Deep Autoregressive Models for Join Cardinality Estimation

Amog Kamsetty

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-123
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-123.html

May 29, 2020

Acknowledgement

I would like to thank the entire NeuroCard team: Zongheng Yang for providing me the opportunity to collaborate with you and for your research guidance and mentorship, Prof. Ion Stoica for advising me and providing the vision for this project, Prof. Joe Hellerstein for sparking my interest in databases through your excellent teaching, and Eric Liang and Frank Luan for being amazing teammates. And of course, thank you to my family for always being there for me.

Deep Autoregressive Models for Join Cardinality Estimation

by Amog Kamsetty

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Ion Stoica

Professor Ion Stoica
Research Advisor

05/27/2020

(Date)

* * * * * * *

Professor Joseph M. Hellerstein
Second Reader

5/28/2020

(Date)

Abstract

Deep Autoregressive Models for Join Cardinality Estimation[1]

by

Amog Kamsetty

Master of Science in Electrical Engineering Computer Sciences

University of California, Berkeley

Professor Ion Stoica, Chair

Database query optimizers rely on accurate cardinality estimates to produce optimal execution plans. Despite decades of research, existing cardinality estimators are inaccurate for complex queries, due to making lossy modeling assumptions and not capturing inter-table correlations. In this work, we show that it is possible to learn the correlations across all tables in a database without any independence assumptions. We present NeuroCard, a join cardinality estimator that builds a single neural density estimator over an entire database. Leveraging join sampling and modern deep autoregressive models, NeuroCard makes no inter-table or inter-column independence assumptions in its probabilistic modeling and is able to efficiently capture the rich multivariate distributions of relational data. In addition, NeuroCard is able to summarize the data with out any supervision and therefore does not require execution of any training queries and is robust to workload shifts. NeuroCard achieves orders of magnitude higher accuracy than the best prior methods (a new state-of-the-art result of 8.5× maximum error on JOB-light) and scales to dozens of tables, while being compact in space (several MBs) and efficient to construct or update (seconds to minutes).

[1] This report is adapted from the following conference papers: **Deep Unsupervised Cardinality Estimation** by Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, & Ion Stoica published in VLDB 2020 and **NeuroCard: One Cardinality Estimator for All Tables** by Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, & Ion Stoica, in submission to VLDB 2021. Please cite these conference papers instead of this report.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Database query optimizers are used to translate declarative queries into optimal execution plans, with optimality usually determined by execution time. These optimizers are critical not only for relational databases, but even for modern analytical engines such as Spark [1] and Presto [39]. While query optimizers are comprised of multiple components, *cardinality estimation* often plays a larger role than the cost model to join order enumeration [24]. The goal of cardinality estimation is to predict the selectivity of a query result, or the number of rows in the output, without actually executing the query itself. These estimates are used as input to a cost model, and these costs are used to score various query plans. Therefore, accurate cardinality estimates lead to more accurate costs and ultimately better execution plans. However, despite its importance, cardinality estimation is an extremely difficult problem, especially for complex queries containing multiple joins. There is wide agreement that this problem is still unsolved [25]. In fact, open source and commercial database systems often produce up to $10^4 - 10^8\times$ errors on complex queries containing many joins and tables with many attributes.

The fundamental difficulty of cardinality estimation comes from condensing information about data into summaries [14]. The predominant approach in database systems today is to collect single-column summaries (e.g., histograms and sketches) for each relation, and to combine these coarse-grained models assuming column independence. This represents one end of the spectrum, where the summaries are fast to construct and cheap to store, but compounding errors occur due to the coarse information and over-simplifying independence assumptions. On the other end of the spectrum, when given the *joint data distribution* of a relation (the frequency of each unique tuple normalized by the relation's cardinality), perfect selectivity "estimates" can be read off or computed via integration over the distribution. However, the exact *joint* is intractable to compute or store for all but the tiniest datasets. Thus, traditional selectivity estimators face the hard tradeoff between the amount of information captured and the cost to construct, store, and query the summary.

An accurate and compact *joint approximation* would allow better design points in this tradeoff space. Recent advances in deep unsupervised learning have offered promising tools in this regard. While it was previously thought intractable to approximate the data distribution

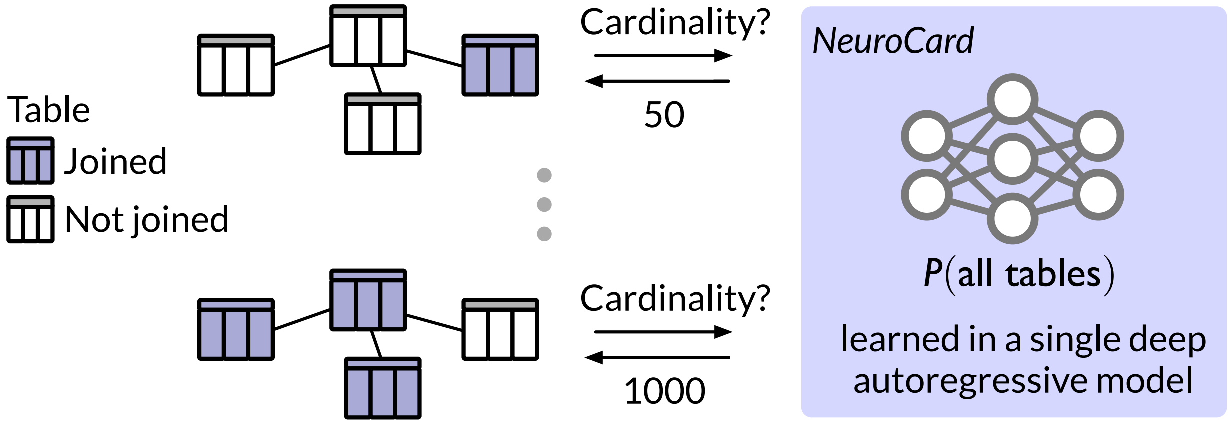


Figure 1.1: NeuroCard uses a single probabilistic model, which learns all possible correlations among all tables in a database, to estimate join queries on any subset of tables.

of a relation in its full form [7, 2], *deep autoregressive models*, a type of density estimator, have succeeded in modeling high-dimensional data such as images, text, and audio [44, 36, 45, 46, 3]. Deep autoregressive models allow us to reject the tradeoff that traditional cardinality estimators face, as these types of models provide an extremely accurate approximation of the full joint data distribution in an efficient manner.

In this work, we propose NeuroCard, a novel cardinality estimator that leverages these new autoregressive density estimators. NeuroCard's distinctive feature is the ability to capture the correlations across multiple joins in a single deep autoregressive model, without any independence assumptions. Once trained, this model can handle all queries issuable to the schema, regardless of what subset of tables is involved, as shown in Figure 1.1. Having a single estimator has two key benefits: simplicity and accuracy. Having multiple estimators—each covering a specific join template (a table subset)—does not scale for a large number of tables, as the number of possible join templates increases exponentially. In addition, it is easier for a DBMS to operationalize a single estimator rather than many estimators. Most importantly, having multiple estimators can hurt accuracy. This is because estimating the cardinality of a query on a table subset not covered by any single estimator, but by multiple estimators, requires some form of independence assumption to combine these estimators. If the independence assumption does not hold, the accuracy will suffer.

Figure 1.2 shows the high-level architecture of NeuroCard. The primary component of NeuroCard is the autoregressive core, a single autoregressive model that learns the joint distribution of the full join (§3). Because we are using a autoregressive model, we make no independence assumptions across tables and are therefore able to accurately learn the distribution of the join.

In order to train the autoregressive model, we prepare an unbiased join sampler by building or loading existing single-table indexes on join keys and computing join count tables

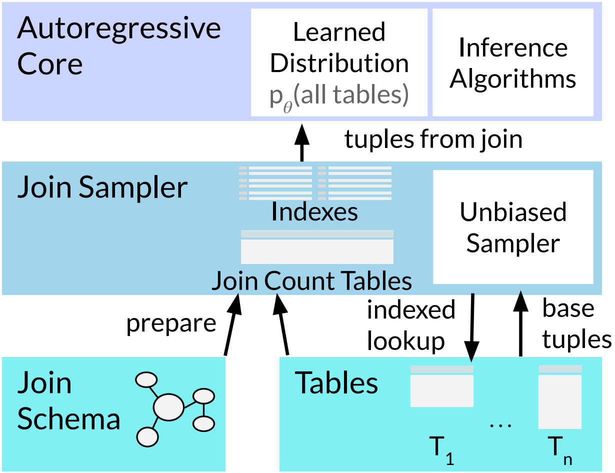


Figure 1.2: Overview of NeuroCard. The Join Sampler (§4) provides correct training data (sampled tuples from join) by using unbiased join counts. Sampled tuples are streamed to an autoregressive model for maximum likelihood training (§3). Inference algorithms (§5) use the learned distribution to estimate query cardinalities.

for the specified join schema (§4). We train the deep AR model by repeatedly requesting batches of sampled tuples from the sampler, usually 2K at a time. The sampler fulfills this request in the background, potentially using multiple sampling threads.

Once the estimator is built, it is ready to compute the cardinality estimates for given queries. For each query, we use probabilistic inference algorithms (§5) to compute the cardinality estimate by (1) performing Monte Carlo integration on the learned AR model, and (2) handling schema subsetting. With these inference algorithms, a single estimator can handle queries joining any subset of tables, with arbitrary range selections.

By combining all of these ingredients, NeuroCard achieves state-of-the-art estimation accuracy, including in the challenging tail quantiles (§6).

Chapter 2

Related Work

At a high level, there are two approaches to cardinality estimation: *query-driven* and *data-driven*. While query-driven estimators rely on query feedback to produce better cardinality estimates, data-driven estimators attempt to approximate the data distribution, usually disregarding queries. From a machine learning perspective, query-driven estimators are considered to be a supervised learning approach. These estimators rely on collecting (encoded query, true cardinality) pairs which are used as training data to learn a mapping from the query representation to a predicted cardinality. Learning based data-driven estimators are considered unsupervised or self-supervised because they directly learn from the actual data itself, and do not need to collect training data through actual query execution. We discuss various lines of research from these two approaches below. We also look at other applications of machine learning to database systems.

Unsupervised data-driven cardinality estimators.

This family of estimators approximates the data distribution and dates back to System R's use of 1D histograms [37]. These histograms are calculated on a per-attribute basis for each table. Any output cardinality can be predicted by combining various estimates from these histograms. However, the main drawback with using histograms for cardinality estimation are the assumptions that are made. In particular, combining estimates from histograms relies on independence assumption across attributes and tables which often do not hold. In addition, the granularity of histogram bins plays an important role as uniformity is assumed within each bin. Due to these assumptions, simply using 1D histograms leads to poor cardinality estimates, and alternate approaches with higher quality density models have been proposed throughout the years:

Classical Methods

Multidimensional histograms [33, 10, 30, 34] are more precise than 1D histograms by capturing inter-column correlations. Probabilistic relational models (PRMs) [7] rely on a Bayes

Net (conditional independence DAG) to factor the joint into materialized conditional probability tables. Tzoumas *et al.* [43] propose a variant of PRMs optimized for practical use. Dependency-based histograms [2] make partial or conditional independence assumptions to keep the approximated joint tractable (factors stored as histograms). Despite their improvements over 1D Histograms, these approaches either still make certain independence assumptions or are not feasible for practice.

Sum-Product Networks

SPNs, a tree-structured density estimator, were proposed about 10 years ago [32]. Each leaf is a coarse histogram of a slice of an attribute, and each intermediate layer uses either × and + to combine children information. Due to their heuristics (e.g., inter-slice independence), SPNs have *limited expressiveness*: there exists simple distributions that cannot be efficiently captured by SPNs of any depth [29]. DeepDB [13] is a recent cardinality estimator that uses SPNs. NeuroCard is similar to DeepDB in the following aspects. *(S1)* Both works use the formulation of learning the full outer join of several tables. *(S2)* Our "schema subsetting" capability builds on the querying algorithms proposed by DeepDB, while generalizing them by removing their PK-FK assumptions (§5.3).

NeuroCard differs from DeepDB in the following. *(D1) Modern density model:* NeuroCard's choice of a deep autoregressive model is a universal function approximator hence fundamentally more expressive. Unlike SPNs, no independence assumption is made in the modeling. *(D2) Correlations learned:* NeuroCard argues for capturing as much correlation as possible across tables, and proposes learning the full outer join of all tables of a schema. DeepDB, due to limited expressiveness, learns multiple SPNs, each on a heuristically chosen table subset ($\sim$ 1–3 tables). Conditional independence is assumed across table subsets. *(D3) Sampling from true data distribution of joins:* NeuroCard identifies the key requirement of sampling from the data distribution in an unbiased fashion. In contrast, DeepDB obtains join tuples either from full computation or using the Index Based Join Sampling algorithm [23] which samples from a biased distribution. Due to these differences, NeuroCard outperforms DeepDB by up to 70× in accuracy and is much faster to construct (§6). We expect that our components (e.g., unbiased sampling (§4), column factorization (§3.5) can improve DeepDB in accuracy or space; similarly, their probabilistic inference algorithms can be added to NeuroCard to handle approximate query processing.

Deep autoregressive models

A breakthrough in density estimation, deep AR models are the current state-of-the-art density models from the ML community [35, 6, 46, 3]. They tractably learn complex, high-dimensional distributions in a neural net, capturing all possible correlations among attributes. Distinctively, AR models provide access to all conditional distributions among input attributes. Because of their expressiveness and efficiency, NeuroCard leverages these models for cardinality estimation.

