Fauce: Fast and Accurate Deep Ensembles with Uncertainty for Cardinality Estimation

Jie Liu[†], Wenqian Dong[†], Qingqing Zhou[*], Dong Li[†]
[†]University of California, Merced [*]Tencent
{jliu279,wdong5,dli35}@ucmerced.edu,hewanzhou@tencent.com

ABSTRACT

Cardinality estimation is a fundamental and critical problem in databases. Recently, many estimators based on deep learning have been proposed to solve this problem and they have achieved promising results. However, these estimators struggle to provide accurate results for complex queries, due to not capturing real inter-column and inter-table correlations. Furthermore, none of these estimators contain the uncertainty information about their estimations. In this paper, we present a join cardinality estimator called Fauce. Fauce learns the correlations across all columns and all tables in the database. It also contains the uncertainty information of each estimation. Among all studied learned estimators, our results are promising: (1) Fauce is a light-weight estimator, it has 10× faster inference speed than the state of the art estimator; (2) Fauce is robust to the complex queries, it provides 1.3×-6.7× smaller estimation errors for complex queries compared with the state of the art estimator; (3) To the best of our knowledge, Fauce is the first estimator that incorporates uncertainty information for cardinality estimation into a deep learning model.

PVLDB Reference Format:
Jie Liu[†], Wenqian Dong[†], Qingqing Zhou[*], Dong Li[†]. Fauce: Fast and Accurate Deep Ensembles with Uncertainty for Cardinality Estimation. PVLDB, 14(11): 1950-1963, 2021.
doi:10.14778/3476249.3476254

1 INTRODUCTION

Cardinality estimation is fundamental and critical in databases. It is widely applied to query optimization, query processing approximation, database tuning, etc. For example, the query optimizer uses the results of the cardinality estimation to determine the best execution plans. However, the cardinality estimation can be challenging. In some cases with complex queries where there are correlated columns or large number of joins, the accuracy of the cardinality estimation drops dramatically.

Recently, the researchers have been actively using the machine learning technique to estimate the cardinality [11, 17–19, 21, 24, 56–58]. These approaches can be mainly classified as two types: *data-driven* and *query-driven* estimators. Both of them have limitation.

The data-driven estimators such as Naru [57], NeuroCard [56], and MADE [18] leverage the *deep autoregressive (AR) models* [9, 14] to approximate the data distribution of a table or joint tables. Deep AR models capture the data distribution of a table by multiplying the estimated data distribution of each column, based on an implicit assumption that each column is dependent on all the previous columns. However, such an assumption is oversimplified. In DBMS, the dependent relationship between columns can be complex. For example, some columns are independent from each other, while others have correlation. As a result, the deep AR models result in large errors for those queries on correlated columns. Furthermore, recent study [52] reveals that the data-driven estimators tend to output large errors when the data are skewed.

The query-driven estimators [10, 11, 17, 24, 25, 39] rely on some regression models to properly learn function mapping between queries and cardinalities. Since the input of the regression models are real-valued vectors, the query-driven estimator must use a *query featurization* method to convert the queries into feature vectors. Those vectors should contain informative features of the queries. A good query featurization method is critical, because it can generate highly informative feature vectors, which are useful to improve the accuracy of the regression models. The existing query featurization methods [11, 18, 24, 46] apply techniques like one-hot encoding, binary encoding [43], basic statistics, or bitmap [5] to convert queries into feature vectors. While those methods are simple to use, they cannot capture the fine-grained correlations between columns and between tables. As a result, the feature vectors generated by the existing query featurization methods are not informative enough, and using such feature vectors for cardinality estimation can be erroneous. Furthermore, the existing query featurization methods focus on static data [52]. However, in a dynamic scenario where the data are dynamically updated, the existing methods cannot adapt to the new data, hence degrading the accuracy of cardinality estimation significantly.

Furthermore, both query-driven and data-driven estimators do not give any quantification of uncertainty or confidence level of the estimation. The estimation is used in database based on an implicit assumption that the estimation is always safe to be used. However, this assumption is not always valid, and using error estimation can be problematic. For example, an erroneous estimation, when used by the query optimizer, can lead to bad execution plans. A better estimation approach is to output the estimated cardinality together with the corresponding uncertainty. Based on the uncertainty, DBMS can determine when to actually trust the estimator and use its estimations. However, how to quantify an estimator's uncertainties for various queries and leverage the uncertainties to boost model accuracy remains to be studied.

To address the above limitation, we propose a new cardinality estimator, Fauce. Fauce includes a new query featurization method

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476254

(§3) that leverages semantic information contained in the database and captures real dependent relationships between table columns to encode the queries into more informative feature vectors. Furthermore, we mathematically define the uncertainty of the estimator and introduce a new model that incorporates the uncertainty estimation into Fauce (§5). Fauce also includes a new learning paradigm that leverages the uncertainties to boost the estimation results and make Fauce robust to be applied in dynamic databases (§5.3).

A query consists of four components, tables, joins, columns, and predicate values. We leverage the semantic information contained in the database (e.g., the relationships between two tables) to featurize the tables and joins of a query (§3.1). Fauce captures the join relationships among database tables. Those relationships are represented as join graphs, where vertices are tables and each edge connects two joinable tables. We make the representations of the tables and joins of a query contain more informative features by analyzing this graph.

To capture the real correlations across all the table columns in a database (§3.2), we introduce dependency graphs to capture dependent relationships across columns, and based on the graphs we embed the columns into a vector to boost the estimation accuracy. Using a data structure to capture dependency requires the capture of implicit dependency relationships in columns across tables. We introduce a hierarchical dependency graph. In particular, we first build a local columns-dependency graph for each table. Then we build the global columns-dependency graph for all the columns in the database based on the local columns-dependency graphs developed in the first step. Finally, we use an embedding technique [16] to represent each column into a vector based on the global columns-dependency graph. Such vectors can convey real correlations among the columns.

To include the uncertainties of the cardinality estimator into Fauce, we design a model based on *deep ensembles* (§5.3) to comprehensively quantify the uncertainty. The uncertainty of the cardinality estimator comes from multiple sources. First, we are uncertain about whether the learned model parameters can best describe the distribution of the queries in the query space. This is referred to as *model uncertainty*. Second, the query-based estimators train the model based on the generated training dataset. But the training dataset can not well reflect the features for all the queries. That is to say, there is always a data shift between the training dataset and the inference queries. This data shift can be large especially for dynamic databases. Thus, we are also uncertain about whether the data used to train model can well represent the features for inference queries, this is referred as *data uncertainty*. These two types of uncertainty consist the uncertainty of the learned estimator.

The two types of uncertainty are difficult to quantify. To address the above problems, we design a model called *deep ensembles with uncertainty* to estimate the cardinality and the corresponding uncertainty. We use the ensemble technique, because it generally produces the best results among all neural network-based approaches. Furthermore, it provides the benefit of being able to separately determine model and data uncertainties.

We conducted an extensive set of experiments over IMDB, a real-world dataset that exhibits complex correlation and conditional independence between table columns and have been extensively used in prior work [21, 24, 56–58]. On the created JOB-base benchmark, a schema that contains 6 tables and correlated filters. Fauce achieves 1.16-4.5× higher accuracy over the state of the art estimator. To check whether Fauce is robust to complicated queries with large number of filters, we create a more difficult benchmark, JOB-more-filters. On this benchmark, Fauce achieves 1.31-45.9× higher accuracy than previous estimators, including IBJS [30], MSCN [24], DeepDB [21],and NeuroCard [56]. Lastly, to test Fauce 's ability to handle queries with more complex join relations, we created JOB-complex-joins which has 15 tables and complex joins. Experimental results show that Fauce scales well to this benchmark, it has at least 1.28× higher accuracy than baselines. The contributions in the paper are summarized as below:

- We design and implement Fauce, the first learned cardinality estimator that contains the uncertainties for its results. It is also light weight with fastest inference time and leading accuracy among the learned methods we studied in the paper.
- Fauce includes a new *query featurization* (§3) method that can encode the queries into more informative feature vectors by leveraging the *join schema* of the database and capturing the real correlations across the table columns.
- Fauce mathematically defines the uncertainty of the estimator and designs a model called *deep ensembles with uncertainty* (§5.3) to estimate the cardinality.
- Fauce includes an uncertainty management module (§5.3). We also show that how the uncertainty management can be leveraged to further boost Fauce's accuracy.

2 PROBLEM DESCRIPTION

In this section, we introduce some notations and describe why the cardinality estimation can be solved as a regression problem.

2.1 Notations

Consider a database $\mathcal{D}$ contains m tables, $\mathcal{D} = \{T_i\}_{i=1}^{m}$. Each table T_i has a number of numeric columns, represented as $T_i = \{Col_i^1, ..., Col_i^{c_k}\}$, where c_k is the total number of columns in Table T_i. The total number of columns in $\mathcal{D}$ is denoted as C, where $C = \sum_{k=1}^{m} c_k$. We define the actual cardinality of a query q as the number of rows in joint tables that satisfy all predicates in q, and denote it as $Act(q)$. Similarly, we use $Card(q)$ to represent the estimated cardinality for the query q. Each query q can be represented as a collection of four sets: $\langle Tables \rangle$, $\langle Joins \rangle$, $\langle Columns \rangle$, $\langle Values \rangle$, and each set is defined as below.

- $\langle Tables \rangle$: the set of the tables in q's FROM clause.
- $\langle Joins \rangle$: the set of the join relations in q's WHERE clause.
- $\langle Columns \rangle$: the set of the columns involved in q's WHERE clause.
- $\langle Values \rangle$: the set of the predicates values in q's WHERE clause.

These four sets together depict the features of a query.

2.2 Formulation as a Regression Problem

As the cardinality of a query is a real-valued number, we develop a regression model $\mathcal{M}$, such that for any range query q on joint tables, the estimated cardinality $Card(q)$ produced by $\mathcal{M}$ matches or closes to the actual cardinality $Act(q)$.

