Learned Cardinality Estimation: A Design Space Exploration and A Comparative Evaluation

Ji Sun[†*], Jintao Zhang[†*], Zhaoyan Sun[†], Guoliang Li[†], Nan Tang[‡]
[†]Department of Computer Science, Tsinghua University, [‡]QCRI
{sun-j16,sunzy18}@mails.tsinghua.edu.cn,jtzhang6@gmail.com,liguoliang@tsinghua.edu.cn,ntang@hbku.edu.qa

ABSTRACT

Cardinality estimation is core to the query optimizers of DBMSs. Non-learned methods, especially based on histograms and samplings, have been widely used in commercial and open-source DBMSs. Nevertheless, histograms and samplings can only be used to summarize one or few columns, which fall short of capturing the joint data distribution over an arbitrary combination of columns, because of the oversimplification of histograms and samplings over the original relational table(s). Consequently, these traditional methods typically make bad predictions for hard cases such as queries over multiple columns, with multiple predicates, and joins between multiple tables. Recently, *learned* cardinality estimators have been widely studied. Because these learned estimators can better capture the data distribution and query characteristics, empowered by the recent advance of (deep learning) models, they outperform non-learned methods on many cases. The goals of this paper are to provide a design space exploration of learned cardinality estimators, and to have a comprehensive comparison of the SOTA learned approaches so as to provide a guidance for practitioners to decide what method to use under various practical scenarios.

PVLDB Reference Format:
Ji Sun,Jintao Zhang,Zhaoyan Sun,Guoliang Li,Nan Tang. Learned Cardinality Estimation: A Design Space Exploration and A Comparative Evaluation. PVLDB, 15(1): XXX-XXX, 2022.
doi:10.14778/3485450.3485459

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/jt-zhang/CardinalityEstimationTestbed.

1 INTRODUCTION

The problem of cardinality estimation is vital to DBMS query optimizer [14, 26]. Despite of its importance, the cardinality estimators in modern DBMSs are still suboptimal (i.e., the error is even higher than 10,000 on some queries), which is mainly due to the inherent hardness of estimating complicated queries and the increasing complexity of data, for complicated predicates and multiple tables.

Non-learned Methods. These include histograms and samplings [20, 27, 29, 31, 38], which are also referred to as *traditional methods.* A histogram is an approximate representation of the distribution of numerical data. Roughly speaking, it first divides the entire range of values into a series of intervals, and then counts how many values fall into each interval. Histograms can be used for either one column, or multiple columns. *Sampling*-based methods [27, 29, 31, 38] sample tuples from database, and then apply given queries to these samples. The cardinality on the full dataset can be estimated based on the result on samples, e.g., by scaling up the number from the samples to the entire dataset.

Limitations. Histogram-based methods typically rely on the Attribute Value Independence (AVI) assumption and fall short of capturing the correlations among many and arbitrary columns. Sampling-based methods assume that the distribution of samples is identical to the full dataset, which is often violated in practice.

Learned Query Models. They learn *a mapping function* between an SQL query and its cardinality on a database. They treat cardinality estimation as a typical regression problem. They first train query models using trained queries and their corresponding cardinalities, and then use the trained models to estimate the cardinalities of online SQL queries. Many models – including statistic-based models (e.g., XGBoost [12]) and neural networks (e.g., Multi-layer perceptron [12, 37]) – can be utilized to train query models.

Learned Data Models. They treat cardinality estimation as a density estimation problem, which learns a joint data distribution (e.g., Gaussian distribution or uniform distribution) of each data point. These models could be learned in either an unsupervised or a supervised fashion. **Unsupervised data models** directly learn from the data (e.g., autoregressive model [15, 40, 41] and sum product network [17]). **Supervised data models** learn by using some SQL queries and their real cardinalities (e.g., kernel-based density estimation (KDE) [16, 22] based on Gaussian models and uniform mixture model [33]). Given an SQL query, these methods first sample some points that satisfy the query, and then sum up the probability of these sampled data points to estimate the cardinality.

Our Goals. Because learned methods have shown superior performance than non-learned methods for cardinality estimation [23, 36, 41], we will focus on learned cardinality estimators in this paper. In particular, we have three main goals:

(1) ***A design space exploration.*** We define a space of learned solutions for cardinality estimation. We provide a categorization of these solutions that factors out their commonalities. We further present a unified workflow to show how different design choices are materialized to form different solutions.

(2) ***A comparative evaluation.*** We design a comprehensive comparison of different learned solutions by varying many parameters. We summarize the results to guide practitioners to make the right decision under various practical scenarios.

Proceedings of the VLDB Endowment, Vol. 15, No. 1 ISSN 2150-8097.
doi:10.14778/3485450.3485459

[*] The first authors contributed equally. Guoliang Lis is the corresponding author.

		Model	Method	Parameter Optimizer	SQL Parser	Sampling	Join Decomposition
Data Model	Unsupervised	Bayesian Network	Baysian	/	Column Predicates	/	Fanout Scaling
		Autoregressive	NeuroCard	Gradient-based	Column Predicates	Progressive Sampling	Fanout Scaling
			Naru	Gradient-based	Column Predicates	Progressive Sampling	/
			DLM	Gradient-based	Column Predicates	Importance Sampling	/
		Sum Product Network	DeepDB	Gradient-based	Column Predicates	/	Fanout Scaling
	Supervised	Gaussian Kernel	Feedback-KDE	Gradient-based	Column Predicates	Random Sampling	/
		Uniform Mixture Model	QuickSel	Analytic-based	Column Ranges	Query-aware Sampling	/

		Model	Method	Parameter Optimizer	Feature	Encoding
Query Model	Supervised	XGBoost	LocalXGB	/	Column Ranges	range bounds vector
		Multi-set Convolutional Network	MSCN	Gradient-based	Join Conditions/Tables/Filter Conditions/samples	one-hot/normalized float/bitmap
		Recurrent Neural Network	RNN	Gradient-based	Join Conditions/Tables/Filter Conditions	one-hot/normalized float
		Neural Network	LocalNN	Gradient-based	Column Ranges	range bounds vector

Figure 1: A summary of the SOTA learned cardinality estimators

(3) ***A cardinality-estimation testbed.*** We provide a testbed with many reusable components, which can facilitate researchers/practitioners to design new cardinality-estimation models for ad-hoc applications with lower design and implementation overhead.

Experimental Findings. We have tested eight state-of-the-art learned methods, including learned query models and data models, on four real-world datasets (with one or several tables) and 256 synthetic datasets to comprehensively test different methods under different conditions (e.g., the number of columns, the number of distinct values, the data skewness, the number of training queries, and so forth). We summarize our main experimental findings below.

1. Data Models DeepDB and Naru are the most effective methods for single tables.
2. Query Model MSCN is the most effective for multiple tables.
3. Query Models are more efficient than Data Models.
4. Data Models are more robust than Query Models.
5. Training queries are vital to Query Models.
6. Samples are crucial to Data Models.
7. Estimators based on neural network are more accurate than statistic-based estimators.
8. Statistic-based query model is the most efficient.

2 A DESIGN SPACE EXPLORATION

Let $\mathcal{D}$ be a database with a set of relational tables $\{T_1, T_2, \cdots, T_n\}$. Each table $T \in \mathcal{D}$ consists of a set of attributes as $\{A_1, A_2, \cdots, A_m\}$. Each row in a table is denoted as $r = \{v_1, v_2, \cdots, v_m\}$ where $v_j = r[A_j]$ for $i \in [1, m]$. Let $P(r)$ be the probability of the tuple r in the corresponding table T.

Cardinality Estimation. Given a database $\mathcal{D}$ and an SQL query Q, the *cardinality* of Q *w.r.t.* the database D, i.e., $|Q(\mathcal{D})|$, is the number of rows returned by executing the query Q over the database $\mathcal{D}$. The problem of *cardinality estimation* is to predict the cardinality $|Q(\mathcal{D})|$ without actually executing the query Q over $\mathcal{D}$.

Learned Cardinality Estimation. The problem of *learned cardinality estimation* is to learn a method $f()$ (or an ML-based cardinality estimator), such that $f(Q, \mathcal{D})$ can give an estimated cardinality $|\widehat{Q(\mathcal{D})}|$, with the object that $|\widehat{Q(\mathcal{D})}|$ is as close to $|Q(\mathcal{D})|$ as possible.

The Design Space. Figure 1 shows a unified view for different learned cardinality estimators (more details will be discussed in Section 3). Next, we provide more details for query modeling (Section 2.1) and data modeling (Section 2.2).

2.1 Query Modeling

Problem. They learn *a mapping model* $f(Q)$ between an SQL query Q and its cardinality $|Q(\mathcal{D})|$ on database $\mathcal{D}$. They treat cardinality estimation as a typical regression problem in ML. They first train the model $f(\cdot)$ and then use the model to estimate the cardinality of an SQL query. Note that methods for query modeling must be supervised. Next, we give a unified design for this case.

The Design Space for Supervised Query Methods.