Supervised query-driven cardinality estimators

Leveraging past or collected queries to improve estimates dates back to LEO [40]. Interest in this approach has seen a resurgence partly due to an abundance of query logs [47] or better function approximators (neural networks) [18, 41] that map featurized queries to predicted cardinalities. Hybrid methods that leverage query feedback to improve density modeling have also been explored, e.g., KDE [11, 15] and mixture of uniforms [52]. Supervised estimators can easily leverage query feedback, handle complex predicates (e.g., UDFs), and are usually more lightweight [4]. NeuroCard has demonstrated superior estimation accuracy to representatives in this family §6, while being fundamentally more robust since it is not affected by out-of-distribution queries. Complex predicates can also be handled by executing on tuples sampled from NeuroCard's learned distribution.

Join sampling

Extensive research has studied join sampling, a fundamental problem in databases. NeuroCard leverages a state-of-the-art join sampler to obtain training tuples representative of a join. NeuroCard adopts the linear-time Exact Weight algorithm from Zhao *et al.* [53], which is among the top-performing samplers they study. This algorithm provides uniform and independent samples, just as NeuroCard requires. While IBJS [23] and Wander Join [26] provide unbiased estimators for counts and aggregates, they do not provide uniform samples of a join and thus are unsuitable for collecting training data. Lastly, we show that it is advantageous to layer a modern density model on join samples.

Learned Database Components

ML for Query Optimizers

There are other approaches revolving around using machine learning for full query optimization. For example, in Neo [28], a learned query optimizer, embeddings for all attributes are pre-trained, and later, a network takes them as input and additionally learns to correct or ignore signals from the input. Neo along with other approaches such as DQ [22] and ReJOIN [27] use reinforcement learning to produce the final optimal plan, and do not deal with cardinality estimation directly. As a result, these approaches may benefit from NeuroCard's improved cardinality estimates.

ML for other Database Components

A great deal of work has recently applied either classical ML or modern deep learning to other various database components, e.g., indexing [21], data layout [51]. Being able to model inter-table and inter-column correlations without any independence assumptions, NeuroCard's use may go beyond query optimization to other tasks that require an *understanding of tables and attributes* (e.g., data imputation [48] or indexing [49]).

Chapter 3

Autoregressive Models for Cardinality Estimation

In this section, we describe the autoregressive core of NeuroCard. We give an overview of autoregressive models, how they can be adapted to learn the distribution of relational data, and how to train these models.

Problem Overview

We estimate the selectivities of queries of the following form. Consider a set of tables, $T_1, \ldots, T_N$. We define their *join schema* as the graph of join relationships, where vertices are tables, and each edge connects two joinable tables. We consider a query to be executed over any arbitrary subgraph of the overall schema. We assume the schema and queries submitted to the estimator are acyclic (§4.2 discusses relaxations), so they can be viewed as trees. A query on a subgraph is a conjunction of single-column boolean predicates, over arbitrary subsets of columns in the join result of the subgraph. A predicate contains an attribute, an operator, and a literal, and is read as $A_i \in R_i$ (attribute i takes on values in valid region R_i). Our formulation includes the usual $=, \neq, <, \leq, >, \geq$ predicates, the rectangular containment $A_i \in [l_i, r_i]$, or even IN clauses. For ease of exposition, we use *range* to denote the valid region R_i or, for the whole query, the composite valid region $R_1 \times \cdots \times R_n$. We assume the *domain* of each column, A_i, is finite: since a real dataset is finite, we can take the empirically present values of a column as its finite domain.

We make a few remarks. First, disjunctions of such predicates are supported via the inclusion-exclusion principle. Second, our formulation follows a large amount of existing work on this topic [2, 7, 11, 33, 52] and, in some cases, offers more capabilities. Certain prior work requires each predicate be a rectangle [18, 11] or columns be real-valued [20, 11]; our "region" formulation supports complex predicates and does not make these assumptions. Lastly, the relation under estimation can either be a base table or a join result.

3.1 Probabilistic Modeling of Tables

Consider a table T with column domains $\{A_1, \ldots, A_n\}$. This table induces a discrete *joint data distribution*, defined as the probability of occurrence of each tuple ($f(\cdot)$ denotes number of occurrences):

$$P(a_1, \ldots, a_n) = f(a_1, \ldots, a_n)/|T|.$$

The n-dimensional data distribution (the *joint*) $P(\cdot)$ allows us to compute a query's cardinality as follows. Define a query Q as $\sigma : A_1 \times \cdots \times A_n \to \{0, 1\}$. Then, the *selectivity*—the fraction of records that satisfy the query—can be computed as a probability: $P(Q) = \sum_{a_1 \in A_1} \cdots \sum_{a_n \in A_n} \sigma(a_1, \ldots, a_n) \cdot P(a_1, \ldots, a_n)$. The *cardinality* is obtained by multiplying it with the row count: $|Q| = P(Q) \cdot |T|$.

Data-driven cardinality estimators can be grouped along two axes: (1) joint factorization, and (2) the density estimator used.

Joint factorization, or the modeling assumption, determines how precisely data distribution p is factored. Any modeling assumption risks losing information about correlations across columns, which ultimately leads to a loss in accuracy. For example, the widely used 1D histogram technique assumes the columns are independent. As a result, it factors p into a set of 1D marginals, $\widehat{P} \approx \prod_{i=1}^{n} \widehat{P}(A_i)$, which can lead to large inaccuracies when the columns' values are strongly correlated. Similarly, other data-driven cardinality estimators such as graphical models [7, 8, 2, 42, 43] either assume conditional independence or partial independence among columns. For instance, Probabilistic Relational Models [7, 8] from the early 2000s leverage the conditional independence assumptions of Bayesian Networks (e.g., joint factored into smaller distributions, $\{\widehat{P}(A_1|A_2, A_3), \widehat{P}(A_2), \widehat{P}(A_3)\}$. Dependency-Based Histograms [2] use decomposable interaction models and rely on partial independence between columns (e.g., $\widehat{P}(A_1, A_2, A_3) \approx \widehat{P}(A_1)\widehat{P}(A_2, A_3)$). Both methods are marked improvements over 1D histograms since they capture more than single-column interactions. However, the tradeoff between richer factorizations and costs to store or integrate is still unresolved. Obtaining selectivities becomes drastically harder due to the integration now crossing multiple attribute domains. Most importantly, the approximated joint's precision is compromised since some forms of independence are still assumed.

One exception is the autoregressive (product-rule) factorization,

$$\widehat{P} = \prod_{i=1}^{n} \widehat{P}(A_i|A_{<i}), \tag{3.1}$$

which precisely expresses the overall joint distribution as the product of the n conditional distributions.

The density estimator determines how precisely the aforementioned factors are actually approximated. The most accurate "estimator" would be recording these factors exactly in a hash table. Unfortunately, this leads to enormous construction and inference costs (e.g., in the case of $\widehat{P}(A_n|A_{1:n-1})$). Recently, *deep autoregressive (AR) models* [3, 35, 36] have emerged as the density estimator of choice. Deep AR models compute $\{\widehat{P}(A_i|A_{<i})\}$ without explicitly materializing them by learning the n conditional distributions in compact neural networks. Deep AR models achieve state-of-the-art precision, and, for the first time, provide a tractable solution for implementing the autoregressive factorization.

3.2 Deep Autoregressive Models

We now show how we use deep autoregressive models to approximate the joint data distribution.

Overview

NeuroCard uses a deep autoregressive model to approximate the joint distribution. We overview the statistical features they offer and how those relate to selectivity estimation.

Access to point density $\widehat{P}(\mathbf{x})$.

Deep autoregressive models produce point density estimates $\widehat{P}(\mathbf{x})$ after training on a set of n-dimensional tuples $T = \{x_1, \dots\}$ with the unsupervised maximum likelihood objective. Many network architectures have been proposed in recent years, such as masked multi-layer perceptrons (e.g., MADE [6], ResMADE [3]) or masked self-attention networks (e.g., Transformer [46]).

Access to conditional densities $\{\widehat{P}(x_i|\mathbf{x_{<i}})\}$.

Additionally, autoregressive models also provide access to all conditional densities present in the product rule:

$$\begin{aligned}\widehat{P}(x) &= \widehat{P}(x_1, x_2, \cdots, x_n)\\ &= \widehat{P}(x_1)\widehat{P}(x_2|x_1)\cdots\widehat{P}(x_n|x_1,\dots,x_{n-1})\end{aligned}$$

Namely, given input tuple $x = (x_1, \cdots, x_n)$, one can obtain from the model the n conditional density estimates, $\{\widehat{P}(x_i|x_{<i})\}$. The model can be architected to use any ordering(s) of the attributes (e.g., (x_1, x_2, x_3) or (x_2, x_1, x_3)). In our exposition we assume the left-to-right schema order.

NeuroCard chooses autoregressive models for selectivity estimation for two important reasons. First, autoregressive models have shown superior modeling precision in learning images [44, 36], audio [45], and text [46]. All these domains involve correlated, high-dimensional data akin to a relational table. Second, as we will show in §5.1, access to conditional densities is critical in efficiently supporting range queries.

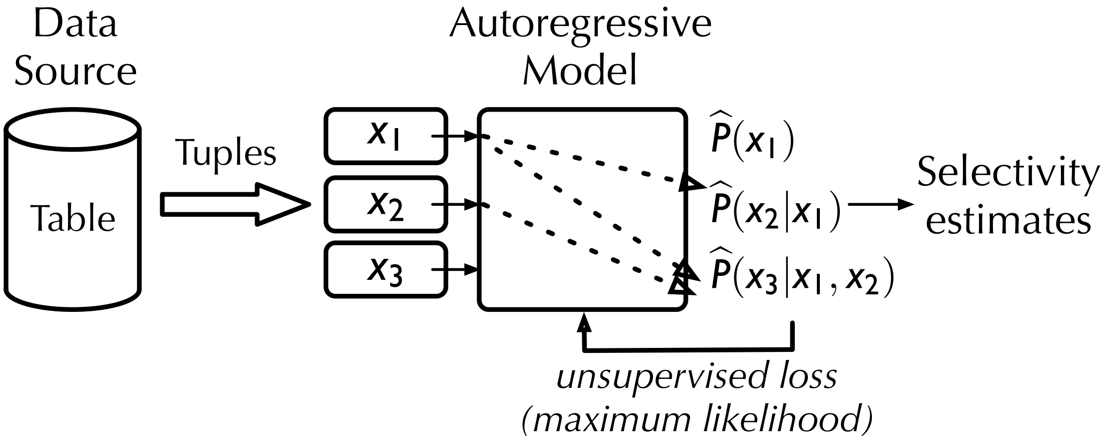


Figure 3.1: Overview of the estimator framework. NeuroCard is trained by reading data tuples and does not require supervised training queries or query feedback, just like classical synopses.

Autoregressive Models for Relational Data

NeuroCard allows any autoregressive model $\mathcal{M}$ to be plugged in. In general, such model has the following functional form:

$$\mathcal{M}(x) \mapsto \Big[\widehat{P}(X_1), \widehat{P}(X_2|x_1), \cdots, \widehat{P}(X_n|x_1, \ldots, x_{n-1})\Big] \tag{3.2}$$

Namely, one tuple goes in, a list of conditional density distributions comes out, each being a *distribution* of the ith attribute conditioned on previous attributes. (The *scalars* required to compute the point density, $\{\widehat{P}(x_i|x_{<i})\}$, are read from these conditional distributions.) How can a neural net $\mathcal{M}$ attain the autoregressive property, e.g., that $\widehat{P}(X_3|x_1, x_2)$ only depends on, or "sees", the information from the first two attribute values (x_1, x_2) but not anything else?

Information masking is a common technique used to implement autoregressive models [6, 46, 44]; here we illustrate the idea by constructing an example architecture for relational data. Suppose we assign each column i its own compact neural net, whose input is the aggregated information about *previous* column values $x_{<i}$. Its role is to use this context information to output a distribution over its own domain, $\widehat{P}(X_i|x_{<i})$. Consider a travel_checkins table with columns city, year, stars. Assume the model is given the input tuple, ⟨Portland, 2017, 10⟩. First, column-specific encoders $E_{\texttt{col}}()$ transform each attribute value into a numeric vector suitable for neural net consumption, $[E_{\texttt{city}}(\texttt{Portland}), E_{\texttt{year}}(\texttt{2017}), E_{\texttt{stars}}(\texttt{10})]$.

Then, appropriately aggregated inputs are fed to the per-column neural nets $\mathcal{M}_{\texttt{col}}$:

$$\begin{gathered} 0 \to \mathcal{M}_{\texttt{city}} \\ E_{\texttt{city}}(\texttt{Portland}) \to \mathcal{M}_{\texttt{year}} \\ \oplus\left(E_{\texttt{city}}(\texttt{Portland}), E_{\texttt{year}}(\texttt{2017})\right) \to \mathcal{M}_{\texttt{stars}} \end{gathered}$$

where $\oplus$ is the operator that aggregates information from several encoded attributes. In practice, this aggregator can be vector concatenation, a set-invariant *pooling* operator (e.g., elementwise sum or max), or even self-attention [46].

Notice that the first output, from $\mathcal{M}_{\texttt{city}}$, does not depend on any attribute values (its input 0 is arbitrarily chosen). The second output depends only on the attribute value from `city`, and the third depends only on both `city` and `year`. Therefore, the three outputs can be interpreted as

$$\left[\widehat{P}(\texttt{city}), \widehat{P}(\texttt{year}|\texttt{city}), \widehat{P}(\texttt{stars}|\texttt{city}, \texttt{year})\right]$$

Thus, autoregressiveness is achieved via such input masking.