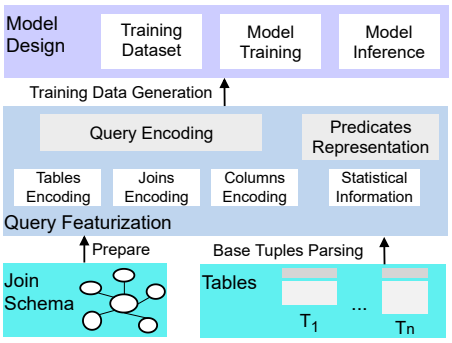


Figure 1: Overview of Fauce. The Query Featurization (§3) transforms the queries into vectors, it includes Tables Encoding (§3.1), Joins Encoding(§3.1), Columns Encoding (§3.2), and basic statistical information (§3.3). The generated training dataset (§4.2) is used to train the appropriately designed regression model (§5). At last, the trained model is used to estimate the query cardinalities (§5.2).

The input of the model $\mathcal{M}$ must be a real-valued vector. Therefore, we must transform the query q into a real-valued vector which represents the features of q. This transformation is called *query featurization*. For a query $q = \langle Tables\rangle, \langle Joins\rangle, \langle Columns\rangle, \langle Values\rangle$, we transform q into a query feature vector $\vec{f} = \langle f_T, f_J, f_C, f_V\rangle$, where f_T, f_J, f_C, and f_V are the features extracted from $\langle Tables\rangle$, $\langle Joins\rangle$, $\langle Columns\rangle$, and $\langle Values\rangle$ respectively. The vector $\vec{f}$ serves as the input to the regression model $\mathcal{M}$. The actual cardinality, $Act(q)$, serves as the labels, which guides the model training. Given a training set of labeled queries $\mathcal{S}$, the model $\mathcal{M}$ trained on $\mathcal{S}$ is expected to produce accurate cardinalities for unseen queries.

2.3 Overview of Fauce

Figure 1 shows the architecture of Fauce at a high level. Fauce consists of two stages. First, Fauce transforms input queries into feature vectors through a new query featurization method (§3), including tables encoding and joins encoding). Tables encoding (§3.1) is based on a graph embedding method that can capture semantic information of the database tables and achieve more accurate encoding results than widely used one-hot encoding and binary encoding methods. Joins encoding (§3.1) is based on our proposed *joins2vec* algorithm to featurize joins into vectors. Without any assumption on the independence of columns, our column encoding (§3.2) can capture real dependency information among the columns. Besides the encoding information, Fauce also collects statistics of the database tables (e.g., row counts and domain bounds) to represent the point predicate and/or range predicate of a query (§3.3).

Second, we train the model $\mathcal{M}$ based on the generated training dataset (§4.2). Once the training is finished, the model is ready to estimate the cardinalities for a given query. For each input query, we use a query featurization method to transform the query into a feature vector. This vector is plugged into the model $\mathcal{M}$, and the output of $\mathcal{M}$ is the estimated cardinality together with the corresponding uncertainty. The trained model $\mathcal{M}$ can handle queries joining any subset of tables, with arbitrary range selection.

3 QUERY FEATURIZATION

Before using the model $\mathcal{M}$ to estimate the cardinality, we must convert input queries into vectors. A query q can be represented as: $\langle Tables\rangle$, $\langle Joins\rangle$, $\langle Columns\rangle$, and $\langle Values\rangle$. Each of them is represented by a vector. These four vectors combined together is the outcome of the *query featurization* for q. The result is directly plugged into the model for both training and inference. Section 3.1 introduces how to encode $\langle Tables\rangle$ and $\langle Joins\rangle$ into vectors; Section 3.2 introduces the method to encode $\langle Columns\rangle$; and Section 3.3 introduces how to represent $\langle Values\rangle$ of a query.

Algorithm 1: Joins2Vec $(JS, D, \lambda, \epsilon)$

Input: $JS = (V, E)$: The join schema of a database
D: Maximal number of allowed joins in a query
λ: Encoding size of each join relationship
ϵ: Number of the epochs
Output: Matrix of vector representations of joins: Θ

```
JGs = {}; // Initialize an empty join graph set
foreach t ∈ V do
    for d = 0 to D do
        JGs∪ =GetJoinGraphs(JS, t, d); // Algorithm 2
Initialize Θ ∈ R^{|JGs|×λ}; // Uniform initialization
for e = 0 to ϵ do
    foreach t ∈ V do
        for d = 0 to D do
            jg_t^(d) := GetJoinGraphs(JS, t, d);
            context_t^(d) = {};
            foreach t' ∈ Neighbours(JS, t) do
                foreach ϕ ∈ {d − 1, d, d + 1} do
                    if ϕ ≥ 0 and ϕ ≤ D then
                        context_t^(d) ∪ =
                          GetJoinGraphs(JS, t', ϕ);
            foreach jg_cont ∈ context_t^(d) do
                Loss(Θ) = -log Pr(jg_cont|Θ(jg_t^(d)));
                Θ = Θ − α ∂Loss(Θ)/∂Θ;
return Θ
```

3.1 Tables and Joins Encoding

Tables encoding. Instead of using one-hot and binary encoding methods, we use a graph embedding method [36] to encode the database tables. The *join schema* of a database is considered as an undirected graph G, where vertices are tables and each edge connects two joinable tables. We use G as the input for the graph emebedding method, and the output is a group of vectors. Each vector is the encoding result for a corresponding table. In a database $\mathcal{D} = \{T_i\}_{i=1}^{m}$, if a table is not involved in a query, we use a vector with all zeros to represent this table. Similar to the binary encoding, our tables encoding method represents each table as a $\lceil \log(m+1) \rceil$ dimensional vector, where m is the number of tables in a database. Finally, the $\langle Tables\rangle$ of a query q is represented as a vector f_T with length of $m\lceil \log(m+1) \rceil$.

Joins encoding. Using the existing coarse-grained joins encoding methods [24] for *query featurization* always causes large errors in

cardinality estimation. We propose a new fine grained algorithm called *Joins2Vec* (Algorithm 1) for the joins encoding.

Algorithm 2: GetJoinGraphs (G, t, d)

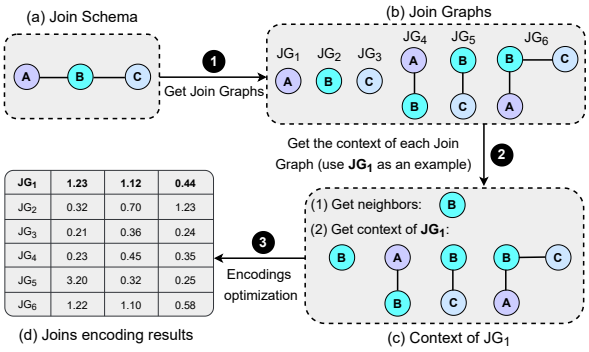
Input: $JS = (V, E)$: The join schema of a database
t: Table which is the root of a join relationship
d: Neighbours considered for extracting join graph
Output: $jg_t^{(d)}$: rooted join graph of degree d around table t

1 $jg_t^{(d)} = \{\}$;
2 **if** $d = 0$ **then**
3 $\quad jg_t^{(d)} := t$;
4 **else**
5 $\quad \mathcal{N}_t := Neighbours(G, t)$; // Breath First Search
6 $\quad M_t^d := \{\text{GetJoinGraphs}(G, t^{'}, d-1) | t^{'} \in \mathcal{N}_t\}$;
7 $\quad jg_t^{(d)} := jg_t^{(d)} \cup \text{GetJoinGraphs}(G, t, d-1) \oplus M_t^d$;
8 **return** $jg_t^{(d)}$

The $\langle Joins \rangle$ of a query q is based on the *join graphs* derived from the join schema JS. The algorithm *Joins2Vec* consists of two main components; the first component discovers all the possible *join graphs* based on the join schema JS, and the second component gets the encodings for all the *join graphs*. The goal of Algorithm 1 is to learn a λ dimensional encoding for each *join graph*. We first search all the join graphs, JGs (Lines 2-4) (extensive details are depicted in Algorithm 2). Then the encodings for the join graphs in JGs are initialized as a matrix: $\Theta \in \mathbf{R}^{|JGs| \times \lambda}$ (Line 5) where $|JGs|$ is the number of possible join graphs extracted from JS. After that, we learn the encoding result Θ (Lines 6-18). These steps are explained in detail in the following two paragraphs.

(1) Get all the join graphs. First, we introduce how to use each table t in the database $\mathcal{D}$ as a root to build the join graphs. The join graph $jg_t^{(d)}$ rooted at the table t with different numbers of joinable tables d in a given join schema JS is extracted (Line 9). The join graphs discovering process is separately explained in Algorithm 2. The Algorithm 2 takes the join schema JS, table t, and degree of the joins d as inputs and returns the intended join graph $jg_t^{(d)}$. When $d = 0$, no join graphs need to be extracted and the table t is returned (Line 3). For the case when $d > 0$, we get all the (breadth-first) neighbours of t in $\mathcal{N}_t$ (Line 5), and the neighbours of t are those tables that can be joined with the table t. Then for each joinable table, $t^{'}$, we get its $(d-1)$-degree join graphs and save them in $M_t^{(d)}$ (Line 6), where $M_t^{(d)}$ is a list to store the rooted d-degree join graphs around table t. Finally, we get the $(d-1)$-degree join graph around the table t and concatenate these join graphs with $M_t^{(d)}$ to obtain the intended join graphs $jg_t^{(d)}$ (Line 7).

(2) Get the context for each join graph. Then, we introduce how to get the context for each join graph based on the results of Algorithm 2. Once the join graphs $jg_t^{(d)}$ of table t is extracted, we learn the encoding of a target join graph using its surrounding context in a given join schema JS (Lines 10-17). We define the context of a d-degree join graph $jg_t^{(d)}$ of the table t as the set of join graphs of $(d-1)$, d and $(d+1)$-degree rooted at each of the neighbours of t (Lines 10-14 in Algorithm 1). Note that we consider



JG1	1.23	1.12	0.44
JG2	0.32	0.70	1.23
JG3	0.21	0.36	0.24
JG4	0.23	0.45	0.35
JG5	3.20	0.32	0.25
JG6	1.22	1.10	0.58

Figure 2: An example of *Joins2Vec*. (a) A join schema with three tables. (b) Get all the possible join graphs based on (a) using Algorithm 2. (c) Get the context of each join graph. (d) Use Algorithm 1 to encode the join graphs into vectors.

join graphs of $(d-1)$, d and $(d+1)$-degree to be in the context of a join graph of d-degree, because a d-degree join graph is likely to be rather similar to the join graphs of degrees that are closer to d (e.g., $d-1$ and $d+1$) and not just the d-degree join graphs only.