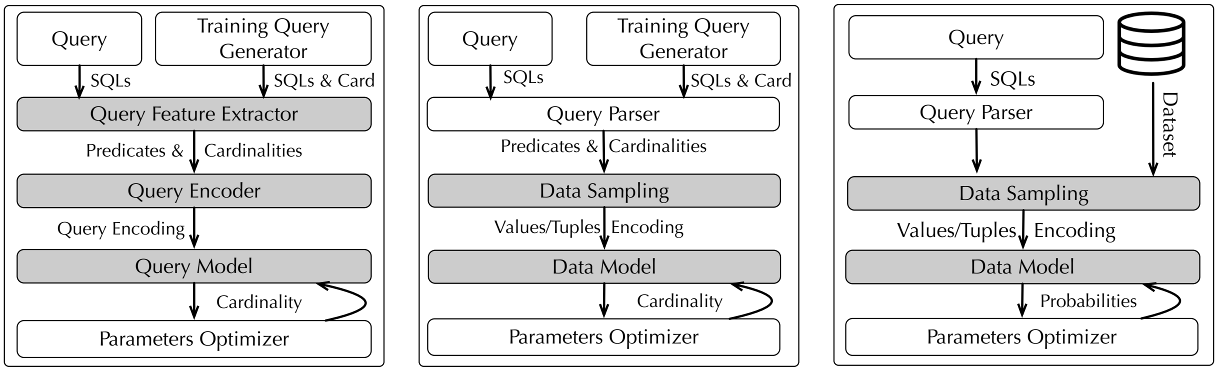
Figure 2(a) depicts a unified workflow for this case.

Model Training. We build a ***Training Query Generator*** for testing all supervised cardinality estimation methods. The generator first samples query tables and columns from schema, and then samples values from each column for predicates. We also build a unified ***Parameter Optimizer*** to train the model. Practitioners need to specify *Query Feature Extractor*, *Query Encoder* and *Query Model* modules. More specifically, practitioners should first decide which features are useful for estimating the cardinality in a query (e.g., tables, predicates, join conditions), i.e., the *Query Feature Extractor* module. They then need to encode all features in a single vector (e.g., one-hot encoding), i.e., the *Query Encoder* module. Afterwards, they select appropriate model for modeling the query features, i.e., the *Query Model* module. The *Query Model* aims to solve a regression problem, and learns the mapping between query and cardinality.

Model Inference. The inference phase is similar to the training phase. All modules in training can be reused here. Moreover, if the query is a join query, the architecture selects a model corresponding to the join pattern, and estimates the cardinality.

Training Data. It is a set of tuples $(Q, \mathcal{D}, |Q(\mathcal{D})|)$, where Q is a query, $\mathcal{D}$ is a dataset and $|Q(\mathcal{D})|$ is the real cardinality of Q.

Query Models. Many models, including both traditional statistic-based models and neural networks, can be utilized to solve the



(a) Supervised Query Methods **(b) Supervised Data Methods** **(c) Unsupervised Data Methods**

Figure 2: Unified workflows for different methods

problem. (1) XGBoost [35] is a statistic-based model, which employs a tree-based ensemble method. Statistic-based models are light-weighted and fast, but they fall short of supporting complex queries and joins. (2) Neural networks use more parameters and gradient-based parameter optimizer, and are more powerful to fit complex distributions. When estimating the cardinality of a query, they first extract and encode features of the query Q, i.e., the embedding of Q denoted by $\mathbf{Q}$, and then use different models to learn the mapping from a query to a cardinality. Different proposals use different DL models (see Section 3.3 for more details).

2.2 Data Modeling

Problem. They treat cardinality estimation as a density estimation problem, which learns a joint data distribution (e.g., Gaussian or uniform distribution) of each data point. Then, given an SQL query, they sample some points that satisfy the query, and sum up the probability of these sampled data points to estimate the cardinality.

Generally speaking, there are two types of methods to learn data distributions: supervised and unsupervised.

Supervised Data Model Training. Supervised data models learn the data distribution by using some SQL queries and their real cardinalities using e.g., kernel-based density estimation (KDE) [16, 22] based on Gaussian models.

Unsupervised Data Model Training. Unsupervised data models directly use the data to train the models. During the training phase, they scan dataset (or data samples) to learn the probability of different values.

Inference. After these models have been trained to learn the data distribution, estimating the cardinality of a query is typically done by uniformly sampling tuples and estimating the cumulative probability of selected tuples from samples.

Data Models. Probabilistic graphic model, neural network, and statistic-based model can be utilized to solve the problem. (1) Probabilistic graphic model is a graph/tree structured model, which can fit data distribution with conditional independent assumption. Bayesian network [13] splits a dataset by columns. If two columns are correlated, they are connected by a directed edge. Sum product network splits the dataset into partitions by rows and columns, and probabilities of partitions are merged by sum and product operators. (2) Autoregressive model [15, 40] is a neural network model. It factorizes the joint distribution as $P(v) = P(v_1)P(v_2|v_1), \cdots, P(v_m|v_1, v_2, \cdots, v_{m-1})$ and solves conditional probability estimation problems for joint distribution estimation. Autoregressive model does not make any independent assumption, and can fit distribution well. (3) Gaussian-based kernel model [16] and Uniform mixture model [33] are statistic-based data models. Gaussian-based kernel model builds smooth kernel models on randomly selected samples and Uniform mixture model is the weighted sum of several uniform distributions. Statistic-based data models are light-weighted and can also learn from queries.

The Design Space for Supervised Data Methods. Figure 2(b) shows a unified workflow for this case.

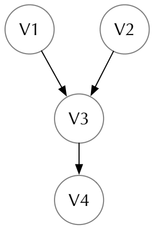
Model Training. It also needs the same ***Training Query Generator*** as discussed for supervised query methods. We also offer a unified ***Query Parser***, which can apply the query predicates on any data tuple to output 1 if selected; and 0 otherwise.

In this case, practitioners should first sample tuples from dataset by using random sampling or query-based sampling, i.e., the *Data Sampling* module. It then builds a distribution model based on those samples, i.e., the Data Model module. The distribution model will output the probability of the query range, and the estimator can train a unique model for each join pattern in a workload to support different join queries.

Model Inference. Practitioners should select a data sampling method (which may not be the same as data sampling for training). The cumulative densities of all selected samples are the estimated selectivity, from which we can easily induce the cardinality. For join queries, practitioners can either combine models on different join patterns or conduct join decomposition.

The Design Space for Unsupervised Data Methods. Figure 2(c) shows a unified workflow for this case.

Model Training. It uses the same ***Query Parser*** as used in supervised data methods. These methods learn joint data distribution from datasets. Practitioners should first sample reasonable amount of dataset uniformly, i.e., the Data Sampling module. They then input the data tuples into data model to learn the joint distributions, i.e., the *Data Model* module. Note that, if the dataset is too large (e.g.,



V1	probability
0	0.4
1	0.6

V2	probability
0	0.7
1	0.3

V1	0		1	
V2	0	1	0	1
V3	probability			
0	0.1	0.3	0.2	0.4
1	0.5	0.1	0.1	0.3

V3	0	1
V4	probability	
0	0.1	0.3
1	0.5	0.1

Figure 3: An example of bayesian network

n-table outer joins) to keep in memory, online sampling methods (e.g., weighted join sampling) should be considered.

Model Inference. The model inference phase is the same as model inference for supervised data methods.

3 CATEGORIZING THE STATE OF THE ART

Figure 1 summarizes the SOTA learned cardinality estimators under each category. In particular, for each method, it provides the used parameter optimizer, SQL parser, sampling method, join decomposition for data models, and the used parameter optimizer, features and encoding methods for query models.

3.1 Unsupervised Data Model

The basic idea of unsupervised data model is to learn the joint data distribution directly from the dataset. The joint data distribution of table T is an aggregation of probabilities of all tuples in T, which is denoted as $P(r = (v_1, v_2, \cdots, v_m))$. Cardinality estimation by unsupervised data model aims to estimate the cumulative probabilities of tuples selected by an SQL query. Existing methods use different types of models (e.g., probability graph and neural network) to fit the joint data distribution.

Probabilistic Graph Models (PGM). *Probabilistic graphic model* is a graph/tree structured model, where each node denotes a part of dataset (e.g., columns/rows), and each edge denotes the dependencies of different parts. There have been two lines of research for cardinality estimation under probabilistic graph models (PGM), based on either Bayesian networks or Sum Product Network (SPN).

▶ **Bayesian.** Bayesian network constructs a directed acyclic graph (DAG) based on the dependencies between every two columns, where the dependency is computed by searching methods, e.g., search by K2 score [9, 24]. The distribution of each column is conditioned by its parents. Bayesian [10] adopts Bayesian Network to estimate cardinality. Bayesian takes each attribute in table T as a variable, constructs the DAG to model the data distribution, and estimates the cardinalities based on the learned data distribution.

For example, Figure 3 shows 4 columns (or variables), v_1, v_2, v_3 and v_4, where v_3 depends on v_1 and v_2, and v_4 depends on v_3. Conditional independence (i.e., no edge between two nodes) means that v_4 and v_1 are independent, and $P(v_1, v_2, v_3)$ can be factorized as $P(v_1)P(v_2)P(v_3|v_1, v_2)P(v_4|v_3) = P(v_1)P(v_2)\frac{P(v_1,v_2,v_3)}{P(v_1,v_2)}\frac{P(v_3,v_4)}{P(v_3)}$. Since Bayesian network makes conditional independent assumption, the probability of each variable is only conditioned by its parents, and the joint distribution is a product of conditional probabilities of ancestor variables.

