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:

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
We first build a local columns-dependency graph for each table. We embed the columns into a vector to boost the estimation accuracy. Then we build the global columns-dependency graph for all the 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 graphs developed in the first step.

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 BJS [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 featureization (§3) that leverages the uncertainties to boost the estimation results and make Fauce robust to be applied in dynamic databases (§5.3).
- 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).
- Fauce 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).
- 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 \( T_i = \{ \text{Col}_i^1, \ldots, \text{Col}_i^{c_i} \} \), where \( c_i \) 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 \( \text{Act}(q) \). Similarly, we use \( \text{Card}(q) \) to represent the estimated cardinality for the query \( q \). Each query \( q \) can be represented as a collection of four sets: \( \{ \text{Tables} \}, \{ \text{Joins} \}, \{ \text{Columns} \}, \{ \text{Values} \} \), and each set is defined as below.

- \( \{ \text{Tables} \} \): the set of the tables in \( q \)’s FROM clause.
- \( \{ \text{Joins} \} \): the set of the join relations in \( q \)’s WHERE clause.
- \( \{ \text{Columns} \} \): the set of the columns involved in \( q \)’s WHERE clause.
- \( \{ \text{Values} \} \): 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 \( \text{Card}(q) \) produced by \( \mathcal{M} \) matches or closes to the actual cardinality \( \text{Act}(q) \).
The input of the model \( M \) 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 \( q = (Tables), (Joins), (Columns), (Values) \), we transform \( q \) into a query feature vector \( \bar{f} = (f_r, f_j, f_c, f_v) \), where \( f_r \), \( f_j \), \( f_c \), and \( f_v \) are the features extracted from \( (Tables) \), \( (Joins) \), \( (Columns) \), and \( (Values) \) respectively. The vector \( \bar{f} \) serves as the input to the regression model \( M \). The actual cardinality, \( \text{Act}(q) \), serves as the labels, which guides the model training. Given a training set of labeled queries \( S \), the model \( M \) trained on \( S \) 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 \( \text{Joins}2\text{Vec} \) 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 \( M \) 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 \( M \), and the output of \( M \) is the estimated cardinality together with the corresponding uncertainty. The trained model \( M \) can handle queries joining any subset of tables, with arbitrary range selection.

### 3 QUERY FEATURIZATION

Before using the model \( M \) to estimate the cardinality, we must convert input queries into vectors. A query \( q \) can be represented as \((Tables), (Joins), (Columns), \) and \((Values)\). 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 \((Tables)\) and \((Joins)\) into vectors; Section 3.2 introduces the method to encode \((Columns)\); and Section 3.3 introduces how to represent \((Values)\) of a query.

#### 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 embedding method, and the output is a group of vectors. Each vector is the encoding result for a corresponding table. In a database \( D = \{T_i\}_{i=1}^m \), 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 \([\log(m+1)] \) dimensional vector, where \( m \) is the number of tables in a database. Finally, the \((Tables)\) of a query \( q \) is represented as a vector \( f_T \) with length of \( m[\log(m+1)] \).

**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: JG = (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] = {t};
2. if d = 0 then
3.    Nt := Neighbours(G, t); // Breath First Search
4.    M0 := |GetJoinGraphs(G, t, d − 1)| ∈ Nt;
5.    jG(t)[d] := jG(t)[d] ∪ GetJoinGraphs(G, t, d − 1) ⊆ M0;
6. return jG(t)[d];

The (Joins) 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: Θ ∈ R|JGs|×|d| (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 encode each table t in the database D as a root to build the join graphs. The join graph j(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 j(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 Nt (Line 5), and the neighbours of t are those 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−1] (Line 6), where M(t)[d−1] 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−1] to obtain the intended join graphs jG(t)[d−1] (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

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 contextt[d] is learnt using the process at Lines 15-17 in Algorithm 1. Given the current representation of the target join graph Θ(jG(t)[d]), we want to maximize the probability of every join graph in its context jGcont (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 Joins 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 Joins 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 Tj from the database 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 gj for each table Tj. The graph gj 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 gj by using depth first search to find whether a path exists between their corresponding vertices.