(3) Optimize the encodings for the join graphs. The encoding of a target join graph, $jg_t^{(d)}$, with the context $context_t^{(d)}$ is learnt using the process at Lines 15-17 in Algorithm 1. Given the current representation of the target join graph $\Theta(jg_t^{(d)})$, we want to maximize the probability of every join graph in its context jg_{cont} (Line 16). Here, we learn such posterior distribution using logistic regression classifier. Finally, the encodings of all the join graphs are optimized by gradient descent (Line 17).

Using Algorithms 1 and 2, we get the encoding result for $\langle Joins \rangle$ of a query. Assume there are n possible join graphs in a database and the encoding size λ is equal to n, the representation of $\langle Joins \rangle$ of the query q is a n dimensional vector. Figure 2 shows an example of applying *Joins2Vec* on a join schema with three tables, A, B, and C. All the join graphs derived from this join schema are encoded into vectors (see (d) in Figure 2).

3.2 Columns Encoding

The correlations of table columns can be utilized as useful information to facilitate the columns encoding. We propose a method called *Columns2Vec*, which encodes the columns by using real correlations among the columns. This method includes three steps.

(1) Build local columns-dependency graphs. We calculate the Randomized Dependence Coefficient [32] (RDC) values for each pair of columns in each table T_i from the database $\mathcal{D}$. If the RDC value for two columns exceeds a threshold τ, then those two columns are dependent with each other; otherwise, they are independent. Using a small value of τ overestimates the columns-dependency, while using a large value of τ underestimates the columns-dependency. Here, we set τ as 0.4. Based on the RDC values of each pair of columns, we can build a local columns-dependency graph g_i for each table T_i. The graph g_i is a DAG. Once there exists a connection (i.e., an edge) between two columns (i.e., vertices), the graph shows those two columns are correlated. We get dependency information between any pair of columns in g_i by using depth first search to find whether a path exists between their corresponding vertices.

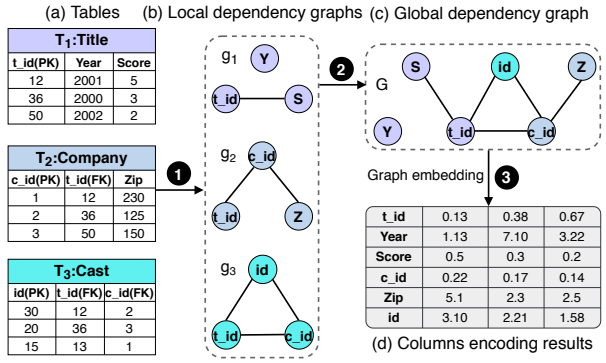


Figure 3: End-to-end example of *Columns2Vec*. (a) Three tables, and their columns to be encoded. (b) Local columns-dependency graph for each table, vertices are columns, an edge represents the correlations between two columns. (c) Global columns-dependency graph. (d) Use the graph from (c) as the input for the graph embedding method. Finally, each column in the database is represented as a vector.

(2) Build a global columns-dependency graph for the database. This graph is represented as G. It is built based on the local column-dependency graphs. Assume that there are two tables T_i and T_j and their local columns-dependency graphs are g_i and g_j respectively. We merge g_i and g_j if T_i and T_j are joinable. Thus, we can build the global columns-dependency graph G by checking whether the pair of tables T_i, T_j in a database are joinable or not.
(3) Use graph embedding for encoding. We use a graph embedding method [16] to encode each vertex in G into a vector. The results of *Columns2Vec* are used to represent $\langle Columns \rangle$ of a query q as a vector f_C with the length C, where C is the total number of different columns in the database. Figure 3 shows an example of applying *Columns2Vec* to three tables, `Title`, `Company`, and `Cast`. Multiple columns from these three tables are encoded into vectors (see (d) in Figure 3).

3.3 Range Representation

In this section, we discuss how to represent $\langle Values \rangle$ of a query. In a database $\mathcal{D} = \{T_i\}_{i=1}^{m}$, any conjunctive query q on numeric columns of the database $\mathcal{D}$ can be represented as a subset of $(lb_1^1 \leq Col_1^1 \leq ub_1^1) \wedge ... \wedge (lb_m^{c_m} \leq Col_m^{c_m} \leq ub_m^{c_m})$, where Col_i^j is the j^{th} column of the table T_i, lb_i^j and ub_i^j are the lower bound and upper bound on values in the column Col_i^j respectively, and $\{c_i\}_{i=1}^{m}$ is the number of columns in the tables $\{T_i\}_{i=1}^{m}$. Let the domain of the j^{th} column in table T_i be $dom(Col_i^j) = [min_i^j, max_i^j]$. If a query does not contain predicate on column Col_i^j, then we have $lb_i^j = min_i^j$ and $ub_i^j = max_i^j$. Then, the predicate on the column Col_i^j becomes $min_i^j \leq Col_i^j \leq max_i^j$. It means that the predicate on Col_i^j does not filter out any row. For instance, assume that there are two columns Col_i^j and Col_i^k from the table T_i, each of which is in the domain [0,100]. Then, the predicate $10 \leq Col_i^j \leq 20$ would have the following representation: $(10 \leq Col_i^j \leq 20) \wedge (0 \leq Col_i^k \leq 100)$.

Table 1: Query Features Segmentation

Type	Table	Join	Column	Predicate
Segment	$\langle Tables \rangle$	$\langle Joins \rangle$	$\langle Columns \rangle$	$\langle Values \rangle$
Method	Embedding	Joins2Vec	Columns2Vec	Range
Seg. Size	$m\lceil \log(m+1) \rceil$	n	C	$2 \times C$

The above definition includes one-sided range predicates and point predicates, i.e., $Col_i^j = x$ can be specified as $lb_i^j = x$ and $ub_i^j = x$.

Finally, we use a vector f_V with $2C$ dimensions to represent $\langle Values \rangle$ of a query: $\langle lb_1^1, ub_1^1, ..., lb_m^{c_m}, ub_m^{c_m} \rangle$. This vector is used as a part of input features for the model $\mathcal{M}$. To facilitate learning, all the vectors constructed by the *query featurization* have the same dimension and the same format as depicted in Table 1, where m is the number of tables in the database, n is the number of possible join relationships among the database tables, and C is the total number of different columns in the database. Therefore, the feature vector for the query q after the *query featurization* has a length of $L = m\lceil \log(m+1) \rceil + n + 3C$.

4 CHOICE OF REGRESSION METHODS

We use an ensembles of *deep neural networks* (DNNs), or *deep ensembles* for short, to estimate the cardinalities. We choose DNN, because the distribution of queries is very complex and DNNs are powerful models that have achieved impressive accuracy on many tasks. Furthermore, previous work [11, 40] has shown the advantage of using the ensemble technique to boost the cardinality estimation.

Deep ensembles. *Deep ensembles* is a learning paradigm where a collection of a finite number of DNNs is trained for the same task. In general, *deep ensembles* is constructed in two steps: (1) training a number of DNNs in parallel without any interaction, and (2) calculating the weighted average of the estimation results of each DNN as the final output of the *deep ensembles*.

4.1 Cardinality Transformation

In this section, we discuss how to create proper training labels through transformations. We generate a set of labeled queries (§4.2) $\mathcal{S} = (q_1 : Act(q_1)), ..., (q_N : Act(q_N))$ with actual cardinality as the label, where $\mathcal{S}$ contains N labeled queries. The cardinality variation across different queries in $\mathcal{S}$ can be huge, and the distribution of the actual cardinalities for different queries can be skewed. Building an accurate model on such data is challenging. We alleviate this problem by normalizing the actual cardinalities in $\mathcal{S}$ before training (i.e., the normalized values of the actual cardinalities belong to $[0, 1]$). We use the *log transformation* and *min-max scaling* to do the transformation. At runtime when using Fauce for estimation, we apply inverse transformation to get the true estimations.
Log transformation. The log transformation allows the model $\mathcal{M}$ to capture the abrupt variation. We apply log transformation (using the base 2) on the cardinalities to mitigate such variation.
Min-max scaling. We rescale the outcomes of the *log transformation* into the range $[0, 1]$ using *min-max scaling*. Given a set of log transformed cardinalities CARD = $\{card_1, card_2, ..., card_n\}$, the max cardinality in CARD (max_{card}), the min cardinality in CARD (min_{card}), the result of the *min-max scaling* for $card_i$ in

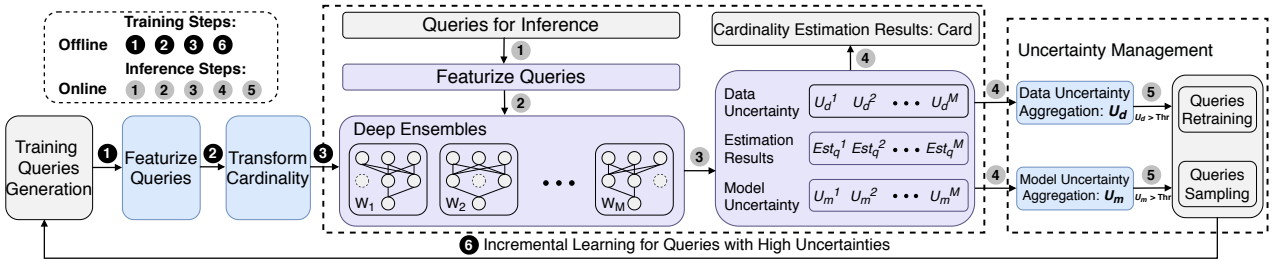


Figure 4: Training and inference process of Fauce. We first train the model offline, then we use the trained model for the inference online. The outcome of the inference includes, estimated cardinality and both model uncertainty and data uncertainty. $Card$ denotes the estimated cardinality; U_m is the model uncertainty with respect to the estimation; U_m is the query-dependent data uncertainty. Queries with high uncertainties are used for the incremental learning. Dotted neurons represent Dropout.