Model Training. Bayesian learns the DAG structure and its conditional distributions. First, it conducts an outer join on all tables into a large table $\mathbf{T} = T_1 \bowtie T_2 \bowtie \cdots \bowtie T_n$ and records the fanout of each join key in table $\mathcal{F}$. It then searches an optimal probability graph structure according to the dependency scores among different attributes of $\mathbf{T}$. Second, it factorizes the joint distribution according to the graph structure with conditional independent assumption, and it can get an explicit function mapping marginal distributions to joint distribution.

Consider an example in Figure 4, Bayesian is built on $T_1 \bowtie T_2 \bowtie T_3$, and all the joins are primary key-foreign key joins. Column **Freq()** records the frequency of each join key in the foreign key side table. Given a query with only T_1, it first finds the marginal distribution of T_1 in $T_1 \bowtie T_2 \bowtie T_3$, and reduces the probability of each row according to the foreign key frequencies. The probability of the first row of T_1 is reduced by a factor $2 \times 3 = 6$.

Model Inference. It first finds distinct values selected by given query in each column. It then combines them into tuples, and finds the probabilities for all distinct tuples from the model and sums them up. It can also answer different join queries based on fanout scaling techniques proposed in [40].

▶ **Sum Product Network (SPN) [DeepDB].** SPN is also a probabilistic graphic model which can fit joint data distribution. As different tuples may have different distributions, SPN splits the tuples via clustering such that the tuples in the same cluster (sub-node) have similar data distribution. Then SPN uses a SUM operator to add the estimated cardinalities of the clusters (sub-nodes). As different columns may have correlations, SPN partitions the columns into different groups (sub-nodes) via column correlation such that the columns in the same group have high correlations. Then SPN uses a PRODUCT operator to multiply the estimated cardinalities in different groups (sub-nodes). So SPN contains two types of shared nodes, Sum and Product. The complexity of deep SPN model is polynomial, which is much smaller than the Bayesian network.

Model Training. DeepDB [17] adopts SPN to support cardinality estimation. DeepDB constructs the SPN model in three phases. First, DeepDB outer joins all tables as dataset $\mathbf{T} = T_1 \bowtie T_2 \bowtie \cdots \bowtie T_n$. Second, DeepDB recursively splits the dataset (data clustering for rows and correlation identifying for columns). Third, DeepDB learns the weights of edges connecting to sum nodes by fitting the joint probability of data samples, and the weights decide how does each partition contribute to the joint probabilities.

Model Inference. For point and range queries, DeepDB calculates selectivities for selected tuples from leaf nodes to the root. For join query, if the query only contains a subset of tables, then DeepDB uses fanout scaling technique to calibrate the selectivity. For example, given a dataset as shown on the left of Figure 5. The SPN can be constructed in two steps. The first two rows are assigned to the left of a sum node, and columns production_year and kind_id are partitioned by the product node. The weights of sum terms can be learned by gradient decent. In this example, the joint distribution is approximated by the weights 0.8 (left) and 0.2 (right).

Autoregressive Model. Autoregressive (AR) model is designed to predict the next value for a given sequence of values. It factorizes

	T1		T2			T3
#row	Attr	PK	Freq(T2.FK)	Attr	PK	Freq(T3.FK)
1	a	1	2	b	2	3
2	b	1	2	b	2	3
3	c	2	1	a	2	3
4	d	2	1	a	1	2
5	e	2	1	c	1	2

Figure 4: An example of fanout scaling table ($\mathcal{F}$)

production_year	kind_id	probability
1994	1	0.38
1995	1	0.39
1994	2	0.10
1995	2	0.11

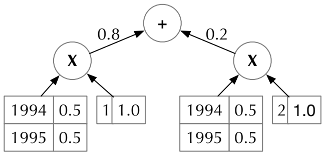


Figure 5: An example of Sum Product Network

the joint distribution as:

$$P(v) = P(v_1)P(v_2|v_1), \cdots, P(v_m|v_1, v_2, \cdots, v_{m-1}) \qquad (1)$$

It first predicts, for one input tuple, a list of conditional distributions, where each being a probability of the i-th attribute conditioned on previous attributes. It then integrates probabilities of data sample selected by queries and estimates the cardinality. Next, we present three cardinality estimation methods using autoregressive models, namely `Naru` [41], `DLM` [15] and `NeuroCard` [40].

▶ **`Naru`**. In `Naru` [41], a sequence is a row of dataset and each variable is a column in table T. `Naru` encodes each row into a vector: encoding each discrete value in a column as a one-hot vector if the number of distinct values on that column is small (e.g., 10); otherwise, encoding each value by dense embedding. Then `Naru` scans the dataset, and sends encoded rows into an AR model. Next, `Naru` computes cross entropy loss for each output value and input value, and minimizes the mean loss by using gradient-based parameter optimization. For a point query, `Naru` encodes the selected row and gets the probability by comparing with output vectors from AR model. For a range query, `Naru` samples dataset column by column. On each column, it selects values according to the probabilities conditioned by previous column samples, and the previous samples are all in the query range. `Naru` multiplies these conditional probabilities and estimates the selectivity.

▶ **`DLM`**. `DLM` [15] also treats a row of dataset as a sequence. However, `DLM` encodes each value as a binary vector according to the value id and takes each bit as a variable in the sequence. Then `DLM` optimizes the parameters of an AR model to minimize the cross entropy loss. For a point query, `DLM` multiplies the conditional probabilities of bits in vector to estimate the cardinality. For a range query, `DLM` uses *Adaptive Important Sampling* to improve the uniform sampling (i.e., the probability of a row being sampled relies on previous samples). `Naru` and `DLM` show similar performance as verified in [15].

▶ **`NeuroCard`**. `NeuroCard` [40] extends `Naru` to support join queries.

Model Training. Given a dataset $\mathbf{T} = T_1 \bowtie T_2 \bowtie \cdots \bowtie T_n$, `NeuroCard` uniformly picks join samples T_s with all the columns by using weighted sampling. In each training epoch, the model scans T_s in batches. For each batch of rows, `NeuroCard` first lookups an embedding for each attribute value and input into the model. The model then updates its parameters to reduce the cross entropy loss between input and output.

Model Inference. Given a query Q, `NeuroCard` uses progressive sampling to obtain the marginal probability of each value given values being selected in previous columns by Q. In this way, we can easily estimate the cardinality of Q if Q contains all tables. While if Q only contains a part of tables, `NeuroCard` also uses progressive sampling to get a small sample of rows with probabilities, and then `NeuroCard` calibrates probabilities according to the fanout scaling coefficient (as shown in Figure 4). Afterwards, `NeuroCard` computes the cardinality of query Q.

3.2 Supervised Data Model

The basic idea of supervised data model is to learn data distributions from query cardinalities. In general, the supervised data model should be able to optimize parameters by minimizing the loss between predicted cardinalities and the true ones for all queries. Cardinality estimation with supervised data model aims to estimate the cumulative probabilities of distinct tuples selected by the query by utilizing density models.

Kernel Density Model. *Kernel density model* builds smooth kernel models on samples, and the probability at random tuple is the sum of outputs from all models. It formalizes the probability as:

$$P(v) = \frac{1}{N \cdot B} \sum_{h=1}^{N} f(\frac{v - S_h}{B}) \qquad (2)$$

where N is the number of samples, S_h is a sample and B controls the scale of each kernel model, and f is a smooth function (e.g., Gaussian) for easier computation. It integrates probabilities of all data sample selected by queries and estimates the cardinality.

▶ **`Feedback-KDE`**. `Feedback-KDE` [16] integrates Gaussian kernel-based estimator into cost estimator of PostgreSQL. It assumes that each tuple follows a Gaussian distribution and aims to learn the bandwidth parameter.

Model Training. `Feedback-KDE` first randomly selects sample tuples from dataset, and builds Gaussian models on data samples. Then `Feedback-KDE` collects training queries and optimizes the bandwidth of Gaussian kernel by using true cardinalities of training queries. The parameter optimizer can be any gradient-based optimizer. As `Feedback-KDE` directly computes the cumulative distribution without density on each tuple, it cannot support join queries by using fanout scaling. Instead, we train Gaussian models for all possible join patterns to support different join queries.

Model Inference. Given a query Q, `Feedback-KDE` finds the model with the same join pattern. If Q is a point query, `Feedback-KDE` computes values of all Gaussian models of different samples on the query point. If Q is a range query, `Feedback-KDE` deduces the integral formation of cumulative probability, and computes the selectivity within the range.

Uniform Mixture Model. Uniform Mixture Model (UMM) is a member of mixture model family [28], it is a weighted sum model based on several uniform functions, and the density at point v can be formalized as:

$$P(v) = \sum_{h=1}^{N} w_h \cdot f_u^h(v) \tag{3}$$

where each uniform distribution f_u^h is defined within a multi-dimensional range R_h, and the centroid of the range is a sample S_h drawn from a dataset. We have $0 < f_u^h(v) < 1$ if $v \in R_h$; otherwise $f_u^h(v) = 0$. Since $\sum_{v \in R_h} f_u^h(v) = 1$, we can infer that $f_u^h(v) = \frac{1}{|R_h|}$. Parameters w_i can be learned from query cardinalities, and it's easy to answer probability density at any given point.

Given a range R_Q defined by query Q, the cumulative probability can be calculated from the overlaps of R_Q and all sample ranges $\{R_1, R_2, \cdots, R_h\}$. This can be formulated as:

$$\int_{R_Q} P(v)dv = \sum_{h=1}^{s} w_h \frac{|R_h \cap R_Q|}{|R_Q|} \tag{4}$$

where $|R_Q|$ is the volume of query range, s is the number of samples.