Training these model outputs to be as close as possible to the true conditional densities is done via maximum likelihood estimation. Specifically, the *cross entropy* [5] between the data distribution P and the model estimate $\widehat{P}$ is calculated over all tuples in relation T and used as the loss:

$$\mathcal{H}(P, \widehat{P}) = -\sum_{x \in T} P(x) \log \widehat{P}(x) = -\frac{1}{|T|} \sum_{x \in T} \log \widehat{P}(x) \tag{3.3}$$

It can be fed into a standard gradient descent optimizer [16]. Lastly, the Kullback-Leibler divergence, $\mathcal{H}(P, \widehat{P}) - \mathcal{H}(P)$, is the *entropy gap* (in bits-per-tuple) incurred by the model. A lower gap indicates a higher-quality density estimator; thus, it serves as a monitoring metric during and after training.

3.3 Join Problem Formulation

A join schema induces the full outer join of all tables in the schema, $T = T_1 ⟗ \cdots ⟗ T_N$. Our goal is to build a single autoregressive probabilistic model on the full join consisting of all tables' columns:

$$\text{Model:} \quad \widehat{P}(T) \equiv \widehat{P}(T_1.\text{col}_1, T_1.\text{col}_2, \ldots, T_N.\text{col}_k) \tag{3.4}$$

In this setting, we define T as the full outer join of all tables within a schema. Consequently, the deep AR model learns the correlations across all tables. Next, we need to sample tuples with probabilities prescribed by the true distribution P. Otherwise, $\widehat{P}$ would approximate an incorrect, biased distribution. To achieve this, we use a sampler that emits *simple random samples* from the full join T (§4).

3.4 Workflow & Architecture

Figure 3.1 outlines the workflow of building the autoregressive core for NeuroCard. Batches of random tuples from T (the full outer join) are read to train NeuroCard. For each tuple, NeuroCard encodes each attribute value using column-specific strategies (§3.4). The encoded batch then gets fed into the model to perform a gradient update step to maximize the predicted likelihood of the data:

$$\text{Sample i.i.d.} \quad x \sim p \tag{3.5}$$

$$\text{Take gradient steps to maximize} \quad \log p_\theta(x) \tag{3.6}$$

The various encoding strategies that NeuroCard employs are described below.

Encoding and Decoding Strategies

NeuroCard models a relation as a high-dimensional discrete distribution. The key challenge is to *encode* each column into a form suitable for neural network consumption, while preserving the column semantics. Further, each column's output distribution $\widehat{P}(X_i|x_{<i})$ (a vector of scores) must be efficiently *decoded* regardless of its datatype or domain size.

For each column NeuroCard first obtains its domain A_i either from user annotation or by scanning. All values in the column are then dictionary-encoded into integer IDs in range $[0, |A_i|)$. For instance, the dictionary can be `Portland` $\mapsto 0$, `SF` $\mapsto 1$, etc. For a column with a natural order, e.g., numerics or strings, the domain is sorted so that the dictionary order follows the column order. Overall, this pre-processing step is a lossless transformation (i.e., a bijection).

Next, column-specific encoders $E_{\texttt{col}}()$ encode these IDs into vectors. The ML community has proposed many such strategies before; we make sensible choices by keeping in mind a few characteristics specific to relational datasets:

Encoding small-domain columns: one-hot.

For such a column $E_{\texttt{col}}()$ is set to *one-hot encoding* (i.e., indicator variables). For instance, if there are a total of 4 cities, then the encoding of `SF` is $E_{\texttt{city}}(1) = [0, 1, 0, 0]$, a 4-dimensional vector. The small-domain threshold is configurable and set to 64 by default. This encoding takes $O(|A_i|)$ space per value.

Encoding large-domain columns: embedding.

For a larger domain, the one-hot vector wastes space and computation budget. NeuroCard uses embedding encoding in this case. In this scheme—a preprocessing step in virtually all natural language processing tasks—a learnable embedding matrix of type $\mathbb{R}^{|A_i| \times h}$ is randomly initialized, and $E_{\texttt{col}}()$ is simply row lookup into this matrix. For instance, $E_{\texttt{year}}(4) \mapsto$ row 4 of embedding matrix, an h-dimensional vector. The embedding matrix gets updated during gradient descent as part of the model weights. Per value this takes $O(h)$ space (NeuroCard defaults h to 64). This encoding is ideal for domains with a meaningful semantic distance (e.g., cities are similar in geo-location, popularity, relation to its nation) since each dimension in the embedding vector can learn to represent each such similarity.

Decoding small-domain columns.

Suppose domain A_i is small. In this easy case, the network allocates an *output layer* to compute a *distribution* $\widehat{P}(X_i|x_{<i})$, which is a $|A_i|$-dimensional vector of probabilities used for selectivity estimation. We use a fully connected layer, $\mathsf{FC}(F, |A_i|)$, where F is the hidden unit size. For example, for a city column with three values in its domain, the output distribution may be $[\mathsf{SF} = 0.2; \mathsf{Portland} = 0.5; \mathsf{Waikiki} = 0.3]$. During optimization, the training loss seeks to minimize the divergence of this output from the data distribution.

Decoding large-domain columns: embedding reuse.

If the domain is large, however, using a fully connected output layer $\mathsf{FC}(F, |A_i|)$ would be inefficient in both space and compute. Indeed, an `id` column in a dataset we tested on has a large domain size of $|A_i| = 10^4$, inflating the output layer beyond typical scales.

NeuroCard solves this problem by an optimization that we call “embedding reuse”. In essence, we replace the potentially large output layer $\mathsf{FC}(F, |A_i|)$ with a much smaller version, $\mathsf{FC}(F, h)$ (recall that h is the typically small embedding dimensions; defaults to 64). This immediately yields a saving ratio of $|A_i|/h$. The goal of decoding is to take in inputs $x_{<i}$ and output $|A_i|$ probability scores over the domain. With the shrunk-down output layer, inputs $x_{<i}$ would pass through the net arriving at an h-dimensional feature vector, $H \subseteq \mathbb{R}^{1\times h}$. We then calculate HE_i^T, where $E_i \subseteq \mathbb{R}^{|A_i|\times h}$ is the *already-allocated* embedding matrix for column i, obtaining a vector $\mathbb{R}^{1\times|A_i|}$ that can be interpreted as the desired scores after normalization. We have thus decoded the output while cutting down the cost of compute and storage. This scheme has proved effective in other large-domain tasks [35].

Model Architecture

For our experiments, we utilize a standard AR architecture, ResMADE [3]. As has been studied [50], a simple masked multi-layer perceptron such as ResMADE strikes a good balance between efficiency and accuracy. The design of NeuroCard can accommodate any deep AR architecture if more advanced architectures are desired.

3.5 Optimization: Reducing Model Size with Lossless Column Factorization

A key challenge of using an autoregressive model for high-cardinality data is that the size of the model parameters can scale linearly with the numbers of distinct values in the columns. In

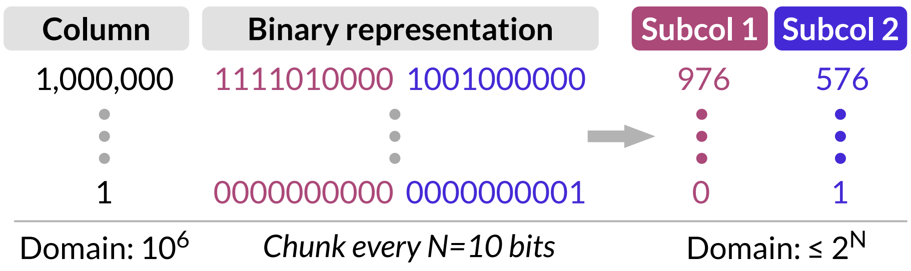


Figure 3.2: Lossless column factorization (§3.5).

the model architecture we use (§3.4), each column (any data type; categorical or numerical) is first dictionary-encoded into integer token IDs. Then a per-column *embedding* layer is applied on these token IDs. The size of the trainable embedding matrix (essentially, a hash table) for each column C scales linearly with $|A_C|$, i.e., the number of distinct values in the domain. Even a moderately sized column with up to 10^6 distinct values, therefore, easily takes up 128 MB of space, assuming 32-dimensional embeddings are used.

To handle high-cardinality columns efficiently, we propose an optimization that we call *lossless column factorization*. This optimization is inspired by the popular use of "subword units" [38] in modern natural language processing, and also shares characteristics with "bit slicing" in the indexing literature [31]. Different from subword units, column factorization does not use a statistical algorithm such as byte pair encoding to determine what subwords to use (a potential optimization). Different from bit slicing, we slice a value into groups of bits and convert them back into base-10 integers.

Figure 3.2 illustrates the idea on a simple example. Suppose a column (any datatype) has a domain size of $|A_C| = 10^6$. Naively supporting this column would require allocating $|A_C| \cdot h$ floats as its embedding matrix, where h is the embedding dimension. Instead, NeuroCard factorizes each value *on-the-fly* during training: we convert an original-space value into its binary representation, then slice off every N bits, the *factorization bits* hyperparameter. Each sliced off portion becomes a *subcolumn*, now in base-10 integer representation. These subcolumns are now treated as regular columns to learn over by the autoregressive model. Crucially, a much smaller embedding matrix is now needed for each subcolumn containing at most $2^N \cdot h$ floats. In this example, we can reduce 128 MB to 250 KB—a more than 500× space reduction.

Lossless = factorization + autoregressive modeling.

With factorization, a column is factorized into multiple subcolumns, which are then fed into a downstream density estimator. However, if a density estimator with independence assumptions, e.g., 1D histograms, is used, then this whole process is *lossy*. By modeling $p(\mathsf{subcol}_1, \mathsf{subcol}_2) \approx p(\mathsf{subcol}_1)p(\mathsf{subcol}_2)$, histograms would fail to capture any potential correlation between the two subcolumns. In other words, other estimators *could* read in subcolumn values and potentially reduce space usage, but their inherent quality and assumptions determine how much information is learned about the subcolumns, and about

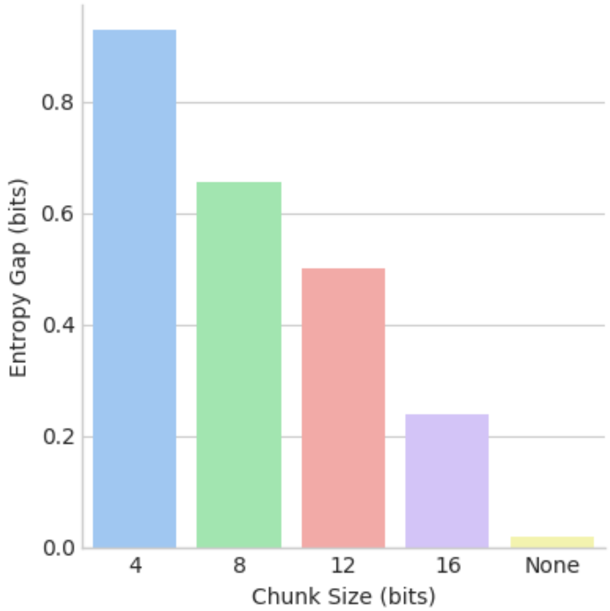


Figure 3.3: Entropy Gap with varying chunk sizes.

their correlations with other columns. By using autoregressive modeling, NeuroCard forces the AR model to explicitly capture such correlation, namely (ignoring other columns):

$$p(\mathsf{col}) \equiv p(\mathsf{subcol}_1, \mathsf{subcol}_2) = p(\mathsf{subcol}_1)p(\mathsf{subcol}_2|\mathsf{subcol}_1),$$

which has no inherent loss of information. Hence, we call the unique combination of factorization and autoregressive modeling *lossless*.

Model size vs. statistical efficiency.

As discussed in the previous section, in theory, by using autoregressive modeling no information is lost in this translation so the precision of the learned distributions is not affected. However, in practice, this isn't the case, as shown by Figure 3.3. In this experiment, we trained various autoregressive models on a single column table containing 4.5M rows and 135k distinct values. Each model has the same configuration, except for a different chunk size bits N. To evaluate how the chunk size affects the model's learning, we look at the KL divergence, or entropy gap, between the model's learned distribution and the true distribution of the data. As we can see, as we increase N, the entropy gap decreases, and the model does a better job in estimating the true distribution. This is because since as we decrease N in each chunk, we have more subcolumns, resulting in a more difficult learning task for the model due to long-range dependencies. Therefore, choosing the factorization bits N enables a tradeoff between model size vs. statistical efficiency.

Lower factorization bits, i.e., slicing into more subcolumns, generally underperform higher ones that use more space. Because of this, we choose the factorization bits N to be as high as possible as long as the model is still under the space usage budget.