CARD is calculated as, $card_i^{'} = \frac{card_i - min_{card}}{max_{card} - min_{card}}$. Therefore, the final cardinality estimation is formulated by inverse transformation: $est(q_i) = 2^{card_i^{'} \times (max_{card} - min_{card}) + min_{card}}$.

4.2 Training Data Generation

Since the distribution of the cardinalities for the queries can be easily skewed, naive sampling from the space of all queries can result in a highly non-uniform training dataset and a sub-optimal cardinality estimator. In order to generate a uniform training dataset, our training data generation uses the following two rules: (1) **Generality.** The queries should come from different *join graphs* derived from the *join schema* of the database; (2) **Diversity.** The training data of the queries should be diverse in the number of predicates and their cardinalities. Based on these two rules, our training data is generated as follows.

We make the generated queries uniformly distributed to each *join graph*. To generate a query to a *join graph*, we first draw a tuple from the inner join result and get the number of non-null columns of this tuple, denoted as N_c. Second, we choose the number of predicates $n_p \in \{2, 3, ..., N_c\}$ uniformly at random. Then we randomly choose n_p columns, and randomly place n_p comparison operators associated with these columns based on whether each column can support range ($\{\leq, \geq, =\}$) or equality filters ($=$). These two steps guarantee a diverse set of multi-predicate queries.

5 MODEL DESIGN

Fauce includes two complementary approaches that operate in two phases, shown in Figure 4. In the offline phase, we train the model $\mathcal{M}$ based on the generated training data (Section 4.2); In the online phase, the model $\mathcal{M}$ accepts queries and outputs their estimated cardinalities together with the estimation uncertainties. The training first generates a set of labeled queries $\mathcal{S}$ (§4.2). Then we apply *query featurization* (§3) and *cardinality transformation* (§4.1) on $\mathcal{S}$ to get the training dataset D. D consists of N featurized queries $\{x_i, y_i\}_{i=1}^N$, where $x_i \in \mathbf{R}^L$ represents the L-dimensional query features, and $y_i \in \mathbf{R}$ is a real value in the range of $[0, 1]$. Let K denote the number of DNNs in the deep ensemble, and $W = \{w_i\}_{i=1}^K$ denote the parameters of the ensemble where w_i is the parameters of a DNN. Once the training is finished offline, the parameters W will be used for inference online.

A query for inference is featurized as a real valued vector, and then this vector is plugged into the trained model to estimate the cardinality and uncertainty of this estimation. The uncertainty consists of *model uncertainty* and *data uncertainty*, where the model uncertainty describes how confident the learned model is, and the data uncertain measures how noisy the collected query data are. These two types of uncertainty values will be leveraged to boost the model accuracy. The high model uncertainty means the learned parameters W cannot best describe the distribution of the features of a query. In this case, this query will be collected for the future retraining. The high data uncertainty means the noisy of the query data (e.g., new updated data in database) is high. In this case, we generate a bunch of new training queries based on this query with the high data uncertainty by a sampling method [6], and use these queries for the future training. That is, we use an *incremental learning* strategy to boost the model accuracy.

5.1 Uncertainty Quantification

Model uncertainty can be quantified using the Bayesian neural network [45, 53] (BNN) that captures uncertainty about the learned parameters. Data uncertainty describes the shift between the generated training data and input queries. To quantify the uncertainty, we use the following definition of the total variance in each estimated cardinality, based on [2]. Assuming x is the feature vector of the query q, y is q's estimated cardinality before inverse-transformation, the variance in y is formulated as follows.

$$Var(y) = Var(\mathbf{E}[y|x]) + \mathbf{E}[Var(y|x)] \tag{1}$$

Based on Equation 1, we define the model uncertainty and data uncertainty as follows.

$$U_m(y|x) = Var(\mathbf{E}[y|x]) \tag{2}$$

$$U_d(y|x) = \mathbf{E}[Var(y|x)] \tag{3}$$

where U_m and U_d represent the model and data uncertainty respectively. We can see that both uncertainties explain the variance in the estimation. The model uncertainty explains the variance related to $\mathbf{E}[y|x]$, and the data uncertainty explains the variance inherent to the conditional distribution $Var(y|x)$.

Model uncertainty. BNNs are used to find the posterior distribution of parameters W for Fauce, given the dataset $D = \{x_i, y_i\}_{i=1}^N$. Assume that the posterior distribution of W is $p(W|D)$, and $f_W = \{f_{w_i}\}_{i=1}^N$ is the function mapping for the deep ensembles between $\{x_i\}_{i=1}^N$ and $\{y_i\}_{i=1}^N$. Given an inference query q^*, its feature vector is x^*. The estimated cardinality is calculated by marginalizing over

the posterior distribution, shown as follows.

$$p(y^*|x^*, D) = \int_W p(y^*|f_W(x^*))p(W|D)dW \quad (4)$$

In Equation 4, y^* is the estimated cardinality for the query q^* before the inverse- transformation. Here, the exact computation for $p(W|D)$ is intractable, so we use a variational inference method [13] to find an approximation $q(W)$ to the posterior distribution $p(W|D)$. The estimation distribution is approximated by switching $p(W|D)$ to $q(W)$ in Equation 4 and performing the Monte Carlo integration, $\mathbf{E}(y^*|x^*) \approx \frac{1}{K}\sum_{i=1}^{K} f_{w_i}(x^*)$. The predictive variance can also be approximated as, $Var(y^*) \approx \frac{1}{K}\sum_{i=1}^{K} f_{w_i}(x^*)^2 - \mathbf{E}(y^*|x^*)^2$. $Var(y^*)$ arises because of the uncertainty about the model parameters W. We use $Var(y^*)$ to quantify the model uncertainty in Fauce.

Data uncertainty. Data uncertainty is dependent on the input queries. We need a model that not only estimates the output cardinalities, but also estimates the variances of the cardinalities given the input queries. That is, the model must give an estimation of $Var(y|x)$ mentioned in Equation 3. Assume that $\mu(x)$ and $\sigma(x)$ are the functions parameterized by W that calculate the mean and standard deviation of the estimation for a query q respectively, and x is q's feature vector. We have $y \sim \mathcal{N}(\mu(x), \sigma(x)^2)$, and the negative log likelihood is written as follows.

$$Loss(W) = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{\log \sigma^2(x_i)}{2} + \frac{(y_i - \mu(x_i))^2}{2\sigma^2(x_i)} + \frac{1}{2}log2\pi\right) \quad (5)$$

Comparing Equation 5 with a standard mean squared loss used in the traditional regression, we can see that the ensemble model introduces higher estimation variances for queries where the mean of estimated cardinality $\mu(x_i)$ is more deviated from the true cardinality y_i. On the other hand, a regularization term on $\sigma(x_i)$ prevents the model from introducing high estimation variances for all queries. After the model is optimized, we use $\sigma^2(x^*)$ to estimate the data uncertainty of a new query q^*, where x^* is the feature vector after q^* is featurized.

5.2 Training and Inference

Ensembles training. Fauce uses the entire training dataset D to train each DNN. To improve the model's robustness, Fauce also includes the *adversarial training*. In particular, we use the fast gradient sign method [15] to generate adversarial query examples. Given a query q, x as q's feature vector, and y as the query's true cardinality, an adversarial example is generated by $x' = x + \eta sign(\nabla_x Loss(W, x, y))$, where Loss(W, x, y) is from Equation 5. Here, η is a small value to bound the max perturbation. Those adversarial examples generated by the above formulation are used to augment the original training set D by treating (x', y) as additional training examples.

Ensembles inference. We treat the ensemble as a uniformly-weighted mixture model to calculate the final estimation results. Assume that x^* is the feature vector of the query q^*. The estimated cardinality of q^* is calculated with, $Card(q^*) = \mathbf{E}(y^*|x^*) \approx \frac{1}{K}\sum_{i=1}^{K}\mu_{w_i}(x^*)$. The model uncertainty for the query q^* is calculated with, $U_m(q^*) = \frac{1}{M}\sum_{i=1}^{M}\mu_{w_i}(x^*)^2 - \mathbf{E}(y^*|x^*)^2$. The data uncertainty for the query q^* is calculated with $U_d(q^*) = \frac{1}{K}\sum_{i=1}^{K}(\sigma^2_{w_i}(x^*) + \mu^2_{w_i}(x^*)) - \mathbf{E}(y^*|x^*)^2$.

5.3 Management of Estimation Uncertainty

Uncertainty management. We propose an algorithm called *manage uncertainty* (Algorithm 3) to use the uncertainties to make the estimation safer to use and improve model accuracy. In Algorithm 3, ϕ_m and ϕ_d are two thresholds for the model uncertainty $U_m(q^*)$ and data uncertainty $U_d(q^*)$ respectively.

When comparing the uncertainty values with the thresholds, we have three situations. First, $U_m(q^*) \le \phi_m$ and $U_d(q^*) \le \phi_d$. This means that Fauce is confident on its estimation, so the estimated cardinality is safe to use (Lines 2-3). Second, $U_m(q^*) > \phi_m$ and $U_d(q^*) \le \phi_d$. This happens when the training dataset D well represents the features of q^*, but the trained parameters underestimate q^*. We store the query q^* into a buffer B for an *incremental learning* strategy to eliminate the underestimation (Lines 4-5). Third, $U_m(q^*) > \phi_m$ and $U_d(q^*) > \phi_d$. This happens when the training dataset D cannot represent the features of the query q^*, and the parameters underestimate q^*. Besides storing the query q^* into the buffere B, we enlarge the number of queries in B by sampling [6] additional training data based on q^* for the incremental learning (Lines 6-9). At last, we update the model $\mathcal{M}$ (Line 10).

