# Machine Learning for Query Optimization

by

Zongheng Yang

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Chair
Professor Joseph M. Hellerstein
Professor Pieter Abbeel
Assistant Professor Sanjay Krishnan

Summer 2022

Machine Learning for Query Optimization

## Abstract

Machine Learning for Query Optimization

by

Zongheng Yang

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

Data has been growing at an unprecedented rate in the past two decades. As a result, systems that store, process, and analyze data have become mission-critical. Crucial to the performance of data systems is the query optimizer, which translates high-level declarative queries (e.g., SQL) into efficient execution plans. However, query optimization is highly complex, leading to two key challenges. First, optimizers use a myriad of hand-designed heuristics to tame the complexity, but heuristics leave performance on the table. Second, optimizers are highly costly to develop, where human experts may spend months writing a first version and years refining it.

This dissertation applies and enhances machine learning advances to tame the complexity in query optimization. First, we remove for the first time decades-old and accuracy-impacting heuristics in cardinality estimation—the Achilles' heel of optimizers where heuristics particularly abound—thereby significantly improving estimation accuracy. We present Naru and NeuroCard, two cardinality estimators based on self-supervised learning advances that learn the joint data distribution of tables without any heuristic assumptions. Our estimators improve the accuracy of cardinality estimation by orders of magnitude compared to the prior state of the art. Second, we show that automatically learning to optimize SQL queries, without learning from an expert-designed optimizer, is both possible and efficient, thereby potentially alleviating the high development cost. We introduce Balsa, a deep reinforcement learning agent that automatically learns to optimize SQL queries by trial-and-error. Balsa can learn to outperform the optimizers of PostgreSQL—one of the most popular database systems—and a commercial database engine with a few hours of learning.

Overall, by enhancing machine learning advances with new, carefully designed systems and ML techniques, this line of work improves existing query optimizers, while opening the possibility of alleviating the complex optimization in future environments and engines.

*To my family*

# Contents

# List of Figures

# List of Tables

## Acknowledgments

First and foremost, I would like to thank Ion Stoica, my advisor, for his guidance and support. Ion taught me the value of focusing on impact and doing great work. I learned from him the importance of striving for simplicity and insights at the same time, a most brilliant skill that he taught by example. Throughout my PhD, Ion has always offered encouragement and reminders of the big picture when I needed them. I am still constantly inspired by and imitating his uncanny ability to always remain constructive and looking ahead.

Joe Hellerstein has been instrumental in sharpening my research rigor. I vividly remember the many lively discussions Joe and I had on either research problems, tweaking presentations, or career advice. In these, Joe has inspired me to set a high bar for scholarship and critical thinking, which I will carry forward in my career.

I also thank the other members of my dissertation committee: Pieter Abbeel and Sanjay Krishnan. Pieter's cutting-edge research in self-supervised learning and reinforcement learning piqued my interests in these recent advances. These techniques subsequently became the foundation for many of the ideas developed in this dissertation. Sanjay took the time (and chance) to collaborate with me, then a fresh graduate student, and introduced me to the area that ended up becoming the main topic of this research.

I am blessed to have worked with great collaborators in graduate school. Eric Liang was a firm collaborator in my first few years and projects. We spent much time together coding, discussing ideas, and honing each other's thinking—all of which I enjoyed greatly. The Naru, NeuroCard, and Balsa projects included in-depth contributions from my co-authors: Pieter Abbeel, Peter Chen, Wei-Lin Chiang, Rocky Duan, Joe Hellerstein, Amog Kamsetty, Sanjay Krishnan, Eric Liang, Frank Luan, Michael Luo, Gautam Mittal, Chenggang Wu, and Ion Stoica. I learned much from each and every one of them through our close collaborations.

In the past year, I had the fortune to start an entirely different project related to Sky Computing with Scott Shenker, Ion Stoica, Romil Bhardwaj, Wei-Lin Chiang, Woosuk Kwon, Michael Luo, Zhanghao Wu, and Siyuan Zhuang. Working on something new with them has been exhilarating.

During my undergraduate study, Shivaram Venkataraman, Rachit Agarwal, and Anurag Khandelwal gave me my first lessons on how research should be done. The research experience with them inspired me to pursue a PhD. I also thank Reynold Xin, who kindly introduced me to my first research opportunity in the AMPLab. When I was at Google prior to starting my PhD, Zhifeng Chen offered me well-thought-out engineering and research projects, which was an experience I remember fondly.

Many people in and outside the RISELab made my PhD experience much more joyful. I cherish the camaraderie with Wen Zhang and Chenggang Wu; they were always game for conversations, be it social and casual, or serious and intellectual. Michael Whittaker is the best paper reviewer I know: I sent most, if not all, of my paper drafts to him and he provided thoughtful and actionable feedback every time. Richard Liaw had a knack for offering unexpected perspectives in our conversations. Staff members of the lab—Kattt Atchley, Boban Zarkovich, Jon Kuroda, David Schonenberg, Shane Knapp, Ivan Ortega—extended tremendous help to my undergraduate and graduate life. I am privileged to have interacted with and learned from many more people, including Philipp Moritz, Robert Nishihara, Alexey Tumanov, Stephanie Wang, Nilesh Tripuraneni, Paras Jain, Eyal Sela, Daniel Rothchild, Justin Wong, Melih Elibol, Nathan Pemberton, Devin Petersohn, Johann Schleier-Smith, Joao Carreira, Ankur Dave, Qifan Pu, Wenting Zheng, Silvery Fu, Vivian Fang, Samyukta Yagati, Vikram Sreekanti, Xin Wang, Hong Zhang, Kirthevasan Kandasamy, and the entire PS2 group.

Last but certainly not least, I am deeply indebted to and grateful for my family. I thank Lysia, my partner and champion, for her heart, wisdom, and utmost patience with me, all of which continue to surprise me every day. Thank you Mom and Dad, for your unconditional love, understanding, and support in all my years in this world. I would not have been able to do this without my family—I dedicate this dissertation to them.

# Chapter 1

# Introduction

In the past two decades, organizations large and small have been accumulating data at an ever increasing rate. With this growth, systems that store, process, and analyze data have attracted significant attention and investment. Such data systems—including database management systems (DBMS), data analytics systems (often dubbed "big data systems"), among others—provide critical value by answering and serving *queries*, questions that organizations may ask about the data they collected.

The most widely used queries today are SQL queries on tabular data. SQL allows users to express questions about data in a declarative way. The query only specifies the "what" (what target quantities and outputs are desired) and the data system is responsible for figuring out the "how" (the exact sequence of processing to execute the query to return answers). This separation of concerns enables the users to focus on their analytics tasks without having to program the detailed processing steps to retrieve and process data.

However, this decoupling also introduces challenges to the data system—most notably, making queries run fast. For a given SQL query, there are often an enormous number of different ways to execute it, termed *query execution plans.* While all plans for the same query eventually produce the same correct output, their efficiency can differ by orders of magnitude. Therefore, most data systems employ a *query optimizer*, a critical component that is responsible for producing an efficient plan to carry out each query. This process is called *query optimization.*

Figure 1.1 illustrates the principal components of the classical cost-based query optimizer. At a high level, given a query, the optimizer works by *enumerating* many candidate plans and scoring them using an analytical *cost model* that quickly estimates each plan's efficiency. The most accuracy-impacting inputs to the cost model are the *cardinality estimates*, i.e., the estimated numbers of tuples in some subplans satisfying the query's conditions. The cardinality estimator is typically queried many times for different partial plans during the optimization. At the end of the optimization, the "best" query plan for the query—the plan having the lowest estimated cost among all enumerated candidates—is emitted for execution.

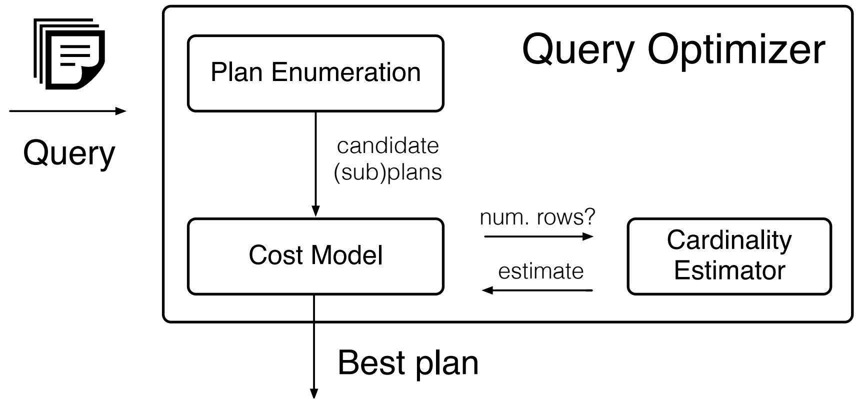


**Figure 1.1:** Principal components of the classical cost-based query optimizer.

## 1.1 Challenges in Query Optimization

Acting as the "brain" of a data system, the query optimizer is highly complex. It must avoid picking poorly performing plans, because they can significantly slow down the query response time. On the other hand, the optimizer must pick out good plans out of a vast search space, which is most likely explored non-exhaustively due to runtime constraints. This high complexity nature has long been acknowledged in the field, with Chaudhuri stating in 1998 that "*building a good optimizer is an enormous undertaking*" [11], and Hellerstein et al. writing in 2007 that "*optimizers are among the most complex components in current-generation commercial DBMSs*" [41].

The issue of high complexity remains in today's optimizers, which leads to the following two key challenges.

First, optimizers use a myriad of heuristics to tame the complexity of the optimization problem, but heuristics leave performance on the table. For example, the cardinality estimator is a component in optimizers that employs particularly many heuristics, resulting in highly inaccurate estimates in today's data systems [64, 65]. These oversimplifying heuristics include data summaries (e.g., 1D histograms), modeling assumptions (e.g., assuming independence between any pair of columns), and beyond. Moreover, effects of inaccurate cardinality estimates *propagate*: inaccurate cardinalities lead to inaccurate cost model scoring (see Figure 1.1), which can ultimately lead the optimizer to output lower-quality plans at best and disastrously slow ones at worst. This problem has attracted significant attention, with some declaring cardinality estimation the *Achilles' heel* of query optimization [70].

Second, as a result of their complexity, optimizers are highly costly to develop. Human experts may spend months writing a first version and years refining it. For example, PostgreSQL, one of the most widely used database systems in the world, has seen a continuous stream of fine-tuning to its optimizer more than 20 years after it was released [17]. Due to the high development costs, some data systems settle for heuristic optimizations and postpone building full-fledged cost-based optimizers—leaving performance on the table. As examples, Spark SQL [115] was introduced in 2014 but only added a cost-based optimizer (CBO) in 2017, while CockroachDB shipped the first version of its CBO in v2.1 after "9 months of intense effort" by a team [46].

Moreover, as data use cases grow in variety and volume, new data systems continue to be invented to address the new demands. These new data systems may go beyond our knowledge of classical query optimization based on relational DBMS (RDBMS). For example, Materialize [76], a streaming database engine based on materialized views, stated that their optimization goal is "off the beaten path" [83], in that it prioritizes minimizing memory usage rather than execution cost (as in classical RDBMS). Another example is dataframe systems [91], which may optimize for a new set of operators designed for dataframes, deviating from the classical model. Therefore, new data systems may have an even higher cost for developing performant optimizers suitable for the new workloads and environments.

## 1.2 Machine Learning for Query Optimization

To address these challenges, this dissertation explores using machine learning to tame the complexity in query optimization.

Machine learning is a promising tool for these challenges in three regards. First, it is a methodology that replaces manually designed heuristics with patterns automatically learned from data. While heuristics are designed for the "average case", they break down when the underlying assumptions do not hold for the data at hand (e.g., two columns are in fact dependent). In contrast, patterns learned from data "fit" that data by construction. Thus, machine learning may enable us to get rid of the most accuracy-impacting heuristics in optimizers, such as the ones causing the high inaccuracy in cardinality estimation.

Second, reinforcement learning, a subset of machine learning, offers the ability to automatically learn to perform a task by trial-and-error. This technique has been successfully applied to solve challenging problems with large search spaces, such as learning to play the game of Go [112, 113], learning to play complex strategy games [131], and learning robotic skills [2]. Therefore, reinforcement learning may open the possibility of automatically learning to optimize queries, potentially alleviating the high development cost of optimizers.

Third, machine learning has been undergoing rapid advancements in the past 10 years, most notably in the form of deep learning. This *new* tool has achieved breakthroughs on problems that could not be solved before, in a wide range of domains such as vision, language, audio and video, games, and beyond. Enabling the successful application of deep learning advances to query optimization is the central theme of this dissertation.

**Roadmap.** In this dissertation, we apply machine learning (ML) advances to:

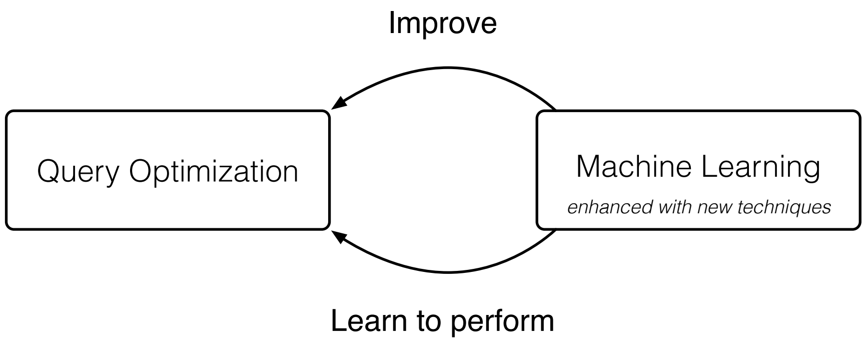


**Figure 1.2:** Using machine learning to both *improve* and *learn to perform* query optimization.

- Remove decades-old heuristics in cardinality estimation for the first time, thereby significantly improving its accuracy (Chapter 3, Chapter 4);
- Automatically learn to optimize queries, thereby alleviating the high development cost of optimizers (Chapter 5).

Figure 1.2 summarizes our overall approach schematically. As we will see, rather than directly applying existing ML methods, we will *enhance* them with new systems and machine learning techniques to realize this approach.

The rest of this dissertation is organized as follows. First, in Chapter 2 we give background information on query optimization and recent machine learning advances. We expand on *why* and *how* ML advances will be leveraged in this dissertation.

In Chapter 3, we present Naru, a learned cardinality estimator for tabular data that removes heuristic modeling assumptions for the first time. Naru uses deep autoregressive models, a new advance in self-supervised learning, to capture the high-dimensional joint data distribution of a table without making any independence assumptions. However, direct application of these models leads to a limited estimator that is prohibitively expensive to evaluate for range or wildcard predicates. To obtain a truly usable estimator, we develop a Monte Carlo integration algorithm on top of autoregressive models, called *progressive sampling*, that efficiently handles range queries touching dozens of columns or more. We augment it further with *wildcard skipping*, an optimization to handle wildcard predicates.

Unlike previous cardinality estimators, Naru approximates the joint data distribution of a table without any independence assumptions. As a result, when evaluated on real-world datasets and compared against real systems (PostgreSQL and a commercial DBMS) and dominant families of cardinality estimators, Naru achieves single-digit multiplicative errors at tail, an up to 90× accuracy improvement over the second best method, and is space- and runtime-efficient.

In Chapter 4, we present NeuroCard, which extends Naru from supporting a single source table to supporting joins involving multiple tables. NeuroCard is a join cardinality estimator that builds a single neural density estimator over an entire database. Leveraging Naru and enhancing it with new techniques (join sampling, column factorization, inference algorithms for joins), NeuroCard shows that it is possible to learn the correlations across all tables in a database without making any inter-table or inter-column independence assumptions. By removing these accuracy-impacting heuristics, NeuroCard achieves orders of magnitude higher accuracy than the best prior methods, scales to more than a dozen tables, while being compact in space (MBs) and efficient to construct or update (seconds to minutes).

Chapter 5 turns the attention to the second key challenge: alleviating the high development cost of optimizers. It introduces Balsa, a deep reinforcement learning agent that automatically learns to optimize SQL queries by trial-and-error. Balsa demonstrates for the first time that learning to optimize queries without learning from an existing expert-designed optimizer is both possible and efficient.

To achieve this arguably surprising result, Balsa tackles the key challenge of mitigating disastrously slow plans during the agent's learning process. It first learns basic knowledge from a simple, engine- and environment-agnostic simulator, followed by learning in real execution, which is guarded by new *safe execution* and *safe exploration* techniques.

On the challenging Join Order Benchmark designed to stress test query optimizers, Balsa matches the performance of two expert optimizers, from PostgreSQL and a commercial engine, with two hours of learning, and outperforms them by up to 2.8× in workload runtime after a few more hours. Balsa thus opens the possibility of automatically learning to optimize in future compute environments and engines where expert-designed optimizers do not exist.

Finally, in Chapter 6 we reflect on lessons learned for related data challenges and future ML-for-systems problems, discuss possible extensions, and conclude this dissertation.

In summary, this dissertation makes the following contributions:

- Significantly improving the accuracy of cardinality estimation for base tables, by directly learning all correlations of the underlying joint data distributions and removing heuristic modeling assumptions (e.g., inter-column independence assumptions).
- Extending this result from base tables to modeling a join schema of multiple tables: all cross-table and cross-column correlations in the schema are learned, without using heuristic inter-table or inter-column independence assumptions.
- Demonstrating that learning to optimize queries by trial-and-error, without learning from an expert-designed optimizer, is not only possible and efficient, but can also result in a highly competitive learned optimizer that can outperform mature optimizers.

**Previously published materials.** This dissertation includes previously published and co-authored work as follows. Chapter 3 includes materials from [142]. Chapter 4 includes materials from [143]. Chapter 5 includes materials from [141].

# Chapter 2

# Background

In this chapter, we first give some background information on query optimization, then proceed to review machine learning advances and discuss *why* and *how* they will be leveraged in this dissertation.

## 2.1 Query Optimization

Query optimization is the process of translating high-level, declarative queries into low-level, efficient execution plans, much like a compiler. Each execution plan is a tree of physical operators—algorithms that the system actually implements to carry out the computation. Query optimization has a long history of work in the research literature as well as in practice, by having a presence in most, if not all, working data systems, dating back to at least the pioneering System R project in the 1970s [107].

Next, we start by reviewing two influential optimizer architectures, both of which are implemented in many of today's systems: the bottom-up System R-style framework, and the top-down Cascades-style framework. We then put the contributions of this dissertation in context.

**System R.** In this framework, queries are optimized *bottom-up*: it starts with finding the best ways to access base tables, then gradually builds up larger plans that include more tables. It instantiates the architecture in Figure 1.1 by using a *dynamic programming* algorithm as the plan enumeration module. We illustrate the key idea of join ordering and plan construction next, and omit many technical details (e.g., handling subqueries, interesting orders) which can be found in its classical paper [107].