Chapter 4

Sampling From Joins

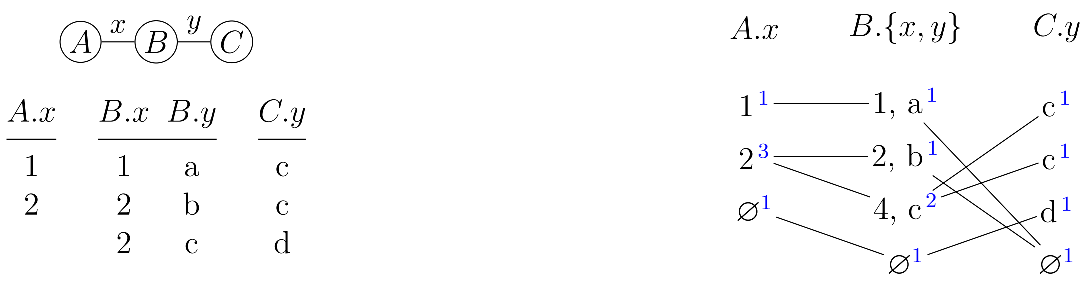
As discussed in the previous section, a key challenge in NeuroCard is computing an *unbiased* sample of the full join (§3.3) to ensure that the learned distribution faithfully approximates the full join distribution. Namely, every tuple in the full join J (a multiset) must be sampled equally likely with probability $1/|J|$. The samples should also be i.i.d., as required by Equation 3.5. To access tuples from the full join, a straightforward treatment is to compute it, then take uniform samples. Unfortunately, even on a small 6-table schema (the JOB-light workload), the full join contains *two trillion* ($2 \cdot 10^{12}$) tuples, making it infeasible to compute in practice. Instead, NeuroCard meets these requirements by using a sampler that produces *simple random samples with replacement* without having to materialize the join result.

4.1 Algorithm

A tuple in the full join contains *join key columns* and *content columns*. Our sampler exploits this decomposition. The first step of the sampler is to precompute *join count tables*, which are per-table statistics that reflect the occurrence counts of the join keys in the full join. The sampler then samples the join keys, table-by-table, with occurrence probabilities proportional to their join counts. Lastly, it selects content columns from the base tables by looking up the drawn join keys. This completes a batch of samples, which is sent to the model for training, and the procedure repeats on demand.

Computing join counts. Zhao et al. [53] provide an efficient algorithmic framework of join sampling that produces simple random samples from general multi-key joins. NeuroCard implements the Exact Weight algorithm from Zhao et al., adapted to full outer joins.

We illustrate the algorithm on a join schema (a tree) consisting of tables $T_1, \ldots, T_N$. For exposition, assume they only involve join keys (content columns are gathered later). Without loss of generality, let T_1 be the root table. The key idea is to sample a tuple through the join tree, using the correct *join counts* as weights at each table. The join count of a tuple $t \in T_i$ is the total number of tuples in the full outer join of all of T_i's descendants that can



$A.x$	$B.x$	$B.y$	$C.y$
1	1	a	c
2	2	b	c
	2	c	d

(a) Schema and base tables

(b) Join counts

$A.x$	$B.x$	$\mathcal{F}_{B.x}$	$B.y$	$C.y$	$\mathcal{F}_{C.y}$	$\mathbb{1}_A$	$\mathbb{1}_B$	$\mathbb{1}_C$
1	1	1	a	$\varnothing$	1	1	1	0
2	2	2	b	$\varnothing$	1	1	1	0
2	2	2	c	c	2	1	1	1
2	2	2	c	c	2	1	1	1
$\varnothing$	$\varnothing$	1	$\varnothing$	d	1	0	0	1

(c) Full outer join, with virtual columns in blue

```
-- In full join, |A.x=2|=3.
-- Q1. True answer is 2.
SELECT COUNT(*)
FROM A JOIN B ON x
       JOIN C ON y
WHERE A.x = 2;
-- Q2. True answer is 1.
SELECT COUNT(*)
FROM A WHERE A.x = 2;
```

(d) Schema subsetting

Figure 4.1: End-to-end example. (a) A join schema of three tables and their join key columns. Content columns are omitted. (b) Join counts (blue) enable uniform sampling of the full outer join and are computed in linear time by dynamic programming. Here, edges connect join partners. (c) Learning target: the full outer join of the schema, with *virtual columns* in blue. We show the *fanouts* $\mathcal{F}$, the number of times a join key value appears in the corresponding base table, for keys $B.x$ and $C.y$. The fanouts for $A.x$ and $B.y$ are all 1 and omitted. Each *indicator* $\mathbb{1}_T$ denotes whether a tuple has a match in table T. *(*d) Examples of schema subsetting, i.e., queries that touch a subset of the full join (§5.3).

join with t. It is recursively defined as:

$$w_i(t) = \prod_{T_j \in \text{Children}(T_i)} \sum_{t' \in t \rtimes T_j} w_j(t') \quad \forall i, \forall t \in T_i \tag{4.1}$$

where $t \rtimes T_j$ denotes all tuples in T_j that join with t. For a leaf table with no descendants, $w_i(\cdot)$ is defined as 1. At the root table T_1, $w_1(t)$ represents the count of $t \in T_1$ in the entire full outer join. The join counts can be computed recursively bottom-up using dynamic programming. The time complexity is therefore linear in the number of tuples in all tables, $O(|T_1| + \cdots + |T_N|)$, in contrast with the exponential worst-case complexity of computing the full join.

Sampling. Once the join counts are computed, the sampler produces a sample by traversing the join tree in a top-down fashion. It starts by drawing a sample t_1 from the root table T_1 using weights $\{w_1(t) : t \in T_1\}$ (i.e., with probabilities $\{w_1(t)/\sum_{t' \in T_1} w_1(t')\}$). It then samples through all descendants of T_1 in the breadth-first order. At a child table, say T_2, it samples t_2 from $t_1 \rtimes T_2$ (all tuples in T_2 that join with t_1) using weights $\{w_2(t) : t \in t_1 \rtimes T_2\}$. The procedure continues recursively until all tables are visited, and thus produces a sample $(t_1, \cdots, t_N)$, each t_i being a tuple of join keys from the respective table.

Example. Consider the schema in Figure 4.1a. Figure 4.1b shows the computed join counts. The leaf table C has a count of 1 for every tuple. In B, since $(2, c)$ can join with two tuples in C, its join count is $2 = 1 + 1$. Similar propagation happens for $A.x = 2$ which gets a count of $3 = 1 + 2$. Physically, we store the join counts indexed by join keys (e.g., for C, only one mapping $c \to 1$ is kept). For sampling, suppose $A.x = 2$ is first sampled. It has two matches in B with weights 1 and 2, so the second match, $(2, c)$, has an inclusion probability of 2/3.

`NULL` handling. To support full outer joins, we handle `NULL` keys as follows. We add a virtual $\varnothing$ tuple (which denotes `NULL`) to each table T_i, and make it join with all normal $t \in T_j$ that have no matches in T_i, where $T_j \in \text{Children}(T_i)$. Similarly, any normal $t \in \text{Parent}(T_i)$ that has no match in T_i joins with T_i's $\varnothing$. All-`NULL` is invalid. Propagation proceeds as before; Figure 4.1b shows examples.

Constructing complete sample tuples. In the prior example, suppose $\langle 2; 2, c; c \rangle$ is drawn. We gather the content columns of A by looking up $A.x = 2$ and similarly for $(B.x, B.y) = (2, c)$[1] and $C.y = c$. On multiple matches, we pick a row uniformly at random. Their concatenation represents a sampled tuple from the full join.

Computing the size of the full join (normalizing constant). Recall from §3.1 that the row count $|J|$ (the *normalizing constant* in probabilistic terms) is required to convert selectivities into cardinalities. With join counts it can be computed exactly: $|J| = \sum_{t \in T_1} w_1(t)$.

Parallel sampling. Finally, the sampling procedure is embarrassingly parallel: after the join count tables $\{w_i(\cdot)\}$ are produced, parallel threads can be launched to read the join counts and produce samples. Computation of the join count tables is also parallelizable,

[1] Either intersect two matching lists from both columns' index lookups, or do a single lookup if a composite index is available.

although it is an one-time effort. Sampling correctness is preserved even in the presence of parallelism due to the i.i.d. property.

4.2 Comparison with other samplers

Our key requirements of uniform and i.i.d. samples from the full join render many related sampling algorithms unsuitable. If either property is not satisfied, the sampling distribution would be biased and thus compromise the quality of the learned AR model. As examples, Index-based Join Sampling (IBJS) [23] is neither uniform nor independent; Wander Join [26] produces independent but non-uniform samples. Both approaches do produce unbiased estimators for counts or other aggregate statistics, but are not designed to return uniform join samples. Reservoir sampling, a well-known technique, draws samples without replacement (thus, non-independent) and requires a full scan over the full join, which is not scalable. Lastly, the Exact Weight algorithm NeuroCard implements is among the most efficient in Zhao *et al.* [53]. They provide additional extensions to support general, potentially cyclic joins (e.g., a cycle can be *broken*), which NeuroCard can leverage to broaden our formulation.

Chapter 5

Querying NeuroCard

Once built, the autoregressive model summarizes the entire full outer join

$$\text{Model:} \quad \widehat{P}(T) \equiv \widehat{P}(T_1.\text{col}_1, T_1.\text{col}_2, \ldots, T_N.\text{col}_k) \tag{5.1}$$

Now, we describe how to query the learned autoregressive model to produce cardinality estimates. In the general case, assume we have learned the following distribution $\widehat{P}(X_1, X_2, ...X_n)$. A query is then issued asking for the selectivity of the conjunction, $\mathsf{sel}(\theta) = P(X_1 \in R_1, \ldots, X_n \in R_n)$, where each range R_i can be a point (equality predicate), an interval (range predicate), or any subset of the domain (IN). The calculation of this density is fundamentally summing up the probability masses distributed in the cross-product region, $R = R_1 \times \cdots \times R_n$. We first discuss the straightforward support for equality predicates, then move on to how NeuroCard solves the more challenging problem of range predicates.

Equality Predicates

When values are specified for *all* columns, estimating conjunctions of these equality predicates is straightforward. Such a point query has the form $P(X_1 = x_1, \ldots, X_n = x_n)$ and requires only a single forward pass on the point, $(x_1, \ldots, x_n)$, to obtain the sequence of conditionals, $[\widehat{P}(X_1 = x_1), \widehat{P}(X_2 = x_2|X_1 = x_1), \ldots, \widehat{P}(X_n = x_n|X_1 = x_1, \ldots, X_{n-1} = x_{n-1})]$, which are then multiplied.

Range Predicates

It is impractical to assume a workload that only issues point queries. With the presence of any range predicate, or when some columns are not filtered, the number of points that must be evaluated through the model becomes larger than 1. (In fact, it easily grows to an astronomically large number for the majority of workloads we considered.) We discuss two ways in which NeuroCard carries out this operation. *Enumeration* exactly sums up the densities when the queried region R is sufficiently small: all discrete points in this region

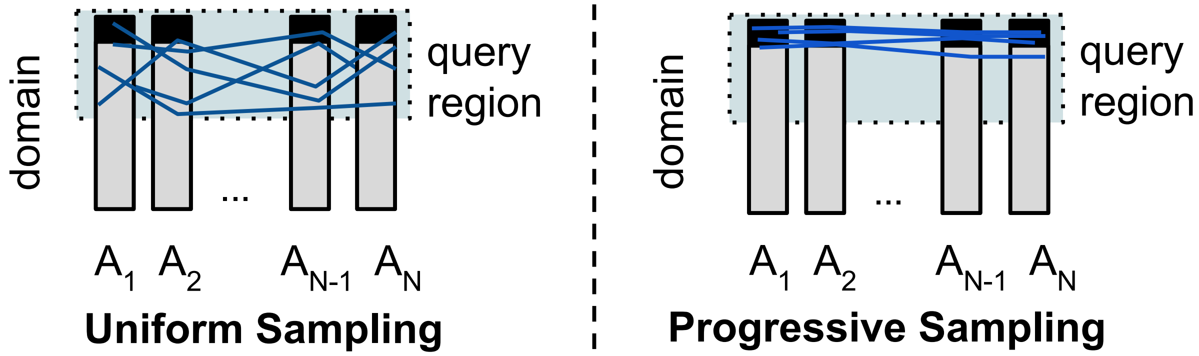


Figure 5.1: The intuition of progressive sampling. Uniform samples taken from the query region have a low probability of hitting the high-mass sub-region of the query region, increasing the variance of Monte Carlo estimates. Progressive sampling avoids this by sampling from the estimated data distribution instead, which naturally concentrates samples in the high-mass sub-region.

are enumerated and fed into the model, in a batching fashion, whose corresponding point densities are then summed up:

$$\mathsf{sel}(X_1 \in R_1, \ldots, X_n \in R_n) \approx \sum_{x_1 \in R_1} \cdots \sum_{x_n \in R_n} \widehat{P}(x_1, \ldots, x_n).$$

When the region R is deemed too big—almost always the case in the datasets and workloads we considered—we instead use a novel approximate technique termed *progressive sampling* (described next), an unbiased estimator that works surprisingly well on the relational datasets we considered.

Lastly, queries with out-of-domain literals can be handled via simple rewrite. For example, suppose year's domain is $\{2017, 2019\}$. A range query with an out-of-domain literal, say "year < 2018", can be rewritten as "year ≤ 2017" with equivalent semantics. For equality predicates with out-of-domain literals, NeuroCard simply returns a cardinality of 0. Hereafter we consider in-domain literals and valid regions.

5.1 Range Queries via Progressive Sampling

The queried region $R = R_1 \times \cdots \times R_n$ in the worst case contains $O(\prod_i D_i)$ points, where $D_i = |A_i|$ is the size of each attribute domain. Clearly, computing the likelihood for an exponential number of points is prohibitively expensive for data/queries with even moderate dimensions. NeuroCard proposes an approximate integration scheme to address this challenge.

First attempt (Figure 5.1, left). The simplest way to approximate the sum is via uniform sampling. First, sample $x^{(i)}$ uniformly at random from R. Then, query the model to compute $\widehat{p}_i = \widehat{P}(x^{(i)})$. By naive Monte Carlo, for S samples we have $\frac{|R|}{S}\sum_{i=1}^{S}\widehat{p}_i$ as an unbiased estimator to the desired density. Intuitively, this scheme is randomly throwing points into target region R to probe its average density.