► **`Quicksel`**. `Quicksel` [33] uses Uniform Mixture Model to fit the cardinalities of given training queries and constructs a density model to estimate cardinality. Query range can be defined by predicates in SQL. For example, given an SQL query "*SELECT * FROM A WHERE A.a between 1 and 3 and A.b between 2 and 5 and A.c between 10 and 13*", then the query range R_Q is a rectangle whose volume is $2 \times 3 \times 3 = 18$.

Model Training. Given a set of training queries, `Quicksel` randomly selects samples from query ranges and builds uniform models on them. It then computes the overlaps between query range R_Q and sample ranges R_h to get $|R_h \cap R_Q|$, and next the weights optimization problem can be solved by quadratic programming. It transforms both point and range predicates to range predicates, and computes overlaps and the cumulative distribution (i.e., selectivity) according to the mixture density function. Similar to `Feedback-KDE`, we also train UMMs for all possible join patterns.

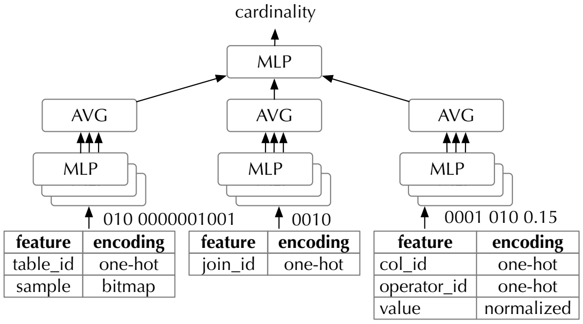
Model Inference. Given a query Q, `Quicksel` first finds the mixture model with the same join pattern with Q, and then `Quicksel` computes the overlaps between query range and sample ranges to get $|R_h \cap R_Q|$, and next `Quicksel` computes selectivity by weighted summing all overlaps.

3.3 Supervised Query Model

This line of research aims to learn a function mapping query and cardinality. Supervised query model is suitable for two scenarios: (1) full data is not available but query logs are available, (2) query is similar but complicated (e.g., non-key join).

Neural Networks (NNs). NNs [18, 25] are a powerful tool to learn the representations of complex structures. It is composed of linear computation units (i.e., neurons) and activation functions. Different applications require customized NN model. Generally, the design space of NN includes (1) neural network structure, (2) loss functions, and (3) feature encoding.

► **Multi-set Convolutional Network (MSCN).** MSCN [23] proposes multi-set convolutional neural network to model SQL queries. MSCN divides an SQL query into three sets, including tables in FROM clause, join conditions and filter conditions. As Figure 6 shows, inputs are transformed by fully connected neural network. Table is encoded as a global table id and a sample bitmap. Samples are selected from each table and the bitmap is a binary vector indicating which sample rows are selected by the query. Join condition is encoded as global join ID. Predicate is encoded as column id, operator id and a normalized numeric value. For each set, all the embeddings are reduced to one vector by average pooling layer, and three vectors are concatenated and fed into the final MLP neural network. The final layer outputs the min-max normalized cardinality. MSCN naturally supports join queries, and has good generalization for different join patterns.



feature	encoding
table_id	one-hot
sample	bitmap

feature	encoding
join_id	one-hot

feature	encoding
col_id	one-hot
operator_id	one-hot
value	normalized

Figure 6: An example of MSCN

► **Fully-connected Neural Network.** Local Neural Network [37] considers different predicates on a fixed join path. Comparing to the representation learning for arbitrary query, learning for a join path is easier because of the smaller query space. Moreover, if join conditions are fixed, the key features of queries on joined table are predicates only and an MLP layer is enough for modeling this. Input of each Local NN is a vector where each 4 number encodes a filter predicate on an attribute in joined table, first three binary number indicates the operation (<, >, =), and the last number is the normalized value. To make the vector same size for all queries, each attribute has and only has one predicate encoding position, and a predicate is encoded as zeros if nonexistent. Although Local NN is more light-weighted than MSCN, it builds a model for each join pattern, and needs to train several hundred models.

► **Recurrent Neural Network (RNN).** RNN [32] is widely used in Natural Language Processing because it's expert in modeling sequence. An SQL query can be viewed as a meaningful sequence. For example, *"SELECT * FROM A, B where A.id = B.id and A.year < 2010 and B.type_id = 5;"* can be translated as two steps, (i) select table A with predicates $A.year < 2010$, and (ii) join with table B using predicates $A.id = B.id$ and $B.type\_id = 5$. Each step is encoded as a vector with table id, filter selectivities and join conditions, and the output of hidden layer of a step would be fed into the next step, and the model outputs the cardinality finally. In paper [32], table id is encoded with one-hot encoding, selectivity is encoded as a float number (zero) for each column with (without) predicates, and a join condition is encoded with a binary indicator.

Statistical (Tree-based Ensembles).

Ensemble methods improve the accuracy of simple regression models (e.g., decision tree), and can be divided into two categories, bagging[7] and boosting[21, 34]. Bagging method trains models

Table 1: Testbed for Learned Cardinality Estimators

Methods	Datasets	Join Sampling	Join Pattern Grouping	Training Queries
Bayesian	√	√	×	×
NeuroCard	√	√	×	×
Naru	√	×	×	×
DeepDB	√	×	×	×
MSCN	×	×	×	√
Quicksel	×	×	√	√
LocalNN	×	×	√	√
LocalXGB	×	×	√	√
Feedback-KDE	√	√	√	√

on L subsets uniformly sampled from training sets, and gets final results by averaging or majority voting (e.g., Random Forest). Boosting method sequentially regresses the residual error produced by previous models, and adds all outputs together. Generally speaking, tree-based boosting method is composed of a set of CART regression tree-based models [8], and each model fits the residual error produced by previous models. Each CART regression model recursively splits feature dimensions with the highest gain (i.e., the ground truth is more concentrated in each partition after splitting).

▶ **XGBoost.** XGBoost [35] proposes a tree-based ensemble method which encodes query as a sequence of selection ranges. For example, given a query q on table A with attributes x_1, x_2 and x_3, “*SELECT * FROM A WHERE* x_1 *<= 3 AND* x_1 *>= 1 AND* x_2 *<= 10 AND* x_2 *>= 3 AND* x_3 *<= 100 AND* x_3 *>= 50*”. The query q can be encodes as [1, 3, 3, 10, 50, 100], and if some columns have no predicate, they are considered as selection range from minimum to maximum.

4 EXPERIMENT

We conducted a comprehensive comparison to answer the following questions. (Exp-1) What is the overall comparison result of learned methods on real datasets? (Exp-2) How does the number of columns affect the accuracy? (Exp-3) How does the number of distinct values affect the accuracy? (Exp-4) How does the correlation between columns affect the accuracy? (Exp-5) How does the skew of columns affect the accuracy? (Exp-6) How does the size of training set affect the accuracy of supervised methods? (Exp-7) How does the size of join samples affect the accuracy of unsupervised methods? (Exp-8) What is the efficiency of training and estimation? (Exp-9) What is the efficiency for incremental data updates?

4.1 Experimental Setting

Methods. Table 1 summarizes the implementation techniques used in our testbed for different methods: a “✓” means that a column (e.g., Datasets) is needed for a method (or a row e.g., Bayesian). We prepared datasets for data models Bayesian, NeuroCard, Naru, DeepDB and Feedback-KDE. We conducted weighted join sampling [42] for Bayesian, NeuroCard, and Feedback-KDE. We trained on each join pattern for Quicksel, LocalNN, LocalXGB and Feedback-KDE. We provided generated training queries for supervised methods MSCN, Quicksel, LocalNN, LocalXGB and Feedback-KDE.

- **Bayesian.** We implemented Bayesian based on the package PyPGM [6]. In order to avoid “*Out Of Memory*” exception for large domains, we discretize distinct values in groups.

Table 2: Statistics of Datasets

Dataset	#Table	#Rows	#Columns	Domain Size
Forest [11]	1	581K	9	10^{24}
Power [11]	1	2.1M	7	10^{17}
IMDB [19]	6	1.3M-36.2M	12	10^{30}
XueTang [39]	5	8.5M-9.9M	10	10^{31}
Synthetic	256	500K	2-8	10^2–10^{32}

Table 3: Synthetic dataset parameters

#Columns	2, 4, 6, 8, 12	Correlations	0.2, 0.4, 0.6, 0.8
#Distinct Values	10, 100, 1,000, 10,000	Skew	0.2, 0.4, 0.6, 0.8

- **NeuroCard.** We adopted the source code implemented by the authors [5], and extended the dataset reader module for our datasets.

- **DeepDB.** We employed the source code implemented by the authors [2], and extended the dataset reader module for our datasets and support single table queries.

- **MSCN.** We used the source code implemented by the authors [1], and extended sample bitmap generator for our workloads.

- **Quicksel.** We adopted the source code by the authors [3], and modified the code and adapted it to support different join queries.

- **LocalNN.** We implemented LocalNN with PyTorch-1.8, and trained a model for each join pattern, in order to support different join queries.

- **LocalXGB.** We coded LocalXGB with XGBoost-1.4, and trained a model for each join pattern to support different join queries.

- **Feedback-KDE.** We deployed Feedback-KDE [4] in our system, and built kernel-based models on uniform join samples to support different join queries. We also implemented a unified interface for Feedback-KDE by using python.