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.
In a database are joinable or not. For instance, assume that there are two tables, and their columns to be encoded. (b) Local columns-dependency graph. (c) Global columns-dependency graph. (d) Use the graph from (c) as the input for the graph embedding method. Finally, (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 (Columns) 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 (Values) of a query. In a database $D = \{T_i\}_{i=1}^m$, any conjunctive query $q$ on numeric columns of the database $D$ can be represented as a subset of $(lb_i^1 \leq Col_i^j \leq ub_i^1) \wedge \cdots \wedge (lb_i^m \leq Col_i^j \leq ub_i^m)$, 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: $(10 \leq Col_i^j \leq 20) \wedge (0 \leq Col_i^k \leq 100)$, \[ \begin{align*} \text{Table 1: Query Features Segmentation} \\
<table>
<thead>
<tr>
<th>Type</th>
<th>Table</th>
<th>Join</th>
<th>Column</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEGMENT</td>
<td>(Tables)</td>
<td>(Joins)</td>
<td>(Columns)</td>
<td>(Values)</td>
</tr>
<tr>
<td>METHOD</td>
<td>Embedding</td>
<td>Joins2Vec</td>
<td>Columns2Vec</td>
<td>RANGE</td>
</tr>
<tr>
<td>SEG. SIZE</td>
<td>$m[\log(m + 1)]$</td>
<td>$n$</td>
<td>$C$</td>
<td>$2 \times C$</td>
</tr>
</tbody>
</table>

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 (Values) of a query: $(lb_i^1, ub_i^1, \ldots, lb_i^m, ub_i^m)$. This vector is used as a part of input features for the model $M$. 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 $L = m[\log(m + 1)] + n + 3C$.

## 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) $S = (q_1 : Act(q_1), \ldots, q_N : Act(q_N))$ with actual cardinality as the label, where $S$ contains $N$ labeled queries. The cardinality variation across different queries in $S$ 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 $S$ 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 Faeze for estimation, we apply inverse transformation to get the true estimations.

**Log transformation.** The log transformation allows the model $M$ 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 $\text{CARD} = \{\text{card}_1, \text{card}_2, \ldots, \text{card}_n\}$, the max cardinality in $\text{CARD}$ ($\text{max}_{\text{CARD}}$), the min cardinality in $\text{CARD}$ ($\text{min}_{\text{CARD}}$), the result of the min-max scaling for $\text{card}_i$ in
CARD is calculated as, \( \text{card}' = \frac{\text{card} - \min\text{card}}{\max\text{card} - \min\text{card}} \). Therefore, the final cardinality estimation is formulated by inverse transformation:
\[
est(q_i) = 2^\text{card}' \times (\max\text{card} - \min\text{card}) + \min\text{card}.
\]

### 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.

### 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 \( M \) based on the generated training data (Section 4.2). In the online phase, the model \( M \) accepts queries and outputs their estimated cardinalities together with the estimation uncertainties. The training first generates a set of labeled queries \( S \) (§4.2). Then we apply query featurization (§3) and cardinality transformation (§4.1) on \( S \) to get the training dataset \( D \). \( D \) consists of \( N \) featurized queries \( \{x_i, y_i\}_{i=1}^N \), where \( x_i \in \mathbb{R}^K \) represents the \( L \)-dimensional query features, and \( y_i \in \mathbb{R} \) 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.
\[
\text{Var}(y) = \text{Var}(E[y|x]) + E[\text{Var}(y|x)] \quad (1)
\]

Based on Equation 1, we define the model uncertainty and data uncertainty as follows.
\[
U_m(y|x) = \text{Var}(E[y|x]) \quad (2)
\]
\[
U_d(y|x) = E[\text{Var}(y|x)] \quad (3)
\]
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 \( E[y|x] \), and the data uncertainty explains the variance inherent to the conditional distribution \( \text{Var}(y|x) \).

**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 \( f_W = \{f_w\}_{i=1}^N \) 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, as shown.
\[
p(y^*|x^*, D) = \int_W p(y^*|f_W(x^*)) p(W|D) dW \quad (4)
\]

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, \( E(y^*|x^*) \approx \frac{1}{K} \sum_{i=1}^K f_m(x^*)^2 - E(y^*|x^*)^2 \). The predictive variance can also be approximated as, \( \text{Var}(y^*) \approx \frac{1}{K} \sum_{i=1}^K f_m(x^*)^2 - E(y^*|x^*)^2 \).
arises because of the uncertainty about the model parameters $W$. We use $Var(y^q)$ 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 $y \sim N(\mu(x), \sigma(x)^2)$, and the negative log likelihood is written as follows.