To understand the failure mode of uniform sampling, consider a relation T with n correlated columns, with each column distribution skewed so that 99 of the probability mass is contained in the top 1 of its domain (Figure 5.1). Take a query with range predicates selecting the top 50 of each domain. It is easy to see that uniformly sampling from the query region will take in expectation $1/(0.01/0.5)^n = 1/0.02^n$ samples to hit the high-mass region we are integrating over. Thus, the number of samples needed for an accurate estimate increases exponentially in n. Consequently, we find that this sampler collapses catastrophically in the real-world datasets that we consider. It has the worst errors among all baselines in our evaluation.

Progressive sampling (Figure 5.1, right). Instead of uniformly throwing points into the region, we could be more selective in the points we choose—precisely leveraging the power of the trained autoregressive model. Intuitively, a sample of the first dimension $x_1^{(i)}$ would allow us to *"zoom in" into the more meaningful region of the second dimension.* This more meaningful region is exactly described by the second conditional output from the autoregressive model, $\widehat{P}(X_2|x_1^{(i)})$, a distribution over the second domain given the first dimension sample. We can obtain a sample of the second dimension, $x_2^{(i)}$, from this space instead of from Unif(R_2). This sampling process continues for all columns. To summarize, progressive sampling consults the autoregressive model to steer the sampler into the high-mass part of the query region, and finally compensating for the induced bias with importance weighting.

Example. We show the sampling procedure for a 3-filter query. Drawing the i-th sample for query $P(X_1 \in R_1, X_2 \in R_2, X_3 \in R_3)$:

1. Forward 0 to get $\widehat{P}(X_1)$. Compute and store scalar $\widehat{P}(X_1 \in R_1)$ by summing over R_1. Then draw $x_1^{(i)} \sim \widehat{P}(X_1|X_1 \in R_1)$.
2. Forward $x_1^{(i)}$ to get $\widehat{P}(X_2|x_1^{(i)})$. Compute and store scalar $\widehat{P}(X_2 \in R_2|x_1^{(i)})$ by summing over R_2. Draw $x_2^{(i)} \sim \widehat{P}(X_2|X_2 \in R_2, x_1^{(i)})$.
3. Forward $(x_1^{(i)}, x_2^{(i)})$ to get $\widehat{P}(X_3|x_1^{(i)}, x_2^{(i)})$. Compute and store scalar $\widehat{P}(X_3 \in R_3|x_1^{(i)}, x_2^{(i)})$.

The summation and sampling steps are fast since they are only over single-column distributions. This is in contrast to integrating or summing over all columns at once, which has an exponential number of points. The product of the three stored intermediates,

$$\widehat{P}(X_1 \in R_1) \cdot \widehat{P}(X_2 \in R_2|x_1^{(i)}) \cdot \widehat{P}(X_3 \in R_3|x_1^{(i)}, x_2^{(i)}) \tag{5.2}$$

is an unbiased estimate for the desired density. By construction, the sampled point satisfies the query ($x_1^{(i)}$ is drawn from range R_1, $x_2^{(i)}$ from R_2, and so forth). It remains to show that this sampler is approximating the correct sum:

Algorithm 1 Progressive Sampling: estimate the density of query region $R_1 \times \cdots \times R_n$ using S samples.

```
function ProgressiveSampling($S; R_1, \ldots, R_n$)
    $\widehat{P} = 0$
    for $i = 1$ to $S$ do                                        ▷ Batched in practice
        $\widehat{P} = \widehat{P} + \text{Draw}(R_1, \ldots, R_n)$
    return $\widehat{P}/S$

function Draw($R_1, \ldots, R_n$)                                ▷ Draw one tuple
    $\widehat{p} = 1$
    $s = 0_n$                                                    ▷ The tuple to fill in
    for $i = 1$ to $n$ do
        Forward pass through model: $\mathcal{M}(s)$
        $\widehat{P}(X_i|s_{<i})$ = the $i$-th model output        ▷ Eq. 3.2
        Zero-out probabilities in slots $[0, D_i) \setminus R_i$
        Re-normalize, obtaining $\widehat{P}(X_i|X_i \in R_i, s_{<i})$
        $\widehat{p} = \widehat{p} \times \widehat{P}(X_i \in R_i|s_{<i})$
        Sample $s_i \sim \widehat{P}(X_i|X_i \in R_i, s_{<i})$
        $s[i] = s_i$
    return $\widehat{p}$                                         ▷ Density of the sampled tuple $s$
```

Theorem 1 *Progressive Sampling estimates are unbiased.*

The proof uses basic probability rules and is show in the Appendix. Algorithm 1 shows the pseudocode for the general n-filter case. For a column that does not have an explicit filter, it can in theory be treated as having a wildcard filter, i.e., $R_i = [0, D_i)$. We describe our more efficient treatment, *wildcard-skipping*, in §5.2. Our evaluation shows that the sampler can cover both low and high density regions, and handles challenging range queries for large numbers of columns and joint spaces.

Progressive sampling bears connections to sampling algorithms in graphical models. Notice that the autoregressive factorization corresponds to a complex graphical model where each node i has all nodes with indices $< i$ as its parents. In this interpretation, progressive sampling extends the *forward sampling with likelihood weighting* algorithm [19] to allow variables taking on *ranges of values* (the former, in its default form, allows equality predicates only).

5.2 Optimizations

Reducing Variance with Wildcard Skipping

NeuroCard utilizes *wildcard-skipping*, a simple optimization to efficiently handle wildcard predicates. Instead of sampling through the full domain of each wildcard column in a query, $X_i \in *$, we could restrict it to a special token, $X_i = \mathsf{MASK}_i$. Intuitively, MASK_i signifies column i's *absence* and essentially marginalizes it. In our experiments, wildcard-skipping can reduce the variance of worst-case errors by several orders of magnitude.

During training, we perturb each tuple so that the training data contains MASK tokens. We uniformly sample a subset of columns to *mask out*—their original values in the tuple are discarded and replaced with corresponding $\mathsf{MASK}_{\text{col}}$. For an n-column tuple, each column has a probability of w/n to be masked out, where $w \sim \text{Unif}[0, n)$. The output target for the cross-entropy loss still uses the original values.

Inference for Factorized Subcolumns

Recall in §3.5, we describe how columns with a large domain size can be *factorized* into subcolumns via bit chunking, leading to smaller model sizes. This technique requires some modifications for inference as well. During probabilistic inference, a filter on an original column needs to be translated into *equivalent* filters on subcolumns. We modify the standard progressive sampling procedure to handle subcolumns by respecting each filter's semantics. Going back to our example in Figure 3.2, consider the filter $\mathsf{col} < 1{,}000{,}000$. The filter for the high-bits subcol_1 is *relaxed* to ≤ 976 (note the less-equal). The inference procedure would draw a subcol_1 value in this range, based on which the low-bits filter is relaxed appropriately. If the drawn subcol_1 is 976, then the filter on subcol_2 is set to "< 576"; otherwise, the high-bits already satisfy the original filter so a wildcard is placed on the low-bits subcolumn. This is an adaptation of range predicate processing on bit-sliced indexes [31]; NeuroCard applies these processing logic in the new context of probabilistic inference for autoregressive models.

5.3 Schema Subsetting

In the previous sections, we described how NeuroCard can support all predicate types in an efficient manner. However, another challenge with querying this probabilistic model for a selectivity estimate is that the query may *restrict the space it touches to a subset of the full join*—a phenomenon we term *schema subsetting.* Consider:

$$T_1.\text{id} : [1,2] \quad T_2.\text{id} : [1,1] \quad \longrightarrow \quad T_1 ⟗ T_2 : [(1,1),(1,1),(2,\varnothing)]$$
$$\text{Query:} \quad \sigma_{\text{id}=1}(T_1)$$

The correct selectivity is $\frac{1}{2}$ (1 row). However, in the full join distribution, $P(T_1.\text{id} = 1) = \frac{2}{3}$ (2 rows). This is because we have not accounted for the *fanout* produced by the missing table, T_2. Since the selectivity estimate returned by the model assumes the probability space to be the full outer join, rather than the query-specific restricted space, the estimate should be *downscaled* appropriately during probabilistic inference.

NeuroCard handles schema subsetting by building off of ideas proposed by Hilprecht et al. [13] for querying subsets of tables on sum-product networks. We state their algorithms below and discuss how to adapt them into our framework, thereby generalizing these algorithms to (1) non PK-FK joins and (2) a new type of probabilistic model.

Basic case: no table omitted.

The simplest case of schema subsetting is an *inner* join query on all tables. Consider the example data in Figure 4.1a and an inner join query Q1 in Figure 4.1d. The query, $\sigma_{A.x=2}(A \bowtie_x B \bowtie_y C)$, restricts the probability space from the full join to the inner join. Naively querying the model for $|A.x = 2|$ would return a cardinality of $|J| \cdot (3/5) = 3$ rows, as 3 out of 5 rows in the full join J (Figure 4.1c) satisfy the filter. However, the correct row count for this query is 2 (two rows in the inner join; both pass the filter). Left/right outer joins can also exhibit this behavior.

To correct for this, Hilprecht et al. propose a simple solution by adding an *indicator column* per table into the full join. A binary column $\mathbb{1}_T$ is added for each table T, with value 1 if a tuple (in the full join) has a non-trivial join partner with table T, and 0 otherwise.

NeuroCard adopts this solution as follows. First, during training, the sampler is tasked with appending these *virtual* indicator columns on-the-fly to sampled tuples. Recall that each sampled tuple is formed by querying base-table indexes with sampled join keys. If a table T contains a join key, we set that sampled tuple's $\mathbb{1}_T$ to 1, and 0 otherwise (see Figure 4.1c). The autoregressive model treats these indicator columns as regular columns to be learned.

Second, during inference, NeuroCard adds equality constraints on the indicator columns, based on what tables are present in the query. The progressive sampling routine (Algorithm 1) not only gets the usual filter conditions, $\{X_i \in R_i\}$, but also $\{\mathbb{1}_T = 1\}$ for any table T that appears in the inner-join query graph[1]. In summary, for the no-omission case, the routine now estimates the probability:

$$P(\{X_i \in R_i\} \wedge \{\mathbb{1}_T = 1 : \text{for all table } T\}) \tag{5.3}$$

Example. Coming back to the example query Q1, $\sigma_{A.x=2}(A \bowtie_x B \bowtie_y C)$, we compute the selectivity under the full join as $P(A.x = 2 \wedge \mathbb{1}_A = \mathbb{1}_B = \mathbb{1}_C = 1)$. Reading from Figure 4.1c, this probability is 2/5, so the cardinality is correctly computed as $5 \cdot (2/5) = 2$ rows.

[1] The indicator columns can also be constrained appropriately for left or right joins.

Omitting tables and fanout scaling.

The less straightforward case is if a query *omits*, i.e., does not join, certain tables. Consider Q2 in Figure 4.1d: $\sigma_{A.x=2}(A)$. When restricting the scope to table A, the row count of $A.x = 2$ is 1, different from $|J| \cdot P(A.x = 2 \wedge \mathbb{1}_A = 1) = 3$ rows. The fundamental reason this happens is because the operation of a full join has *fanned out* tuples from base tables. To correctly downscale, Hilprecht *et al.* propose recording a per-join *fanout* column. We adapt this solution in NeuroCard[2].

Specifically, for each join key column $T.k$, we insert into the full join a virtual fanout column, $\mathcal{F}_{T.k}$, defined as the number of times each value appears in $T.k$. For example, 2 appears twice in $B.x$, so its fanout is $\mathcal{F}_{B.x}(2) = 2$; see Figures 4.1a and 4.1c. Again, we task the join sampler with adding these fanout values on-the-fly to each batch of sampled tuples. The inclusion of fanouts is piggybacked onto the index lookup path (querying the size of each lookup result list), which adds negligible overheads.

On the inference side, Hilprecht *et al.* showed that the correct cardinality with omitted tables can be computed via *fanout scaling*:

$$\begin{aligned} \text{Cardinality(query Q)} &= |J| \cdot P(\{X_i \in R_i\} \text{ subsetted to query Q}) \\ &= |J| \cdot \mathop{\mathbb{E}}_{X \sim J}\left[\frac{\mathbb{1}_{\{X_i \in R_i\}} \cdot \prod_{T \in Q} \mathbb{1}_T}{\prod_{R \notin Q} \mathcal{F}_{R.\text{key}}}\right]. \end{aligned} \tag{5.4}$$

In essence, the numerator handles the basic case above, while the denominator counts the total number of times omitted tables $\{R \notin Q\}$ have fanned out each tuple in query Q. It loops through each omitted table R, finds its unique join key R.key that connects to Q in the schema (discussed in detail below), and looks up the associated fanout value $\mathcal{F}_{R.\text{key}}$. We incorporate this scaling as follows. Since the fanout columns are learned by the model, we modify progressive sampling to draw a concrete value for each relevant $\mathcal{F}_{R.\text{key}}$ per progressive sample, compute the product of these fanouts, and divide the progressive sample's estimated likelihood by this product.

Example. Coming back to Q2, $\sigma_{A.x=2}(A)$, the constraints are $\{A.x = 2, \mathbb{1}_A = 1\}$. Reading from Figure 4.1c, three rows satisfy the constraints and the relevant downscaling keys are $B.x$ and $C.y$. Thus the expectation expands as: $\frac{1}{5} \cdot (\frac{1}{2 \cdot 1} + \frac{1}{2 \cdot 2} + \frac{1}{2 \cdot 2}) = \frac{1}{5}$. Multiplying with $|J| = 5$ arrives at the correct cardinality of 1 row.