Remark. Both our experiments and existing comparison [15, 32] show that Naru and DLM are very similar to NeuroCard in model performance, and RNN is similar to LocalNN. Therefore, we do not include DLM and RNN in our evaluation due to the space limitations. We use Naru for synthetic dataset and NeuroCard for real datasets, because NeuroCard supports join queries.

Datasets. We conduct experiments on both real datasets and synthetic datasets, as shown in Table 2, where the first four are real-world datasets and the last one is for synthetic datasets.

- Forest and Power [11] are widely used for testing cardinality estimation methods. The data types of them are all integers and can be supported by any method naturally. By using these datasets, experiments can focus on the performance of all the methods for multi-attribute cardinality estimation.

- IMDB is widely used in query optimization and cost estimation for joins [26], because of the high skewness and correlations. We select 12 columns from 5 tables, and columns have different numbers of distinct values. All 6 tables are joined by using key *movie_id* and *id*(title). Note that, (1) although *phonetic_code* is string type data, it supports range queries by an alphabet order, and thus we update values in *phonetic_code* column to order ids for running all methods on it; and (2) null values cannot be supported by all methods, and thus we fill blank cells with values sampled from distinct values of each column. We also vary domain sizes of each table.

- XueTang is a real-world OLTP benchmark for online education. We select five tables from it, including *auth_user*,

Table 4: Overall Accuracy Comparison on Real Datasets

Datasets	**Forest**					**Power**					**IMDB**					**XueTang**				
Methods	median	mean	90%	95%	max	median	mean	90%	95%	max	median	mean	90%	95%	max	median	mean	90%	95%	max
Quicksel	2.73	217	126	731	2e4	6.27	670	598	2e3	4e4	10.4	667	1e3	2e3	3e4	15.6	482	503	1434	3e4
Feedback-KDE	1.11	2.23	3.15	6.12	173	1.10	2.01	4.00	5.34	61	11.5	257	143	366	1e3	32.0	1283	1100	6725	4e4
Bayesian	1.13	2.37	5.60	7.00	1218	1.15	11.2	2.10	3.00	3e4	12.5	306	78.3	521	5e3	1.40	**1.81**	**3.0**	12	230
Naru (NeuroCard)	1.14	**2.24**	3.01	**4.79**	122	1.07	**1.30**	**1.75**	**2.00**	**15.0**	3.85	6.85	9.66	11.2	**477**	**1.26**	2.82	6.0	10.0	30.0
DeepDB	**1.06**	2.51	**2.56**	4.97	117	**1.03**	1.72	2.28	3.76	77.2	4.26	8.28	13.5	25.3	789	1.47	6.23	20.0	30.3	92.7
MSCN	1.91	5.17	12.7	20.0	**96.0**	1.80	5.30	11.2	22.2	84.0	**1.82**	**6.59**	**5.62**	**9.88**	536	1.33	2.33	5.0	**6.0**	**19.0**
LocalNN	1.94	4.64	9.15	13.9	136	1.77	3.88	6.75	11.0	105	9.38	18.3	19.7	22.1	965	4.0	55.0	11.1	17.8	3508
LocalXGB	2.70	7.42	10.9	20.4	511	1.93	3.27	5.29	8.16	73.4	8.12	20.2	18.3	25.3	1e3	3.48	82.3	8.92	17.1	3748

student_courseenrollment, *organization_acount_userorgprofile*, *auth_userprofile* and *bbs_comment*. All tables are joined by using *user_id* and *id*.

Synthetic Dataset. In order to better test which method is the best for different data distributions. We synthesize 256 different datasets from 4 perspectives, as shown in Table 3.

1. Domain size *dom* is the number of distinct values on each column. We set each column with the same domain size.
2. Correlation *corr* is the probability of each value in one column identical to another. For each pair of column combination, we consider the values in the same position of these two columns to have the same *corr* probability.
3. Skewness *skew* is a parameter of density distribution $f(x) = (1+x\cdot(skew-1))^{-1-\frac{1}{skew-1}}$. Given a linearly increasing x, the distribution is close to exponential if $skew = 1$; otherwise, the distribution is close to uniform.
4. For the convenience of correlation setting, we set the number of columns *col* as an even number so that we can group each two of them together.

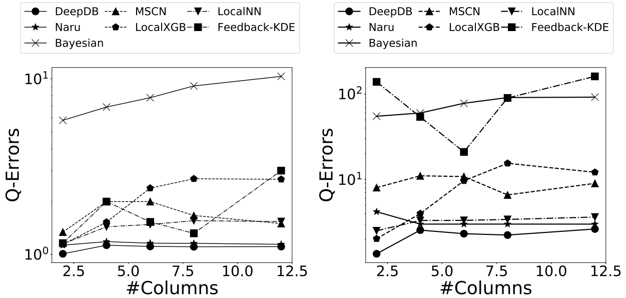
Workloads. Next we present the workloads for real and synthetic datasets. On single table datasets *Forest*, *Power* and a set of *Synthetic*, for generating a query, we randomly select several attributes from each table, and then sample a value from each attribute, next construct a conjunctive predicate. On multiple tables *IMDB*, we generate queries involving different tables for *IMDB* dataset. We randomly generate 2-column queries on $title$, 4-column queries on $title$ and $cast\_info$, 6-column and 8-column queries on $title$, $cast\_info$, $movie\_info$, *movie_info_idx* and *movie_keyword*. On *XueTang*, we collected user submitted queries over 2-5 tables as the overall test workload. We generated training queries randomly on these tables. The maximal #columns of queries is 10.

Metrics. We use Q-error [30] to measure the accuracy of cardinality estimators: $Q\text{-}error(\widehat{card}, card) = \frac{max(\widehat{card},card)}{min(\widehat{card},card)}$ where $\widehat{card}$ is the estimated cardinality and *card* is true cardinality. If the estimated cardinality is 0, we add it to 1 for avoiding exception.

Machine. We conduct our experiments on a machine with Intel E5-2630 2.20GHz CPU, GTX 2080ti GPU and 128GB memory.

4.2 Accuracy

Exp-1: Overall Comparisons on Real Datasets. Table 4 shows the Q-errors of all existing learned cardinality estimators on real datasets *Forest*, *Power*, *IMDB* and *XueTang*. From Table 4, we have the following observations. Unsupervised estimators, Naru and DeepDB, outperform other methods on single real tables *Forest* and *Power*. The reason is that Naru and DeepDB can better capture the data distribution and column correlations. Quicksel fails on both datasets, because the accuracy of model used by Quicksel heavily relies on what does the query look like. In other words, it has limited generalization ability. Bayesian produces small median error but large max error, as its conditional independent assumption may fail on some values. MSCN outperforms other data model based methods on *IMDB* [19] that requires to join several tables. The reason is that multi-table joins produce a large sized result, and it's hard for other methods to fit the data distribution by learning from join samples. Bayesian and MSCN outperform other methods on dataset *XueTang* [39]. Bayesian produces smaller mean and 90th percentile errors but larger max and 95th percentile errors because it models the joint distribution precisely. Note, however, that Bayesian is much more slower than other methods.



(a) Median error **(b) 95th percentile error**

Figure 7: [Synthetic] Cardinality Errors on Varying #Columns (#distinct value=1000, correlation=0.6, skew=0.6)

Exp-2: Varying the #Columns.

Single tables. Figure 7 shows experiment results on single tables with different number of columns. From Figure 7, we have the following observations. LocalNN, Naru and DeepDB outperform other methods on accuracy. The reason is that Feedback-KDE, LocalXGB and Bayesian are less powerful in modeling data and query, and MSCN encodes operator as a one-hot vector for each predicate instead of a value range. Bayesian, LocalNN and LocalXGB perform worse when the number of column increases, because more columns make the data distribution more complicated, and it's harder to fit such data distribution for these models. However, Naru, MSCN and DeepDB still perform well on dataset with 8 columns, that's because Autoregressive model in Naru and SPN in DeepDB can fit multi-attributes dataset well, and sample bitmap in MSCN improves the accuracy of query model significantly. DeepDB produces small errors when the number of column is 2, that's because it's easier to capture

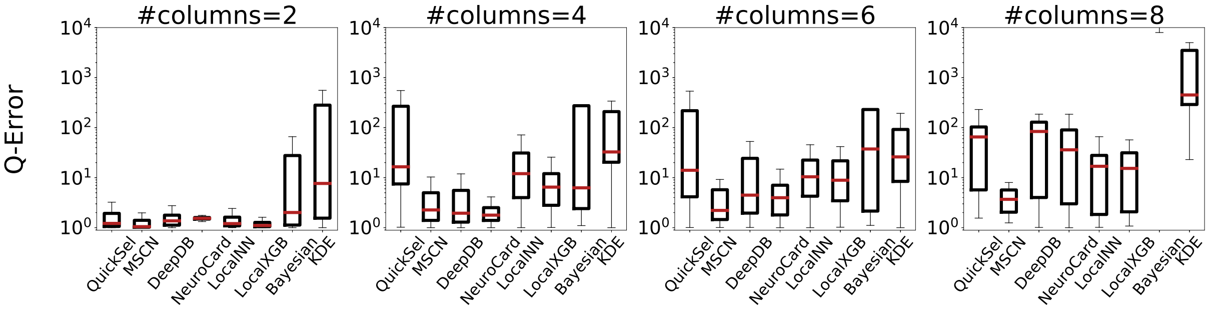


Figure 8: [IMDB] Cardinality Estimation Errors on Varying #Columns (median, quantile, max)