Algorithm 3: ManageUncertainty($U_m(q^*), U_d(q^*), \phi_m, \phi_d$)

Input: $U_m(q^*)$: Model uncertainty for new query q^*
$U_d(q^*)$: Data uncertainty for new query q^*
$Card(q^*)$: Estimated cardinality for q^*
ϕ_m, ϕ_d: Threshold for the model, data uncertainty
Output: $Safe\_Card$: Cardinality that is safety to use
Output: $\mathcal{M}^*$: updated model $\mathcal{M}$

1 $B = \{\}$; `// Buffer to store queries for retraining`
2 **if** $U_m(q^*) \le \phi_m$ ***and*** $U_d(q^*) \le \phi_d$ **then**
3 $\quad Safe\_Card := Card(q^*)$;
4 **else if** $U_m(q^*) > \phi_m$ ***and*** $U_d(q^*) \le \phi_d$ **then**
5 $\quad B = B \oplus q^*$;
6 **else if** $U_m(q^*) > \phi_m$ ***and*** $U_d(q^*) > \phi_d$ **then**
7 $\quad B = B \oplus q^*$;
8 $\quad B\cup = \text{Sampling}(q^*)$; `// Sampling queries[6]`
9 $\mathcal{M}^* \leftarrow \text{IncrementalLearning}(\mathcal{M}, B)$;
10 **return** $Safe\_Card, \mathcal{M}^*$

When is the incremental learning triggered? In Algorithm 3, the incremental learning can be triggered when the number of queries in B is beyond B's maximal size. In Fauce, we set the maximal size of B as 2000 queries. A small maximal size of B can frequently trigger the incremental learning, which increases the overhead of using the incremental learning. In contrast, a large maximal size of B may rarely trigger the incremental learning, which means that a large number of queries will be estimated by a stale model. As a consequence, the estimation quality of Fauce is decreased.

Re-encoding for tables/joins/columns. The tables and joins encoding is based on the join schema of a database. Thus, re-encoding of them is not necessary when incremental learning happens in both static and dynamic environments. The columns encoding is based on the inter-column correlations. Such correlations do not change in a static environment. Thus, re-encoding of columns is not necessary. However, in a dynamic environment, the inter-column

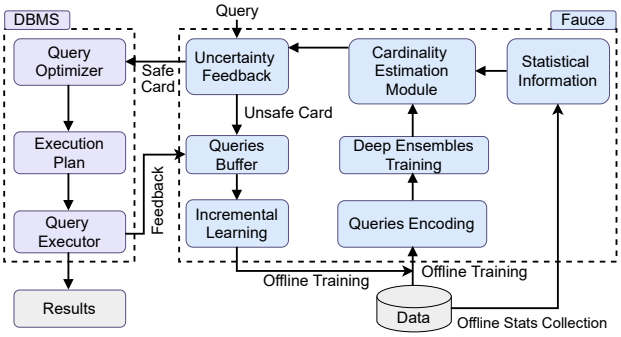


Figure 5: Integration of Fauce with existing DBMS

correlations could change when data are continuously updated. However, Fauce does not need to re-encode of all the columns from scratch. First, Fauce calculates the inter-column correlations only based on the new coming data. Then, Fauce filters out those pairwise columns whose correlations are significantly changed, and only re-encodes those columns. Therefore, re-encoding of columns only happens on a portion of columns in a dynamic environment.

5.4 Integration with DBMS

Figure5 shows how Fauce is integrated into a DBMS. Fauce is performed before the query optimizer as an additional phase that estimates the cardinalities and uncertainties. If the estimated uncertainties are less than the threshold ϕ_m and ϕ_m, then the cardinalities are injected into the query optimizer. In Fauce, the cardinality estimation techniques have online and offline phases. The offline phase includes three components: (1) statistical information collection; (2) deep ensembles training; and (3) incremental learning. These three components happen in both static and dynamic scenarios. But in a dynamic scenario, the statistical information can be outdated as new data are continuously coming. Thus, Fauce must update the statistics in the dynamic scenario. In dynamic scenario, the re-encoding of columns is required when incremental learning happens. The updated data may change the correlations among columns. Thus, Fauce must update the global column-dependency graph(§3.2) for the database. The online phase is for inference. This phase is the same in both static and dynamic scenarios. Before inference, a query must be featurized into vectors. As Fauce has received the statistics information about the database tables and encoding results of all the tables, joins, and columns in the offline phase, the time overhead for query featurization is small, which usually takes 2-6ms in our evaluation.

How is the incremental learning tied to the database system? Figure 5 shows that the incremental learning is tied to the database system in two ways. For the first way, we can get uncertainty feedback when use Fauce to estimate the cardinalities. The queries with large estimated uncertainties will be stored in a buffer B for the offline incremental learning. For the second way, we can directly use feedback from the query executor for incremental learning. However, the incremental learning based on new queries could affect the existing queries. In other words, the model "forgets" the old data and focuses exclusively on the new data. We use the Dropout [47] technique to avoid the above problem. In Fauce, we utilize a dropout value of $p = 0.2$ when updating the model $\mathcal{M}$ over the queries in B.

Table 2: Workloads used for evaluation. *Tables*: Number of base tables in each workload. *Rows*: Number of rows after the outer join. *Cols*: Total number of columns in the base tables. *Feature*: Characteristic of the queries in each workload.

Workload	Tables	Rows	Cols	Feature
JOB-base	6	$2 \cdot 10^{12}$	13	correlated filters
JOB-more-filters	6	$2 \cdot 10^{12}$	22	+ more filters
JOB-complex-joins	15	$2 \cdot 10^{13}$	22	+ complex joins

6 EVALUATION

We compare Fauce with state-of-the-art cardinality estimators using point and range queries. We aim to answer the following questions:

- Compared with the prior methods, how does Fauce perform in terms of accuracy and efficiency? (§6.2 and §6.4)
- How does the improvement on the cardinality estimation impact the performance of the query optimiser (§6.3)
- How does the Fauce perform in a dynamic environment? (§6.5)
- How does Fauce perform on data profiling task? (§6.7)

6.1 Experimental Setup

Platform. We use a machine with an NVIDIA V100 GPU and an Intel i9 CPU with 128GB RAM, and Tensorflow 2.3.

Workloads. We use a real-world dataset: IMDB [29]. IMDB has complex correlated columns. It consists of 21 tables. We focus on testing queries with correlated filters, larger number of filters, and complex joins in their predicates. In our experiments, each workload contains 2000 testing queries. Those workloads are discussed as follows (see Table 2).

- **JOB-base**: the queries in JOB-base are generated based on numeric columns in JOB-light. The schema in JOB-light is a typical star schema. JOB-light contains six tables, `title` (primary), `cast_info`, `movie_info`, `movie_company`, `movie_keyword`, and `movie_info_idx`. The predicates of the queries have 3-7 filters.
- **JOB-more-filters**: this benchmark tests Fauce 's scalability to complicated predicates. Some queries involve large number of columns in their predicates. The schema is the same as JOB-base's. The predicates of the testing queries have 8-13 filters.
- **JOB-complex-joins**: this benchmark contains 15 tables in IMDB and involves multiple join keys. For instance, `movie_companies` is not only joined with `title` on `movie_id`, but also joined with `company_name` on `company_id`, etc. Each query joins 2–11 tables. JOB-complex-joins is used to test Fauce 's scalability to complicated join conditions.

Evaluation metrics. To evaluate the accuracy of Fauce on the above workloads, we use the q-error metric. The q-error of Fauce on a query q is calculated as, $error = max(\frac{est(q)}{act(q)}, \frac{act(q)}{est(q)})$. Here, we assume that $act(q) \geq 1$ and $est(q) \geq 1$, so the minimum $error$ is 1×. We report the median, $75th$, $90th$, $95th$ and $99th$ percentile errors across all queries.

Baselines. We compare Fauce against a variety of representative cardinality estimators, including:

1) Postgres: Using Postgres, we evaluate the cardinality estimation that can be obtained from a real DBMS. The cardinality estimation in Postgres relies on 1D histograms and heuristics.

Table 3: Estimation errors on the JOB-base, JOB-more-filters, JOB-complex-joins workloads. "MU" and "DU" denote model uncertainty and data uncertainty respectively. "Fauce +MU" means training with model uncertainty only, "Fauce +DU" means training with data uncertainty only, and "Fauce +Both" means training with both model uncertainty and data uncertainty.

Estimator	JOb-base					JOB-more-filters					JOB-complex-joins				
	50th	75th	90th	95th	99th	50th	75th	90th	95th	99th	50th	75th	90th	95th	99th
Postgres	13.4	623	1960	$2 \cdot 10^5$	$7 \cdot 10^5$	8.1	162	1429	$1 \cdot 10^4$	$2 \cdot 10^5$	17.4	1679	7928	$4 \cdot 10^5$	$8 \cdot 10^5$
IBJS	11.6	125	2321	$4 \cdot 10^4$	$7 \cdot 10^6$	7.6	77.1	963	$8 \cdot 10^3$	$4 \cdot 10^5$	14.5	239	5014	$3 \cdot 10^4$	$7 \cdot 10^4$
MSCN	6.13	44.5	142	3568	$2 \cdot 10^4$	4.8	16.3	121	1680	$5 \cdot 10^4$	6.9	34.4	163	2820	$4 \cdot 10^4$
DeepDB	4.61	17.3	145	3348	$3 \cdot 10^4$	4.2	14.5	86	1182	$4 \cdot 10^4$	5.3	17.3	268	3717	$3 \cdot 10^4$
NeuroCard	3.04	10.2	69	1093	$9 \cdot 10^3$	3.8	8.2	59	538	$2 \cdot 10^4$	3.5	8.8	56.7	608	$8 \cdot 10^3$
Fauce +DU	4.12	9.6	46.5	1246	$8 \cdot 10^3$	3.9	7.7	43	464	$1 \cdot 10^4$	4.1	8.4	48.0	598	$9 \cdot 10^3$
Fauce +MU	2.86	5.5	17.2	375	$3 \cdot 10^3$	3.2	5.6	25	245	$5 \cdot 10^3$	3.4	7.2	25.8	366	$5 \cdot 10^3$
Fauce +Both	**2.58**	**5.1**	**15.3**	**279**	$\mathbf{2 \cdot 10^3}$	**2.9**	**5.1**	**21**	**206**	$\mathbf{3 \cdot 10^3}$	**2.7**	**6.3**	**21.6**	**227**	$\mathbf{3 \cdot 10^3}$

2) IBJS: We use the Index-based Join Sampling method (IBJS) [30] as a non-learned baseline. IBJS estimates a query's cardinality using a sampling-based approach based on the query's join graph and executing per-table filters.