[2] Our definition differs slightly from Hilprecht *et al.*. In that work, each fanout column is bound to a PK-FK join and stores the frequency of a value in the FK. Our treatment binds a fanout to each join key, regardless of PK/FK, and is defined as the frequency each value appears in that key column itself. This removes their assumption of PK-FK joins and supports general equi-joins where both join keys can have duplicate values. Note that our formulation still requires the join keys to be pre-declared before model training begins.

Handling fanout scaling for multi-key joins.

Our formulation of fanout scaling supports multi-key joins, e.g., both x and y keys in the example schema $A.x = B.x \wedge B.y = C.y$ (Figure 4.1a). The challenge of fanout scaling in this case is determining the set of omitted keys to downscale. Let V be the set of all tables. Let Q be the set of tables joined in a query, and the complement $O = V \setminus Q$ the omitted tables. Pick any table $T \in Q$. There exists a unique path from each omitted $T_O \in O$ to T, because the join schema graph is a tree (acyclic, connected). The join key attached to the edge incident to T_O on this path is the unique join key for table T_O to downscale. Hence, the fanout downscaling factor in Equation 5.4 is well-defined.

Going back to example Q2 where only A is queried, when considering the omitted table B which has two join keys ($B.x$, $B.y$), we see that $B.x$ is the unique fanout key since it lies on the path $A \longleftrightarrow B$.

Summary of schema subsetting.

To recap, NeuroCard's probabilistic inference leverages the progressive sampling algorithm from Naru and the idea of additional columns from Hilprecht *et al.* that we term *virtual columns*. Our join sampler is modified to logically insert into the full join two types of virtual columns, the indicators and the fanouts. Both are treated as regular columns to be learned over by the density model, and both are used during progressive sampling to handle various cases of schema subsetting.

Ordering virtual columns in the autoregressive factorization.

The autoregressive model requires some fixed ordering of columns in its factorization (§**??**). Naru has shown that different orderings may have different performance in the tail error but not in the lower error quantiles. We adopt the same practice as Naru in using an arbitrary ordering for the content columns. For the virtual columns introduced above, we place them after all the content columns, with indicators before fanouts. The intuition here is to ensure that (1) the conditional distributions involving content columns do not get confused by the presence of virtual columns, and (2) when sampling fanouts, placing them at the end allows for prediction using a maximum amount of prior information.

In our early benchmarks this choice performed better than if virtual columns were placed early in the ordering. We also experimented with *multi-order training* [6] in the autoregressive model, but did not see noticeably better performance. Thus, we opt for a simple treatment and leave such optimizations to future work.

Chapter 6

Evaluation

We evaluate NeuroCard on accuracy and efficiency and compare it with state-of-the-art cardinality estimators. The key takeaways are:

- **NeuroCard outperforms the best prior methods by 4–72× in accuracy**. On the popular JOB-light benchmark, NeuroCard achieves a maximum error of 8.5× using 4 MB.
- **NeuroCard scales well to more complex queries**. On the two new benchmarks JOB-light-ranges (more difficult range filters) and JOB-M (more tables in schema), NeuroCard achieves orders of magnitude higher accuracy than prior approaches.
- **NeuroCard is efficient to construct and query**. A few million tuples, learned in less than 5 minutes, suffice for it to reach best-in-class accuracy.
- **We study the relative importance of each component of NeuroCard**. Out of all factors, learning the correlations across all tables and performing unbiased join sampling prove the most impactful.

6.1 Experimental Setup

Workloads

The various workloads that we use are described in Table 6.1. We adopt the real-world IMDB dataset and schema to test cardinality estimation accuracy. Prior work [25, 24] reported that correlations are abound in this dataset and established it to be a good testbed for cardinality estimators. We test the following query workloads on IMDB:

- JOB-light: a 70-query benchmark used by many recent cardinality estimator proposals [41, 18, 13]. The schema contains 6 tables, title (primary), cast_info, movie_companies, movie_info, movie_keyword, movie_info_idx and is a typical star schema—every non-primary table only joins with title on title.id. The full outer join contains $2 \cdot 10^{12}$ tuples. Each

Table 6.1: Workloads used in evaluation. *Tables*: number of base tables. *Rows, Cols, Dom.*: row count, column count, and maximum column domain size of the full outer join of each schema. *Feature* characterizes the each workload's queries.

Workload	Tables	Rows	Cols	Dom.	Feature
JOB-light	6	$2 \cdot 10^{12}$	8	235K	single-key joins
JOB-light-ranges	6	$2 \cdot 10^{12}$	13	134K	+complex filters
JOB-M	16	10^{13}	16	2.7M	+multi-key joins

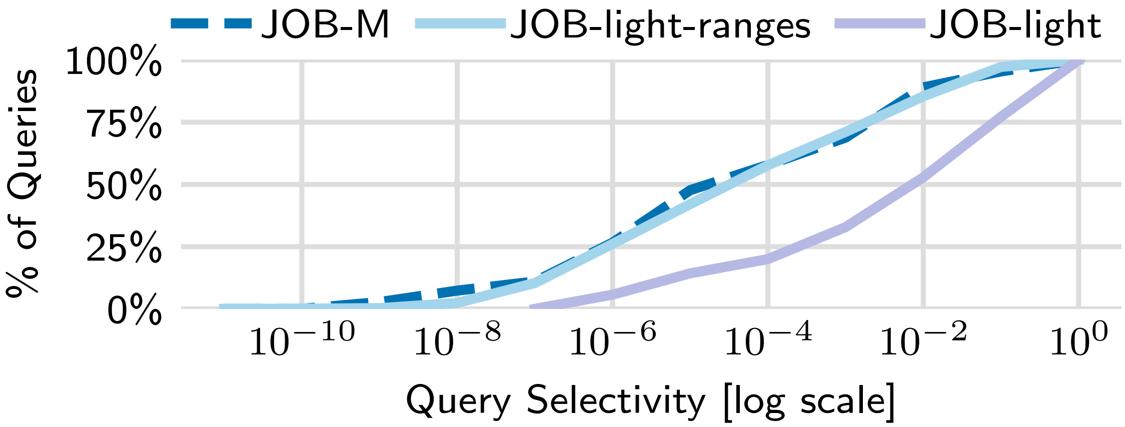


Figure 6.1: Distribution of query selectivity (§6.1).

query joins between 2 to 5 tables, with only equality filters except for range filters on title.production_year.

- JOB-light-ranges: we synthesized this second benchmark containing 1000 queries derived from JOB-light by enriching filter variety. We generate the 1000 queries uniformly distributed to each join graph of JOB-light (18 in total), as follows. For each join graph, using our sampler we draw a tuple from the inner join result. We use the non-null column values of this tuple as filter literals, and randomly place 3–6 comparison operators associated with these literals, based on whether each column can support range (draw one of $\{\leq, \geq, =\}$) or equality filters (=). Overall, this generator (1) follows the data distribution and guarantees non-empty results, and (2) includes more filters, in variety and in quantity, than JOB-light. An example 3-table query is: $\mathsf{mc} \bowtie \sigma_{\mathsf{info\_type\_id=99}}(\mathsf{mi\_idx}) \bowtie \sigma_{\mathsf{episode\_nr}\leq 4 \wedge \mathsf{phonetic\_code}\geq \text{'N612'}}(t)$, where t.id is joined with other tables' movie_id.
- JOB-M: this last benchmark contains 16 tables in IMDB and involves *multiple* join keys. For instance, the table movie_companies is joined not only with title on movie_id, but also with company_name on company_id, and with company_type on company_type_id, etc. We

adapt the 113 JOB queries [24] by allowing each table to appear at most once per query and removing logical disjunctions (e.g., $\mathsf{A.x{=}1} \vee \mathsf{B.y{=}1}$). Each query joins 2–11 tables. We use JOB-M to test NeuroCard's scalability as its full join is 5× larger and has more dimensions than the above (see Table 6.1).

Metric

We report the usual Q-error distribution of each workload, where the Q-error of a query is the multiplicative factor an estimated cardinality deviates from the query's true cardinality:

$$\text{Q-error(query)} := \max\left(\frac{\mathsf{card}_{\mathsf{actual}}}{\mathsf{card}_{\mathsf{estimate}}}, \frac{\mathsf{card}_{\mathsf{estimate}}}{\mathsf{card}_{\mathsf{actual}}}\right).$$

Both actual and estimated cardinalities are lower bounded by 1, so the minimum attainable Q-error is 1×. As reported in prior work [50], reducing high-quantile errors is much more challenging than mean or median; thus, we report the quantiles $p100, p99, p95$, and the median. For timing experiments, we report latency/throughput using an AWS EC2 VM with a NVIDIA V100 GPU and 32 vCPUs.

Benchmark characteristics

Figure 6.1 plots the distributions of selectivities of these workloads, where we calculate each query's selectivity as $\mathsf{card}_{\mathsf{actual}}/\mathsf{card}_{\mathsf{inner}}$ (denominator is the row count of the query join graph—an inner join—without filters). The selectivity spectrums of our two benchmarks (JOB-light-ranges and JOB-M) are much wider than JOB-light due to higher filter variety. The median selectivity is more than 100× lower, while at the low tail the minimum selectivities are 1000× lower.

Compared Approaches

We compare against several prevalent families of estimators. In each family, we aim to choose a state-of-the-art representative. Related Work (§2) includes a more complete discussion on all families and their representative methods.

Supervised query-driven estimators

We use MSCN [18] as a recent representative from this family. It takes in a featurized query, runs the query filters on pre-materialized samples of the base tables, then use these bitmaps as additional network inputs, and predicts a final cardinality. For JOB-light, we used the training queries and sample bitmaps provided in the authors' source code [17]. For JOB-light-ranges, due to new columns, we generated 10K new training queries—generating and executing them to obtain true cardinality labels took 3.2 hours—and used a bitmap size of 2K to match the size of other estimators in this benchmark. For JOB-light, we also cite the best numbers obtained by Sun and Li [41], termed *E2E*, which is a deep supervised net with more effective building blocks (e.g., pooling, LSTM) than MSCN.

Unsupervised data-driven estimators

We use DeepDB [13] as a recent representative in this family. It builds a (non-neural) sum-product network [32] as density estimator on each heuristically chosen table subset. Across table subsets, conditional independence is assumed. In contrast, NeuroCard uses a neural network—the deep autoregressive model—and builds a single learned estimator over all tables in a schema. We use two recommended configurations from DeepDB: a base version that uses four 2-table models and an 1-table model, and a larger version that additionally builds two 3-table models chosen by their inter-column correlation heuristics.

We found that the DeepDB source code [12] did not support range queries on categorical string columns out-of-the-box. Since JOB-light-ranges contains such queries, we perform data and query rewriting for this baseline, by dictionary-encoding the string values into integers. Reported results are with this optimization enabled.

Join sampling

We implement Index-based Join Sampling (IBJS) [23], using 10,000 as the maximum sample size. A query's cardinality is estimated by taking a sample from the query's join graph and executing per-table filters on-the-fly.

Real DBMS

We use Postgres (v12), which carries out cardinality estimation using 1D histograms and heuristics to combine them.

NeuroCard

We implement NeuroCard using the architecture described in §3.4 with layer sizes chosen to match the space usage of baselines. We train NeuroCard on 7M tuples for JOB-light and 10M for JOB-light-ranges/JOB-M. For inference, 512 progressive samples are drawn per query.

6.2 Estimation Accuracy

JOB-light

Table 6.2 reports each estimator's accuracy on the 70 JOB-light queries. Overall, NeuroCard exhibits high accuracy across the spectrum. **It sets a new state-of-the-art maximum error at** 8.5× using 3.8 MB of parameters. This represents an $> 8\times$ improvement over the best prior method when controlling for size.

Table 6.2: JOB-light, estimation errors. Lowest errors are bolded.

Estimator	Size	Median	95th	99th	Max
Postgres	70 KB	7.97	797	$3 \cdot 10^3$	10^3
IBJS	–	1.48	10^3	10^3	10^4
MSCN	2.7 MB	3.01	136	$1 \cdot 10^3$	10^3
E2E (quoting [41])	N/A	3.51	139	244	272
DeepDB	3.7 MB	1.32	4.90	33.7	72.0
DeepDB-large	32 MB	**1.19**	**4.66**	35.0	39.5
NeuroCard	3.8 MB	1.57	5.91	**8.48**	**8.51**

We now discuss a few observations. Not surprisingly, Postgres has the most inaccurate median—indicating a systematic mismatch between the approximated distribution and data—due to its use of coarse-grained density models (histograms) and heuristics. IBJS fares better at the median, but falls off sharply at tail, because samples of a practical size have a small chance to hit low-density queries in a large joint space. Both MSCN and E2E are deep supervised regressors which show marked improvements over prior methods. However, their median and 95th errors are quite similar and have sizable gaps from the two data-driven estimators.

NeuroCard vs. DeepDB shows interesting trends. NeuroCard is up to 4–8× better at tail (99th, max), and DeepDB is slightly better at lower quantiles. NeuroCard is more robust at tail due to (1) a markedly better density model (neural autoregressive vs. non-neural sum-product networks that use inter-column independence assumptions), and (2) learning all possible correlations among the columns of all 6 tables, whereas DeepDB assumes (conditional) independence across several table subsets. DeepDB-large, being 8.4× bigger and trained on 7.7× more (54M) tuples, still trails NeuroCard at tail by more than 4×. NeuroCard slightly trails at the lower quantiles ("easy" queries with high true density) likely due to the mode-covering behavior of KL-divergence minimization [9].