Given a query with $n$ tables to join, a System R optimizer starts by enumerating all scan operators to access each of these tables, each of which has its cost estimated and stored in a memoization table. This step yields the best 1-table subplans (i.e., partial plans). It then proceeds to construct the best 2-table subplans from the smaller, already-optimized subplans: a candidate is formed by picking two 1-table subplans from the memoization table and joining them in some order with a physical join operator (e.g., hash join, sort-merge join). These 2-table candidates are scored by the cost model, and the cheapest plan per table combination is written to the memoization table. The algorithm proceeds similarly for all $k$-table joins, at each "level" composing the smaller optimized subplans from prior levels, eventually producing a final cheapest plan.

The System R optimizer is considered textbook material and is implemented in a wide range of database systems today (e.g., PostgreSQL).

**Cascades.** The Cascades framework [32] came from a line of work on *extensible optimizers* and *optimizer generators*, and it addressed several challenges observed in the prior work Exodus [33] and Volcano [34]. For our purpose, we describe its *top-down* approach of producing an execution plan, to contrast with System R. This framework can be seen as instantiating Figure 1.1 with a rule-based search engine as the enumeration module.

At a high level, a Cascades optimizer starts from a logical plan of the query and repeatedly invokes *rules* to transform the current plan to get closer to a desired final plan. These rules can represent logical-to-logical rewrites (e.g., two relations in a logical join can swap their order) or logical-to-physical rewrites (replacing logical operators with physical operators). Like System R, a cost model and a cardinality estimator are used to quickly estimate the efficiency of the considered plans. At the end of the optimization, the cheapest plan found thus far is emitted.

This framework has the advantage of being more extensible, as new rules can be easily added into the framework in a modular manner. Many ideas developed in the Cascades framework have appeared in widely used commercial systems, including Microsoft SQL Server, SCOPE [146], PDW [109], and Greenplum's Orca [114].

Despite their apparent differences, both influential optimizer architectures described above have enjoyed wide usage in practice and have no clear winner.

**Our work in perspective.** Importantly, both frameworks rely on the core primitives of cost modeling and cardinality estimation for good optimization performance. In this dissertation, Chapter 3 and Chapter 4 focus on removing decades-old, accuracy-impacting heuristics from base-table and join cardinality estimation, respectively. Therefore, our results in those chapters are broadly applicable to a range of optimizers, independent of which framework they are implemented in.

Chapter 5 presents a deep reinforcement learning approach to learn to optimize queries. The learned optimizer described in that chapter (Balsa) implements a bottom-up plan search procedure (as the enumeration module) that is guided by a learned value network (as the cost model module). Thus, Balsa can be seen as replacing the plan enumeration and cost model components shown in Figure 1.1. It is closer in spirit to the System R framework as it does not use transformation rules as the key mechanism to produce plans.

As mentioned, query optimization has a rich history of work. This section cannot and does not attempt to cover the topic in depth. Rather, we present the necessary background information to set up the context for the chapters that follow. For more details, we refer the reader to the surveys by Jarke and Koch [49] and Chaudhuri [11]; for more perspectives, see Hellerstein et al. [41] and the "Red Book" [5].

## 2.2 Machine Learning Advances

Next, we discuss the recent machine learning (ML) advances that serve as the enabling techniques in this work. We first briefly review some historical context of deep learning, then articulate *why* and *how* advances in self-supervised learning and reinforcement learning will be leveraged in the chapters to come in this dissertation.

**Deep learning (DL).** DL, a family of ML techniques based on the use of deep neural networks as function approximators, has become the de facto method of choice in a wide range of ML problems in the last decade. While its core techniques (e.g., neural networks trained by the backpropagation algorithm) were proposed decades ago [102, 62, 101], it was only until around 2011 to 2012 where the confluence of fast accelerator hardware (GPUs), improved software, and the availability of large datasets that made deep learning truly feasible for the first time. This started the so-called "AI boom" or the "deep learning revolution" of the 21st century [106].

The defining transition point was, by general consensus, the "AlexNet moment" in 2012, where a network architecture called AlexNet [61] won the large-scale ImageNet image recognition contest by a large margin, compared to other non-neural, classical methods. The AlexNet moment convincingly demonstrated both the feasibility and the superior performance of deep learning, thus kicking off a swift shift to DL methods in the computer vision community in particular, and the broader ML community in general.

Since then, DL has attracted significant attention, and the capabilities of DL have rapidly improved. These include an expanded list of supported data modalities (from image pixels to text, audio, video, and the combinations thereof), improved neural network architectures (from multilayer perceptrons, to convolutional neural networks, sequence models, Transformer, and numerous variants), better training algorithms (e.g., Adam [52]), hardware and software improvements, and so on. As a result, applications of DL have multiplied. Today, deep learning has been applied to many areas in sciences and powers many widely used applications and web services.

DL has since pushed the boundary of nearly every area in ML. We discuss two particularly active areas that we will leverage: self-supervised learning and reinforcement learning.

**Self-supervised learning.** A subset of unsupervised learning, self-supervised learning refers to a family of methods that learn to predict unseen parts of the input data given the observed parts. In other words, the data itself provides the supervision signals. This is in direct contrast with the more familiar regime of supervised learning, where supervision labels need to be explicitly collected (e.g., by human labelers).

A major direction in self-supervised learning is *generative modeling*, which has been rapidly progressing thanks to the combination of new algorithms and neural networks as powerful function approximators. Notably, while it was previously thought intractable to learn a joint data distribution of high-dimensional data in its full form, *deep autoregressive models*, a new type of density estimator, have succeeded in modeling high-dimensional data such as images [126, 103, 14], audio [127], and text [97, 128, 8, 23, 140] (termed *language models*). Other new generative models include flow models [53, 45, 88, 47, 24], latent variable models [54, 55, 86, 98], and implicit models such as Generative Adversarial Networks [31].

As mentioned, deep autoregressive models for the first time enable us to learn complex high-dimensional joint data distributions without independence assumptions, achieving state-of-the-art accuracy. In Chapter 3 and Chapter 4, we will apply them to first learn the joint data distributions of tabular data without any modeling assumptions, then, crucially, enhance the models with new techniques to turn them into feasible cardinality estimators.

**Reinforcement learning (RL).** RL studies automatically learning to perform a task by trial-and-error. At a high level, RL consists of an *agent* that learns to solve a task by repeatedly interacting with an *environment*. The agent observes the environment's *state* and takes an *action* to maximize a *reward*. Through rewards, the agent is incentivized to reinforce good actions and correct mistakes.

Deep RL, in particular, utilizes neural networks to learn powerful policy and value functions—key components of a typical RL agent—that allow the agent to learn sophisticated policies (behaviors) to solve increasingly complex tasks. In the last few years, deep RL has remarkably solved several previously unsolvable challenges. Notable milestones include learning to play Atari games [78], the AlphaGo project and its successors [112, 113, 111, 104] that learn to play Go to a superhuman level (defeating human champions), learning to play complex multi-player strategy games (StarCraft II, Dota 2) to a superhuman level [131, 7], learning robotic skills [3, 2], to name a few. Importantly, these milestones have contributed many learning algorithms that have been applied to other use cases.

In Chapter 5, we will apply deep RL to learn to optimize SQL queries by trial-and-error. We will leverage value network-guided tree search, an algorithm similar to AlphaGo and expert iteration [4], to allow the agent to navigate a large search space to find high-quality plans. To make it feasible in our setting, we will enhance it with new techniques to resolve the key challenge of mitigating disastrously slow plans in the learning process.

# Chapter 3

# Naru: Deep Unsupervised Cardinality Estimation

As discussed in Chapter 1, cardinality estimation is a component in query optimization where heuristics particularly abound. While heuristic modeling assumptions, e.g., the column independence assumption and the uniformity assumption in histograms, simplify the estimation problem, they leave accuracy on the table and thus hurt the quality of query optimization. As a result, database systems today produce highly inaccurate cardinality estimates, which are found to be the dominant factor for poor execution plans [64, 65].

In this chapter, we build a new cardinality estimator called *Naru*, that learns a table's data distribution while fully removing heuristic modeling assumptions for the first time, by applying and *enhancing* a new statistical model from recent advances in self-supervised learning. Like classical synopses, Naru directly summarizes the data and then uses the summary to estimate the cardinalities of incoming queries or predicates. Unlike previous estimators, Naru approximates the joint data distribution of a table without any independence assumptions, thereby achieving a new level of accuracy in base table cardinality estimation.

## 3.1 Introduction

Cardinality estimation is a core primitive in query optimization [107]. One of its main tasks is to accurately estimate the *selectivity* of a SQL predicate—the fraction of a relation selected by the predicate—without actual execution. Despite its importance, there is wide agreement that the problem is still unsolved [64, 65, 90]. Open-source and commercial DBMSes routinely produce up to $10^4$–$10^8\times$ estimation errors on queries over a large number of attributes [64].

The fundamental difficulty of selectivity estimation[1] comes from condensing information

[1]We use this term and "cardinality estimation" interchangeably, as multiplying a selectivity by the table's row count recovers the cardinality.

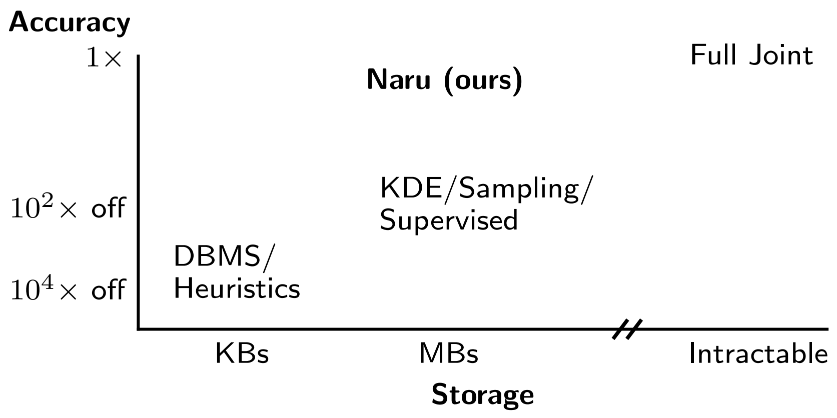


**Figure 3.1:** Approximating the joint data distribution in full, Naru enjoys high estimation accuracy and space efficiency.

about data into summaries [50]. The predominant approach in database systems today is to collect single-column summaries (e.g., histograms and sketches), 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 *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 (Figure 3.1). Recent advances in deep unsupervised learning have offered promising tools in this regard. While it was previously thought intractable to approximate the data distribution of a relation in its full form [28, 22], *deep autoregressive models*, a type of density estimator, have succeeded in modeling high-dimensional data such as images, text, and audio [126, 103, 127, 128]. However, these models only estimate *point densities*—in query processing terms, they only handle equality predicates (e.g., "what is the fraction of tuples with price equal to $100?"). Full-featured selectivity estimation requires handling not only equality but also range predicates (e.g., "what fraction of tuples have price less than $100 and weight greater than 10 lbs?"). Naive estimation of the range density by integrating over the query region requires summing up an enormous number of points. In an 11-dimensional table we consider, a challenging range query has $10^{10}$ points in the query region, which would take more than 1,000 hours to sum over by a naive enumeration scheme. A full-featured selectivity estimator, therefore, requires new techniques beyond the state of the art.

In this chapter, we show that selectivity estimation can be performed with high accuracy by using deep autoregressive models. We first show how relational data—including both numeric and categorical attributes—can be mapped onto these models for effective selectivity estimation of equality predicates. We then introduce a new Monte Carlo integration technique called *progressive sampling*, which efficiently estimates range queries even at high dimensionality. By leveraging the availability of conditional probability distributions provided by the model, progressive sampling steers the sampler into regions of high probability density, and then corrects for the induced bias by using importance weighting. This technique extends the state of the art in density estimation, with particular applicability to our problem of general-purpose selectivity estimation. Our scheme is effective: a thousand samples suffice to accurately estimate the aforementioned $10^{10}$-point query.

To realize these ideas, we design and implement Naru (Neural Relation Understanding), a selectivity estimator that approximates the joint data distribution in its full form, without any column independence assumptions. Approximating the joint in full not only provides superior accuracy, but also frees us from specifying what combinations of columns to build synopses on. We further propose optimizations to efficiently handle wildcard predicates, and to encode and decode real-world relational data (e.g., supporting various datatypes, small and large domain sizes). Combining our integration scheme with these practical strategies results in a highly accurate, compact, and functionality-rich selectivity estimator based on deep autoregressive models.

Just like classical synopses, Naru summarizes a relation in an unsupervised fashion. The model is trained via statistically grounded principles (maximum likelihood) where no supervised signals or query feedback are required. While *query-driven* estimators are optimized with respect to a set of training queries (i.e., "how much error does the estimator incur on these queries?"), Naru is optimized with respect to the underlying data distribution (i.e., "how divergent is the estimator from the data?"). Being data-driven, Naru supports a much larger set of queries and is automatically robust to query distributional shifts. Our evaluation compares Naru to the state-of-the-art unsupervised and supervised techniques, showing Naru to be the only estimator to achieve worst-case *single-digit multiplicative errors* for challenging high-dimensional queries.

In summary, we make the following contributions:

1. We show deep autoregressive models can be used for selectivity estimation (§3.2, §3.3), and propose optimizations to make them suitable for relational data (§3.4).

2. To handle challenging range queries, we develop *progressive sampling*, a Monte Carlo integration algorithm that efficiently estimates range densities even with large query regions (§3.5.1). We augment it with a novel optimization, *wildcard-skipping* (§3.5.2), to handle wildcard predicates. We also propose information-theoretic column orderings (§3.5.3) to reduce estimation variance.

3. We extensively evaluate on real datasets against 8 baselines across 5 different families (heuristics, real DBMSes, sampling, statistical methods, deep supervised regression). Our estimator Naru achieves up to orders-of-magnitude better accuracy with space usage $\sim$1% of data size and $\sim$5−10ms of estimation latency (§3.6).

Naru is open sourced at `https://github.com/naru-project/naru`.

## 3.2 Problem Formulation

Consider a relation $T$ with attribute domains $\{A_1, \ldots, A_n\}$. Selectivity estimation seeks to estimate the fraction of tuples in $T$ that satisfy a particular predicate, $\theta : A_1 \times \cdots \times A_n \to \{0, 1\}$. We define the selectivity to be $\mathsf{sel}(\theta) := |\{\boldsymbol{x} \in T : \theta(\boldsymbol{x}) = 1\}|/|T|$.

The *joint data distribution* of the relation, defined to be

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

is closely related to the selectivity, where $f(a_1, \ldots, a_n)$ is the number of occurrences of tuple $(a_1, \ldots, a_n)$ in $T$. It forms a valid probability distribution since integrating it over the attribute domains yields a value of 1. Thus, exact selectivity calculation is equivalent to integration over the joint:

$$\mathsf{sel}(\theta) = \sum_{a_1 \in A_1} \cdots \sum_{a_n \in A_n} \theta(a_1, \ldots, a_n) \cdot P(a_1, \ldots, a_n).$$

In this chapter, we consider finite relation $T$ and hence its empirical domains $A_i$ are finite. Therefore summation is used in the integration calculation above.

### 3.2.1 Approximating the Joint via Factorization

Given the joint, exact selectivity "estimates" can be calculated by integration. However, the number of entries in the joint—and thus the maximum number of points needed to be summed over in the integration—is $|P| = \prod_{i=1}^{n} |A_i|$, a size that grows exponentially in the number of attributes. Real-world tables with a dozen or so columns can easily have a theoretic joint size of $10^{20}$ and upwards (§3.6). In practice, it is possible to bound this number by $|T|$, the number of tuples in the relation, by not storing any entry with zero occurrence. Algorithmically, to scale construction, storage, and integration to high-dimensional tables, joint approximation techniques seek to *factorize* [35] the joint into some lower-dimensional representation, $\widehat{P} \approx P$.

Classical 1D histograms [107] use the simplest factorization, $\widehat{P}(A_1, \cdots, A_n) \approx \prod_{i=1}^{n} \widehat{P}(A_i)$, where independence between attributes is assumed. The $\widehat{P}(A_i)$'s are materialized as histograms that are cheap to construct and store. Selectivity estimation reduces to calculating per-column selectivities and combining by multiplication,

$$\mathsf{sel}(\theta) \approx \left(\sum_{a_1 \in A_1} \theta_1(a_1)\widehat{P}(a_1)\right) \times \cdots \times \left(\sum_{a_n \in A_n} \theta_n(a_n)\widehat{P}(a_n)\right)$$

where each $\theta_i$ is predicate $\theta$ projected to each attribute (assuming here $\theta$ is a conjunction of single-attribute filters).

Richer factorizations are possible and are generally more accurate. For instance, Probabilistic Relational Models [28, 29] 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 [22] 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.

In this chapter, we consider the richest possible factorization of the joint, using the product rule:

$$\widehat{P}(A_1, \cdots, A_n) = \widehat{P}(A_1)\widehat{P}(A_2|A_1)\cdots\widehat{P}(A_n|A_1, \ldots, A_{n-1})$$

Unlike the previous proposals, the product rule factorization is an *exact* relationship to represent a distribution. It makes no independence assumptions and captures all complex interactions between attributes. Key to this goal is that the factors, $\{\widehat{P}(A_i|A_1, \ldots, A_{i-1})\}$, need not be materialized; instead, they are calculated on-demand by a neural network, a high-capacity universal function approximator [38].

## 3.2.2 Problem Statement

We estimate the selectivities of queries of the following form. A query is a conjunction of single-column boolean predicates, over arbitrary subsets of columns. 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 [22, 28, 40, 95, 89] and, in some cases, offers more capabilities. Certain prior work requires each predicate be a rectangle [56, 40] or columns be real-valued [58, 40]; 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.

## 3.3 Deep Autoregressive Models

### 3.3.1 Overview

Naru 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 $\widehat{P}(\boldsymbol{x})$.** Deep autoregressive models produce point density estimates $\widehat{P}(\boldsymbol{x})$ after training on a set of n-dimensional tuples $T = \{\boldsymbol{x}_1, \dots\}$ with the unsupervised maximum likelihood objective. Many network architectures have been proposed in recent years, such as masked multi-layer perceptrons (e.g., MADE [27], ResMADE [25]) or masked self-attention networks (e.g., Transformer [128]).

**Access to conditional densities $\{\widehat{P}(x_i|\boldsymbol{x}_{<i})\}$.** Additionally, autoregressive models also provide access to all conditional densities present in the product rule:

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

Namely, given input tuple $\boldsymbol{x} = (x_1, \cdots, x_n)$, one can obtain from the model the $n$ conditional density estimates, $\{\widehat{P}(x_i|\boldsymbol{x}_{<i})\}$. 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 (§3.5.3 discusses heuristically picking a good ordering).

