.13478

[2] Tianyi Chen, Jun Gao, Hedui Chen, and Yaofeng Tu. 2023. LOGER: A Learned Optimizer towards Generating Efficient and Robust Query Execution Plans. *Proc. VLDB Endow.* 16, 7 (2023), 1777–1789. https://doi.org/10.14778/3587136.3587150

[3] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek R. Narasayya. 2021. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems. *Proc. VLDB Endow.* 14, 13 (2021), 3376–3388. https://doi.org/10.14778/3484224.3484234

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

[5] Ahmed K. Farahat, Ali Ghodsi, and Mohamed S. Kamel. 2013. Efficient greedy feature selection for unsupervised learning. *Knowl. Inf. Syst.* 35, 2 (2013), 285–310. https://doi.org/10.1007/s10115-012-0538-1

[6] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. *CoRR* abs/1903.02428 (2019). arXiv:1903.02428

[7] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *Proceedings of the International Conference on Machine Learning, ICML 2017 (Proceedings of Machine Learning Research, Vol. 70)*. 1263–1272.

[8] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*. 1024–1034.

[9] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Tan Wei Liang, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *Proc. VLDB Endow.* 15, 4 (2021), 752–765. https://doi.org/10.14778/3503585.3503586

[10] Haoyu He, Tianhao Wei, Huan Zhang, Changliu Liu, and Cheng Tan. 2022. Characterizing Neural Network Verification for Systems with NN4SYSBENCH. In *Workshop on Formal Verification of Machine Learning*.

[11] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction. *Proc. VLDB Endow.* 15, 11 (2022), 2361–2374. https://doi.org/10.14778/3551793.3551799

[12] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *Proc. VLDB Endow.* 13, 7 (2020), 992–1005. https://doi.org/10.14778/3384345.3384349

[13] Alan Jovic, Karla Brkic, and Nikola Bogunovic. 2015. A review of feature selection methods with applications. In *International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2015*. 1200–1205. https://doi.org/10.1109/MIPRO.2015.7160458

[14] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. 2022. Learned Cardinality Estimation: An In-depth Study. *SIGMOD Rec.*, 1214–1227. https://doi.org/10.1145/3514221.3526154

[15] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015*.

[16] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *Proc. Conf. on Innovative Data Systems Research (CIDR)*.

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

[18] Jie Liu, Wenqian Dong, Dong Li, and Qingqing Zhou. 2021. Fauce: Fast and Accurate Deep Ensembles with Uncertainty for Cardinality Estimation. *Proc. VLDB Endow.* 14, 11 (2021), 1950–1963. https://doi.org/10.14778/3476249.3476254

[19] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2022. Bao: Making Learned Query Optimization Practical. *SIGMOD Rec.* 51, 1 (2022), 6–13. https://doi.org/10.1145/3542700.3542703

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

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

[22] Magnus Müller, Lucas Woltmann, and Wolfgang Lehner. 2023. Enhanced Featurization of Queries with Mixed Combinations of Predicates for ML-based Cardinality Estimation. In *Proceedings International Conference on Extending Database Technology, EDBT 2023*. 273–284. https://doi.org/10.48786/edbt.2023.22

[23] Parimarjan Negi, Ryan C. Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *Proc. VLDB Endow.* 14, 11 (2021), 2019–2032. https://doi.org/10.14778/3476249.3476259

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

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

[26] Hoifung Poon and Pedro M. Domingos. 2011. Sum-Product Networks: A New Deep Architecture. In *UAI 2011, Proceedings of the Conference on Uncertainty in Artificial Intelligence*. 337–346.

[27] Saúl Solorio-Fernández, Jesús Ariel Carrasco-Ochoa, and José Fco. Martínez-Trinidad. 2020. A review of unsupervised feature selection methods. *Artif. Intell. Rev.* 53, 2 (2020), 907–948. https://doi.org/10.1007/s10462-019-09682-y

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

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*. 5998–6008.

[30] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations, ICLR 2018*.

[31] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready For Learned Cardinality Estimation? *Proc. VLDB Endow.* 14, 9 (2021), 1640–1654. https://doi.org/10.14778/3461535.3461552

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

[33] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. 2022. FactorJoin: A New Cardinality Estimation Framework for Join Queries. *CoRR* (2022). https://doi.org/10.48550/arXiv.2212.05526 arXiv:2212.05526

[34] Ziniu Wu and Amir Shaikhha. 2020. BayesCard: A Unified Bayesian Framework for Cardinality Estimation. *CoRR* (2020). arXiv:2012.14743

[35] Ziniu Wu, Pei Yu, Peilun Yang, Rong Zhu, Yuxing Han, Yaliang Li, Defu Lian, Kai Zeng, and Jingren Zhou. 2022. A Unified Transferable Model for ML-Enhanced DBMS. In *Conference on Innovative Data Systems Research, CIDR 2022*.

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

[37] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *Proc. VLDB Endow.* 14, 1 (2020), 61–73. https://doi.org/10.14778/3421424.3421432

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

[39] Jiaxuan You, Zhitao Ying, and Jure Leskovec. 2020. Design Space for Graph Neural Networks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020*.

[40] Xiang Yu, Chengliang Chai, Guoliang Li, and Jiabin Liu. 2022. Cost-based or Learning-based? A Hybrid Query Optimizer for Query Plan Selection. *Proc. VLDB Endow.* 15, 13 (2022), 3924–3936. https://doi.org/10.14778/3565838.3565846

[41] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabás Póczos, Ruslan Salakhutdinov, and Alexander J. Smola. 2017. Deep Sets. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*. 3391–3401.

[42] Shuo Zhang and Lei Xie. 2020. Improving Attention Mechanism in Graph Neural Networks via Cardinality Preservation. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI 2020*. 1395–1402. https://doi.org/10.24963/ijcai.2020/194

[43] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: A Tree Transformer Model for Query Plan Representation. *Proc. VLDB Endow.* 15, 8 (2022), 1658–1670. https://doi.org/10.14778/3529337.3529349

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

[45] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2021. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation. *Proc. VLDB Endow.* 14, 9 (2021), 1489–1502. https://doi.org/10.14778/3461535.3461539