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

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 $\sigma(x)$ 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 featureized.

### 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 $x' = x + \eta \text{sign}(\nabla_x Loss(W, x, y))$, where $Loss(W, x, y)$ is from Equation 5. Here, $\eta$ 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, $Card(q^*) = E(y|x^*) \approx \frac{1}{K} \sum_{i=1}^{K} \mu_{w_i}(x^*)$. The model uncertainty for the query $q^*$ is calculated with, $U_m(q^*) = \frac{1}{K} \sum_{i=1}^{K} \mu_{w_i}^2(x^*) - E(y|x^*)^2$. The data uncertainty for the query $q^*$ is calculated with $U_d(q^*) = \frac{1}{K} \sum_{i=1}^{K} (\sigma_{w_i}^2(x^*) + \mu_{w_i}^2(x^*)) - E(y|x^*)^2$.

### 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, $\phi_m$ and $\phi_d$ 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, $U_m(q^*) \leq \phi_m$ and $U_d(q^*) \leq \phi_d$. 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 $U_d(q^*) \leq \phi_d$. 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 buffer $B$, 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 $M$ (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 $q^*$
$\phi_m, \phi_d$: Threshold for the model, data uncertainty

**Output:** Safe.Card: Cardinality that is safety to use
$M^*$: updated model $M$

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

**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 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

Figure 5 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 $\phi_m$ and $\phi_d$, then the cardinalities are estimated without executing the query.
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 featureized 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 featureization is small, which usually takes 2-6ms in our evaluation.

Figure 5: Integration of Fauce with existing DBMS

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 optimizer? (§6.3)
- How does the Fauce perform in a dynamic environment? (§6.5)
- How does Fauce perform on data profiling task? (§6.7)

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.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Tables</th>
<th>Rows</th>
<th>Cols</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOB-base</td>
<td>6</td>
<td>2 · 10^2</td>
<td>13</td>
<td>correlated filters</td>
</tr>
<tr>
<td>JOB-more-filters</td>
<td>6</td>
<td>2 · 10^2</td>
<td>22</td>
<td>+ more filters</td>
</tr>
<tr>
<td>JOB-complex-joins</td>
<td>15</td>
<td>2 · 10^3</td>
<td>22</td>
<td>+ complex joins</td>
</tr>
</tbody>
</table>

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_companies, 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, \( error = \max(\text{act}(q), \text{est}(q)) - \text{act}(q) \). Here, we assume that \( \text{act}(q) \geq 1 \) and \( \text{est}(q) \geq 1 \), 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.
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.
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 both model uncertainty and data uncertainty.

<table>
<thead>
<tr>
<th>Estimator</th>
<th>JOB-base (50th, 75th, 90th, 95th, 99th)</th>
<th>JOB-more-filters (50th, 75th, 90th, 95th, 99th)</th>
<th>JOB-complex-joins (50th, 75th, 90th, 95th, 99th)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Postgres</td>
<td>13.4, 623, 1960, 2 \cdot 10^5, 7 \cdot 10^5</td>
<td>8.1, 162, 1429, 1 \cdot 10^4, 2 \cdot 10^3</td>
<td>17.4, 1679, 7928, 4 \cdot 10^3, 8 \cdot 10^3</td>
</tr>
<tr>
<td>IBJS</td>
<td>11.6, 125, 2321, 4 \cdot 10^4, 7 \cdot 10^6</td>
<td>7.6, 77.1, 963, 8 \cdot 10^3, 4 \cdot 10^3</td>
<td>14.5, 239, 5014, 3 \cdot 10^4, 7 \cdot 10^4</td>
</tr>
<tr>
<td>MSCN</td>
<td>6.13, 44.5, 142, 3568, 2 \cdot 10^4</td>
<td>4.8, 16.3, 121, 1680, 5 \cdot 10^4</td>
<td>6.9, 34.4, 163, 2820, 4 \cdot 10^4</td>
</tr>
<tr>
<td>DeepDB</td>
<td>4.61, 17.3, 145, 3348, 3 \cdot 10^1</td>
<td>4.2, 14.3, 86, 1182, 4 \cdot 10^4</td>
<td>5.3, 17.3, 268, 3717, 3 \cdot 10^4</td>
</tr>
<tr>
<td>NeuroCard</td>
<td>3.04, 12.2, 69, 1093, 9 \cdot 10^3</td>
<td>3.8, 8.2, 59, 538, 2 \cdot 10^4</td>
<td>3.5, 8.8, 56.7, 668, 8 \cdot 10^3</td>
</tr>
</tbody>
</table>