JOB-light-ranges

This 1000-query benchmark adds equality/range filters on more content columns, using the same join templates as JOB-light (which has range filters on one column only). Results are shown in Table 6.3.

NeuroCard achieves the best accuracy across all error quantiles, and improves on the best prior methods by up to 15–72×. It is also the only estimator with a $< 2\times$ median and a two-digit 95%-tile errors. Overall, all estimators produce less accurate cardinalities, though the drops are of varying degrees. Compared with MSCN, NeuroCard improves by 2× at median, 7× at 95th, 15× at 99th, and 2× at max. Compared with DeepDB, NeuroCard improves the four quantiles by 2×, 9×, 21×, and 23×, respectively.

Table 6.3: JOB-light-ranges, estimation errors. Lowest errors bolded.

Estimator	Size	Median	95th	99th	Max
Postgres	70 KB	13.8	$2 \cdot 10^3$	$2 \cdot 10^4$	$5 \cdot 10^6$
IBJS	–	10.1	$4 \cdot 10^4$	10^6	10^8
MSCN	4.5 MB	4.53	397	$6 \cdot 10^3$	$2 \cdot 10^4$
DeepDB	4.4 MB	3.40	537	$8 \cdot 10^3$	$2 \cdot 10^5$
DeepDB-large	33.6 MB	2.35	441	$1 \cdot 10^4$	$3 \cdot 10^5$
NeuroCard	4.1 MB	1.87	57.1	375	8169
NeuroCard-large	23 MB	**1.49**	**44.0**	**300**	**4116**

Table 6.4: JOB-M, estimation errors. Lowest errors are bolded.

Estimator	Size	Median	95th	99th	Max
Postgres	120 KB	174	$1 \cdot 10^4$	$8 \cdot 10^4$	$1 \cdot 10^5$
IBJS	–	61.1	$3 \cdot 10^5$	$4 \cdot 10^6$	$4 \cdot 10^6$
NeuroCard	27.3 MB	**3.2**	**283**	**1297**	$1 \cdot 10^4$

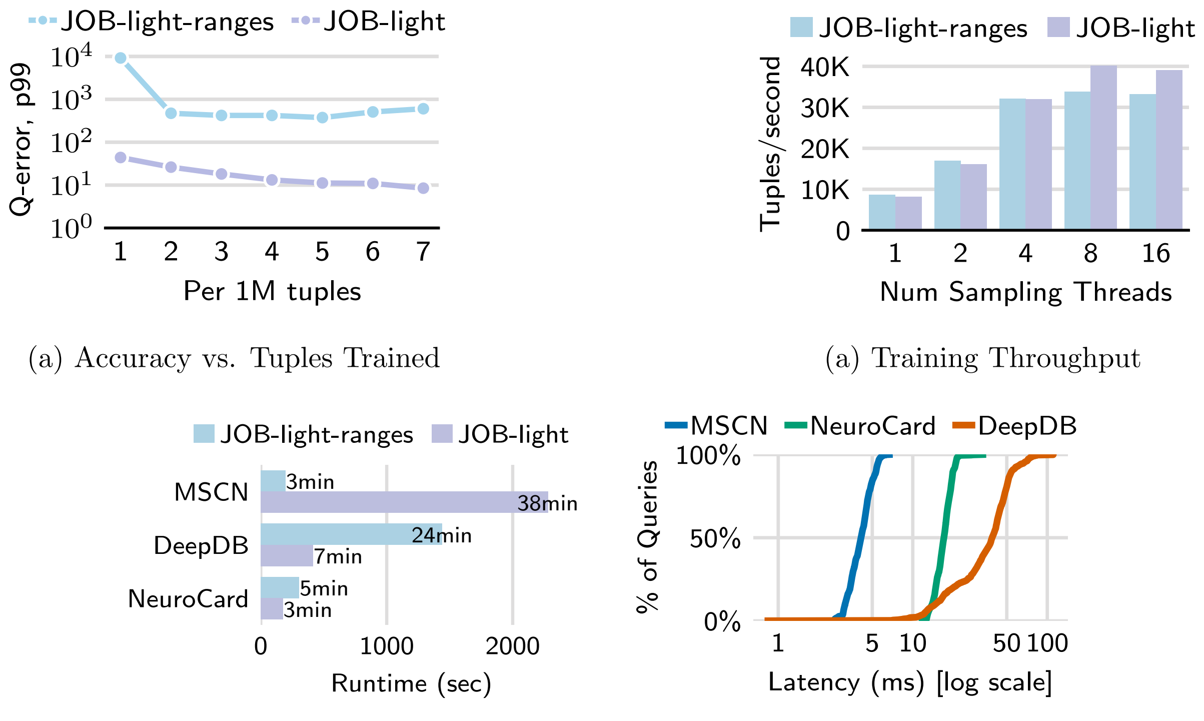
Comparing the enlarged versions of the two estimators (suffixed with -large), the accuracy gains become $1.5\times, 10\times, 33\times$ and $72\times$, respectively.

NeuroCard's improvements over baselines significantly widen in this benchmark, due to prior approaches failing to capture the more complex inter-column correlations being tested.

JOB-M

This final benchmark tests NeuroCard's ability to scale to a much larger and more complex join schema. Different from the JOB-light schema, JOB-M contains 16 tables, with each query joining 2–11 tables on multiple join keys (in addition to movie_id only in JOB-light). For baselines, we only include Postgres and IBJS, because MSCN's query encoding does not support the complex filters in this benchmark and DeepDB had intractable training time (did not finish in 22 hours) on this 16-table dataset due to high-cardinality categorical columns.

Results in Table 6.4 show that **NeuroCard's accuracy remains high on this complex schema**. Postgres produces large errors, and IBJS also struggles, due to many intermediate samples becoming empty as the number of joins grows. NeuroCard overcomes this challenge and offers more than 10× better accuracy across the board. In terms of space efficiency, since the model needs to be trained on the full outer join of 16 tables and the maximum domain size exceeds 2 million, a vanilla NeuroCard would require 900 MB in model size. With column factorization (§3.5), the model size is reduced to 27MB—less than 1% of the total size of all tables.



(a) Accuracy vs. Tuples Trained

(a) Training Throughput

(b) Training Comparison

(c) Inference Comparison

Figure 6.2: Statistical and physical efficiency of NeuroCard.

6.3 Efficiency

Having established that NeuroCard achieves the best accuracy, we now study the statistical and physical efficiency of NeuroCard.

How many tuples are required for good accuracy?

Figure 6.2a plots accuracy (p99 on JOB-light and JOB-light-ranges) vs. number of tuples trained. *About 2–3M tuples are sufficient for NeuroCard to achieve best-in-class accuracy* (compare with Tables 6.2 and 6.3). Using more samples helps, but eventually yields diminishing returns. Reaching high accuracy using a total of $\sim 10^7$ samples out of a population of 10^{12} data points (i.e., only 0.001% of the data)—many queries would inevitably touch unseen data points—shows that NeuroCard generalizes well and is *statistically efficient*.

How does sampling affect training throughput?

Figure 6.2a plots the training throughput, in tuples per second, vs. the number of sampling threads used to provide training data. Four threads suffice to saturate the GPU used for training. At lower thread counts, the device spends more time waiting for training data than doing computation. With a peak throughput of $\sim$ 40K tuples/second, NeuroCard can finish training on 3M tuples in about 1.25 minutes.

Wall-clock training time comparison

Figure 6.2b compares the wall-clock time used for training the MSCN, DeepDB, and NeuroCard configurations reported in Tables 6.2 and 6.3. DeepDB runs on CPU, hence takes the longest; its construction time is significantly increased on JOB-light-ranges, which has more columns to learn. MSCN requires a separate phase of executing training queries to collect true cardinalities, which takes much longer (3.2 hours for 10K queries) than just the training time shown here. NeuroCard starts training after calculating the join count tables, which takes 13 seconds for both datasets. Its construction is the most efficient due to the use of parallel sampling and accelerated GPU computation.

Wall-clock inference time comparison

Lastly, Figure 6.2c plots the latency CDF of the learning approaches for 1000 JOB-light-ranges queries. As before, we use the base configurations reported in the accuracy Tables. MSCN and NeuroCard run on GPU while DeepDB runs on CPU; all three approaches are implemented in Python. MSCN is fastest because its lightweight network has fewer calculations involved. DeepDB's latencies span a wide spectrum, from $\sim$ 1 ms for queries with low complexity (numbers of joins and filters involved) to $\sim$ 100 ms for queries with the highest complexity. NeuroCard's latencies are more predictable, with 17 ms at median and 12 ms at minimum: this is due to the higher number of floating point operations involved in the neural autoregressive model. All approaches can be sped up by engineering efforts (e.g., if run in a native language). For NeuroCard, model compression or weight quantization can also reduce the computational cost.

6.4 Dissecting NeuroCard

To gain insights, we now evaluate the relative importance of primary components of NeuroCard, by varying them and measuring the change in estimation accuracy on JOB-light-ranges. We use the smaller NeuroCard in Table 6.3 as the *Base* configuration, and ablate each component in isolation. Table 6.5 presents the results.

In (A), using IBJS adapted for full joins[1] as a *biased* sampler significantly decreases the learned estimator's accuracy. The large increase in the median error implies a systematic distribution mismatch. Overall, this design choice is the second most important.

[1] The fact table title is ordered at front and a large intermediate size of 10^6 is used.

Table 6.5: Ablation studies: varying primary components of NeuroCard. Unlisted values are identical to the Base configuration. We show the impact of the sampler (A), column factorization bits (B), autoregressive model size (C), inter-table correlations learned (D), and whether to use an autoregressive model at all (E) on the 50% and 99%-tile errors of JOB-light-ranges.

	Sampler	Fact. Bits	$d_{\text{ff}}; d_{\text{emb}}$	Correlations Learned	**p50**	**p99**
Base (4.1 MB)	unbiased	14	128; 16	all tables in one AR	1.9	375
(A)	biased				33	$1 \cdot 10^4$
(B)		10 (2.2 MB)			2.2	2811
		12 (2.6 MB)			2.0	936
		None (12 MB)			1.6	375
(C)			128; 64 (23 MB)		1.5	300
			1024; 16 (31 MB)		1.7	497
(D)				one AR per table	40	$7 \cdot 10^6$
(E)	No model; uniform join samples only				4.0	$3 \cdot 10^6$

Rows in group (B) vary the column factorization granularity. Using smaller bits results in more subcolumns and yields a small drop in accuracy. Disabling factorization uses the most space and appears to perform the best.

Group (C) varies the size of the autoregressive model, by changing the dimension of the feedforward linear layers (d_{ff}) or the embeddings (d_{emb}). An enlarged embedding proves markedly more useful than enlarged linear layers, likely because each token's captured semantics becomes more finetuned during optimization.

In group (D) we vary the correlation learned by NeuroCard. While all configurations above learn the distribution of all tables in a single model—capturing all possible correlations among them—here we build one model (same architecture as Base) per table. Queries that join across tables are estimated by combining individual models' estimates via independence. Without modeling inter-table correlations, this variant yields the lowest accuracy.

Finally, group (E) ablates away the AR model altogether. We test *uniform join samples* as a standalone estimator: it uses our sampler (§4) to draw 10^4 simple random samples (actual tuples in the database) from each query's join graph. While the median error is reasonable, it is $10^4\times$ less accurate than an autoregressive model at tail as many queries

Table 6.6: Updating NeuroCard, fast and slow. JOB-light.

Strategy	Update Time	Error	Partitions Ingested 1	2	3	4	5
stale	None	p95	2.82	1848	10^5	10^4	10^4
		p50	1	1	5.69	207	408
fast update	$\sim$ 3 sec	p95	2.82	5.39	12.84	12.85	14.3
		p50	1	1	1.32	1.37	1.51
retrain	$\sim$ 3 min	p95	2.82	5.87	6.08	7.53	6.43
		p50	1	1	1.16	1.20	1.52

have no sample hits. The AR model is more statistically efficient than sampling, because it provides access to conditional probability distributions—these conditional contributions enable an efficient probabilistic inference procedure, progressive sampling, which cannot be used otherwise.

6.5 Update Strategies

NeuroCard handles new data by either retraining, or taking additional gradient steps, i.e., incremental training. To test both strategies, we simulate the practice of *time-ordered partition appends*: table title is range-partitioned on a year column into 5 partitions. Each partition defines a distinct snapshot of the entire database and the full join, so running the same set of queries at different partition count yields 5 sets of true cardinalities. We compare three update strategies, all of which are trained fully for 7M tuples after the first ingest: (1) *stale*, trained once on the first snapshot and never updated, (2) *fast update*, incrementally updated after each new ingest on 70k additional samples (1% of the number of samples used for the initial ingest), and (3) *retrain*, completely retraining a new model with 7M tuples after each ingest. We also show the latency required to perform additional gradient steps.

Results are shown in Table 6.6. Without update, the stale NeuroCard significantly degrades in accuracy, which is expected as each partition adds a significant amount of new information. A fast updated NeuroCard recovers most of the accuracy, incurring a minimal overhead. Even fully retraining only requires a few minutes and yields the highest accuracy. Both the statistical efficiency (number of tuples needed vs. accuracy) and the physical efficiency of NeuroCard contribute to these highly practical update strategies.