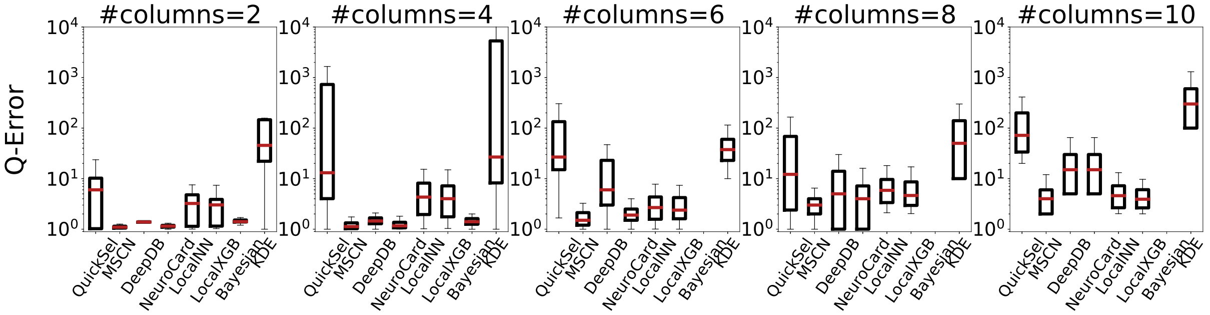


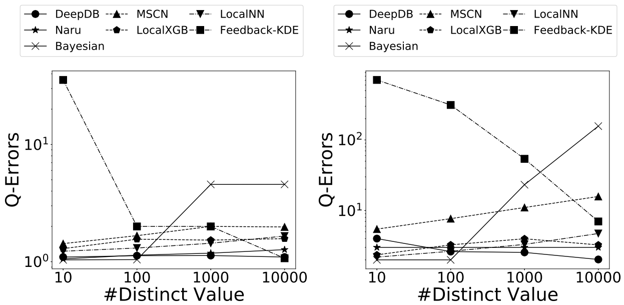
Figure 9: [XueTang] Cardinality Estimation Errors on Varying #Columns (median, quantile, max)

data distribution of only 2 columns. However, when the number of columns increases from 4 to 8, the error slightly decreases because SPN makes more partitions for more columns, and the partition number dominates the accuracy on single table. Feedback-KDE produces smaller 95th percentile error when the number of columns increases, that's because Feedback-KDE is prone to overfit training queries on dataset with less columns. Bayesian produces the largest error because data discretizing losses accuracy for distribution learning. LocalXGB performs similar to Naru on dataset with smaller column number, but makes much larger error when the number of column increases, that's because statistical model can fit 2-column distribution well but is powerless with more columns.

Multiple tables. Figure 8 shows the results on IMDB database and Figure 9 shows the results on XueTang database. These two datasets involve join queries and more challenging than single tables. From Figures 8 and 9, we have the following observations. Overall, more columns bring larger errors for all methods. On IMDB, Bayesian and Feedback-KDE perform the worst because Bayesian suffers from data discretization, and Feedback-KDE cannot fit complicated data distributions with only one tunable parameter. On XueTang, Bayesian cannot support queries with more than 6 columns because of Out of Memory Exceptions. On both IMDB and XueTang, Supervised methods MSCN, LocalXGB and LocalNN outperform unsupervised methods DeepDB and NeuroCard on queries with larger #columns (e.g., 8 or 10), because DeepDB and NeuroCard support join queries by learning from uniform join samples instead of full datasets because there are billions of rows for 3 tables outer join, and the sparsity of join samples reduces the accuracy. MSCN outperforms LocalXGB and LocalNN because it uses one model to fit all join queries, and it has better generalization for varying join patterns. Moreover, the models LocalXGB and LocalNN use are too simple to capture the complicated distributions.

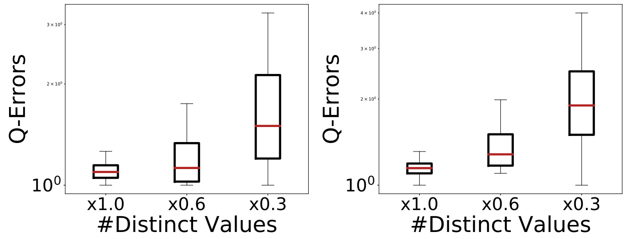
Exp-3: Varying #Distinct Values. Figure 10 shows the cardinality estimation errors on synthetic tables with varying domain sizes. From median and 95th percentile highest errors, we make the following observations. The accuracy of learned estimators based on query model decreases significantly with domain size increasing, that's because larger domain size makes the query space sparser, and the knowledge of test queries may not be covered by training set. Bayesian outperforms other methods on datasets with domain size 10 and 100 because Bayesian can fit data distribution on small domains precisely. However, Bayesian becomes unusable when domain size increases to 1,000 and 10,000 because value discretization losses too much accuracy for less space overhead. Overall, DeepDB performs the best among all learned estimators with larger domain size, and the 95th percentile error decreases with domain size increasing. The most likely reason is that it's would be much easier to find independent partitions for large domain size. 95th percentile error of Feedback-KDE decreases a lot with domain size increasing. The reason is that Feedback-KDE overfits training queries when domain size is small. With domain size increasing, the accuracy of supervised methods MSCN, LocalNN and LocalXGB also increases, because large domain size makes the query space sparser, and reduces distribution similarities between training queries and test queries. Figure 11 shows the Q-errors on IMDB, which join multiple tables. It shows that on join dataset, the estimation results of NeuroCard and MSCN are affected by domain size significantly. Both methods perform better on IMDB with smaller domain size.

Exp-4: Varying Correlations. Figure 12 shows the accuracy comparison of different learned estimators on dataset with varying correlations. From both median and 95th percentile highest errors, we have the following observations. The accuracy of most of the methods decreases when data correlation becomes larger. That's



(a) Median error (b) 95th percentile error

Figure 10: [Synthetic] Cardinality Errors on Varying #Distinct Values (correlation=0.6, column=4, skew=0.6)



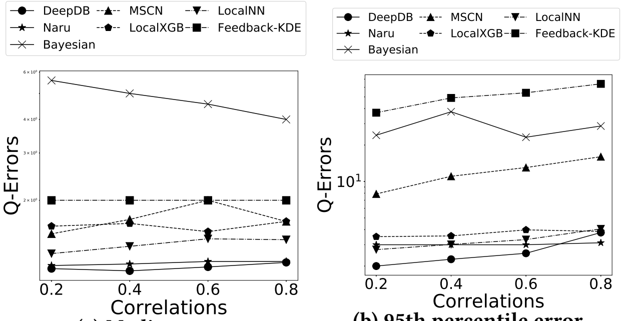
(a) MSCN (b) NeuroCard

Figure 11: [IMDB] Cardinality Estimation Errors on Varying #Distinct Values (median, quantile, max)

because larger correlation means the probabilities of values in different columns are correlated, and it's challenging to fit all conditional distributions in one model. Bayesian produces large errors because of data discretization, but it can search the optimal probabilistic graph and fit larger correlations. DeepDB fails on dataset with larger correlations because it makes independent assumptions between vertical column groups. Moreover, DeepDB produces very large errors on a small part of queries (95th percentile highest error). Naru also faces accuracy decay when correlation becomes larger, but Naru outperforms other estimators on dataset with high correlation because autoregressive model uses lossless distribution factorization to fit the dataset, and be able to learn the correlations.

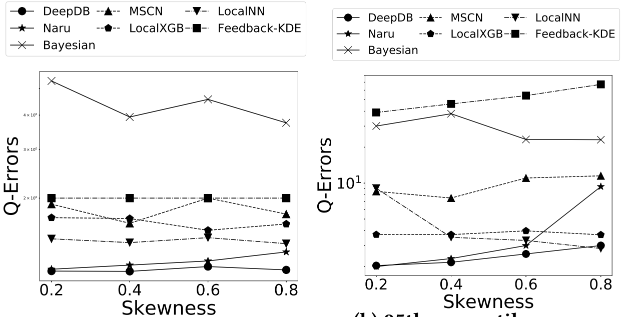
Supervised methods support dataset with different correlations, because they learn the cardinalities from training queries. The accuracy of supervised method also decreases because larger correlation makes the training queries insufficient for all the joint distributions.

Exp-5: Varying Skewness. Figure 13 shows the estimation errors of different methods on datasets with varying skewness. We have the following observations. Median and 95th percentile errors of Naru and DeepDB increase with skewness increasing. That's because (a) Naru conducts progressive sampling when estimating a query, and sampling involves 0-tuple problem (i.e., values with low frequency may be lost), and (b) DeepDB stores a frequency table in each leaf node when training, and the frequency table may loss low frequency values because of sampling. Supervised methods MSCN and LocalXGB produce similar median errors for all skewness, because most of the values in predicates can appear in training queries. Instead, LocalNN reduces its errors with skewness increasing, because LocalNN can better capture corner cases. Feedback-KDE produces the largest 95th percentile errors and errors increase with skewness increasing, because it relies on the data sample and cannot estimate all queries properly on large skewed dataset.