3) MSCN: This is a representative supervised query-driven estimator [24]. We generate 10K training queries for each workload to train the model and use a bitmap size of 2K.

4) DeepDB: This is an unsupervised data-driven estimator [21]. DeepDB uses a non-neural sum-product network as the density estimator for each table subset chosen by correlation tests. Conditional independence is assumed across subsets.

5) NeuroCard: This is also an unsupervised data-driven estimator [56]. NeuroCard is a join cardinality estimator that builds a single neural density estimator over the entire database.

6.2 Estimation Quality

Tables 3 shows that Fauce exceeds the baseline estimators on all the three workloads (§ 6.1).

(1) Results on JOB-base.

Postgres has the largest median, 75th, and 90th error. Postgres only relies on 1D histogram and heuristics, and does not contain cross-column statistics. Thus, Postgres cannot fully capture the characteristics of queries, and has high estimation error.

IBJS has the largest 95th and 99th errors. IBJS is a sampling based method. We set the maximum sampling budget as 10,000, as a larger sampling budget does not bring too much benefit [30]. Because the joint space is very large, those samples have small chances to hit testing queries, hence can cause large estimation errors. IBJS's inference time varies from 3 to 20 ms.

MSCN has large estimation errors on some queries with small true cardinalities. MSCN's training is based on a number of featurized queries. MSCN does not contain uncertainty information about testing queries. Furthermore, its query featurization method cannot leverage semantic information in a database. As a result, MSCN has large errors on some queries.

DeepDB has large errors on high quantiles. DeepDB uses a sum-product network to estimate the density for each table subset. Each table subset is chosen based on the correlations among tables in the database. DeepDB assumes conditional independence across table subsets. But this assumption is not always true in real databases as some table subsets may have close relationships. Furthermore, DeepDB assumes inter-column independence when building the density model via the sum-product network. Therefore, it does not reflect real column dependencies in the databases. As a result, Fauce's accuracy gain on each quantile is 1.8×, 3.4×, 9.5×, 12×, and 15×, compared with DeepDB.

Fauce exceeds NeuroCard. NeuroCard uses deep autoregressive models as a density estimator to learn high-dimensional data distributions. It works as follows. Given a range query with K predicates, first, NeuroCard obtains the probability of i-th predicate conditioned on previous values. Then, it generates a sample value for i-th column. Finally, the conditional probabilities are multiplied together to estimate the cardinality. We find NeuroCard tends to have large errors on some range queries with correlated columns in their predicates. In contrast, Fauce is robust to this kind of queries. The overall Fauce 's accuracy gain on each quantile is 1.16×, 2×, 4.5×, 3.9×, and 4.5×, compared with NeuroCard.

(2) Results on JOB-more-filters. This workload is used for testing Fauce 's scalability on queries with a large number of filters in their predicates. As Table 3 shows, all estimators produce less accurate cardinalities than Fauce. Compared with Postgres, Fauce's accuracy gain is from 2.8× to 70×, because the accumulative error caused by the 1D histogram grows as the number of filters grows. Compared with IBJS, Fauce's accuracy gain is from 2.6× to 46×. This is because the sampling results can easily be empty as the number of filter grows. Compared with MSCN, Fauce's accuracy gain is 1.7×, 3.2×, 5.9×, 8.1× and and 16.7× at median, 75th, 90th, 95th, and 99th respectively. Compared with DeepDB, Fauce improves the accuracy by 1.5×, 2.8×, 4×, 5.7×, and 13.3× at median, 75th, 90th, 95th, and 99th respectively. At last, compared with NeuroCard, Fauce's accuracy gains is 1.3×, 1.6×, 2.8×, 2.6× and 6.7× at median, 75th, 90th, 95th, and 99th respectively. Existing estimators fail to capture the more complex inter-column correlations. As a result, their estimations are vulnerable to queries with a large number of columns in predicates. The results demonstrate Fauce's scalability to the number of filters.

(3) Results on JOB-complex-joins. This workload is used for testing the Fauce's ability to scale to queries with a large number of filters and multiple join keys. The number of filters in the predicates of the queries varies from 4 to 13; The possible number of join keys varies from 2 to 10, and the predicates of the queries can have multiple join keys. Table 3 shows that Fauce 's accuracy remains high on this complex schema. Postgres and IBJS have the largest errors, because many intermediate samples become empty. Compared with MSCN and DeepDB, Fauce's accuracy improvement is

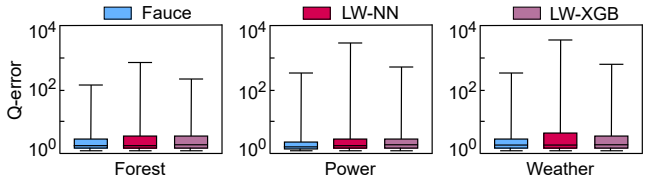


Figure 6: Estimation errors on various datasets with no joins.

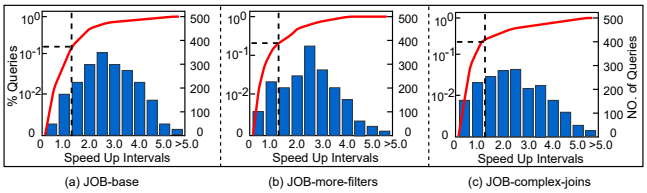


Figure 7: The impact of the improved cardinality estimation of Fauce on query performance.

up to 13.3× and 10× respectively. NeuroCard also achieves high accuracy, but it still has large estimation errors on queries with correlated columns in predicates. Fauce overcomes this challenge and offers better accuracy than NeuroCard.

(4) Results on queries with no joins. We use the methods in [11] as the baseline. The estimators in [11] are based on light-weight models (i.e., simple NNs and boosting trees). We refer to the methods using NNs and boosting trees in [11] as "LW-NN" and "LW-XGB" respectively. Our experiments use the same datasets as [11], including "Forest", "Power", and "Weather". Fauce, LW-NN, and LW-XGB are trained and tested on the same queries. Figure 6 shows the testing results. We can see Fauce has the higher accuracy than LW-NN and LW-XGB on all the datasets. The main difference between Fauce and LW-NN lies in the *query featurization* method. LW-NN and LW-XG extract features from $\langle Values \rangle$ of queries, and use AVI [28], EBO[3], and MinSel[35] as extra features during *query featurization*. Fauce, besides extracting features from $\langle Value \rangle$, uses the *Columns2vec* algorithm to extract features from $\langle Column \rangle$ of a query. The higher accuracy of Fauce on these datasets indicates that Fauce's featurization method can capture more informative features of a query than the query featurization method used in [11].

6.3 Impacts on Query Performance

We evaluate whether the improvement of cardinality estimation in Fauce leads to better query performance. Our evaluation is based on the workloads introduced in Section 6.1. We test 2000 queries for each workload. After we get the estimated cardinalities from Fauce, these estimations are then fed into a version of PostgreSQL modified to accept external cardinality estimations [4]. Figure 7 shows the performance impact of the cardinalities estimated by Fauce, compared to the default cardinality estimations from PostgreSQL. For the Job-base (Figure7(a)), the execution time for these queries ranges from < 1s up to 200s. Fauce improves the performance of 81.4% of the queries. For the Job-more-filters (Figure7(b)), the majority of the queries' runtime ranges from 0.5 to 350s. Fauce improves the performance for 80.3% of the queries. 8.4% of the queries' execution time is extended, and the rest of queries have the same performance as PostgreSQL. For the Job-complex-joins (Figure7(c)), Fauce improves the performance for 78.2% of the queries.

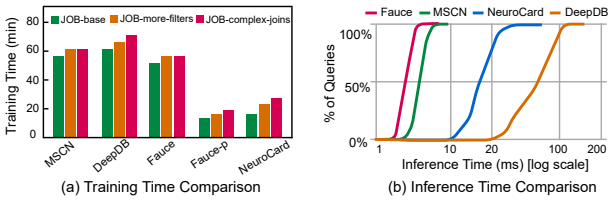


Figure 8: Physical efficiency of Fauce.

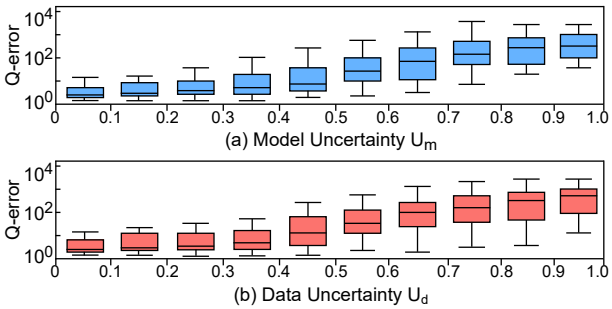


Figure 9: Q-error of queries with different uncertainties.

6.4 Efficiency of Fauce

Training time comparison. Figure 8(a) shows the training time. Once the training queries are collected, training MSCN takes 55-60 mins for the three workloads. DeepDB can only run on parallel CPUs (not on GPU as other methods), so DeepDB takes the longest training time (60-75 mins). Fauce has higher efficiency on *query featurization* compared with MSCN, so training Fauce requires less time than MSCN: Fauce takes 50-54 mins on all the three workloads. Note that DNNs in the ensemble are independent in Fauce and can be trained independently. Therefore, Fauce's training time can be further optimized through parallel training, which reduces the training time to less than 20 mins (see Fauce-p in Figure 8(b)). Training NeuroCard has to calculate the join count tables and perform parallel sampling first, and then trains the auto-regressive model for some epochs. Even if training process of NeuroCard is accelerated by GPUs, training Neurocard still takes more than 20 mins on Job-more-filters and Job-complex-joins workloads.

Inference time comparison. Figure 8(b) shows the inference time of MSCN, DeepDB, NeuroCard, and Fauce on JOB-base workload. MSCN, NeuroCard, and Fauce run on GPU while DeepDB runs on CPU; These estimators are implemented in Python. Fauce and MSCN are the fastest because they are based on lightweight network and involve fewer calculation during the inference. DeepDB's inference time spans from 1 ms to 200ms, and its inference time is short for queries with a small number of joins and filters. However, its inference time can be more than 150ms for complex queries. NeuroCard's inference time is smaller than DeepDB, but it is still 2 to 10× larger than those of Fauce and MSCN. The inference time of DeepDB and NeuroCard is more sensitive to the number of the predicates in a query than Fauce and MSCN.