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 featureized queries. MSCN does not contain uncertainty information about testing queries. Furthermore, its query featureization 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 1.16 \times 10^{-1}, 2 \times 10^{-1}, and 4.5 \times 10^{-1}, respectively. Compared with NeuralCard, Fauce’s accuracy improvement is from 2.8 \times 10^{-1} to 70 \times 10^{-1}, because the accumulative error caused by the 1D histogram grows as the number of filters grows.

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 \times 10^{-1} to 70 \times 10^{-1}, 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 \times 10^{-1} to 46 \times 10^{-1}. 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 \times 10^{-1}, 3.2 \times 10^{-1}, 5.9 \times 10^{-1}, and 16.7 \times 10^{-1} at median, 75th, 90th, 95th, and 99th respectively. Compared with DeepDB, Fauce improves the accuracy by 1.5 \times 10^{-1}, 2.8 \times 10^{-1}, 4 \times 10^{-1}, 5.7 \times 10^{-1}, and 13.3 \times 10^{-1} at median, 75th, 90th, 95th, and 99th respectively. At last, compared with NeuroCard, Fauce’s accuracy gains is 1.3 \times 10^{-1}, 1.6 \times 10^{-1}, 2.8 \times 10^{-1}, 2.6 \times 10^{-1}, and 6.7 \times 10^{-1} 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 up to 13.3 \times 10^{-1} and 10 \times 10^{-1} 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 their predicates. In contrast, Fauce is robust to this kind of queries. The overall Fauce’s accuracy gain on each quantile is 1.16 \times 10^{-1}, 2 \times 10^{-1}, 4.5 \times 10^{-1}, and 4.5 \times 10^{-1}, compared with NeuroCard.
We evaluate whether the improvement of cardinality estimation in Fauce on query performance.

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 (Figure 7(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 (Figure 7(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 (Figure 7(c)), Fauce improves the performance for 78.2% of the queries.

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 featureization 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 10x 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 $\phi_m$ and $\phi_d$ to control the model uncertainty and data uncertainty respectively. Here, we discuss how we set proper values for $\phi_m$ and $\phi_d$ as the thresholds. In our study, the threshold values for $\phi_m$ and $\phi_d$ 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 $\phi_m$ higher than 0.5 or $\phi_d$ 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 $\phi_m$ or a query’s data uncertainty is below $\phi_d$, 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 $T = T_{i+1} - T_i$. 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 $\lceil n \cdot (\frac{T_f - T_i}{T_{i+1} - T_i}) \rceil$ queries, their cardinalities are estimated using the stale model $M_{\text{stale}}$. For the remaining $\lfloor n \cdot (1 - \frac{T_f - T_i}{T_{i+1} - T_i}) \rfloor$ queries, the updated model $M_{\text{update}}$ 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 update 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 (Figure 11(a)). If the frequency of the data update is high (shown in the left figure in Figure 11(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 Figure 11(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 (Figure 11(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 $q = \langle \text{Tables}, \text{Joins}, \text{Columns}, \text{Values} \rangle$ after the query featurization (§3) has the length of $L = m[\log(m + 1)] + n + 3C$ (see Table 1). As the inter-column correlations have been significantly changed, the features extracted from (Columns) can not reflect the new inter-column correlations. The ratio for the features extracted from (Columns) among the total length of the feature vector is: $\frac{C}{L}$, where $C$ is the length of features extracted from (Columns). Here, the schema of the database remains the same, so the features extracted from (Tables) and (Joins) 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 (Values) 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.
We compare the efficiency of Fauce with other four data profiling methods: Pyro [27], Tane [22], Ducc/Dfd [1], and Fdep [12]. 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. (2) Impact of cardinality transformation. We use log transformation and min-max scaling (§4.1) to normalize the cardinalities before model training. Table 5 shows that the transformation reduces the error, especially the high-quantile errors. This is because the cardinalities of queries are skewed, and the models directly trained on such skewed dataset have large errors. Using the transformation reduces the gap between large and small cardinalities.

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 6 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 (10^4 s). 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 RefIns 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.

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 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.