Naru chooses autoregressive models for selectivity estimation for two important reasons. First, autoregressive models have shown superior modeling precision in learning images [126, 103], audio [127], and text [128]. All these domains involve correlated, high-dimensional data akin to a relational table. Second, as we will show in §3.5.1, access to conditional densities is critical in efficiently supporting range queries.

### 3.3.2 Autoregressive Models for Relational Data

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

$$\mathcal{M}(\boldsymbol{x}) \mapsto \left[\widehat{P}(X_1), \widehat{P}(X_2|x_1), \cdots, \widehat{P}(X_n|x_1, \dots, x_{n-1})\right] \quad (3.1)$$

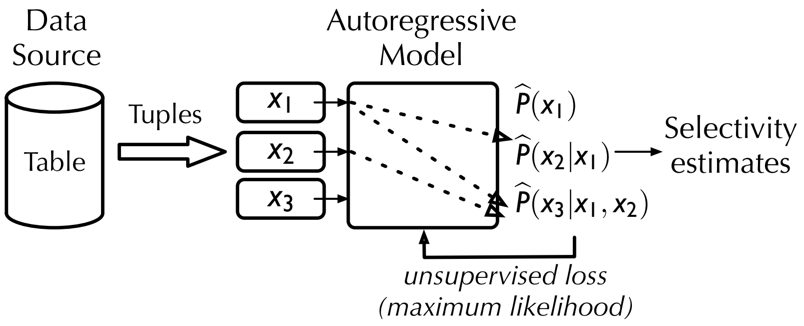


**Figure 3.2:** Overview of the estimator framework. Naru is trained by reading data tuples and does not require supervised training queries or query feedback, just like classical synopses.

Namely, one tuple goes in, a list of conditional density distributions comes out, each being a *distribution* of the $i$th attribute conditioned on previous attributes. (The *scalars* required to compute the point density, $\{\widehat{P}(x_i|\boldsymbol{x}_{<i})\}$, 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 [27, 128, 126]; 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 $\boldsymbol{x}_{<i}$. Its role is to use this context information to output a distribution over its own domain, $\widehat{P}(X_i|\boldsymbol{x}_{<i})$. Consider a `travel_checkins` table with columns `city`, `year`, `stars`. Assume the model is given the input tuple, $\langle \texttt{Portland}, \texttt{2017}, \texttt{10}\rangle$. 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 $\mathcal{M}_{\texttt{col}}$:

$$\mathbf{0} \to \mathcal{M}_{\texttt{city}}$$
$$E_{\texttt{city}}(\texttt{Portland}) \to \mathcal{M}_{\texttt{year}}$$
$$\oplus\,(E_{\texttt{city}}(\texttt{Portland}), E_{\texttt{year}}(\texttt{2017})) \to \mathcal{M}_{\texttt{stars}}$$

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 [128].

Notice that the first output, from $\mathcal{M}_{\texttt{city}}$, does not depend on any attribute values (its input $\mathbf{0}$ 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* [38] between the data distribution $P$ and the model estimate $\widehat{P}$ is calculated over all tuples in relation $T$ and used as the loss:

$$\mathcal{H}(P, \widehat{P}) = -\sum_{\boldsymbol{x} \in T} P(\boldsymbol{x}) \log \widehat{P}(\boldsymbol{x}) = -\frac{1}{|T|} \sum_{\boldsymbol{x} \in T} \log \widehat{P}(\boldsymbol{x}) \tag{3.2}$$

It can be fed into a standard gradient descent optimizer [52]. 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.

## 3.4 Estimator Construction

We now discuss practical issues in constructing Naru.

### 3.4.1 Workflow

Figure 3.2 outlines the workflow of building a Naru estimator. After specifying a table $T$ to build an estimator on, batches of random tuples from $T$ are read to train Naru. In practice, a snapshot of the table can be saved to external storage so normal DBMS activities are not affected. Neural network training can be performed either close to the data (at periods of low activity) or offloaded to a remote process.

For a batch of tuples, Naru encodes each attribute value using column-specific strategies (§3.4.2). The encoded batch then gets fed into the model to perform a gradient update step. Our evaluation (§3.6.4) empirically observed that one pass over data is sufficient to achieve a high degree of accuracy (e.g., outperforming real DBMSes by $10-20\times$), and more passes are beneficial until model convergence.

Appends and updates may cause statistical staleness. Naru can be fine-tuned on the updated relation to correct for this, as we show in §3.6.8.3. Further, if new data comes in per-day partitions, then each partition can train its own Naru model.

**Joins.** The estimator does not distinguish between the type of table it is built on. To build an estimator on a joined relation, either the entire joined relation can be pre-computed and materialized, or multi-way join operators [129, 130] and samplers [63, 12] can be used to produce batches of tuples on-the-fly. Given access to tuples from the joined result, no changes are needed to the estimator framework. Once trained, the estimator supports queries that filter any column in the joined relation. This treatment follows prior work [74, 89, 51] and is conceptually clean.

In Chapter 4 we will study this in depth, by extending Naru to efficiently support joins.

### 3.4.2 Encoding and Decoding Strategies

Naru 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 $\widehat{P}(X_i|\boldsymbol{x}_{<i})$ (a vector of scores) must be efficiently *decoded* regardless of its datatype or domain size.

For each column Naru 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. Naru 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 (Naru 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* $\widehat{P}(X_i|\boldsymbol{x}_{<i})$, 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.

Naru 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 $\boldsymbol{x}_{<i}$ and output $|A_i|$ probability scores over the domain. With the shrunk-down output layer, inputs $\boldsymbol{x}_{<i}$ 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 [97].

### 3.4.3 Model Choice

As discussed, any autoregressive model can be plugged in, taking advantage of Naru's encoding/decoding optimizations as well as querying capabilities (§3.5). We experiment with three representative architectures: (A) Masked Autoencoder (MADE) [27], a standard multi-layer perceptron with information masking to ensure autoregressiveness; (B) ResMADE [25], a simple extension to MADE where residual connections are introduced to improve learning efficiency; and (C) Transformer [128], a class of self-attentional models driving recent state-of-the-art advances in natural language processing [23, 140]. Table 3.7 compares the tradeoffs of these building blocks. We found that, under similar parameter count, more advanced architectures (B, C) achieve better entropy gaps; however, the smaller entropy gaps do not automatically translate into better selectivity estimates and the computational cost can be significantly higher (for C).

## 3.5 Querying the Estimator: Progressive Sampling

Once an autoregressive model is trained, it can be queried to compute selectivity estimates. Assume a query $\mathsf{sel}(\theta) = P(X_1 \in R_1, \ldots, X_n \in R_n)$ asking for the selectivity of the conjunction, where each range $R_i$ can be a point (equality predicate), an interval (range predicate), or any subset of the domain ($\mathsf{IN}$). 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 Naru 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 Naru carries out this operation. *Enumeration* exactly sums up the densities when the queried region $R$ is sufficiently small:

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

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 $\leq 2017$" with equivalent semantics. For equality predicates with out-of-domain literals, Naru simply returns a cardinality of 0. Hereafter we consider in-domain literals and valid regions.

## 3.5.1 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. Naru proposes an approximate integration scheme to address this challenge.

**First attempt** (Figure 3.3, left). The simplest way to approximate the sum is via uniform sampling. First, sample $\boldsymbol{x}^{(i)}$ uniformly at random from $R$. Then, query the model to compute $\widehat{p}_i = \widehat{P}(\boldsymbol{x}^{(i)})$. 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.

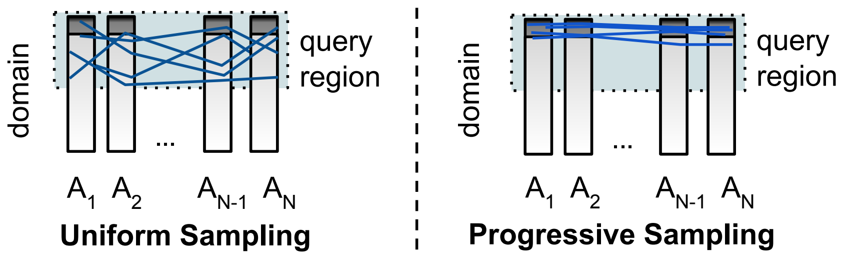


**Figure 3.3:** The intuition of progressive sampling. Uniform samples taken from the query region have a low probability of hitting the high-mass sub-region of the query region, increasing the variance of Monte Carlo estimates. Progressive sampling avoids this by sampling from the estimated data distribution instead, which naturally concentrates samples in the high-mass sub-region.

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 3.3). 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 3.3, 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 $\mathbf{0}$ to get $\widehat{P}(X_1)$. Compute and store $\widehat{P}(X_1 \in R_1)$ by summing. Then draw $x_1^{(i)} \sim \widehat{P}(X_1|X_1 \in R_1)$.