6.5 Handling Data Updates

We analyze how Fauce performs in a dynamic environment.

Threshold values. In Algorithm 3, we use two thresholds ϕ_m and ϕ_d to control the model uncertainty and data uncertainty respectively. Here, we discuss how we set proper values for ϕ_m and ϕ_d

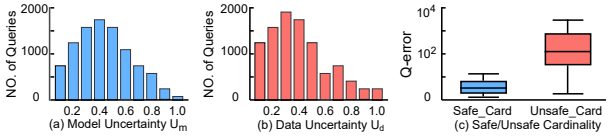


Figure 10: Statistical information for the queries.

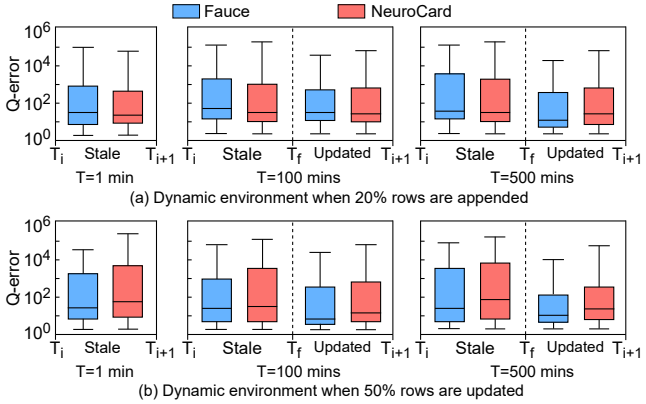


Figure 11: Estimation quality under dynamic environment.

as the thresholds. In our study, the threshold values for ϕ_m and ϕ_d are measured based on additional 10K queries, not those for testing. Those queries are derived from JOB-light. We estimate cardinalities and uncertainties for those 10K queries. The uncertainty value of each query is in the range of [0,1]. We set the length of a uncertainty interval as 0.1 and use ten uncertainty intervals. We count the number of queries in each of the ten uncertainty intervals. Figure 9(a) and (b) show that queries with ϕ_m higher than 0.5 or ϕ_d higher than 0.4 tend to have large errors. Based on the above observation, we set the threshold value for model uncertainty and data uncertainty as 0.5 and 0.4 respectively. In Fauce, if a query's model uncertainty is below ϕ_m or a query's data uncertainty is below ϕ_d, then its estimated cardinality is safe to use. We refer to such cardinality as "safe_card". Figure 10(a) and (b) show the number of queries within the ten intervals. We can see the percentage of safe_card is about 70%. The errors of queries within safe_card and unsafe_card are shown in Figure 10(c), based on which we conclude that the queries with safe_card have much smaller errors than with unsafe_card.

Dynamic environment setup. Suppose that there are n queries uniformly distributed in a time range $[T_i, T_{i+1}]$, and $T = T_{i+1} - T_i$. The queries based on updated data begin to come at timestamp T_i. Those queries with high uncertainties are stored in the buffer B for the incremental learning. Once the number of queries in the buffer B is beyond B's maximum capacity, the model update begins. Suppose the model update finishes at timestamp T_f ($T_i < T_f \leq T_{i+1}$). For the first $\lceil n \cdot \frac{T_f - Ti}{T} \rceil$ queries, their cardinalities are estimated using the stale model $\mathcal{M}_{stale}$. For the remaining $\lfloor n \cdot (1 - \frac{T_f - Ti}{T}) \rfloor$ queries, the updated model $\mathcal{M}_{update}$ are used. Since some queries are handled by the (inaccurate) stale model, the estimation results for these queries can be erroneous.

Data update. We use the real-world dataset IMDB [29] for testing under a dynamic environment. Our experiment is based on two different kinds of data updates. The first kind of data updates leads to significant changes in pair-wise correlations, while the second kind of data updates does not. (a) In the first kind of data update, we use the similar method introduced in [11] to update the dataset. In particular, we update 50% tuples of the dataset, which results in huge change in data distribution; (b) In the second kind of data update, we partition the table `title` into two parts on the `year` column. The part with the latest `year` is used as the new data to be appended into dataset, the pair-wise correlations for this method are not significantly changed. This kind of data update is used in [56]. After data updates, we apply our workload generation method on the updated dataset to generate 10K queries for testing. These queries are uniformly distributed in $[T_i, T_{i+1}]$. T, which is equal to $T_{i+1} - T_i$, is a parameter, which represents how "frequently" the data are updated. **Model update.** We update Fauce and NeuroCard, and then compare their estimation quality. NeuroCard is a data-driven estimator, so NeuroCard is updated by retraining on the entire new updated dataset. Fauce is a query-driven estimator, it is updated via the incremental learning once there are 2K queries contained in the buffer B.

Estimations in a dynamic environment. We test the estimation quality of Fauce and NeuroCard in a dynamic environment. The value of T is varied to control the frequency of data update. We set three levels of the frequency: high (1 min), medium (100 mins), and low (500 mins). The estimation quality of NeuroCard and Fauce in the dynamic environment is shown in Figure 11.

First, we compare Fauce with NeuroCard when 20% of rows in the table `title` is appended (Figure11(a)). If the frequency of the data update is high (shown in the left figure in Figure11(a)), both Fauce and NeuroCard cannot finish the model update, then the stale models for Fauce and NeuroCard are used for testing. When the data update does not change the data distribution, data distribution learned by NeuroCard still works. As a result, NeuroCard performs better than Fauce. When the data update frequency is medium and slow (the right two figures in Figure11(a)), both Fauce and NeuroCard can finish model update. We set the time interval for data updates as $T = T_{i+1} - T_i$, for the queries coming within $[T_i, T_f]$, and those queries are tested by the stale models. Here, T_f is the time when the model updates are finished. Queries coming within $[T_f, T_{i+1}]$ are tested by the updated models. For queries coming within $[T_i, T_f]$, NeuroCard performs better than Fauce. This is because the appended data does not change the data distribution in the database. So NeuroCard can still work well. Queries coming within $[T_f, T_{i+1}]$ are tested by the updated models in Fauce and NeuroCard. We can see Fauce performs better than NeuroCard.

Second, we compare Fauce and NeuroCard when 50% rows in table `title` are updated (Figure11(b)). Overall, Fauce performs better than NeuroCard for queries coming within $[T_i, T_f]$ (when the stale models are used) and $[T_f, T_{i+1}]$ (when the updated models are used). This is because the inter-column correlations in this scenario have been significantly changed, so the data distribution learned by NeuroCard is outdated. In Fauce, the feature vector of a query $q = \langle Tables \rangle, \langle Joins \rangle, \langle Columns \rangle, \langle Values \rangle$ after the *query featurization* (§3) has the length of $L = m\lceil \log(m+1) \rceil + n + 3C$ (see Table1). As the inter-column correlations have been significantly changed, the features extracted from $\langle Columns \rangle$ can not reflect the new inter-column correlations. The ratio for the features extracted from $\langle Columns \rangle$ among the total length of the feature vector is: $\frac{C}{L}$, where C is the length of features extracted from $\langle Columns \rangle$.

Table 4: Impact of encoding methods(§3). "Ours" denotes our encoding method. The lowest errors are bolded.

Workload	Encoding	50th	90th	95th	99th
JOB-base	One-hot	4.53	87	2029	3×10^4
	Binary	4.24	62	1586	2×10^4
	Ours	**2.58**	**15.3**	**279**	$\mathbf{2 \times 10^3}$
JOB-more -filters	One-hot	5.23	84	862	2×10^4
	Binary	4.82	69	754	2×10^4
	Ours	**2.9**	**21**	**206**	$\mathbf{3 \times 10^3}$
JOB-complex -joins	One-hot	5.62	78	1446	2×10^4
	Binary	4.77	62	1193	1×10^4
	Ours	**2.7**	**21.6**	**227**	$\mathbf{3 \times 10^3}$

Here, the schema of the database remains the same, so the features extracted from $\langle Tables \rangle$ and $\langle Joins \rangle$ can be reused. Those features are not required for re-encoding. If the domain of some columns is changed, we need to update the domain for featurizing $\langle Values \rangle$ of a query q. Updating the domain of the columns can be finished in a very short time (similar with updating the 1D histogram in DBMSs). We conclude that only the $\frac{C}{L}$ portion of features after the *query featurization* is influenced by data updates. For IMDB [29], C is relatively small, compared with L, so Fauce's featurization method still work well in this scenario. That is why Fauce performs better than NeuroCard when 50% of rows is updated.

6.6 How Encoding Methods Impact Fauce

We explore how the encoding methods (§3.1 and §3.2) impact Fauce's accuracy.

Impact of encoding methods. We encode $\langle Tables \rangle$, $\langle Joins \rangle$, and $\langle Columns \rangle$ of a query with our encoding method. Some existing estimators [19, 24, 40] use one-hot or binary encoding methods. Table 4 shows the impact of encoding methods on the errors over the three workload (§6.1). We can see the impact brought by different encoding methods for low-quantile errors is small. However, the encoding methods have large impact on high-quantile errors. Our encoding method's accuracy gain is up to 7.3× and 15× on high-quantile errors, compared with the one-hot and binary encodings respectively.

6.7 Data Profiling

The column encoding method (§3.2) in Fauce can be used to find approximate functional dependencies (AFDs) among database columns. We compare the efficiency of Fauce with other four data profiling methods: Pyro [27], Tane [22], Ducc/Dfd [1], and Fdep [12]. Table 5 shows the information of the datasets we use for data profiling. The results are shown in Figure 12. We can see that Fauce finishes the job of finding the AFDs on all the datasets within a time limit (10^4s). Fauce's execution time for data profiling is the shortest on the datasets DB Status, Census, and Entity source. These datasets have unknown or large number of AFDs. When profiling on the datasets Reflns and Spots, Fauce's execution time is still lower than Tane, Ducc/Dfd, and Fdep (except for Pyro). For an easy-to-process dataset with a smaller number of rows/columns and AFDs (e.g., the Wiki image), Fauce is outperformed by the baselines. But for the hard-to-process datasets (i.e., DB Status, Census, and Entity source), the speedup of Fauce is larger than 10×, compared with baselines.