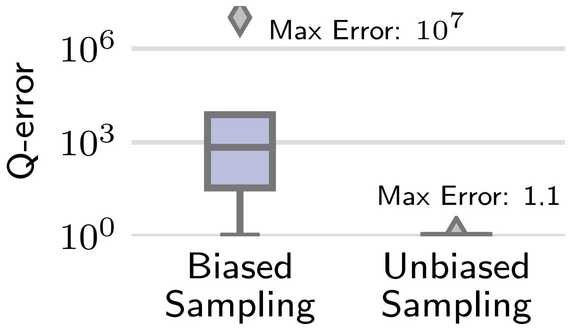


Figure 6.3: Impact of biased vs. unbiased sampling (§6.6).

6.6 Biased vs. unbiased join sampling

In this section, we further dissect the impact of biased vs. unbiased join sampling by considering the worst case for biased join sampling. We synthesize a simple two-table schema to highlight why biased sampling fails. Table A contains distinct keys $\{1, \ldots, 10^6\}$, and table B contains the same keys but with a heavy hitter value, $500K$, duplicated 10^7 times. The join schema is $A \bowtie B$ (which equals B) with a total size of $11M$ rows. We evaluate on 50 queries all of which are intervals that include the heavy hitter; we vary the intervals in size so that the lowest-density query is an equality lookup on the heavy hitter, and the highest queries the entire join result.

We compare the impact of biased vs. unbiased join sampling on this *worst-case* scenario. As an example of biased join sampling, we run the DeepDB source code, which uses IBJS to collect samples from a join [12]. As discussed in §4, it yields a biased data distribution. As comparison, we build a NeuroCard estimator which uses unbiased join sampling. Both methods are trained on $110K$ samples.

Figure 6.3 shows the Q-error distributions. As expected, unbiased sampling produces an estimator that is 7 orders of magnitude more accurate. Biased sampling fails because its samples all fail to hit the heavy hitter, so by training on these biased samples, the estimator is essentially approximating A (the uniform keys) instead of the true distribution $A \bowtie B$. We also see a 1000× difference in the median. The gaps become smaller as the true query densities increase—for the easiest scan-all query, biased sampling becomes on par.

Takeaway: biased sampling can be made arbitrarily inaccurate. Various optimizations or datasets with less severe skews may mitigate this problem, but NeuroCard's proposal of unbiased join sampling is a more principled solution and should be used to build unsupervised cardinality estimators.

Chapter 7

Conclusion

NeuroCard is built on a simple idea: learn the correlations across all tables in a database without making any independence assumptions. NeuroCard applies established techniques from join sampling and deep self-supervised learning to cardinality estimation, a fundamental problem in query optimization. It learns from data—just like classical data-driven estimators—but captures all possible inter-table correlations in a probabilistic model: $p_\theta(\text{all tables})$. To our knowledge, NeuroCard is the first cardinality estimator to achieve assumption-free probabilistic modeling of more than a dozen tables. NeuroCard achieves state-of-the-art accuracy for join cardinality estimation (4–72× better than prior methods) using a single per-schema model that is both compact and efficient to learn. We believe NeuroCard will be used as a robust and accurate cardinality estimator in classical and future learning-based query optimizers.

Bibliography

[1] Michael Armburst et al. “Spark SQL: Relational Data Processing in Spark”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’15. Melbourne, Victoria, Australia: ACM, 2015, pp. 1383–1394. ISBN: 978-1-4503-2758-9.

[2] Amol Deshpande, Minos Garofalakis, and Rajeev Rastogi. “Independence is good: Dependency-based histogram synopses for high-dimensional data”. In: *ACM SIGMOD Record* 30.2 (2001), pp. 199–210.

[3] Conor Durkan and Charlie Nash. “Autoregressive Energy Machines”. In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. Long Beach, California, USA: PMLR, Sept. 2019, pp. 1735–1744.

[4] Anshuman Dutt et al. “Selectivity estimation for range predicates using lightweight models”. In: *Proceedings of the VLDB Endowment* 12.9 (2019), pp. 1044–1057.

[5] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*. Springer series in statistics New York, 2001.

[6] Mathieu Germain et al. “MADE: Masked autoencoder for distribution estimation”. In: *International Conference on Machine Learning*. 2015, pp. 881–889.

[7] Lise Getoor, Benjamin Taskar, and Daphne Koller. “Selectivity estimation using probabilistic models”. In: *ACM SIGMOD Record*. Vol. 30. 2. ACM. 2001, pp. 461–472.

[8] Lise Getoor et al. “Learning probabilistic models of relational structure”. In: *ICML*. Vol. 1. 2001, pp. 170–177.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[10] Dimitrios Gunopulos et al. “Selectivity estimators for multidimensional range queries over real attributes”. In: *The VLDB Journal* 14.2 (2005), pp. 137–154.

[11] Max Heimel, Martin Kiefer, and Volker Markl. “Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’15. Melbourne, Victoria, Australia: ACM, 2015, pp. 1477–1492. ISBN: 978-1-4503-2758-9.

[12] Hilprecht et al. *Github repository, deepdb-public*. `github.com/DataManagementLab/deepdb-public`. [Online; accessed April, 2020]. 2020.

[13] Benjamin Hilprecht et al. "DeepDB: Learn from Data, not from Queries!" In: *Proceedings of the VLDB Endowment* 13.7 (2020), pp. 992–1005.

[14] Raghav Kaushik et al. "Synopses for Query Optimization: A Space-complexity Perspective". In: vol. 30. 4. New York, NY, USA: ACM, Dec. 2005, pp. 1102–1127.

[15] Martin Kiefer et al. "Estimating join selectivities using bandwidth-optimized kernel density models". In: *Proceedings of the VLDB Endowment* 10.13 (2017), pp. 2085–2096.

[16] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. 2015.

[17] Kipf et al. *Github repository, learnedcardinalities*. `https://github.com/andreaskipf/learnedcardinalities`. [Online; accessed April, 2020]. 2019.

[18] Andreas Kipf et al. "Learned Cardinalities: Estimating Correlated Joins with Deep Learning". In: *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. 2019.

[19] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[20] Flip Korn, Theodore Johnson, and HV Jagadish. "Range selectivity estimation for continuous attributes". In: *Proceedings. Eleventh International Conference on Scientific and Statistical Database Management*. IEEE. 1999, pp. 244–253.

[21] Tim Kraska et al. "The case for learned index structures". In: *Proceedings of the 2018 International Conference on Management of Data*. ACM. 2018, pp. 489–504.

[22] Sanjay Krishnan et al. "Learning to optimize join queries with deep reinforcement learning". In: *arXiv preprint arXiv:1808.03196* (2018).

[23] Viktor Leis et al. "Cardinality Estimation Done Right: Index-Based Join Sampling." In: *CIDR*. 2017.

[24] Viktor Leis et al. "How good are query optimizers, really?" In: *Proceedings of the VLDB Endowment* 9.3 (2015), pp. 204–215.

[25] Viktor Leis et al. "Query optimization through the looking glass, and what we found running the join order benchmark". In: *The VLDB Journal* (2018), pp. 1–26.

[26] Feifei Li et al. "Wander join: Online aggregation via random walks". In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 615–629.

[27] Ryan Marcus and Olga Papaemmanouil. “Deep Reinforcement Learning for Join Order Enumeration”. In: *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. aiDM’18. Houston, TX, USA: ACM, 2018, 3:1–3:4. ISBN: 978-1-4503-5851-4. DOI: `10.1145/3211954.3211957`. URL: `http://doi.acm.org/10.1145/3211954.3211957`.

[28] Ryan Marcus et al. “Neo: A Learned Query Optimizer”. In: *PVLDB* 12.11 (2019), pp. 1705–1718.

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

[30] M Muralikrishna and David J DeWitt. “Equi-depth multidimensional histograms”. In: *ACM SIGMOD Record*. Vol. 17. 3. ACM. 1988, pp. 28–36.

[31] Patrick O’Neil and Dallan Quass. “Improved query performance with variant indexes”. In: *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*. 1997, pp. 38–49.

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

[33] Viswanath Poosala and Yannis E Ioannidis. “Selectivity estimation without the attribute value independence assumption”. In: *VLDB*. Vol. 97. 1997, pp. 486–495.

[34] Viswanath Poosala et al. “Improved Histograms for Selectivity Estimation of Range Predicates”. In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’96. Montreal, Quebec, Canada: ACM, 1996, pp. 294–305. ISBN: 0-89791-794-4. DOI: `10.1145/233269.233342`. URL: `http://doi.acm.org/10.1145/233269.233342`.

[35] Alec Radford et al. “Language models are unsupervised multitask learners”. In: *URL https://openai.com/blog/better-language-models* (2019).

[36] Tim Salimans et al. “PixelCNN++: Improving the PixelCNN with Discretized Logistic Mixture Likelihood and Other Modifications”. In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. 2017.

[37] P Griffiths Selinger et al. “Access path selection in a relational database management system”. In: *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. ACM. 1979, pp. 23–34.

[38] Rico Sennrich, Barry Haddow, and Alexandra Birch. “Neural Machine Translation of Rare Words with Subword Units”. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725.

[39] R. Sethi et al. “Presto: SQL on Everything”. In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 2019, pp. 1802–1813.

[40] Michael Stillger et al. “LEO-DB2’s learning optimizer”. In: *VLDB*. Vol. 1. 2001, pp. 19–28.

[41] Ji Sun and Guoliang Li. “An end-to-end learning-based cost estimator”. In: *Proceedings of the VLDB Endowment* 13.3 (2019), pp. 307–319.

[42] Kostas Tzoumas, Amol Deshpande, and Christian S Jensen. “Efficiently adapting graphical models for selectivity estimation”. In: *The VLDB Journal* 22.1 (2013), pp. 3–27.

[43] Kostas Tzoumas, Amol Deshpande, and Christian S Jensen. “Lightweight graphical models for selectivity estimation without independence assumptions”. In: *Proceedings of the VLDB Endowment* 4.11 (2011), pp. 852–863.

[44] Aaron Van den Oord et al. “Conditional image generation with pixelcnn decoders”. In: *Advances in neural information processing systems*. 2016, pp. 4790–4798.

[45] Aaron Van den Oord et al. “WaveNet: A generative model for raw audio”. In: *arXiv preprint arXiv:1609.03499* (2016).

[46] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[47] Chenggang Wu et al. “Towards a learning optimizer for shared clouds”. In: *Proceedings of the VLDB Endowment* 12.3 (2018), pp. 210–222.

[48] Richard Wu et al. “Attention-based Learning for Missing Data Imputation in HoloClean”. In: *Proceedings of Machine Learning and Systems* (2020), pp. 307–325.

[49] Yingjun Wu et al. “Designing succinct secondary indexing mechanism by exploiting column correlations”. In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 1223–1240.

[50] Zongheng Yang et al. “Deep Unsupervised Cardinality Estimation”. In: *Proceedings of the VLDB Endowment* 13.3 (2019), pp. 279–292.

[51] Zongheng Yang et al. “Qd-tree: Learning Data Layouts for Big Data Analytics”. In: *Proceedings of the 2020 International Conference on Management of Data*. SIGMOD ’20. 2020.

[52] Barzan Mozafari Yongjoo Park Shucheng Zhong. “QuickSel: Quick Selectivity Learning with Mixture Models”. In: *SIGMOD* (2020).

[53] Zhuoyue Zhao et al. “Random sampling over joins revisited”. In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 1525–1539.

Appendix A

Proof of Theorem 1

The proof uses only basic probability rules. For ease of exposition, we prove the 3-column case; the general N-column case follows the exact structure. Specifically, we need to show the expectation of Equation 5.2,

$$\mathop{\mathbb{E}}_{x_1^{(i)},x_2^{(i)}}\Big[\widehat{P}(X_3 \in R_3|x_1^{(i)},x_2^{(i)})\widehat{P}(X_2 \in R_2|x_1^{(i)})\widehat{P}(X_1 \in R_1)\Big]$$

equals the desired density. First, expanding the expectation over $x_1^{(i)}$ gives

$$\mathop{\mathbb{E}}_{x_2^{(i)}}\big[\sum_{x_1=K\in R_1}\widehat{P}(x_1=K|x_1 \in R_1)\widehat{P}(X_3 \in R_3|x_1=K,x_2^{(i)})$$
$$\widehat{P}(X_2 \in R_2|x_1=K)\widehat{P}(X_1 \in R_1)\big]$$

Applying Bayes' rule to the new conditional term,

$$\mathop{\mathbb{E}}_{x_2^{(i)}}\big[\sum_{x_1=K\in R_1}\frac{\widehat{P}(x_1=K)}{\widehat{P}(X_1 \in R_1)}\widehat{P}(X_3 \in R_3|x_1=K,x_2^{(i)})$$
$$\widehat{P}(X_2 \in R_2|x_1=K)\widehat{P}(X_1 \in R_1)\big]$$

Similarly, we expand the expectation over $x_2^{(i)}$ and applying the same rule to get

$$\sum_{\substack{x_1=K\in R_1\\ x_2=M\in R_2}}\big[\frac{\widehat{P}(x_2=M|x_1=K)}{\widehat{P}(X_2 \in R_2|x_1=K)}\frac{\widehat{P}(x_1=K)}{\widehat{P}(X_1 \in R_1)}$$
$$\widehat{P}(X_3 \in R_3|x_1=K,x_2=M)\widehat{P}(X_2 \in R_2|x_1=K)\widehat{P}(X_1 \in R_1)\big]$$

Canceling terms, we obtain

$$\sum_{\substack{x_1=K\in R_1\\ x_2=M\in R_2}}\big[\widehat{P}(X_3 \in R_3|x_1=K,x_2=M)\widehat{P}(x_2=M|x_1=K)$$
$$\widehat{P}(x_1=K)\big]$$

which is the density $\widehat{P}(X_1 \in R_1, X_2 \in R_2, X_3 \in R_3)$.