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

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

6:  function Draw($R_1, \ldots, R_n$)                            ▷ Draw one tuple
7:      $\widehat{p} = 1$
8:      $\boldsymbol{s} = \boldsymbol{0}_n$                      ▷ The tuple to fill in
9:      for $i = 1$ to $n$ do
10:         Forward pass through model: $\mathcal{M}(\boldsymbol{s})$
11:         $\widehat{P}(X_i|\boldsymbol{s}_{<i})$ = the $i$-th model output    ▷ Eq. 3.1
12:         Zero-out probabilities in slots $[0, D_i) \setminus R_i$
13:         Re-normalize, obtaining $\widehat{P}(X_i|X_i \in R_i, \boldsymbol{s}_{<i})$
14:         $\widehat{p} = \widehat{p} \times \widehat{P}(X_i \in R_i|\boldsymbol{s}_{<i})$
15:         Sample $s_i \sim \widehat{P}(X_i|X_i \in R_i, \boldsymbol{s}_{<i})$
16:         $\boldsymbol{s}[i] = s_i$
17:     return $\widehat{p}$                                     ▷ Density of the sampled tuple $s$
```

2. Forward $x_1^{(i)}$ to get $\widehat{P}(X_2|x_1^{(i)})$. Compute and store $\widehat{P}(X_2 \in R_2|x_1^{(i)})$. 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 $\widehat{P}(X_3 \in R_3|x_1^{(i)}, x_2^{(i)})$.

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,

$$\widehat{P}(X_1 \in R_1) \cdot \widehat{P}(X_2 \in R_2|x_1^{(i)}) \cdot \widehat{P}(X_3 \in R_3|x_1^{(i)}, x_2^{(i)}) \tag{3.3}$$

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:

**Theorem 1** *Progressive Sampling estimates are unbiased.*

The proof only uses basic probability rules and is deferred to our online technical report. 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 §3.5.2. 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[57] to allow variables taking on *ranges of values* (the former, in its default form, allows equality predicates only).

### 3.5.2 Reducing Variance: Wildcard-Skipping

Naru introduces *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, $\mathsf{MASK}_i$ 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 (§3.6.6).

During training, we perturb each tuple so that the training data contains $\mathsf{MASK}$ 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.

### 3.5.3 Reducing Variance: Column Ordering

Naru models adopt a single ordering of columns during construction (§3.4). However, different orderings may have different sampling efficiency. For instance, having `city` as the first column and setting it to `Waikiki` focuses on records only relevant to that city, a data region supposedly much narrower than that from having `year` as the first column.

Empirically, we find that these heuristic orders work well:

1. MutInfo: successively pick column $X_i$ that maximizes the *mutual information* [19] between all columns chosen so far and itself, $\arg\max_i I(X_{\text{chosen}}; X_i)$.

2. PMutInfo: a variant of the above that maximizes the *pairwise* mutual information, $\arg\max_i I(X_{\text{last}}; X_i)$.

Intuitively, maximizing $I(X_{\text{chosen}}; X_i)$ corresponds to finding the next column with *the most information already contained in the chosen columns*. For both schemes, we find that picking the column with the maximum marginal entropy, $\arg\max_i H(X_i)$, as the first works well. Interestingly, on our datasets, the Natural ordering (left-to-right order in table schema) is also effective. We hypothesize this is due to human bias in placing important or "key"-like columns earlier that highly reduce the uncertainty of other columns.

Lastly, we note that *order-agnostic training* has been proposed in the ML literature [27, 125, 140]. The idea is to train the same model on more than one order, and at inference time invoke a (presumably seen) order most suitable for the query. This is a possible future optimization for Naru, though in the preliminary experiments we did not find the performance benefits on top of our optimizations significant.

## 3.6 Evaluation

We answer the following questions in our evaluation:

1. How does Naru compare to state-of-the-art selectivity estimators in accuracy (§3.6.2)? Is it robust (§3.6.3)?
2. How long does it take to train a Naru model to achieve a useful level of accuracy (§3.6.4)?
3. Naru requires multiple inference passes to produce a selectivity estimate. How does this compare with the latency of other approaches (§3.6.5)?
4. How do wildcard-skipping and column orderings affect accuracy and variance (§3.6.6)? How does accuracy change with model choices and sizes (§3.6.7)?

Lastly, a series of microbenchmarks are run to understand Naru's limits (§3.6.8).

### 3.6.1 Experimental Setup

We compare Naru against predominant families of selectivity estimation techniques, including estimators in real databases, heuristics, non-parametric density estimators, and supervised learning approaches (Table 3.2). To ensure a fair comparison between estimators, we restrict each estimator to a fixed *storage budget* (Table 3.1). For example, for the Conviva-A dataset, Naru's model must be less than 3MB in size, and the same restriction is held for all estimators for that dataset when applicable.

#### 3.6.1.1 Datasets

We use real-world datasets with challenging characteristics (Table 3.1). The number of rows ranges from 10K to 11.6M, the number of columns ranges from 11 to 100, and the size of the joint space ranges from $10^{15}$ to $10^{190}$:

**Table 3.1:** List of datasets used in evaluation. "Dom." refers to per-column domain size. "Joint" is number of entries in the exact joint distribution (equal to the product of all domain sizes). "Budget" is the storage budget we allocated to all evaluated estimators, when applicable, relative to the in-memory size of the corresponding original tables.

| Dataset | Rows | Cols | Dom. | Joint | Budget |
|---|---|---|---|---|---|
| DMV | 11.6M | 11 | 2–2K | $10^{15}$ | 1.3% (13MB) |
| Conviva-A | 4.1M | 15 | 2–1.9K | $10^{23}$ | 0.7% (3MB) |
| Conviva-B | 10K | 100 | 2–10K | $10^{190}$ | N/A |

**DMV [117].** Real-world dataset consisting of vehicle registration information in New York. We use the following 11 columns with widely differing data types and domain sizes (the numbers in parentheses): record_type (4), reg_class (75), state (89), county (63), body_type (59), fuel_type (9), valid_date (2101), color (225), sco_ind (2), sus_ind (2), rev_ind (2). Our snapshot contains 11,591,877 tuples. The exact joint distribution has a size of $3.4 \times 10^{15}$.

**Conviva-A.** Enterprise dataset containing anonymized user activity logs from a video analytics company. The table corresponds to 3 days of activities. The 15 columns contain a mix of small-domain categoricals (e.g., error flags, connection types) as well as large-domain numerical quantities (e.g., various bandwidth numbers in kbps). Although the domains have a range (2–1.9K) similar to DMV, there are many more numerical columns with larger domains, resulting in a much larger joint distribution ($10^{23}$).

**Conviva-B.** A small dataset of 10K rows and 100 columns also from Conviva, with a joint space of over $10^{190}$. Though this dataset is trivial in size, this enables the use of an emulated, perfect-accuracy model for running detailed robustness studies (§3.6.8).

#### 3.6.1.2 Estimators

We next discuss the baselines listed in Table 3.2.

**Real databases.** PostgreSQL and DBMS-1 represent the performance a practitioner can hope to obtain from a real DBMS. Both rely on classical assumptions and 1D histograms, while the latter additionally contains cross-column correlation statistics. Every column has a histogram and associated statistics built. PostgreSQL is tuned to use a maximum amount of per-column bins (10,000). For DBMS-1, one invocation of stats creation with all columns specified only builds a histogram on the first column; we therefore invoke stats creation several times so that all columns are covered.

**Independence assumption.** Indep scans each column to obtain perfect per-column selectivities and combines them by multiplication. This measures the inaccuracy solely

**Table 3.2:** List of estimators used in evaluation.

| Type | Estimator | Description |
|---|---|---|
| Heuristic | Indep | A baseline that multiplies *perfect* per-column selectivities. |
| Real System | PostgreSQL | 1D stats and histograms via independence/uniformity assumptions. |
| Real System | DBMS-1 | Commercial DBMS: 1D stats plus inter-column unique value counts. |
| Sampling | Sample | Keeps $p$% of all tuples in memory. Estimates a new query by evaluating on those samples. |
| MHIST | MHIST | The MaxDiff(V, A) histogram [96]. |
| Graphical | BayesNet | Bayes net (Chow-Liu tree [15]). |
| KDE | KDE | Kernel density estimation [40, 51]. |
| Supervised | MSCN | Supervised deep regression net [56]. |
| Deep AR | Naru | (Ours) Deep autoregressive models. |

attributed to the independence assumption.

**Multi-dimensional histogram.** We compare to MHIST, an N-dimensional histogram. We use *MaxDiff* as our partition constraint, *Value* (V) as the sort parameter, and *Area* (A) as the source parameter [96]. We use the MHIST-2 algorithm [95] and the *uniform spread* assumption [96] to approximate the value set within each partition. According to [95], the resulting MaxDiff(V, A) histogram offers the most accurate and robust performance compared to other state-of-the-art histogram variants.

**Bayesian Network.** We use a Chow-Liu tree [15] as the Bayesian Network, since empirically it gave the best results for the allowed space. Since the size of conditional probability tables scales with the cube of column cardinalities, we use equal-frequency discretization (to 100 bins per column) to bound the space consumption and inference cost. Lastly, we apply the same progressive sampler to allow this estimator to support range queries (it does not support range queries out of the box); this ensures a fair comparison between the use of deep autoregressive models and this approach.

**Kernel density estimators & Sampling.** In the non-parametric sampling regime, we evaluate a uniform sampler and a state-of-the-art KDE-based selectivity estimator [40, 51]. Sample keeps a set of $p$% of tuples uniformly at random from the original table. In accordance with our memory budget for each dataset, $p$ is set to 1.3% for DMV and 0.7% for Conviva-A. KDE [40] attempts to learn the underlying data distribution by averaging Gaussian kernels centered around random sample points. The number of sample points is chosen in accordance with our memory budget: 150K samples for DMV and 28K samples for

Conviva-A. The bandwidth for KDE is computed via Scott's rule [105]. The bandwidth for KDE-superv is initialized in the same way, but is further optimized through query feedback from 10K training queries. We modify the source code released by the authors [71] in order to run it with more than ten columns.

**Supervised learning.** We compare to a recently proposed supervised deep net-based estimator termed *multi-set convolutional network* [56], or MSCN. We apply the source code from the authors [80] to our datasets. As it is a *supervised* method, we generate 100K training queries from the same distribution the test queries are drawn, ensuring their "representativeness". The net stores a materialized sample of the data. Every query is run on the sample to get a bitmap of qualifying tuples—this is used as an input additional to query features. We try three variants of the model, all with the same hyperparameters and all trained to convergence: MSCN-base uses the same setup reported originally [56] (1K samples, 100K training queries) and consumes 3MB. We found that MSCN's performance is highly dependent on the samples, so we include a variant with 10× more samples (MSCN-10K: 10K samples, 100K train queries), consuming 13MB (satisfying DMV's budget only). We also run MSCN-0 that stores no samples and uses query features only.

**Deep unsupervised learning (ours).** We train one Naru model for each dataset. All models are trained with the unsupervised maximum likelihood objective. Unless stated otherwise, we employ wildcard-skipping and the natural column ordering. Sizes are reported without any compression of network weights:

- DMV: masked autoencoder (MADE), 5 hidden layers (512, 256, 512, 128, 1024 units), consuming 12.7MB.
- Conviva-A: MADE, 4 hidden layers with 128 units each, consuming 2.5MB. The embedding reuse optimization with $h = 64$ is used (§3.4.2).

For timing experiments, we train and run the learning methods (KDE, MSCN, Naru) on a V100 GPU. Other estimators are run on an 8-core node and vectorized when applicable.

#### 3.6.1.3 Workloads

**Query distribution** (Figure 3.4). We generate multidimensional queries containing both range and equality predicates. The goal is to test each estimator on a wide spectrum of target selectivities: we group them as high (>2%), medium (0.5%–2%), and low ($\leq$ 0.5%). Intuitively, all solutions should perform reasonably well for high-selectivity queries, because dense regions require only coarse-grained modeling capacity. As the query selectivity drops, the estimation task becomes harder, since low-density regions require each estimator to model details in each hypercube. True selectivities are obtained by executing the queries on PostgreSQL.

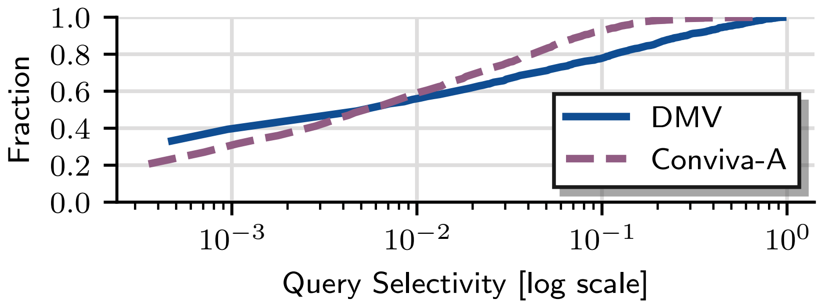


**Figure 3.4:** Distribution of query selectivity (§3.6.1.3).

The query generator is inspired by prior work [56]. Instead of designating a few fixed columns to filter on, we consider the more challenging scenario where filters are randomly placed. First, we draw the number of (non-wildcard) filters $5 \le f \le 11$ uniformly at random. We always include at least five filters to avoid queries with very high selectivity, on which all estimators perform similarly well. Next, $f$ distinct columns are drawn to place the filters. For columns with domain size $\ge 10$, the filter operator is sampled uniformly from $\{=, \le, \ge\}$; for columns with small domains, the equality operator is picked—the intention is to avoid placing a range predicate on categoricals, which often have a low domain size. The filter literals are then chosen from a random tuple sampled uniformly from the table, i.e., they follow the data distribution. For example, a valid 5-filter query on DMV is "($\mathsf{fuel\_type}$ = GAS) $\wedge$ ($\mathsf{rev\_ind}$ = N) $\wedge$ ($\mathsf{sco\_ind}$ = N) $\wedge$ ($\mathsf{valid\_date} \ge$ 2018-03-23) $\wedge$ ($\mathsf{color}$ = BK)". Overall, the queries span a wide range of selectivities (Figure 3.4).

**Accuracy metric.** We report accuracy by the multiplicative error [56, 64, 65] (also termed "Q-error"), the factor by which an estimate differs from the actual cardinality:

$$\text{Error} := \max(\textit{estimate}, \textit{actual}) / \min(\textit{estimate}, \textit{actual})$$

We lower bound the estimated and actual cardinalities at 1 to guard against division by zero. In line with prior work [22, 64], we found that the multiplicative error is much more informative than the *relative* error, as the latter does not fairly penalize small cardinality estimates (which are frequently the case for high-dimensional queries). Lastly we report the errors in quantiles, with a particular focus at the tail. Our results show that all estimators can achieve low median (or mean) errors but with greatly varying performance at the tail, indicating that mean/median metrics do not accurately reflect the hard cases of the estimation task.

### 3.6.2 Estimation Accuracy

In summary, Tables 3.3 and 3.4 show that not only does Naru match or exceed the best estimator across the board, it excels in the *extreme tail of query difficulty*—that is, worst-case errors on low-selectivity queries. For these types of queries, Naru achieves orders of magnitude better accuracy than classical approaches, and up to 90× better tail behavior than query-driven (supervised) methods.

The same Naru model is used to estimate all queries on a dataset, showing the robustness of the model learned. We now discuss these macrobenchmarks in more detail.

#### 3.6.2.1 Results on DMV

Overall, Naru achieves the best accuracy and robustness across the selectivity spectrum. In the tail, it outperforms MHIST by 691×, DBMS-1 by 114×, un-tuned (tuned) MSCN by 115× (33×), BayesNet by 70×, Sample by 47×, and KDE-superv by 21×. We next discuss takeaways from Table 3.3.

**Independence assumptions lead to orders of magnitude errors.** Estimators that assume full or partial independence between columns produce large errors, regardless of query selectivity or how good per-column estimates are. These include Indep, PostgreSQL, and DBMS-1, whose tail errors are in the $10^3 - 10^5\times$ range. Naru's model is powerful enough to avoid this assumption, leading to better results.

**MHIST** outperforms Indep and PostgreSQL by over an order of magnitude. However, its performance is limited by the linear partitioning and uniform spread assumptions.

**BayesNet** also does quite well, nearly matching supervised approaches, but is still significantly outperformed by Naru due to the former's uses of lossy discretization and conditional independence assumptions.

**Worst-case errors are much harder to be robust against.** All estimators perform worse for low-selectivity queries or at worst-case errors. For instance, in the high-selectivity regime PostgreSQL's error is a reasonable 1.55× at 95th, but becomes 1682× worse at the maximum. Also, Sample performs exceptionally well for high and medium selectivity queries, but drops off considerably for low selectivity queries where the sample has no hits. MSCN struggles since its supervised objective requires more training data to cover all possible low-selectivity queries. Naru yields much lower (single-digit) errors at the tail, showing the robustness that results from directly approximating the joint.

**KDE struggles with high-dimensional data.** KDE's errors are among the highest. The reason is that, the bandwidth vector found is highly sub-optimal despite tunings, due to (1) a large number of attributes in DMV, and (2) discrete columns fundamentally do not work well with the notion of "distance" in KDE [40]. The method must rely on query feedback (KDE-superv) to find a good bandwidth.

**MSCN heavily relies on its materialized samples for accurate prediction.** Across the spectrum, its accuracy closely approximates Sample. MSCN-10K has 3× better tail accuracy than MSCN-base due to access to 10× more samples, despite having the same network architecture and trained on the same 100K queries. Both variants' accuracies drop

**Table 3.3:** Estimation errors on DMV. Errors are grouped by true selectivities and shown in percentiles computed from 2,000 queries.

| ESTIMATOR | HIGH ((2%, 100%]) | | | | MEDIUM ((0.5%, 2%]) | | | | LOW ($\leq 0.5\%$) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Median | 95th | 99th | Max | Median | 95th | 99th | Max | Median | 95th | 99th | Max |
| Indep | 1.12 | 1.55 | 46.1 | 2566 | 1.25 | 46.6 | 1051 | $8 \cdot 10^4$ | 1.35 | 225 | 2231 | $2 \cdot 10^4$ |
| PostgreSQL | 1.12 | 1.55 | 46.3 | 2608 | 1.25 | 45.5 | 1070 | $8 \cdot 10^4$ | 1.36 | 227 | 2287 | $2 \cdot 10^4$ |
| DBMS-1 | 1.45 | 3.36 | 5.80 | 12.6 | 2.72 | 6.38 | 9.29 | 10.1 | 5.28 | 83.0 | 417 | 917 |
| Sample | **1.00** | **1.02** | **1.03** | **1.05** | 1.02 | 1.05 | **1.07** | **1.10** | 1.12 | 43.2 | 98.1 | 377 |
| KDE | 10.9 | 1502 | $1 \cdot 10^4$ | $2 \cdot 10^5$ | 48.0 | $2 \cdot 10^4$ | $9 \cdot 10^4$ | $2 \cdot 10^5$ | 38.0 | 3191 | $2 \cdot 10^4$ | $5 \cdot 10^4$ |
| KDE-superv | 1.40 | 3.81 | 4.91 | 13.3 | 1.53 | 4.36 | 8.12 | 16.9 | 1.95 | 30.0 | 98.0 | 175 |
| MSCN-base | 1.17 | 1.42 | 1.47 | 1.58 | 1.14 | 1.65 | 2.53 | 3.96 | 2.95 | 32.5 | 85.6 | 921 |
| MSCN-0 | 16.8 | 92.4 | 195 | 285 | 8.89 | 80.4 | 344 | 471 | 4.79 | 67.1 | 169 | 6145 |
| MSCN-10K | 1.04 | 1.10 | 1.12 | 1.16 | 1.04 | 1.12 | 1.19 | 1.23 | 1.51 | 14.2 | 33.7 | 264 |
| MHIST | 1.70 | 2.65 | 4.11 | 9.20 | 1.65 | 6.00 | 15.1 | 21.1 | 2.81 | 70.3 | 352 | 5532 |
| BayesNet | 1.01 | 1.07 | 1.12 | 1.44 | 1.05 | 1.18 | 1.26 | 1.40 | 3.41 | 16.1 | 79 | 561 |
| Naru-1000 | 1.01 | 1.03 | 1.05 | 1.28 | **1.01** | 1.06 | 1.15 | 1.27 | **1.03** | 1.44 | 2.51 | **8.00** |
| Naru-2000 | 1.01 | 1.03 | 1.04 | 1.16 | **1.01** | **1.04** | 1.09 | 1.38 | **1.03** | **1.41** | **2.18** | **8.00** |