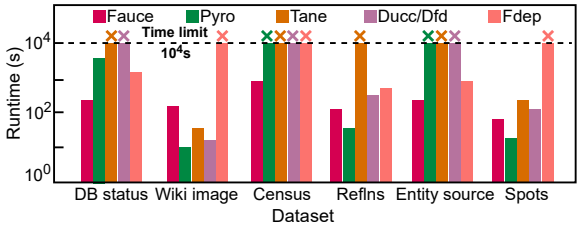


Figure 12: Runtime for data profiling. "x" means out of limit.

Table 5: Datasets used for data profiling.

	DB status	Wiki image	Census	Reflns	Entity	Spots
Cols.	35	12	42	37	46	15
Rows	29,787	777,676	199,524	24,769	26,139	973,510
AFDs	108,003	92	unknown	9,396	unknown	75

7 RELATED WORK

Query-driven cardinality estimators. Recently, there has been a surge of interests in using ML-based methods [7, 8, 31, 38, 54, 55] to solve system problems, especially using ML-based methods to enhance the performance of database components, e.g., indexing [26, 37, 49], query execution [41] and scheduling [33]. Some work [20, 23, 24, 48, 51] targets on leveraging past queries to learn functions mapping a query with a prediction domain. While Kipf et al [24] addressed a generic version of the selectivity estimation problem, the models in this paper are much more succinct, leading to significantly faster estimations.

Data-driven cardinality estimators. The data-driven approaches build unsupervised models, which learn the joint probability density function (PDF) of table attributes to estimate the probability of a query. In recent work, there is extensive work on applying data-driven techniques for solving challenging database problems. *Sample and Kernel-based methods* [20, 21, 30, 30] sample records from tables on-the-fly, or use average kernels centered around sampled points for estimation. *Sum-Product Networks (SPNs)* [34, 42] estimate the PDF results using either sum and product operations to combine children information in a tree structure. *Deep Auto-Regression (DAR)* models are the current state-of-the-art density models [14, 44, 50, 57]. DAR models capture all possible correlations among the attributes of tables to produce selectivity estimates.

8 CONCLUSIONS

It is challenging to make accurate cardinality estimations for complex queries using machine learning models. We introduce Fauce to address this problem. Fauce has a new query featurization method which can make the input feature vectors more informative for the cardinality estimation. It also includes uncertainty information for estimation results. Experimental results show that Fauce has 1.16-6.67× higher accuracy than the state-of-the-art approach when estimating cardinalites for complex queries.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This work was partially supported by U.S. National Science Foundation (CCF-1718194 and CCF-1553645) and the Chameleon Cloud.

REFERENCES

[1] Ziawasch Abedjan, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. 2014. Detecting unique column combinations on dynamic data. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 1036–1047.

[2] binghamton. [n.d.]. Variance proof. https://www2.math.binghamton.edu/lib/exe/fetch.php/people/renfrew/447-4-17.pdf.

[3] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. 2001. STHoles: a multidimensional workload-aware histogram. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. 211–222.

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

[5] Chee-Yong Chan and Yannis E Ioannidis. 1999. An efficient bitmap encoding scheme for selection queries. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*. 215–226.

[6] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. 2007. Optimized stratified sampling for approximate query processing. *ACM Transactions on Database Systems (TODS)* 32, 2 (2007), 9–es.

[7] Wenqian Dong, Jie Liu, Zhen Xie, and Dong Li. 2019. Adaptive neural network-based approximation to accelerate eulerian fluid simulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–22.

[8] Wenqian Dong, Zhen Xie, Gokcen Kestor, and Dong Li. 2020. Smart-PGSim: Using Neural Network to Accelerate AC-OPF Power Grid Simulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 63, 15 pages.

[9] Conor Durkan and Charlie Nash. 2019. Autoregressive energy machines. In *ICML*.

[10] Anshuman Dutt, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2020. Efficiently approximating selectivity functions using low overhead regression models. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2215–2228.

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

[12] Peter A Flach and Iztok Savnik. 1999. Database dependency discovery: a machine learning approach. *AI communications* 12, 3 (1999), 139–160.

[13] Yarin Gal and Zoubin Ghahramani. 2016. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*. 1050–1059.

[14] Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. 2015. Made: Masked autoencoder for distribution estimation. In *International Conference on Machine Learning*. 881–889.

[15] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[16] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 855–864.

[17] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2019. Multi-attribute selectivity estimation using deep learning. *arXiv preprint arXiv:1903.09999* (2019).

[18] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1035–1050.

[19] Rojeh Hayek and Oded Shmueli. 2020. Nn-based transformation of any SQL cardinality estimator for handling distinct, and, OR and NOT. *arXiv preprint arXiv:2004.07009* (2020).

[20] Max Heimel, Martin Kiefer, and Volker Markl. 2015. Self-tuning, gpu-accelerated kernel density models for multidimensional selectivity estimation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1477–1492.

[21] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2019. DeepDB: learn from data, not from queries! *arXiv preprint arXiv:1909.00607* (2019).

[22] Yka Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. 1999. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal* 42, 2 (1999), 100–111.

[23] Martin Kiefer, Max Heimel, Sebastian Breß, and Volker Markl. 2017. Estimating join selectivities using bandwidth-optimized kernel density models. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2085–2096.

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

[25] Andreas Kipf, Dimitri Vorona, Jonas Müller, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2019. Estimating cardinalities with deep sketches. In *Proceedings of the 2019 International Conference on Management of Data*. 1937–1940.

[26] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. 489–504.

[27] Sebastian Kruse and Felix Naumann. 2018. Efficient discovery of approximate dependencies. *Proceedings of the VLDB Endowment* 11, 7 (2018), 759–772.

[28] Byung-Jae Kwak, Nah-Oak Song, and Leonard E Miller. 2005. Performance analysis of exponential backoff. *IEEE/ACM transactions on networking* 13, 2 (2005), 343–355.

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

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

[31] Jie Liu, Jiawen Liu, Zhen Xie, and Dong Li. 2020. FLAME: A Self-Adaptive Auto-labeling System for Heterogeneous Mobile Processors. *arXiv preprint arXiv:2003.01762* (2020).

[32] David Lopez-Paz, Philipp Hennig, and Bernhard Schölkopf. 2013. The randomized dependence coefficient. *Advances in neural information processing systems* 26 (2013), 1–9.

[33] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*. 270–288.

[34] James Martens and Venkatesh Medabalimi. 2014. On the expressive efficiency of sum product networks. *arXiv preprint arXiv:1411.7717* (2014).

[35] microsoft. [n.d.]. queries contain correlations. https://support.microsoft.com/en-us/topic/kb2658214-fix-poor-performance-when-you-run-a-query-that-contains-correlated-and-predicates-in-sql-server-2008-or-in-sql-server-2008-r2-or-in-sql-server-2012-86e1a4a8-5793-f1a4-dd10-bc42347a7208.

[36] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, Yang Liu, and Santhoshkumar Saminathan. 2016. subgraph2vec: Learning distributed representations of rooted sub-graphs from large graphs. *arXiv preprint arXiv:1606.08928* (2016).

[37] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-dimensional Indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 985–1000.

[38] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *arXiv preprint arXiv:2101.04964* (2021).

[39] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2018. Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*. 1–4.

[40] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2019. An empirical analysis of deep learning for cardinality estimation. *arXiv preprint arXiv:1905.06425* (2019).

[41] Yongjoo Park, Ahmad Shahab Tajik, Michael Cafarella, and Barzan Mozafari. 2017. Database learning: Toward a database that becomes smarter every time. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 587–602.

[42] Hoifung Poon and Pedro Domingos. 2011. Sum-product networks: A new deep architecture. In *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*. IEEE, 689–690.

[43] Kedar Potdar, Taher S Pardawala, and Chinmay D Pai. 2017. A comparative study of categorical variable encoding techniques for neural network classifiers. *International journal of computer applications* 175, 4 (2017), 7–9.

[44] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[45] Jost Tobias Springenberg, Aaron Klein, Stefan Falkner, and Frank Hutter. 2016. Bayesian optimization with robust Bayesian neural networks. *Advances in neural information processing systems* 29 (2016), 4134–4142.

[46] SQLServer.2016. [n.d.]. Cardinality estimation for correlated columns in SQLServer 2016. https://blogs.msdn.microsoft.com/sql_server_team/cardinality-estimation-forcorrelated-columns-in-sql-server-2016/.

[47] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[48] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO-DB2's learning optimizer. In *VLDB*, Vol. 1. 19–28.

[49] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: a scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 308–320.

[50] Kostas Tzoumas, Amol Deshpande, and Christian S Jensen. 2011. Lightweight graphical models for selectivity estimation without independence assumptions. *Proceedings of the VLDB Endowment* 4, 11 (2011), 852–863.

[51] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1009–1024.

[52] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2020. Are We Ready For Learned Cardinality Estimation? *arXiv preprint arXiv:2012.06743* (2020).

[53] Yijun Xiao and William Yang Wang. 2019. Quantifying uncertainties in natural language processing tasks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 7322–7329.

[54] Zhen Xie, Wenqian Dong, Jiawen Liu, Hang Liu, and Dong Li. 2021. Tahoe: tree structure-aware high performance inference engine for decision tree ensemble on GPU. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 426–440.

[55] Zhen Xie, Wenqian Dong, Jie Liu, Ivy Peng, Yanbao Ma, and Dong Li. 2021. MD-HM: memoization-based molecular dynamics simulations on big memory system. In *Proceedings of the ACM International Conference on Supercomputing*. 215–226.

[56] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: one cardinality estimator for all tables. *arXiv preprint arXiv:2006.08109* (2020).

[57] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep unsupervised cardinality estimation. *arXiv preprint arXiv:1905.04278* (2019).

[58] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2020. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation. *arXiv preprint arXiv:2011.09022* (2020).