(a) Median error (b) 95th percentile error

Figure 12: [Synthetic] Cardinality Errors on Varying Correlations (#distinct value=1000, column=4, skew=0.6)



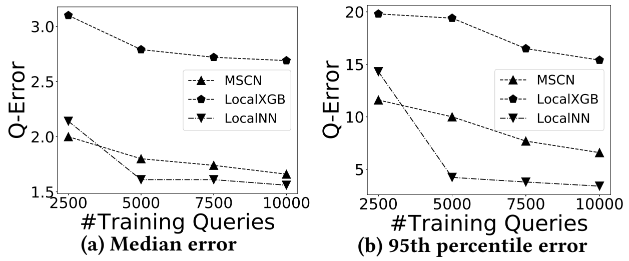
(a) Median error (b) 95th percentile error

Figure 13: [Synthetic] Cardinality Errors on Varying Skews (#distinct value=1000, column=8, correlation=0.6)

Exp-6: Varying the #-Training Queries. Figure 14 shows the accuracy comparison of supervised learning methods on synthetic dataset with different numbers of training queries. We have the following observations. Overall, we can observe that the estimation error significantly reduces with more training queries. For median errors, LocalXGB produces the largest errors with all training set sizes, that's because the model used in LocalXGB is less powerful than neural network on dataset with 8 columns. MSCN can produce high accuracy with only 2,500 training queries, but it improves little with more training queries. LocalNN produces large error with 2,500 training queries, but it improves the accuracy drastically when the number of training queries increases to 5,000, and outperforms LocalXGB and MSCN significantly. This is due to neural network structure and clear predicate range features for single table queries.

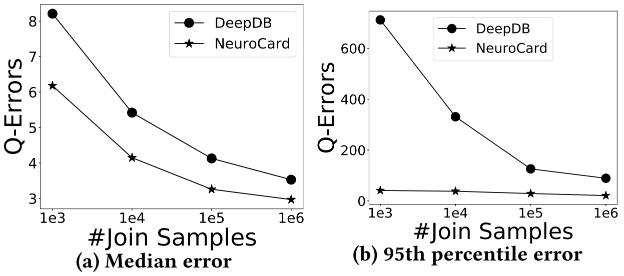
From 95th percentile highest errors, we can observe that increasing training queries can effectively improve accuracy of all supervised methods, that's because more training queries can reveal query semantics and data distributions from more perspectives.

Exp-7: Varying #Join Sample. Figure 15 shows that join sample size affects accuracy significantly. If the data model cannot train enough samples from joined tables, it produces large errors. The accuracy improves a lot when the size of join sample increases.



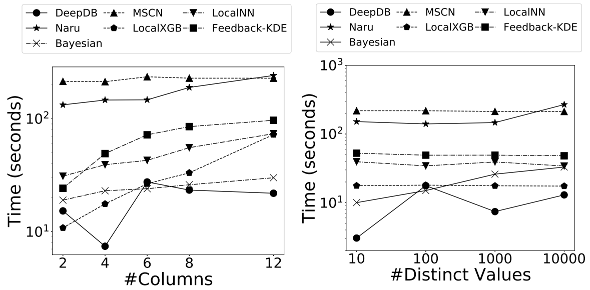
(a) Median error **(b) 95th percentile error**

Figure 14: [Synthetic] Errors of Varying #Training Queries (correlation=0.6, skew=0.6, #columns=8, #distinct=1000)



(a) Median error **(b) 95th percentile error**

Figure 15: [IMDB] Errors of Varying #Join Samples



(a) Varying #Columns (#distinct=1000) **(b) Varying #Distinct Values (#column=4)**

Figure 16: [Synthetic]Training (correlation=0.6, skew=0.6)

4.3 Efficiency

Exp-8: Training Time and Estimation Time.

Training Time. Figure 16a, 17, 19 show the training time on synthetic dataset, IMDB database and XueTang dataset with varying columns. From Figure 16a, we can observe that all the methods spend more training time with the number of columns increasing, in particular: (i) DeepDB requires more partitions and parameters; (ii) Each input of Naru contains more values, and more parameters should be optimized; (iii) Bayesian learns more variable; (iv) Each input query vector for LocalXGB and LocalNN is longer, and more parameters should be optimized; and (v) MSCN encodes and computes more predicates. From Figure 17, we can observe similar conclusions as Figure 16a. We also test Quicksel and Feedback-KDE, the parameter optimization of these two statistical methods requires more computation when the number of column increases. Figure 16b shows the training time of methods on synthetic datasets by varying the domain size. We can observe that Naru and Bayesian need more training time for larger domain size. That's because (i) the embedding space of Naru increases with domain size; and (ii) more probabilities are stored in the tree structure in Bayesian. Instead, domain size takes no effect on supervised methods, because they don't model the data distribution directly.

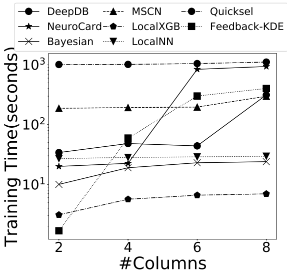


Figure 17: [IMDB] Training Time

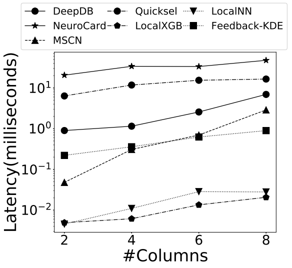


Figure 18: [IMDB] Estimate Latency

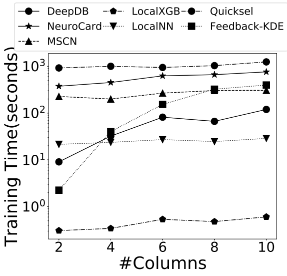


Figure 19: [XueTang] Training Time

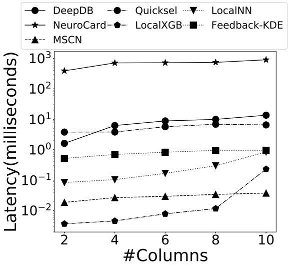
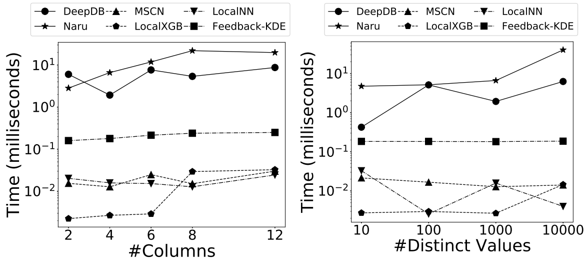


Figure 20: [XueTang] Estimate Latency



(a) Varying #Columns by fixing #distinct=1000 **(b) Varying #Distinct Values by fixing #columns=4**

Figure 21: [Synthetic] Latency (correlation=0.6, skew=0.6)

From Figure 19, we observe that (i) DeepDB spends more time for training when the #column increases, because a larger #columns requires more partitions and parameters; (ii) NeuroCard also spends more time for training because it needs more cost on progressive sampling; (iii) LocalXGB takes more cost when the #column becomes larger, because more predicates make each decision tree model larger; (iv) LocalNN and MSCN keep training time stable as the growing of computations brought by feature size increasing does not dominate the training time. (v) We also test Feedback-KDE, and the parameter optimization of two statistical methods requires more computation when the number of column increases.

Estimation Time. Figure 18, Figure 20 and Figure 21a show estimation latency on datasets with varying columns. Because Bayesian

	High Correlation							Mid Correlation							Low Correlation						
#Table	1				2	3		1				2	3		1				2	3	
#Column	2	4	6	8	4	6	8	2	4	6	8	4	6	8	2	4	6	8	4	6	8
Methods #Domain	H/M/L	H/M/L	H/M/L	H/M/L	H/M/L	H/M/L	H/M/L	H/M/L	H/M/L	H/M/L	H/M/L	H/M/L	H/M/L	H/M/L	H/M/L	H/M/L	H/M/L	H/M/L	H/M/L	H/M/L	H/M/L
Naru	++/+/+	+/++/+	++/++/++	+/+/+	NA	NA	NA	+/+/+	+/+/+	+/+/++	+/+/+	NA	NA	NA	+/+/+	+/+/+	+/+/+	+/+/+	NA	NA	NA
NeuroCard	++/+/+	+/++/+	++/++/++	+/+/+	+/++/++	+/+/++	-/-/+	+/+/+	+/+/+	+/+/++	+/+/+	+/++/++	+/+/++	-/-/+	+/+/+	+/+/+	+/+/+	+/+/+	+/++/++	+/+/++	-/-/+
DeepDB	+/+/+	+/+/+	+/+/+	++/++/++	+/+/+	+/+/+	-/-/-	++/+/+	+/+/+	++/++/+	++/++/++	+/+/+	+/+/+	-/-/-	++/+/+	+/+/+	++/++/++	++/++/++	+/+/+	+/+/+	-/-/-
Bayesian Network	-/++/++	-/+/++	-/-/-	- -/-/-	- -/- -/- -	- -/- -/- -	- -/- -/- -	-/++/++	-/++/++	-/-/-	- -/-/-	- -/- -/- -	- -/- -/- -	- -/- -/- -	-/++/++	-/++/++	-/-/-	- -/-/-	- -/- -/- -	- -/- -/- -	- -/- -/- -
MSCN	-/-/-	- -/-/-	-/-/-	-/-/-	++/+/+	++/++/+	++/++/++	-/-/-	- -/-/-	-/-/-	-/-/-	++/+/+	++/++/+	++/++/++	-/-/-	- -/-/-	-/-/-	-/-/-	++/+/+	++/++/+	++/++/++
LocalNN	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+
LocalXGB	+/+/+	+/+/+	+/+/+	-/-/+	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+	-/-/+	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+	+/+/+	-/-/+	+/+/+	+/+/+	+/+/+
QuickSel	+/+/+	-/-/-	-/-/-	- -/-/-	-/-/-	-/-/-	-/-/-	+/+/+	-/-/-	-/-/-	- -/-/-	-/-/-	-/-/-	-/-/-	+/+/+	-/-/-	-/-/-	- -/-/-	-/-/-	-/-/-	-/-/-
Feedback-KDE	+/+/-	++/-/-	+/-/-	+/+/-	- -/- -/-	-/-/-	-/-/-	+/+/-	++/-/-	+/-/-	+/+/-	- -/- -/-	-/-/-	-/-/-	+/+/-	++/-/-	+/-/-	+/+/-	- -/- -/-	-/-/-	-/-/-