**Table 3.4:** Estimation errors on Conviva-A. Errors grouped by true selectivities and shown in percentiles computed from 2,000 queries.

| Estimator | High ((2%, 100%]) | | | | Medium ((0.5%, 2%]) | | | | Low ($\leq$ 0.5%) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Median | 95th | 99th | Max | Median | 95th | 99th | Max | Median | 95th | 99th | Max |
| DBMS-1 | 1.75 | 5.25 | 7.77 | 9.12 | 3.93 | 13.6 | 19.9 | 31.3 | 8.63 | 176 | 636 | 4737 |
| Sample | **1.02** | **1.06** | **1.09** | **1.11** | **1.04** | **1.14** | **1.18** | **1.23** | 1.18 | 49.3 | 218 | 696 |
| KDE | 105 | $2 \cdot 10^5$ | $5 \cdot 10^5$ | $8 \cdot 10^5$ | 347 | $6 \cdot 10^4$ | $8 \cdot 10^4$ | $8 \cdot 10^4$ | 224 | $1 \cdot 10^4$ | $2 \cdot 10^4$ | $2 \cdot 10^4$ |
| KDE-superv | 1.99 | 7.97 | 14.5 | 33.6 | 2.04 | 8.44 | 17.0 | 49.7 | 2.76 | 74.5 | 251 | 462 |
| MSCN-base | 1.14 | 1.27 | 1.36 | 1.48 | 1.15 | 1.55 | 2.26 | 57.7 | 2.05 | 20.3 | 84.1 | 370 |
| MHIST | 1.54 | 5.24 | 11.2 | $1 \cdot 10^4$ | 2.22 | 10.5 | 30.3 | $3 \cdot 10^4$ | 6.71 | 792 | 4170 | $8 \cdot 10^4$ |
| BayesNet | 1.15 | 1.75 | 2.05 | 2.70 | 1.31 | 2.70 | 4.95 | 47.6 | 1.67 | 14.8 | 78.0 | 998 |
| Naru-1000 | **1.02** | 1.11 | 1.18 | 1.37 | 1.05 | 1.17 | 1.27 | 1.40 | 1.10 | 1.71 | 3.01 | 185 |
| Naru-2000 | **1.02** | 1.10 | 1.16 | 1.28 | 1.05 | 1.17 | 1.27 | 1.38 | **1.09** | 1.66 | 3.00 | 58.0 |
| Naru-4000 | **1.02** | 1.10 | 1.17 | 1.21 | **1.04** | 1.15 | 1.27 | 1.36 | **1.09** | **1.57** | **2.50** | **4.00** |

**Table 3.5:** Robustness to OOD queries. Errors from 2,000 queries.

| Estimator | Median | 95th | 99th | Max |
|---|---|---|---|---|
| MSCN-10K | 23 | 96 | 151 | 417 |
| KDE-superv | 1.00 | 1.00 | 3.67 | 163 |
| Sample | 1.00 | 1.00 | 2.00 | 116 |
| Naru-2000 | 1.00 | 1.00 | 1.26 | 4.00 |

off considerably for low-selectivity queries, since, when there are no hits in the materialized sample, the model relies solely on the query features to make “predictions”. MSCN-0 which does not use materialized samples performs much worse, obtaining a max error of 6145×.

#### 3.6.2.2 Results on Conviva-A

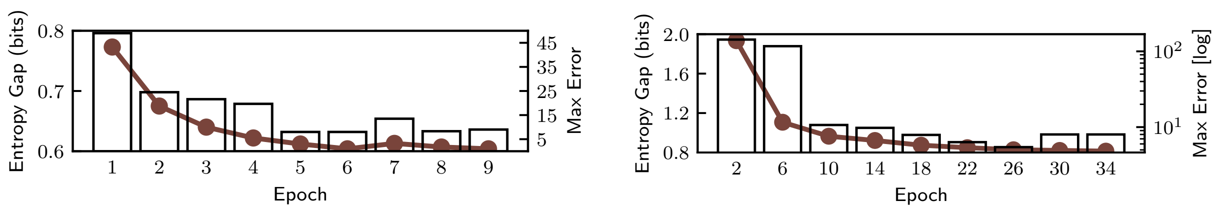
Based on DMV results, we keep only the promising baselines for this dataset. Table 3.4 shows that Naru remains best-in-class for a dataset with substantially different columns and a much larger joint size.

For this dataset, most estimators produce larger errors. This is because Conviva-A has a much larger joint space. DBMS-1, MHIST, BayesNet, and KDE-superv exhibit 5×, 14×, 1.8×, and 2.6× worse max error than before respectively. MSCN-base’s max error in the medium-selectivity regime is also 14× worse. As a non-parametric method covering the full joint space, Sample remains a robust choice.

For Naru, since the sampler needs to cross more domains and a much larger joint space, Naru-1000 becomes insufficient to provide single-digit error in all cases. However, a modest scaling of the number of samples to 4K decreases the worst-case error back to single-digit levels. This suggests that the approximated joint is sufficiently accurate, and that the key challenge lies in *extracting its information.*

### 3.6.3 Robustness to Out-of-Distribution Queries

Our experiments thus far have drawn the filter literals (query centers) from the data. However, a strong estimator must be robust to out-of-distribution (OOD) queries where the literals are drawn from the entire joint domain, which often result in no matching tuples. Table 3.5 shows results on select estimators on 2K OOD queries on DMV, where 98% have a true cardinality of zero. The supervised MSCN-10K suffers greatly (e.g., median is now 23×, up from the 1.51× in Table 3.3) because it was trained on a set of in-distribution queries; at test time, out-of-distribution queries confuse the net. KDE-superv, a sampling-based approach, finds no hits in its sampled tuples, and therefore appropriately assigns zero density mass for all queries.



**(a)** DMV **(b)** Conviva-A

**Figure 3.5:** Training time vs. quality (§3.6.4). Dotted lines show divergence from data; bars show max estimation errors.

Since Naru approximates the data distribution, it correctly learns that out-of-distribution regions have little or no density mass, outperforming KDE by 40× and MSCN by 104×.

### 3.6.4 Training Time vs. Quality

Compared to supervised learning, Naru is efficient to train: no past queries are required; we only need access to a uniform random stream of tuples from the relation. We also find that, surprisingly, it only takes a few epochs of training to obtain a sufficiently powerful Naru estimator.

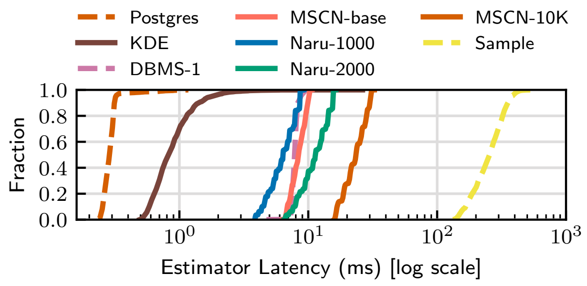
Figure 3.5 shows how two quality metrics, entropy gap and estimation error, change as training progresses. The metrics are calculated after each epoch (one pass over the data) finishes. An epoch takes about 75 seconds and 50 seconds for DMV and Conviva-A, respectively. The number of progressive samples is set to 2K for DMV and 8K for Conviva-A.

Observe that Naru quickly converges to a high goodness-of-fit both in terms of entropy gap and estimation quality. For DMV where a larger Naru model is used, 1 epoch of training suffices to produce the best estimation accuracy compared to all baselines (Table 3.3, last column). For Conviva-A, 2 epochs yields the best-in-class quality and about 15 epochs yields the quality of single-digit max error.

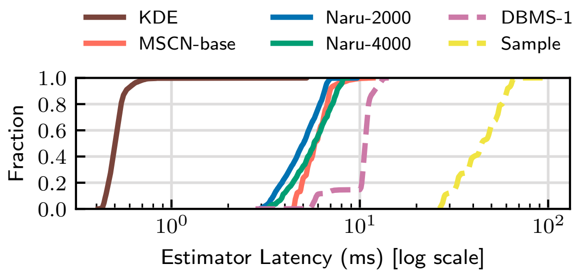
### 3.6.5 Estimation Latency

Figure 3.6 shows Naru's estimation latency against other baselines. On both datasets Naru can finish estimation in around 5-10ms on a GPU, which is faster than scanning samples (Sample and MSCN) and is competitive with DBMS-1. We note the caveat that latencies for PostgreSQL and DBMS-1 include producing an entire plan for each query.

Naive progressive sampling requires as many model forward passes as the number of attributes in the relation. With Naru's wildcard-skipping optimization (§3.5.2), however, we can skip the forward passes that generate distributions for the wildcard columns. Hence, the number of forward passes required is the number of non-wildcard columns in each query.

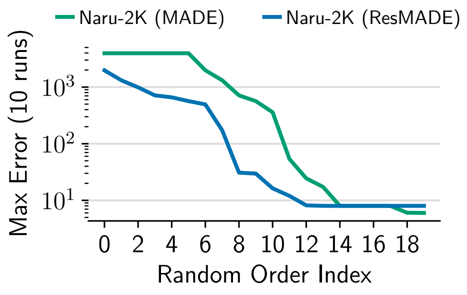


**(a)** DMV

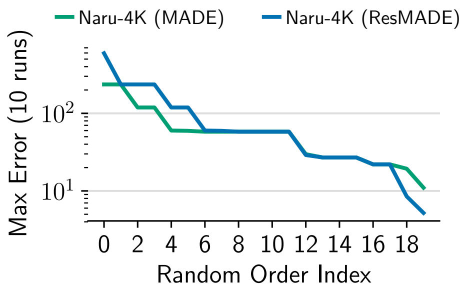


**(b)** Conviva-A

**Figure 3.6:** Estimator latency (§3.6.5). Learning methods are run on GPU; other estimators are run on CPU (dashed lines).



**(a)** DMV



**(b)** Conviva-A

**Figure 3.7:** Variance of random orders. Each dataset's 2000-query workload is run 10 times; the maximum of these 10 max errors are shown. Order indices sorted by descending max error.

This effect manifests in the slightly slanted nature of Naru's CDF curves—queries that only touch a few columns are faster to estimate than those with a larger number of columns. Latency tail is also well-behaved: on DMV, Naru-1000's median is at 6.4ms vs. max at 9.4ms; on Conviva-A, Naru-2000's median is at 5.0ms vs. max at 9.7ms.

Naru's estimation latency can be further minimized by engineering. Naru's sampler is written in Python code and a general-purpose deep learning framework (PyTorch); the resultant control logic overhead from interpretation can be removed by using hand-optimized native code. Orthogonal techniques such as half-precision, i.e., 32-bit floats quantized into 16-bit floats, would shrink Naru's compute cost by half.

### 3.6.6 Variance Analysis

**Effect of random orders.** Figure 3.7 shows the estimation variance of randomly sampled column orders. We sample 20 random orders and train a Naru model on each, varying the autoregressive building block. The result shows that the choice of ordering does affect

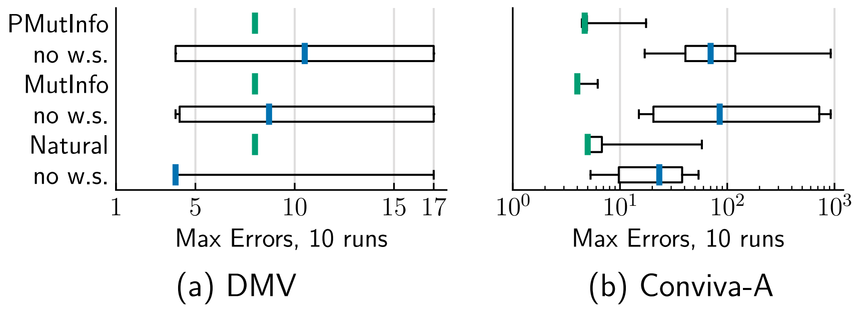


**Figure 3.8:** Wildcard-skipping and heuristic orders. These orders have much lower variance than random orders. Ablation of wildcard-skipping is shown. Distributions of 10 max errors are plotted; whiskers denote min/max and bold bars denote medians.

**Table 3.6:** Larger model sizes yield lower entropy gap. Here we only consider scaling the hidden units of a MADE model.

| Architecture | Size (MB) | Entropy gap, 5 epochs |
|---:|:--|:--|
| $32 \times 32 \times 32 \times 32$ | 0.6 | 4.23 bits per tuple |
| $64 \times 64 \times 64 \times 64$ | 1.1 | 2.25 bits per tuple |
| $128 \times 128 \times 128 \times 128$ | 2.7 | 1.01 bits per tuple |
| $256 \times 256 \times 256 \times 256$ | 3.8 | 0.84 bits per tuple |

estimation variance in the extreme tail. However, we found that on 99%-tile or below, almost all orderings can reach single-digit errors.

**Effect of wildcard-skipping (§3.5.2) and heuristic orders (§3.5.3).** Figure 3.8 shows that the information-theoretic orders have much lower variance than randomly sampled ones. The left-to-right order (Natural) is also shown for comparison. We also find that wildcard-skipping is critical in reducing max error variance by up to several orders of magnitude (e.g., MutInfo's max drops from $10^3$ to $< 10$).

## 3.6.7 Autoregressive Model Choice and Sizing

In Table 3.6, we measure the relationship between model size and entropy gap on Conviva-A. While larger model sizes yield lower entropy gaps, Figure 3.5 shows that this can yield diminishing returns in terms of accuracy.

Table 3.7 compares accuracy and (storage and computation) cost of three similarly sized autoregressive building blocks. The results suggest that ResMADE and regular MADE are

**Table 3.7:** Comparison of autoregressive building blocks (§3.4.3). Max error calculated by running DMV's 2000-query workload 10 times (Naru-2000). FLOPs is the number of floating point operations required per forward pass per input tuple.

| | Params | FLOPs | Ent. Gap | Max Error |
|---|---|---|---|---|
| MADE | 3.3M | 6.7M | 0.59 | 8.0× |
| ResMADE | 3.1M | 6.2M | 0.56 | 8.0× |
| Transformer | 2.8M | 35.5M | 0.54 | 8.2× |

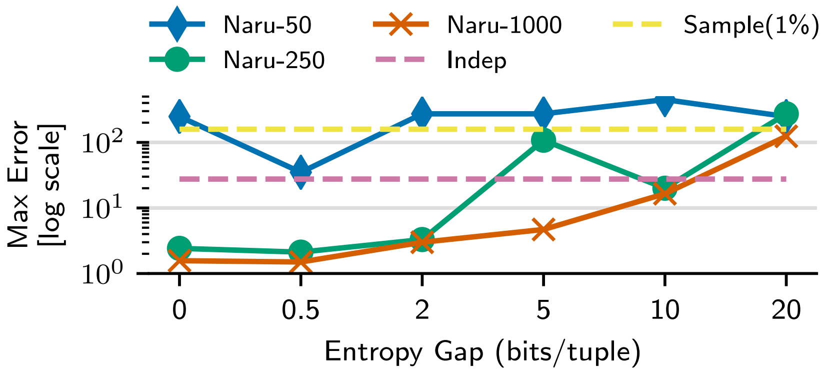


**Figure 3.9:** Accuracy of Naru as an artificial entropy gap is added to an oracle model for Conviva-B projected to the first 15 columns. 50 queries are drawn from the same distribution as in the macrobenchmarks. Naru has the best accuracy for an entropy gap of less than 2 bits, though remains competitive up to a surprisingly large gap of 10 bits. Variance of progressive sampling decreases dramatically when moving from 50 to 250 to 1000 samples.

preferable due to their efficiency. We expect the Transformer—a more advanced architecture—to excel on datasets of larger scale.

### 3.6.8 Understanding Estimation Performance

Naru's accuracy depends critically on two factors: (1) the accuracy of the density model; and (2) the effectiveness of progressive sampling. This section seeks to understand the interplay between the two components and how each contributes to estimation errors. We do this by running microbenchmarks against the Conviva-B dataset, which has only 10K rows but has 100 columns for a total joint space of $10^{190}$. The small size of the dataset makes it possible to run queries against an emulated *oracle model* with perfect accuracy by scanning the data. This allows us to isolate errors introduced by density estimation vs. progressive sampling.

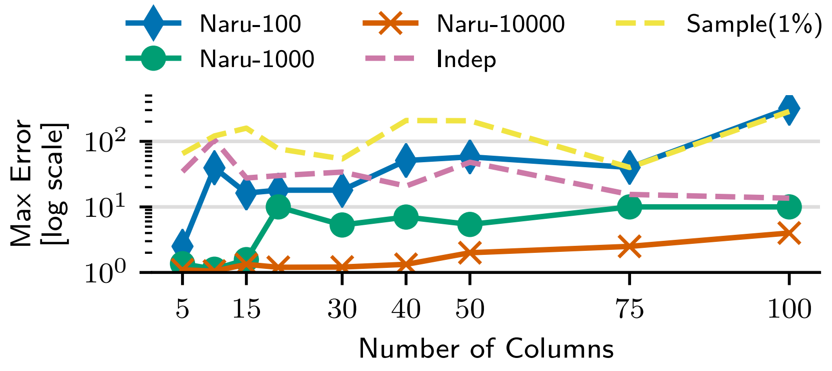


**Figure 3.10:** Accuracy of Naru as we add more columns from Conviva-B. We again use an oracle model (with 0 bits of entropy gap) and 50 randomly generated queries. The number of predicates covers at most 12 columns. The number of progressive sample paths required to accurately query the model increases modestly with the number of columns, but remains tractable even as the joint data space reaches over $10^{190}$ (at 100 columns).

#### 3.6.8.1 Robustness to Increasing Model Entropy Gap

One natural question is: how accurate does the density model have to be? One metric of modeling accuracy is *the fraction of total probability mass assigned to observed data tuples.* For example, a randomly initialized model will assign equal probability mass to all points in the joint space of tuples. As training proceeds, it learns to assign higher probability to tuples actually present in the relation. Under the simplifying assumption that all relation tuples are unique (as they are in Conviva-B), we can quantify this fraction as follows. Suppose the model has an entropy gap of 2 bits; then, the fraction of probability mass assigned to the relation, $f$, satisfies $-\log_2 f = 2$, which leads to $f = 25\%$.

Figure 3.9 shows that Naru achieves the best performance with a model entropy gap of 0-2 bits. A gap of lower than 0.5 bits does not substantially improve performance. This means that for the best accuracy, the model must assign between $25 - 100\%$ of the probability mass to the empirical data distribution. Surprisingly, Naru still outperforms baselines with up to 10 bits of entropy gap, which corresponds to less than $\approx 0.1\%$ probability mass assigned. We hypothesize that the range queries make such modeling errors less critical, because density errors of individual tuples could even out when estimating the density of the region as a whole.

#### 3.6.8.2 Robustness to Increasing Column Counts

While the datasets tested in macrobenchmarks have a good number of columns, using Conviva-B we test how well progressive sampling scales to $10\times$ as many dimensions. Figure 3.10 shows that while the number of columns does significantly increase the variance of estimates, the number of progressive samples required to mitigate this variance remains tractable. A choice of 1000 sample paths produces reasonable worst-case accuracies for up to 100 columns, and 10000 sample paths improves on that by a modest factor.

#### 3.6.8.3 Robustness to Data Shifts

Lastly, we study how Naru reacts to data shifts. We partition DMV by a date column into 5 parts. We then ingest each partition in order, emulating the common practice of "1 new partition per day". Each estimator is built after seeing the first partition. After a new ingest, we test the previously built estimators on queries that touch all data ingested so far. The same query generator as macrobenchmarks is used where the filters are drawn from tuples in the first partition (true selectivities computed on all data ingested so far).

**Table 3.8:** Robustness to data shifts. Errors from 200 queries.

| Partitions Ingested | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Naru, refreshed: max | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 |
| 90%-tile | 1.20 | 1.14 | 1.12 | 1.14 | 1.15 |
| Naru, stale: max | 2.0 | 40.3 | 47.5 | 52.9 | 53.5 |
| 90%-tile | 2.0 | 2.4 | 3.4 | 4.4 | 5.5 |

Table 3.8 shows the results of (1) Naru, no model updates, (2) Naru, with gradient updates on each new ingest. The model architecture is the same as in Table 3.3; 8,000 progressive samples are used since we are interested in learning how much imprecision or staleness presents in the model itself and not the effectiveness of information extraction. The results show that, Naru is able to handle queries on new data with reasonably good accuracy, even without having seen the new partitions. The model has learned to capture the underlying data correlations so the degradation is graceful.

## 3.7 Related Work

Naru builds upon decades of rich research on selectivity estimation and this section cannot replace comprehensive surveys [18]. Below, we highlight the most related areas.

**Joint approximation estimators.** Multidimensional histograms [95, 36, 81, 96] can been seen as coarse approximations to the joint data distribution. Probabilistic relational models (PRMs) [28] rely on a Bayes Net (conditional independence DAG) to factor the joint into materialized conditional probability tables. Tzoumas et al. [124] propose a variant of

PRMs optimized for practical use. Dependency-based histograms [22] make partial or conditional independence assumptions to keep the approximated joint tractable (factors stored as histograms). Naru belongs to this family and applies recent advances from the deep unsupervised learning community. Naru does not make *any* independence assumptions; it directly models the joint distribution and lazily encodes all product-rule factors in a universal function approximator.

**Query-driven estimators.** These are supervised methods that take advantage of past or training queries [13]. ISOMER [116] and STHoles [9] are two representatives that adopt feedback to improve histograms. LEO [118] and CardLearner [136] use feedback to improve selectivity estimation of future queries. Heimel et al. [40] propose query-driven KDEs; Kiefer et al. [51] enhance them to handle joins. Supervised learning regressors [69, 56, 26], some utilizing deep learning, have also been proposed. Naru, an unsupervised data-driven synopsis, is orthogonal to this family. Our evaluation shows that full joint approximation yields accuracy much superior to two supervised methods.

**Machine learning in query optimizers.** Naru can be used as a drop-in replacement of selectivity estimator used in ML-enhanced query optimizers. Ortiz et al. [87] learns query representation to predict cardinalities, a *regression* rather than our *generative* approach. Neo [74], a learned query optimizer, approaches cardinality estimation *indirectly*: embeddings for all attribute values are first pre-trained; later, a network takes them as input and additionally learns to correct or ignore signals from the embeddings. This proposal, as well as reinforcement learning-based join optimizers (DQ [60], ReJOIN [72]), may benefit from Naru's improved estimates.

## 3.8 Summary

We have shown that deep autoregressive models are highly accurate selectivity estimators. They approximate the data distribution without any independence assumptions. We develop a Monte Carlo integration scheme and associated variance reduction techniques that efficiently handle challenging range queries. To the best of our knowledge, these are novel extensions to autoregressive models. Our estimator, Naru, exceeds the state-of-the-art in accuracy over several families of estimators.

Naru can be thought of as an unsupervised *neural synopsis*. In contrast to supervised learning-based estimators, Naru enjoys drastically more efficient training since there is no need to execute queries to collect feedback—it only needs to read the data. Learning directly from the underlying data allows Naru to answer a much more general set of future queries and makes it inherently robust to shifts in the query workload. Our approach is non-intrusive and can serve as an opt-in component inside an optimizer.

# Chapter 4

# NeuroCard: Extending Naru to Joins

Having addressed in Chapter 3 how to remove long-standing heuristics in *base table* cardinality estimation, we now extend this result to cardinality estimation for *joins* over multiple tables. Accurately estimating join cardinalities is more critical than base table estimates [70], because estimation errors propagate exponentially with the number of joins [48]. Inaccurate join estimates would severely hurt the optimizer's quality on queries that touch multiple tables, a common occurrence in today's workloads.

In this chapter, 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. NeuroCard leverages two building blocks from Naru—deep autoregressive modeling (§3.3, §3.4) and the progressive sampling algorithm (§3.5)—and enhances them with new techniques to make learning the distribution of multiple tables without heuristics possible.

By removing both inter-table and inter-column independence assumptions in its probabilistic modeling, NeuroCard achieves orders of magnitude higher accuracy than the best prior methods, scales to more than a dozen tables, while being compact in space (several MBs) and efficient to construct or update (seconds to minutes).

## 4.1 Introduction

There have been two approaches to cardinality estimation: *query-driven* and *data-driven*. Query-driven estimators typically rely on supervised learning to learn a function mapping (featurized) queries to predicted cardinalities. They implicitly assume queries from a production workload are "similar" to training queries—namely, training and test sets of queries are drawn from the same underlying distribution. This assumption can be violated when, for example, users issue unexpected types of queries.

In contrast, data-driven estimators approximate the data distribution of a table—a function mapping each tuple to its probability of occurrence in the table—instead of training

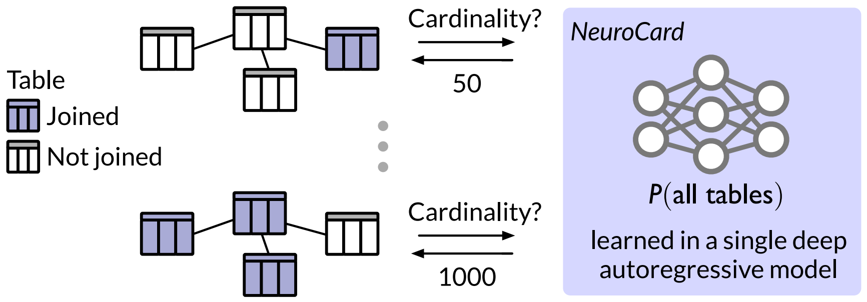


**Figure 4.1:** NeuroCard uses a single probabilistic model, which learns all possible correlations among all tables in a database, to estimate join queries on any subset of tables.

on "representative" queries. A simple method to approximate the data distribution is a histogram. In theory, once we estimate the distribution of each table in a schema, we can estimate the output cardinality of any query. While this approach is more general, it suffers from two drawbacks: (1) *lossy modeling assumptions* (e.g., assume the tables' distributions are independent), and (2) *low precision* (e.g., a limited number of histogram bins). Fortunately, recent advances in machine learning have alleviated both drawbacks. Unlike previous density estimators, *deep autoregressive (AR) models* [103, 127, 97, 27, 25] can learn complex high-dimensional data distributions without independence assumptions, achieving state-of-the-art results in both precision and expressiveness. This has resulted in new data-driven cardinality estimators based on deep AR models, exemplified by the Naru estimator presented in Chapter 3.

However, Naru is limited to handling single tables. There are three challenges that make this approach ineffective for *joins*:

- ***High training cost:*** To learn the distribution of a join, any data-driven estimator needs to see actual tuples from the join result. Unfortunately, for all but the smallest scale, it is expensive, and sometimes infeasible, to precompute the join.
- ***Lack of generality:*** The AR approach builds a probabilistic model for each join, e.g., $T = T_1 \bowtie T_2 \bowtie T_3$, that it estimates. However, the model for $T$ cannot be directly used to estimate a join on a subset of $T$, e.g., $T_2 \bowtie \sigma(T_3)$. Of course, one could train a model for every possible join. This can be prohibitive, as the number of possible joins is exponential in the number of tables.
- ***Large model size:*** The complexity of the learned AR model grows with the cardinality of the dataset. As joins tend to involve columns with high cardinalities, an AR model built on a join may incur a prohibitively large size.

In this chapter, we propose NeuroCard, a learning-based join cardinality estimator that directly learns from data to overcome these challenges. NeuroCard's distinctive feature is the ability to capture the correlations across multiple joins in a single deep AR model, without any independence assumptions (Figure 4.1). Once trained, this model can handle all queries issuable to the schema, regardless of what subset of tables is involved. We address the above challenges using the following key ingredients.

To reduce training cost, NeuroCard samples from a join, instead of computing the join fully (§4.4). The key property of such a sample is to capture the join's distribution: if a key is more frequent in the join result, it should be more frequent in the sample as well. To meet this requirement, we precompute the correct sampling weights for each key. While the worst-case cost of computing the join is exponential in the number of tables, computing the sampling weights is done in time linear with the data size by dynamic programming.

To achieve generality, NeuroCard needs to train a single model to answer queries on any subset of tables (§4.6). The basic idea behind our solution is to train the AR model on samples from the *full outer join* of all tables. The full join contains the values from all the base tables, so it has sufficient information to answer a query touching any subset of tables. At inference time, if a table in the schema is not present in a join query, we need to account for any potential *fanout* effect. Consider an AR model trained on samples from the full join $T = T_1 ⟗ T_2$, and a query $\sigma(T_1)$ whose cardinality we want to estimate. If the join key of $T_2$ is the foreign key of $T_1$, then a tuple of $T_1$ may *appear multiple times* in $T$. NeuroCard learns the probabilities of these "duplicated" tuples and additional bookkeeping information, which enables us to account for fanouts.

Finally, to scale to large-cardinality columns while avoiding prohibitively large models, NeuroCard employs *lossless column factorization* (§4.5). An AR model stores one embedding vector per distinct value, so it could quickly blow up in size for columns with large numbers of distinct values, e.g., 100,000s or more. With factorization, a column is decomposed into several subcolumns, each taking a chunk of bits from the binary representation of the original column values. For instance, a 32-bit ID column $\mathsf{id}$ can be decomposed into $(\mathsf{id}_0, \ldots, \mathsf{id}_3)$ with the first subcolumn corresponding to the first 8 bits, and so on. We then train the autoregressive model on these lower-cardinality subcolumns instead of the full columns.

By combining these ingredients, NeuroCard achieves state-of-the-art estimation accuracy, including in the challenging tail quantiles. On the popular JOB-light benchmark, a schema that contains 6 tables and basic filters, NeuroCard achieves a maximum Q-error of 8.5× using 4 MB. This corresponds to a 4.6× improvement over the previous state of the art. We created a more difficult benchmark, JOB-light-ranges, with a larger variety of content columns and range filters. On this benchmark, NeuroCard achieves up to 15–34× higher accuracy than previous solutions, including DeepDB [44], MSCN [56], and IBJS [63]. Lastly, to test NeuroCard's ability to handle a more complex join schema, we created JOB-M which has 16 tables and multi-key joins. NeuroCard scales well to this benchmark, offering 10× higher accuracy than conventional approaches while maintaining a low model size (27 MB, covering 16 tables).

In summary, this chapter makes the following contributions:

- We design and implement NeuroCard, the first learned data-driven cardinality estimator that learns across joins in a schema without any independence assumptions. All in-schema correlations among the tables are captured by a single autoregressive model, which can estimate any query on any subset of tables.
- NeuroCard learns the correct distribution of a join without actually computing that join. Instead, the model is trained on uniform and independent samples of the join of all tables in a schema.
- We propose lossless column factorization (§4.5), a technique that significantly reduces the size of the autoregressive model, making its use practical for high-cardinality columns.
- Compared to the best prior methods, NeuroCard significantly improves the state-of-the-art accuracy on the JOB-light benchmark. We further propose two new benchmarks, JOB-light-ranges and JOB-M, and show that both are much more challenging and thus better gauges of estimator quality (§4.7).

To invite further research, NeuroCard and the benchmarks used in this chapter are open sourced at `https://github.com/neurocard`.

## 4.2 Overview of NeuroCard

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. A query is a subgraph of the overall schema. If a query joins a table multiple times, our framework duplicates that table in the schema. We assume the schema and queries submitted to the estimator are acyclic (§4.4.2 discusses relaxations), so they can be viewed as trees.

Next, we present an overview of NeuroCard as a sequence of goals and solutions to achieve these goals.

### 4.2.1 Goals and Solutions

**Goal: A single estimator.** Our goal is building a single cardinality estimator for the entire join schema. For example, assuming the schema has three tables, the estimator should handle joins on any subset of tables, e.g., $\sigma(T_2), T_1 \bowtie T_3$, or $T_1 \bowtie T_2 \bowtie \sigma(T_3)$.

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,

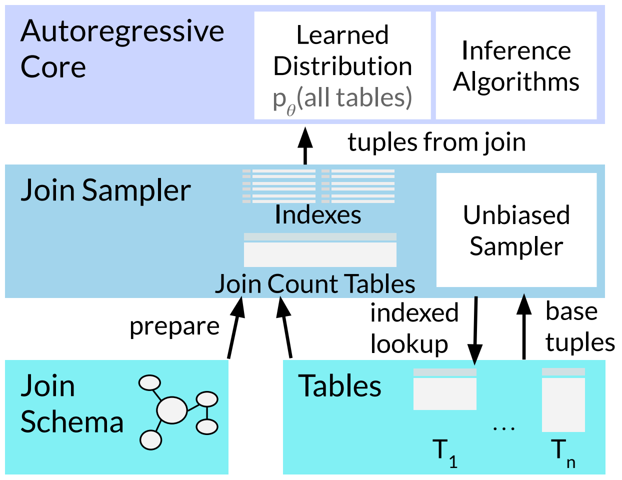


**Figure 4.2:** Overview of NeuroCard. The Join Sampler (§4.4) provides correct training data (sampled tuples from join) by using unbiased join counts. Sampled tuples are streamed to an autoregressive model for maximum likelihood training (§4.3). Inference algorithms (§4.6) use the learned distribution to estimate query cardinalities.

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.

***Solution:*** We build a single cardinality estimator that learns the distribution of *the full outer join of all tables* in the schema (henceforth, full join). For example, for a three-table schema, we learn $p(T_1 ⟗ T_2 ⟗ T_3)$. Note that using the inner join instead of the full join would not work. Indeed, the inner join $T_1 \bowtie T_2 \bowtie T_3$ is the intersection of the three tables. If a query uses only $T_1$ or $T_1 \bowtie T_3$, their tuples may not be fully contained in this intersection, and thus the estimator would have insufficient information to answer these queries.

**Goal: Efficient sampling of the full join.** A data-driven estimator learns a distribution by reading representative tuples from that distribution. To learn the distribution of the full join, a straightforward approach is to compute it and then uniformly draw random samples from the result. 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.

***Solution:*** We perform uniform sampling over the full join *without* materializing it. Specifically, we ensure that any tuple in the full join $J$ (a multiset) is sampled with same probability, $1/|J|$. To achieve this, we leverage a state-of-the-art join sampling algorithm [145] (§4.4). We first precompute *join count tables* that map each table's join keys to their correct sampling weights with respect to the full join. Then, we sample the keys using these counts as weights. Given a sampled key, we construct the full tuple by looking up the remaining columns via indexes[1] from all tables, and then concatenating them. This way, we only need

[1]Like prior work on join sampling [63, 66], we assume base tables have an index built for each join key.

to materialize the join counts as opposed to the full join. Using dynamic programming, computing the join counts takes time linear in the size of the database, and is quite fast in practice (e.g., 13 seconds for 6 tables in JOB-light, and 4 minutes for 16 tables in JOB-M).

**Goal: Support any subset of tables.** Although the full outer join contains all information of the tables, we need to take care when a query involves just a subset of the tables. 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, $T_2$.

***Solution:*** Handle *schema subsetting*: If a query does not include a table, we downscale the estimate by the fanout introduced by that table. In essence, since the learned probability space is the full join, we must downscale appropriately when a query touches a subset and expects the returned selectivity to refer to that subset.

**Goal: Accurate density estimation.** The final ingredient to achieve our goal is an accurate and compact density estimator.

***Solution:*** We leverage *deep autoregressive (AR) models* to implement our density estimator. This family of neural density estimators have been successfully employed on high-dimensional data types such as image [103], audio [127], and text [97]. In Chapter 3, we showed that Naru leverages deep AR models to achieve state-of-the-art accuracy results on estimating the cardinalities of single-table queries, while *learning the correlations among all columns* without independence assumptions. In this chapter, we apply Naru to learn the distribution of the full join, and optimize its construction and inference for our setting.

### 4.2.2 Putting It All Together

Figure 4.2 shows the high-level architecture of NeuroCard.

Building the estimator consists of two stages. First, we prepare the join sampler by building or loading existing single-table indexes on join keys and computing the join count tables for the specified join schema (§4.4). Second, we train the deep AR model by repeatedly requesting batches of sampled tuples from the sampler, usually 2K tuples 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 (§4.6) to compute the

This impacts the efficiency but not correctness of the design.

cardinality estimate by (1) performing Monte Carlo integration on the learned AR model, and (2) handling schema subsetting. A single estimator can handle queries joining any subset of tables, with arbitrary range selections.

## 4.3 Constructing NeuroCard

In this section, we present the background of the techniques used to implement NeuroCard.

### 4.3.1 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 ID histogram technique assumes the columns are independent. As a result, it factors $p$ into a set 1D marginals, $p \approx \prod_{i=1}^{n} p(A_i)$, which can lead to large inaccuracies when the columns' values are strongly correlated. Similarly, other data-driven cardinality estimators such as graphical models [28, 29, 22, 123, 124] either assume conditional independence or partial independence among columns. One exception is the autoregressive (product-rule) factorization,

$$p = \prod_{i=1}^{n} p(A_i | A_{<i}), \tag{4.1}$$

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 $p(A_n | A_{1:n-1})$). At the other end, the 1D histogram has low costs, but this comes at the expense of low precision, as it makes no distinction between the values falling in the same bin. Over the years, a plethora of solutions have been proposed, including kernel density estimators and Bayesian networks. Recently, *deep autoregressive (AR) models* [25, 97, 103] have emerged as the density estimator of choice. Deep AR models compute $\{p(A_i|A_{<i})\}$ 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.

### 4.3.2 Leveraging Naru

NeuroCard builds on Naru, introduced in Chapter 3, a state-of-the-art cardinality estimator that fully captures the correlations among all columns of a single table using a deep AR model. Next, we present a brief review of Naru and in particular discuss how NeuroCard leverages it.

**Construction.** Given table $T$, an AR model $\theta$ takes a tuple $\boldsymbol{x} \in T$ as input, and predicts conditional probability distributions, $\{p_\theta(X_i|\boldsymbol{x}_{<i})\}$, each of which is an 1D distribution over the $i$-th column (conditioned on all prior column values of $\boldsymbol{x}$). The likelihood of the input tuple is then predicted as $p_\theta(\boldsymbol{x}) = \prod_{i=1}^{n} p_\theta(X_i = \boldsymbol{x}_i|\boldsymbol{x}_{<i})$. Any deep AR architecture can instantiate this framework, e.g., ResMADE [25] or Transformer [128]. Training aims to approximate the data distribution $p$ using $p_\theta$, by minimizing the KL divergence [82], $D_{KL}(p||p_\theta)$. This is achieved by maximum likelihood estimation (MLE) and gradient ascent to maximize the predicted (log-)likelihood of data:

$$\text{Sample i.i.d.} \quad \boldsymbol{x} \sim p \tag{4.2}$$

$$\text{Take gradient steps to maximize} \quad \log p_\theta(\boldsymbol{x}) \tag{4.3}$$

In our 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 $p$. Otherwise, $p_\theta$ would approximate an incorrect, biased distribution. To achieve this, we use a sampler that emits *simple random samples* from the full join $T$ (§4.4).

**Estimating query cardinalities.** Once constructed, the Naru estimator estimates the cardinality of a given query. A query is represented as a hyper-rectangle: each column $X_i$ with domain $A_i$ is constrained to take on values in a valid region $R_i \subseteq A_i$:

$$\text{Query:} \quad \wedge \{X_i \in R_i\} \tag{4.4}$$

Next, Naru estimates the probability of the query (an event) using a Monte Carlo integration algorithm, *progressive sampling*:

$$\mathsf{ProgressiveSampling}(\{X_i \in R_i\}): \quad p_\theta(\wedge\{X_i \in R_i\}) \cdot |T| \tag{4.5}$$

It works by drawing imaginary, in-region tuples from the model's learned distributions. Specifically, it draws the first dimension of the sample as $x_1 \sim p_\theta(X_1|X_1 \in R_1)$, the second dimension of the sample as $x_2 \sim p_\theta(X_2|X_2 \in R_2; x_1)$, and so on. The likelihoods of the samples are importance-weighted. This procedure also efficiently supports omitted columns, i.e., wildcards of the form $X_i \in *$.

NeuroCard's inference invokes progressive sampling to estimate cardinalities, but extends it in two ways. First, we apply the column factorization optimization (§4.5), which potentially changes a $X_{i+i}$'s valid region, $R_{i+1}$, based on the value drawn from $X_i$. Second, we add support for schema subsetting (§4.6), by downscaling selectivity $p_\theta(\wedge\{X_i \in R_i\})$ by the corresponding fanout.

### 4.3.3 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 fully autoregressive probabilistic model on the full join consisting of all tables' columns:

$$\text{Model:} \quad p_\theta(T) \equiv p_\theta(T_1.\text{col}_1, T_1.\text{col}_2, \ldots, T_N.\text{col}_k) \tag{4.6}$$

We can then use the probabilistic model to estimate the cardinalities of join queries on any subset of tables in the schema.

**Supported joins.** NeuroCard supports acyclic join schemas and queries containing multi-way, multi-key equi-joins (§4.4.2 discusses how to relax the acyclic requirement). The schema should capture the most common joins. For joins not captured in the schema, their cardinalities can be estimated by first obtaining single-table estimates using NeuroCard, then combining the estimates using classical heuristics [65]. This allows uncommon cases to be handled under the same framework, albeit at the cost of lower accuracy.

**Supported filters.** NeuroCard supports equality and range filters on discrete or numerical columns. These include arithmetic comparison operators ($<, >, \leq, \geq, =$) and IN. More complex filters can also be expressed using the valid region encoding, mentioned in the previous section. Arbitrary forms of AND/OR can be handled via the inclusion-exclusion principle.

### 4.3.4 Model architecture

NeuroCard uses a standard AR architecture, ResMADE [25], which is also employed by Naru; see Figure 4.3. Input tuples are represented as discrete, dictionary-encoded IDs, $(x_1, \ldots, x_n)$, and embedded by per-column embedding matrices. The concatenated embedded vector is fed to a series of residual blocks, each consisting of two masked linear layers (they are masked to ensure the autoregressive property). The output layer produces logits $\{\log p_\theta(X_i|\boldsymbol{x}_{<i})\}$

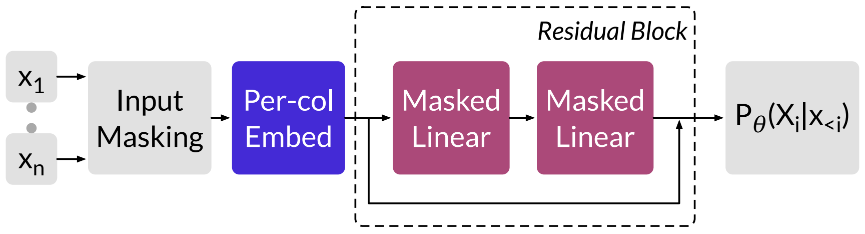


**Figure 4.3:** Default architecture of the autoregressive model.

by dotting the last layer's output with the embedding matrices. Next, we compute a cross-entropy loss on the logits and perform backpropagation. We turn on Naru's *wildcard skipping* optimization, which randomly masks inputs to train special marginalization tokens that aid infer-time estimation (i.e., using these tokens to skip sampling any wildcards in a query).

Masked multi-layer perceptrons such as ResMADE strike a good balance between efficiency and accuracy. NeuroCard can use any advanced AR architectures, if desired. In §4.7, we also instantiate NeuroCard with an advanced architecture (the Transformer [128]).

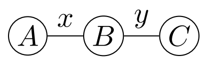
## 4.4 Sampling from Joins

A key challenge in NeuroCard is computing an *unbiased* sample of the full join (§4.2.1) 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 4.2. NeuroCard meets these requirements by using a sampler that produces *simple random samples with replacement*.

### 4.4.1 Algorithm

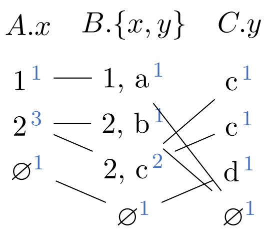
A tuple in the full join contains *join key columns* and *content columns*. Our sampler exploits this decomposition. The first step of the sampler is to precompute *join count tables*, which are per-table statistics that reflect the occurrence counts of the join keys in the full join. The sampler then 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 sample, which is sent to the model for training, and the procedure repeats on demand.

**Computing join counts.** Zhao et al. [145] 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.



| $A.x$ | $B.x$ | $B.y$ | $C.y$ |
|---|---|---|---|
| 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 | ∅ | 1 | 1 | 1 | 0 |
| 2 | 2 | 2 | b | ∅ | 1 | 1 | 1 | 0 |
| 2 | 2 | 2 | c | c | 2 | 1 | 1 | 1 |
| 2 | 2 | 2 | c | c | 2 | 1 | 1 | 1 |
| ∅ | ∅ | 1 | ∅ | 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 4.4:** 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 (§4.6).

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). Let $T_1$ be the root 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 joins with $t$. It is recursively defined as:

$$w_i(t) = \prod_{T_j \in \text{Children}(T_i)} \sum_{t' \in t \rtimes T_j} w_j(t') \quad \forall i, \forall t \in T_i \tag{4.7}$$

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 all $t \in T_1$ in the entire full outer join. The join counts of each table are computed by aggregating over the join counts of all of its child tables, and can thus be computed recursively in a bottom-up fashion. Using dynamic programming, the time complexity is linear in the number of tuples in all tables, $O(|T_1| + \cdots + |T_N|)$.

**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 4.4a. Figure 4.4b 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 4.4b 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[2]

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

$(B.x, B.y) = (2, c)$ 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 §4.3.2 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, although it is an one-time effort. Sampling correctness is preserved even in the presence of parallelism due to the i.i.d. property.

### 4.4.2 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) [63] is neither uniform nor independent; Wander Join [66] 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. [145]. 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 (§4.3.3).

## 4.5 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 the model architecture we use (§4.3.4), 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 $|C|$, 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

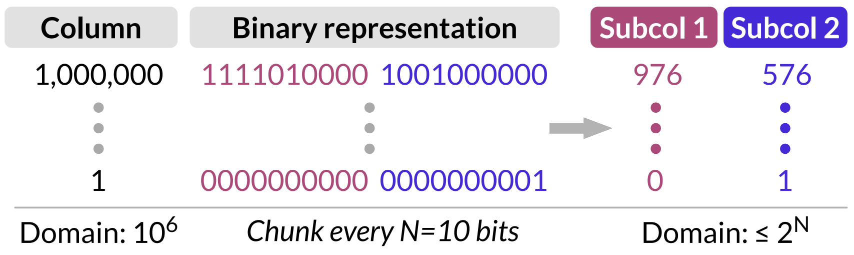


**Figure 4.5:** Lossless column factorization (§4.5).

units" [108] in modern natural language processing, and also shares characteristics with "bit slicing" in the indexing literature [85]. 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 4.5 illustrates the idea on a simple example. Suppose a column (any datatype) has a domain size of $|C| = 10^6$. Naively supporting this column would require allocating $|C| \cdot h$ 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.

**Model size vs. statistical efficiency.** Choosing the factorization bits $N$ enables a trade-off between model size vs. statistical efficiency. By decreasing $N$, we have more subcolumns, each with a smaller domain, but learning across more variables becomes harder. In theory, by using autoregressive modeling no information is lost in this translation, so the precision of the learned distributions is not affected. In practice, we observed that lower factorization bits, i.e., slicing into more subcolumns, generally underperform higher ones that use more space, but not by a significant margin (§4.7.5). We thus set the factorization bits $N$ based on a space usage budget.

**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 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*.

**Filters on subcolumns.** During probabilistic inference, a filter on an original column needs to be translated into *equivalent* filters on subcolumns. Recall from §4.3.2 that the probabilistic inference procedure draws samples that lie inside the queried region. We modify that procedure to handle subcolumns by respecting each filter's semantics. Going back to our example, consider the filter $\mathsf{col} < 1{,}000{,}000$. The filter for the high-bits $\mathsf{subcol}_1$ is *relaxed* to $\leq 976$ (note the less-equal). The inference procedure would draw a $\mathsf{subcol}_1$ value in this range, based on which the low-bits filter is relaxed appropriately. If the drawn $\mathsf{subcol}_1$ is 976, then the filter on $\mathsf{subcol}_2$ 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 reminiscent of processing range predicates on bit-sliced indexes [85]; NeuroCard applies these processing logic in the new context of probabilistic inference for autoregressive models.

## 4.6 Querying NeuroCard

Once built, the autoregressive model summarizes the entire full outer join. The 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*. 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's inference algorithms combine two building blocks. First, Naru introduced *progressive sampling* (§3.5.1), a Monte Carlo algorithm that integrates over an autoregressive model to produce selectivity estimates. We invoke this routine (i.e., Equation 4.5) on the trained autoregressive model with changes outlined in this section. Second, Hilprecht et al. [44] have proposed inference algorithms to query a sum-product network trained on a full outer join. We state their algorithms below and discuss how to adapt these algorithms into our framework, thereby generalizing them to 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 4.4a and an inner join query Q1 in Figure 4.4d. 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 4.4c) 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 4.4c). 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 (Equation 4.5) 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.[3] In summary, for the no-omission case, the routine now estimates the probability:

$$P(\{X_i \in R_i\} \wedge \{\mathbb{1}_T = 1 : \text{for all table } T\}) \tag{4.8}$$

**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 4.4c, this probability is 2/5, so the cardinality is correctly computed as $5 \cdot (2/5) = 2$ rows.

**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 4.4d: $\sigma_{A.x=2}(A)$. 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.[4]

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

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

[4]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.

appears twice in $B.x$, so its fanout is $\mathcal{F}_{B.x}(2) = 2$; see Figures 4.4a and 4.4c. 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*:

$$\begin{aligned} \text{Cardinality(query Q)} &= |J| \cdot P(\{X_i \in R_i\} \text{ subsetted to query Q}) \\ &= |J| \cdot \mathop{\mathbb{E}}_{X \sim J}\left[\frac{\mathbb{1}_{\{X_i \in R_i\}} \cdot \prod_{T \in Q} \mathbb{1}_T}{\prod_{R \notin Q} \mathcal{F}_{R.\text{key}}}\right]. \end{aligned} \tag{4.9}$$

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 4.4c, 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.

**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 4.4a). 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 4.9 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 (§4.3.2). 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* [27] 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.

## 4.7 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 4–34× in accuracy** (§4.7.3). On the popular JOB-light benchmark, NeuroCard achieves a maximum error of 8.5× using 4 MB.
- **NeuroCard scales well to more complex queries** (§4.7.3). 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** (§4.7.4). 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** (§4.7.5). Out of all factors, learning the correlations across all tables and performing unbiased join sampling prove the most impactful.

**Table 4.1:** 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 each workload's queries. Rows in full join: $2 \cdot 10^{12}; 2 \cdot 10^{12}; 10^{13}$.

| 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 |

### 4.7.1 Experimental Setup

**Workloads** (Table 4.1). We adopt the real-world IMDB dataset and schema to test cardinality estimation accuracy. Prior work [65, 64] reported that correlations 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 [119, 56, 44]. 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 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: $\mathsf{mc} \bowtie \sigma_{\mathsf{info\_type\_id}=99}(\mathsf{mi\_idx}) \bowtie \sigma_{\mathsf{episode\_nr}\leq 4 \wedge \mathsf{phonetic\_code}\geq \text{'N612'}}(\mathsf{t})$, 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 [64] by allowing each table to appear at most once per query and removing logical disjunctions (e.g., A.x=1 $\vee$ 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 4.1).

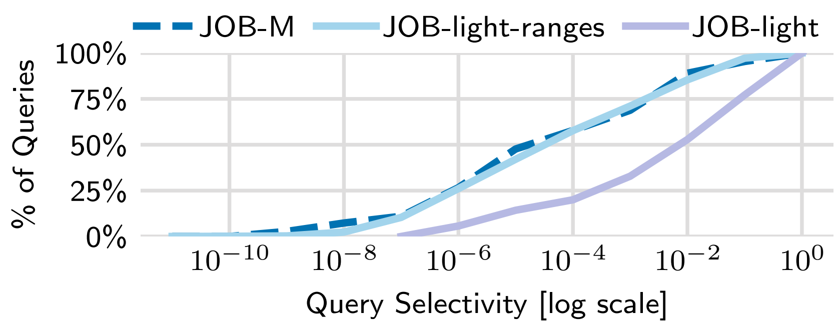


**Figure 4.6:** Distribution of query selectivity (§4.7.1).

The benchmarks are available at `https://github.com/neurocard`.

**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: 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 $1\times$. As reported in Naru's evaluation (§3.6), 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 4.6 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 $100\times$ lower, while at the low tail the minimum selectivities are $1000\times$ lower.

## 4.7.2 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 (§4.8) includes a more complete discussion on all families and their representative methods.

**Supervised query-driven estimators.** We use MSCN [56] 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 [80]. 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 [119], 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 [44] as a recent technique in this family. It uses a non-neural sum-product network [94] as the density estimator for each table subset chosen by correlation tests. Conditional independence is assumed across subsets. In contrast, NeuroCard uses a neural autoregressive model to build a single learned estimator over all tables in a schema. We use two recommended configurations from DeepDB: a base version that learns up to 2-table joins, and a larger version that additionally builds 3-table models. These correspond to their storage-optimized and the standard setups, respectively.

We found that the DeepDB source code [21] 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 the Index-based Join Sampling method (IBJS) [63], 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 PostgreSQL v12, which performs cardinality estimation using 1D histograms and heuristics to combine them.

**Other baselines.** The methods chosen above have been compared to other estimators in prior studies. Naru has shown that estimators based on classical density modeling (KDE; Bayesian networks; the MaxDiff n-dimensional histogram) or random sampling significantly lag behind deep autoregressive models (see §3.6). DeepDB [44] also shows that it significantly outperforms wavelets [10]. We therefore do not compare to these methods.

**NeuroCard.** We implement NeuroCard on top of the Naru source code [84]. We use ResMADE by default. For complex benchmarks we also use the Transformer (§4.3.4), which is suffixed with -large.

**Table 4.2:** JOB-light, estimation errors. Lowest errors are bolded.

| Estimator | Size | Median | 95th | 99th | Max |
|---|---|---|---|---|---|
| PostgreSQL | 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 [119]) | 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** |

## 4.7.3 Estimation Accuracy

### 4.7.3.1 JOB-light

Table 4.2 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.

We now discuss a few observations. Not surprisingly, PostgreSQL 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 [30].

**Table 4.3:** JOB-light-ranges, estimation errors. Lowest errors bolded.

| Estimator | Size | Median | 95th | 99th | Max |
|---|---|---|---|---|---|
| PostgreSQL | 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 | 21 MB | 2.00 | 91.7 | $2 \cdot 10^3$ | $4 \cdot 10^4$ |
| NeuroCard | 4.1 MB | 1.87 | 57.1 | 375 | 8169 |
| NeuroCard-large | 21 MB | **1.40** | **35.1** | **232** | **1029** |

**Table 4.4:** JOB-M, estimation errors. Lowest errors are bolded.

| Estimator | Size | Median | 95th | 99th | Max |
|---|---|---|---|---|---|
| PostgreSQL | 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 | **2.84** | **404** | **1327** | $\mathbf{2 \cdot 10^4}$ |
| NeuroCard-large | 409 MB | **1.96** | **26.4** | **304** | **874** |

#### 4.7.3.2 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 4.3.

**NeuroCard achieves the best accuracy across all error quantiles, and improves on the best prior methods by up to 15−34×.** It is also the only estimator with $< 2\times$ median and 3-digit 99%-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. Comparing the enlarged versions of the two estimators (suffixed with -large), the accuracy gains become $1.4\times, 2.6\times, 9.6\times$ and 34×, 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.

#### 4.7.3.3 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 PostgreSQL and IBJS, because MSCN's query encoding does not support the complex filters in this benchmark and DeepDB ran out of memory on this 16-table dataset due to high-cardinality categorical columns.

Results in Table 4.4 show that **NeuroCard's accuracy remains high on this complex schema**. PostgreSQL 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 (§4.5), the model size is reduced to 27MB—less than 1% of the total size of all tables. We also present a large model NeuroCard-large to demonstrate scalability.

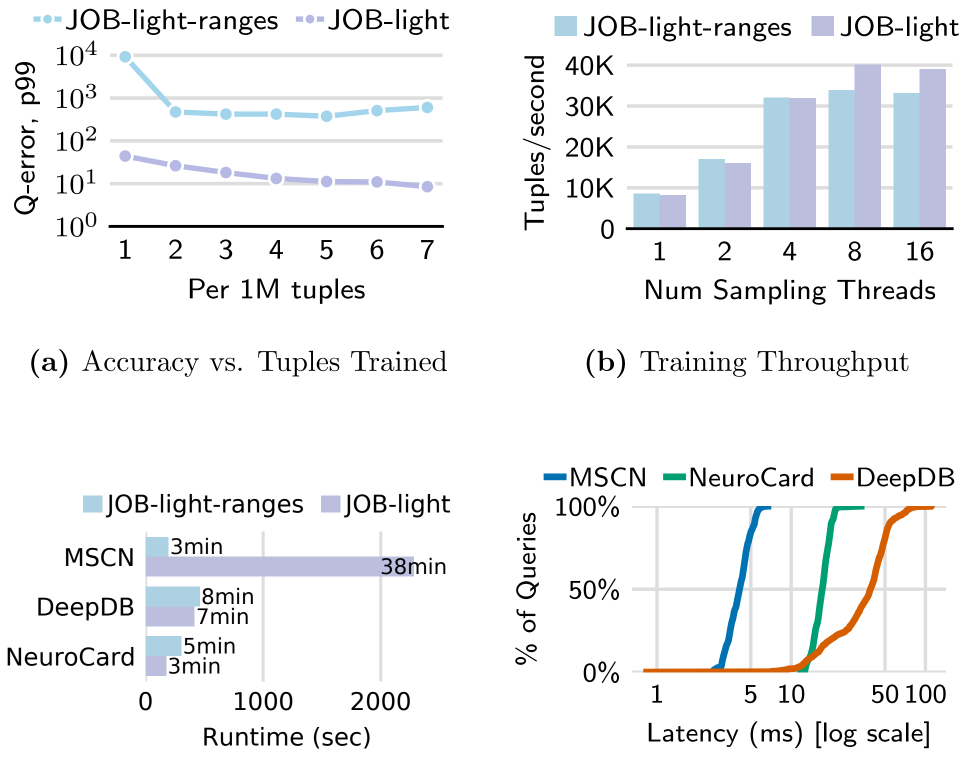
### 4.7.4 Efficiency

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 4.7a 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 4.2 and 4.3). 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 4.7b 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 ~40K tuples/second, NeuroCard can finish training on 3M tuples in about 1.25 minutes.

**Wall-clock training time comparison.** Figure 4.7c compares the wall-clock time used for training the MSCN, DeepDB, and NeuroCard configurations reported in Tables 4.2 and



**(a)** Accuracy vs. Tuples Trained

**(b)** Training Throughput

**(c)** Training Comparison

**(d)** Inference Comparison

**Figure 4.7:** Statistical and physical efficiency of NeuroCard.

4.3. 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. DeepDB runs on parallel CPUs and is quite efficient. NeuroCard starts training/on-the-fly sampling after calculating the join count tables, which takes 13 seconds for both datasets. Its construction is efficient due to parallel sampling and accelerated GPU computation.

**Wall-clock inference time comparison.** Lastly, Figure 4.7d 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

**Table 4.5:** 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 95%-tile errors of JOB-light-ranges.

| | Sampler | Fact. Bits | $d_{\text{ff}}; d_{\text{emb}}$ | Correlations Learned | **p50** | **p95** |
|---|---|---|---|---|---|---|
| Base (4.1 MB) | unbiased | 14 | 128; 16 | all tables in one AR | 1.9 | 57.1 |
| (A) | biased | | | | 33 | 3270 |
| (B) | | 10 (2.2 MB) | | | 2.2 | 173 |
| | | 12 (2.6 MB) | | | 2.0 | 168 |
| | | None (12 MB) | | | 1.6 | 62.7 |
| (C) | | | 128; 64 (23 MB) | | 1.5 | 44.0 |
| | | | 1024; 16 (31 MB) | | 1.7 | 64.0 |
| (D) | | | | one AR per table | 40 | $9 \cdot 10^4$ |
| (E) | No model; uniform join samples only | | | | 4.0 | $2 \cdot 10^5$ |

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.

### 4.7.5 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 4.3 as the *Base* configuration, and ablate each component in isolation. Table 4.5 presents the results.

In (A), using IBJS adapted for full joins[5] 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.

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

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 ($d_{\text{ff}}$) or the embeddings ($d_{\text{emb}}$). 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 (§4.4) 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 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, i.e., progressive sampling, which cannot be used otherwise.

*Tuning guide.* Groups (B) and (C) show that NeuroCard is not overly sensitive to hyperparameters. For new datasets, we recommend starting with the Base configuration and increasing sizing as much as possible up to a size budget. The recommended precedence is: factorization bits; $d_{\text{emb}}$; $d_{\text{ff}}$ and the number of layers. The number of training tuples can be set by early-stopping or a time budget; §4.7.4's results suggest starting with a few to 10+ million.

### 4.7.6 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 1% of original samples (70K), and (3) *retrain*, using 100% of original samples (7M) after each ingest. We also show the latency required to perform additional gradient steps.

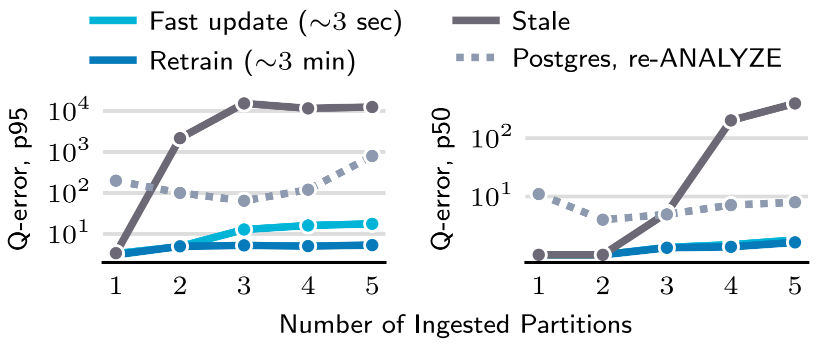


**Figure 4.8:** Updating NeuroCard, fast and slow. JOB-light. Errors (p95, p50) of each strategy are averaged from 10 runs. PostgreSQL is also run as comparison, whose statistics are updated (1∼2 sec.) on each ingest.

Results are shown in Figure 4.8. 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.

## 4.8 Related Work

**Unsupervised data-driven cardinality estimators.** This family approximates the data distribution and dates back to System R's use of 1D histograms [107]. The quality of the density model used has seen steady improvements throughout the years:

**Classical methods.** Multidimensional histograms [95, 36, 81, 96] are more precise than 1D histograms by capturing inter-column correlations. Starting from early 2000s, graphical models were proposed for either single-table or join cardinality estimation [28, 124, 22]. These density models tradeoff precision for efficiency by assuming conditional or partial independence, and require expensive structure learning (finding the best model structure given a dataset).

**Sum-product networks.** SPNs, a tree-structured density estimator, were proposed about 10 years ago [94]. 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 [75]. DeepDB [44] 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 multiple tables. *(S2)* Our "schema subsetting" capability builds on their querying algorithms.

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 table subset (∼1–3 tables) chosen by correlation tests. Conditional independence is assumed across table subsets. *(D3) Correct sampling:* NeuroCard identifies the key requirement of sampling from the data distribution of joins in an unbiased fashion. In contrast, DeepDB obtains join tuples either from full computation or IBJS which samples from a biased distribution. Due to these differences, NeuroCard outperforms DeepDB by up to 34× in accuracy (§4.7).

**Deep autoregressive models.** A breakthrough in density estimation, deep AR models are the current state-of-the-art density models from the ML community [97, 27, 128, 25]. 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. Naru [142] is a single-table cardinality estimator that uses a deep AR model; see Chapter 3. By accessing conditional distributions, Naru proposes efficient algorithms to integrate over an AR model, thereby producing selectivity estimates. NeuroCard builds on single-table Naru and overcomes the unique challenges (§4.2) to support joins.

**Supervised query-driven cardinality estimators.** Leveraging past or collected queries to improve estimates dates back to LEO [118]. Interest in this approach has seen a resurgence partly due to an abundance of query logs [136] or better function approximators (neural networks) [56, 119] that map featurized queries to predicted cardinalities. Hybrid methods that leverage query feedback to improve density modeling have also been explored, e.g., KDE [40, 51] and mixture of uniforms [89]. Supervised estimators can easily leverage query feedback, handle complex predicates (e.g., UDFs), and are usually more lightweight [26]. NeuroCard has demonstrated superior estimation accuracy to representatives in this family, 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. [145], which is among the top-performing samplers they study. This algorithm provides uniform and independent samples, just as NeuroCard requires. NeuroCard may further leverage their extensions to support cyclic join schemas. While IBJS [63] and Wander

Join [66] 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.** A great deal of work has recently applied either classical ML or modern deep learning to various database components, e.g., indexing [59], data layout [144], and query optimization [60, 122, 74]. NeuroCard can be seen as a versatile *core* that can benefit any query engine, learned or not learned. 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 [137] or indexing [138]).

## 4.9 Summary

In this chapter we have presented the NeuroCard estimator. 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: $p_\theta$(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 (4–34× better than prior methods) using a single per-schema model that is both compact and efficient to learn.