Figure 22: Experiment Summary

Table 5: Update (sec)

	Forest			IMDB		
Percent	**5%**	**10%**	**20%**	**5%**	**10%**	**20%**
#-rows	**3e4**	**6e4**	**12e4**	**4e6**	**8e6**	**16e6**
DeepDB-inc	20	71	262	191	513	1571
DeepDB	311	337	391	289	313	362
Naru-inc	158	162	180	206	210	217
Naru	323	338	351	795	864	932
MSCN-inc	162	327	608	736	1681	3214
MSCN	335	646	1247	1081	1712	3378
LocalNN-inc	171	408	826	674	1525	3191
LocalNN	329	728	1332	998	1671	3247
LocalXGB-inc	166	392	818	765	1644	3360
LocalXGB	297	595	1301	1047	1809	3417

method takes too much time (even 10,000 ms) for estimating each query, we do not include it in the figures. From Figure 21a, we can observe that the estimation times of Naru and LocalXGB are strictly increasing with column number increasing. That's because Naru should compute more conditional probabilities for a query, and LocalXGB goes deeper in each regression tree. From Figure 18, estimation time increases significantly of almost all methods when the number of column increases, that's because the number of tables involved in queries also increases. With more tables, MSCN computes more tables and sample bitmaps, DeepDB computes more partitions, NeuroCard computes more conditional probabilities. We also test Quicksel and Feedback-KDE on IMDB database, they take more time for higher dimensional dataset. Figure 21b shows estimation time of different methods on synthetic datasets with varying domain size. Estimation latency of methods Naru, DeepDB, and LocalXGB strictly increases with domain size for the following reasons: (1) progressive sampling in Naru valid more distinct values for each query; (2) DeepDB lookups more probabilities on each column (3) LocalXGB searches a deep regression tree. From Figure 20, estimation time increases of almost all methods when the number of column increases, that's because the number of tables involved in queries also increases. With more tables, DeepDB computes more partitions, NeuroCard computes more conditional probabilities on all columns. We also test Quicksel and Feedback-KDE, they take more time for higher dimensional dataset.

Exp-9: Incremental Data Updates. We test updating time of different cardinality estimators with 5%, 10% and 20% insertions. For these estimators, we evaluate incremental training methods (e.g., DeepDB-inc, Naru-inc, MSCN-inc, LocalNN-inc and LocalXGB-inc), and retraining methods. DeepDB-inc updates the Sum-Product Network for each inserted row. NeuroCard-inc and Naru-inc resample the datasets and train model based on old models for a few epochs. MSCN-inc, LocalNN and LocalXGB updates query labels on inserted data and trains model based on old models.

Tables 5 shows that (i) with small updates (5%), incremental methods outperform retraining methods by 30%–1500%, (ii) with larger updates (e.g., 8e6 rows or 16e6 rows), DeepDB-inc performs slowly because the incremental DeepDB code we use [2] updates for each row of inserted data sample, and (iii) In general, query-based methods spend more time for updating than data-based methods, that's because query labels updating needs much time.

4.4 Summary

According to the experimental analysis above, we summarize the accuracy comparison results of various methods on different settings from four perspectives: column number, domain size, table number, and correlation. From Figure 22, we mainly have the following observations: (1) advanced unsupervised methods Naru and DeepDB outperform others on single tables, but MSCN outperforms others on three table joins. (2) DeepDB is affected by column correlation significantly, and it is outperformed by Naru on single tables with 2,4,6-column and high correlation. (3) Bayesian and Quicksel support small datasets well, but they fail when columns and domain size increase. (4) Feedback-KDE is the only method of which the accuracy increases with domain size increasing, and it outperforms other estimators on 8-column single table with high domain size. (5) LocalNN and LocalXGB outperform MSCN on single tables, but MSCN outperforms other methods for join queries.

5 CONCLUSION

We have systematically studied the design space for learned cardinality estimation methods, and a comparative evaluation of these methods using both real-world and synthetic datasets. Our summarized experimental findings, could serve as a guidance for both researchers and practitioners to design and implement learned estimators for their applications. We also provided a cardinality estimation testbed, and the researchers who want to design new learned estimators could utilize our testbed to significantly reduce the overhead of design and implementation.

Acknowledgement. This paper was supported by NSF of China (61925205, 61632016), Huawei, TAL education, and Beijing National Research Center for Information Science and Technology (BNRist).

REFERENCES

[1] [n.d.]. https://github.com/andreaskipf/learnedcardinalities.

[2] [n.d.]. https://github.com/DataManagementLab/deepdb-public.

[3] [n.d.]. https://github.com/illinoisdata/quicksel.

[4] [n.d.]. https://github.com/martinkiefer/feedback-kde.

[5] [n.d.]. https://github.com/neurocard/neurocard.

[6] [n.d.]. https://github.com/pgmpy/pgmpy.

[7] Leo Breiman. 1994. Bagging Predictors.

[8] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees*. Wadsworth.

[9] AM Carvalho. [n.d.]. Scoring functions for learning Bayesian networks. http://www.lx.it.pt/~asmc/pub/talks/09-TA/ta_pres.pdf

[10] C. K. Chow and C. N. Liu. 1968. Approximating discrete probability distributions with dependence trees. *IEEE Trans. Inf. Theory* 14, 3 (1968), 462–467. https://doi.org/10.1109/TIT.1968.1054142

[11] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

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

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

[14] Guy Lohman. 2014. Is Query Optimization a "Solved" Problem? https://dsf.berkeley.edu/cs286/papers/queryopt-sigmodblog2014.pdf.

[15] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1035–1050. https://doi.org/10.1145/3318464.3389741

[16] 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, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1477–1492. https://doi.org/10.1145/2723372.2749438

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

[18] J J Hopfield. 1982. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences* 79, 8 (1982), 2554–2558. https://doi.org/10.1073/pnas.79.8.2554 arXiv:https://www.pnas.org/content/79/8/2554.full.pdf

[19] IMDb.com. 2019. https://www.imdb.com/. https://www.imdb.com/

[20] Yannis E. Ioannidis. 2003. The History of Histograms (abridged). In *PVLDB*. 19–30. http://www.vldb.org/conf/2003/papers/S02P01.pdf

[21] M. Kearns and L. G. Valiant. 1989. Crytographic Limitations on Learning Boolean Formulae and Finite Automata. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing (STOC '89)*. Association for Computing Machinery, New York, NY, USA, 433–444. https://doi.org/10.1145/73007.73049

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

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

[24] Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press.

[25] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep Learning. *Nature* 521, 7553 (2015), 436–444. https://doi.org/10.1038/nature14539

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

[27] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *CIDR*. http://cidrdb.org/cidr2017/papers/p9-leis-cidr17.pdf

[28] Bruce G. Lindsay and Michael Stewart. 2010. *Mixture Models*. Palgrave Macmillan UK, London, 129–138. https://doi.org/10.1057/9780230280816_17

[29] Richard J. Lipton, Jeffrey F. Naughton, and Donovan A. Schneider. 1990. Practical Selectivity Estimation Through Adaptive Sampling. *SIGMOD Rec.* 19, 2 (May 1990), 1–11. https://doi.org/10.1145/93605.93611

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

[31] Frank Olken and Doron Rotem. 1990. Random sampling from database files: A survey. In *Statistical and Scientific Database Management.*

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

[33] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. 2020. QuickSel: Quick Selectivity Learning with Mixture Models. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1017–1033. https://doi.org/10.1145/3318464.3389727

[34] Robert E. Schapire. 1990. The Strength of Weak Learnability. *Mach. Learn.* 5 (1990), 197–227. https://doi.org/10.1007/BF00116037

[35] Kostas Tzoumas, Amol Deshpande, and Christian S. Jensen. 2011. Lightweight Graphical Models for Selectivity Estimation Without Independence Assumptions. *Proc. VLDB Endow.* 4, 11 (2011), 852–863. http://www.vldb.org/pvldb/vol4/p852-tzoumas.pdf

[36] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2020. Are We Ready For Learned Cardinality Estimation? *CoRR* abs/2012.06743 (2020). arXiv:2012.06743 https://arxiv.org/abs/2012.06743

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

[38] Wentao Wu, Jeffrey F. Naughton, and Harneet Singh. 2016. Sampling-Based Query Re-Optimization. In *SIGMOD*. 1721–1736. https://doi.org/10.1145/2882903.2882914

[39] Tsinghua XueTang. 2019. https://www.xuetangx.com/global. https://www.xuetangx.com/global

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

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

[42] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1525–1539. https://doi.org/10.1145/3183713.3183739