# Chapter 5

# Balsa: Learning to Optimize Queries

Chapter 3 and Chapter 4 removed long-standing heuristics in cardinality estimation. We now turn our attention to the second key challenge addressed in this dissertation: the high development cost of optimizers. As discussed in Chapter 1, optimizers take experts months to write and years to refine due to their high complexity. Instead of having human experts spend years developing a state-of-the-art optimizer, in this chapter we ask the question: *is it possible to use machine learning to automatically learn to optimize queries?*

We answer this question affirmatively by presenting Balsa, a learned query optimizer based on deep reinforcement learning. Balsa demonstrates for the first time that learning to optimize queries without learning from an expert optimizer is both possible and efficient.

To achieve this arguably surprising result, Balsa tackles the key challenge of mitigating disastrously slow plans during the agent's learning process. It first learns basic knowledge from a simple, engine and environment-agnostic simulator, followed by learning in real execution, guarded by new *safe execution* and *safe exploration* techniques. On the challenging Join Order Benchmark designed to stress test query optimizers, Balsa matches the performance of two expert optimizers, from PostgreSQL and a commercial engine, with two hours of learning, and outperforms them by up to 2.8× in workload runtime after a few more hours. Balsa thus opens the possibility of automatically learning to optimize in future compute environments where expert-designed optimizers do not exist.

## 5.1 Introduction

Balsa leverages deep reinforcement learning (RL), which has been successfully employed to learn complex skills [2] and play highly challenging games, defeating human champions [112, 113, 131]. RL consists of an *agent* that learns to solve a task by repeatedly interacting with an *environment*. The agent observes the environment's *state* and takes an *action* to maximize a *reward*. If the actions lead to improved rewards, they are *reinforced*, i.e., the agent is updated to make these actions more likely in the future. For a learned optimizer agent, such as Balsa, the environment is the database; a state is a partial plan for a query; an action is to add operators to the partial plan, and the reward for a complete plan is its execution latency (negated). Using this feedback loop, Balsa learns by trial and error to become increasingly better at generating query execution plans.

In fact, the promise of RL for query optimization has been shown by several recent projects [60, 74, 73]. However, these methods assume the availability of a mature query optimizer to learn from. In contrast, Balsa does not learn from such an expert optimizer. To our knowledge, Balsa demonstrates for the first time that *learning to optimize queries without learning from an expert optimizer is both possible and efficient*. This can have a far reaching impact, as it paves the road towards automatically learning to optimize in new data systems [91, 83] where a mature optimizer does not exist.

A unique challenge in learning to optimize queries without an expert optimizer's guidance is that most execution plans for a query are slow—sometimes orders of magnitude more expensive than the optimal plan [64, 65]. At the beginning of the learning process, the agent has no prior knowledge, so the probability of selecting such disastrous plans is high, which may prevent any progress. This is a unique characteristic of query optimization that is not shared by other successful RL applications such as games. Indeed, with most games (e.g., AlphaGo [112], MuZero [104]), a "bad" action typically leads to a game ending quicker. As a result, bad actions do not hinder learning in those environments.

To avoid disastrous plans, Balsa employs simulation-to-reality learning [121]. In the "simulation" phase, Balsa quickly learns from a simulator how to avoid disastrous plans without executing queries, while in the "reality" phase it learns from real executions to produce high-performance plans. The simulator gives cost feedback to the agent by using a basic, logical-only cost model with a cardinality estimator. For convenience, we use PostgreSQL's cardinality estimator, a simple histogram-based method [64]. We pick an existing estimator since, unlike an optimizer, a cardinality estimator is agnostic to the execution environment, so the same estimator can be used for any environment. (In our evaluation, we use PostgreSQL's estimates for another commercial engine.) Moreover, the estimator needs not be high-quality for effective simulation. In fact, PostgreSQL's estimates can exhibit orders of magnitude errors [64], and we find that even injecting noises to these estimates does not impact Balsa's performance (§5.10). This is because Balsa only uses the simulation to learn to avoid disastrous plans, not to reach expert-level performance. Therefore, basic cost models and estimates suffice.

Next, to vastly improve over the imperfect knowledge acquired from the simulator, Balsa learns in the real environment by actually executing queries. While the simulation knowledge enables the agent to avoid the worst plans, it can still stumble onto bad plans, causing unpredictable stalls in the learning process. Balsa addresses this challenge by using *timeouts*. A query's timeout is set to its best latency so far during learning. If a plan times out, we assign it a predefined low reward (as we do not know its true reward). If the plan finishes, we tighten the timeout for future iterations. Thus, timeouts bound each learning iteration's

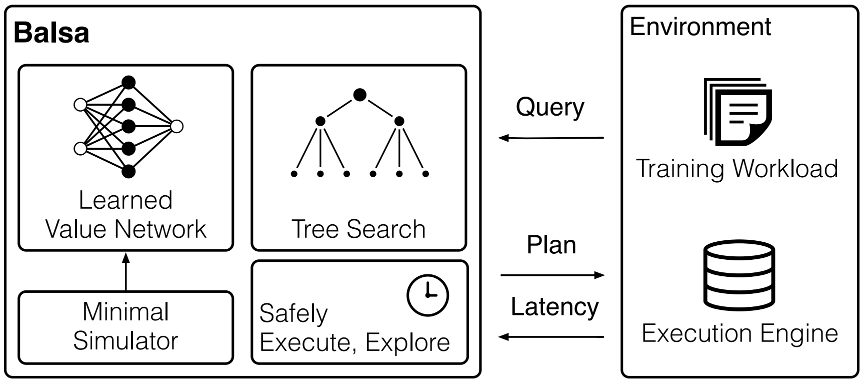


**Figure 5.1:** Balsa's architecture. Balsa learns to optimize queries by executing plans and observing their latency feedback from an engine.

runtime, ensuring *safe execution* that eliminates unpredictable stalls.

Finally, an RL agent must balance exploiting past experiences with exploring new ones to escape local minima. The classic solution is random exploration, i.e., occasionally pick a random plan. Unfortunately, this standard strategy is ineffective, since random plans in the search space are likely to be highly expensive. Instead, Balsa explores from a set of *probably good* plans. During exploration, Balsa generates *several* best predicted plans (instead of the best), then picks the best unseen one out of them. This *safe exploration* approach improves Balsa's plan coverage and performance.

Given a target dataset, Balsa is trained by repeatedly optimizing a set of sample queries by trial and error. After training, we test its *generalization* performance on a new set of unseen queries for the same dataset. We find that all three components of Balsa—simulation learning, safe execution, safe exploration—boost its generalization. They expose Balsa to a higher quantity and variety of plans, thereby enabling it to optimize new queries more robustly—a trait we believe is essential for the practical deployment of learned optimizers. We further propose using *diversified experiences* to enhance generalization (§5.6). We study Balsa's generalization in depth in our evaluation (§5.8.2, §5.8.5), and find that it achieves better performance than two expert optimizers on unseen queries.

We call our approach "**B**ootstrap, **Sa**fe**l**y Execute, **Sa**fely Explore", hence Balsa[1] for short. To our knowledge, Balsa is the first learned optimizer that does not rely on plans (*demonstrations*) generated by an existing expert optimizer. On the Join Order Benchmark [64], a complex workload designed to stress test optimizers, Balsa matches the performance of two expert optimizers with two hours of training, and outperforms them by 2.1–2.8× after a few more hours.

In summary, we make the following contributions:

[1]Balsa wood is famous for its light weight.

- We introduce Balsa, a learned query optimizer that does not learn from an existing, expert optimizer.
- We design a simple approach for learning a query optimizer without expert demonstrations: bootstrapping from simulation (§5.3), safe execution (§5.4), and safely exploring the plan space (§5.5).
- We propose *diversified experiences*, a novel method to further enhance training and generalization performance (§5.6), including generalizing to unseen queries with highly distinct join templates.
- Balsa can outperform both an open-source (PostgreSQL) and a commercial query optimizer, after a few hours of training (§5.8).
- We show that, despite not learning from an expert optimizer, Balsa outperforms the prior state-of-the-art technique that does.

Balsa is open sourced at `https://github.com/balsa-project/balsa`.

### 5.1.1 Differences from Prior Work

To highlight Balsa's contributions, we briefly compare with the most related work and defer a complete discussion to §5.9.

*DQ* [60] learns from an expert optimizer's *cost model*. As such, its performance is bounded by the quality of the cost model, which can be inaccurate. *Neo* [74] takes an opposite approach by learning from an expert optimizer's *plans* and real executions. While this is more accurate than using just a cost model, it is also more expensive. Importantly, these solutions assume either an *expert* cost model or an *expert* optimizer to bootstrap from.

In contrast, Balsa requires neither an expert cost model (as in DQ) nor an expert optimizer (as in Neo) to learn from. Balsa removes these fundamental assumptions by bootstrapping from a minimal, logical-only cost model, followed by safe learning in real execution. For the cost model, Balsa needs a basic cardinality estimator (§5.3.3). We find inaccurate estimates can still lead to successful simulation, and most of Balsa's knowledge is learned after simulation (§5.10).

In summary, this chapter tackles the *new problem* of learning to optimize when an expert optimizer does not exist. (We discuss in §5.10 how Balsa can *better leverage* an expert, if available, than prior work.) To solve this problem, we develop or apply techniques new to the domain of learned optimizers. These include sim-to-real (§5.2), safe execution (§5.4.3), safe exploration (§5.5), on-policy learning (§5.4.1), and enhancing generalization with diversified experiences (§5.6).

## 5.2 Balsa Overview

Balsa's goal is to learn to optimize queries for a given dataset and an execution engine. We assume a training workload is available. At test time, Balsa is asked to optimize unseen queries issued for the same dataset, which can contain new filters and join graphs that are different from those in the training queries.

Balsa learns by trial and error. It optimizes the training queries, producing different plans, then executes them on the engine to observe their runtimes. Based on the runtime feedback, Balsa updates itself to correct mistakes and reward good decisions. As the feedback loop repeats, Balsa gets better at generating good plans.

After training, Balsa can be deployed to optimize an unseen test set of queries. The agent is evaluated by the performance of training plans produced, the performance of testing plans produced (i.e., its generalization ability), and its learning efficiency.

Throughout learning, Balsa accesses the underlying execution engine only to execute plans and observe their runtimes, and does not learn from an existing optimizer. This requirement is informed by the fact that many data systems have execution engines built long before an optimizer becomes available (§5.1).

**Assumptions.** We assume the database content is kept static. Updates to the schema, appends, or in-place updates can be handled by retraining. This assumption implies that the agent need not solve a learning problem with a shifting distribution. Another assumption is that Balsa currently optimizes select-project-join (SPJ) blocks. This is in line with the classical treatment [107] of decomposing a query into simple SPJ blocks and optimizing them block-by-block.

### 5.2.1 Approach

Balsa's architecture is shown in Figure 5.1. It consists of three basic components: bootstrapping a value network in a minimal cost model, fine-tuning the value network in real execution, and using a tree search algorithm to build query plans.

**Classical design: cost models + enumeration.** The classical optimizer design [107] uses an expert-implemented *cost model* that takes in a plan[2] and outputs a cost estimate:

$$C : \mathsf{plan} \to \mathsf{cost}$$

Costs are designed to reflect real execution performance: lower costs should correlate with faster execution. The optimizer produces plans by *enumerating* candidate plans and scoring them using the cost model. For queries with a small number of tables, dynamic programming (DP) is typically used as the enumeration module.

[2]We use "plans" to refer to both complete plans and partial subplans.

**RL: value functions + planning.** Instead of a cost model, which estimates the immediate cost of a plan, Balsa learns a *value function* that estimates the *overall* cost/latency of executing a query when the plan is used as a partial step (subplan):

$$V : (\mathsf{query}, \mathsf{plan}) \to \mathsf{overall\ cost\ or\ latency}$$

Given a value function, we can use it to optimize queries by building a plan bottom-up. Consider a query $Q$ joining tables $\{A, B, C, D\}$. To figure out the best first join to perform, we compare the overall cost/latency, i.e., the value, of all valid first joins:

$$\{A, B, C, D\} \Rightarrow [V(Q, A \bowtie B)); V(Q, A \bowtie C); \ldots]$$

In other words, we use $V$ to score the 2-table joins, which are all partial subplans to complete query $Q$. The best first join is the one with the lowest $V$ value. Suppose $A \bowtie C$ is the best among them, then we can continue the process, scoring all possible second joins:

$$\{A \bowtie C, B, D\} \Rightarrow [V(Q, B \bowtie D); V(Q, B \bowtie (A \bowtie C)); \ldots]$$

Continuing such *planning* leads to a complete query plan.

In contrast to the classical cost model, a value function directly optimizes for the final, overall cost/latency of completing a query—the real objective we care about. Moreover, a *learned* value function can leverage data to tailor to a target database and hardware environment, potentially surpassing heuristics. If the optimal value function $V^*$ is known, then planning would produce optimal plans for queries. Our goal is to approximate $V^*$ as accurately as possible.

**Learned value networks.** Balsa approximates the optimal value function by training a neural network, $V_\theta(\mathsf{query}, \mathsf{plan})$ (with parameters $\theta$), on agent-collected data. The two inputs to the network are featurized into *query features* (encoding joined tables and filters) and *plan features* (encoding the tree structure of the plan and each node's operator type), respectively.

We learn the value function in two stages. First, we learn parameters $\theta_{\text{sim}}$ in a fast simulation environment backed by a minimal cost model. Next, we initialize parameters $\theta_{\text{real}} \leftarrow \theta_{\text{sim}}$ and start fine-tuning the value function in real execution. The two stages produce the value networks[3]:

$$V_{\text{sim}} : (\mathsf{query}, \mathsf{plan}) \to \mathsf{overall\ cost}$$
$$V_{\text{real}} : (\mathsf{query}, \mathsf{plan}) \to \mathsf{overall\ latency}$$

After training, $V_{\text{real}}$ is used with planning to optimize new queries.

[3]For notational convenience, throughout this chapter we use $V_{\text{sim}}$ and $V_{\text{real}}$ to refer to the simulation and real-execution models $V_{\theta_{\text{sim}}}$ and $V_{\theta_{\text{real}}}$, respectively.

**Step 1: bootstrapping from a minimal cost model (§5.3).** Balsa starts learning in a "simulator" of query optimization, i.e., a cost model. The key advantage of using a simulator is that *the agent can learn about disastrous plans without executing them* in the initial phase of learning. The agent bootstraps initial knowledge against an inaccurate but fast-to-query cost model, which provides rapid feedback (cost estimates) for the agent. The cost model is generic and does not model the target engine or hardware.

To train the simulation model $V_{\text{sim}}$, we use a data collection procedure (e.g., DP) to enumerate plans for the training query set and ask the simulator for costs. Each query can yield thousands of training data points, eventually producing a sufficiently large dataset, $\mathcal{D}_{\text{sim}} = \{(\mathsf{query}, \mathsf{plan}, \mathsf{overall\ cost})\}$. $V_{\text{sim}}$ is then trained on this dataset in a standard supervised learning fashion.

**Step 2: fine-tuning in real execution (§5.4).** Next, we transfer the value function from doing well in the simulator to excelling in the real execution environment. The second stage starts by initializing the real-execution model from the trained simulation model: $V_{\text{real}} \leftarrow V_{\text{sim}}$. The fine-tuning of $V_{\text{real}}$ is performed in iterations of query executions and model updates. In each iteration, Balsa uses its current $V_{\text{real}}$ to optimize training queries; these plans are executed with their latencies measured. Balsa then updates its $V_{\text{real}}$ on these collected data to make its latency predictions more accurate.

A key challenge of learning in real execution is mitigating slow plans. We address this as follows. By initializing from $V_{\text{sim}}$, Balsa's behavior in iteration 0 would be much better than random initialization (which amounts to picking plans randomly). After iteration 0, Balsa uses timeouts (determined by earlier runtimes) to early-terminate slow plans (§5.4.3) and also employs safe exploration (§5.5).

**Planning with tree search.** Balsa uses tree search planning on top of the learned value function to optimize queries. The learned $V_{\text{real}}$ guides the search towards the promising regions of the plan space. As $V_{\text{real}}$ becomes more accurate, better plans can be found.

There are many tree search algorithms with different complexity-optimality tradeoffs: from greedy planning, to advanced planning algorithms such as Monte Carlo tree search. We opt for a middle ground by using a simple beam search (§5.4.2).

In the next sections, we describe Balsa's components in detail.

## 5.3 Bootstrapping From Simulation

The first stage of training aims to rapidly impart basic knowledge to the agent, before it starts learning in long-running real executions. We achieve this by bootstrapping Balsa in a minimal simulator, i.e., a cost model. It "simulates" query optimization in that query plans are not actually executed. Instead, the agent issues a large amount of plans to the simulator, which can quickly return cost estimates (rather than measuring their runtimes) as feedback.

**Why is a simulator necessary?** The search space for a query is vast and disastrous execution plans are abundant [64]. Unfortunately, disastrous plans can stall learning progress: an agent may wait for a long time for a slow plan to complete execution, before learning that it is a bad action (if it ever finishes). This property is in direct contrast to other RL use cases such as games. In game environments (e.g., Go, chess, Atari), bad moves typically cause a game to end *sooner*, as the opponent can exploit the agent's mistakes.

A randomly initialized RL agent without training in simulation can quite easily stumble upon such disastrous plans, especially in the early stage of learning. We show this with a simple experiment: we randomly initialize 6 agents without simulation learning, and task them with optimizing 94 queries from the Join Order Benchmark (detailed setup described in §5.8.1). Plans produced by the median random agent execute 45× slower in workload runtime than those produced by an expert optimizer, PostgreSQL. The slowest agent is 79× slower than the expert (2.5 hours vs. 2 minutes).

Next, we describe the specific choice of cost model employed.

## 5.3.1 A Minimal Simulator

Balsa uses a minimal, logical plan-only cost model, which captures the general principle that "fewer tuples lead to better plans". It is *minimal*, because it is free of any prior knowledge about the execution engine and physical operators (e.g., merge vs. hash join).

Formally, we use the $C_{out}$ cost model [16]:

$$C_{out}(T) = \begin{cases} |T| & \text{if } T \text{ is a table/selection} \\ |T| + C_{out}(T_1) + C_{out}(T_2) & \text{if } T = T_1 \bowtie T_2 \end{cases}$$

where $|T|$ denotes the estimated cardinality of a table (with filters taken into account) or a join, obtained from a cardinality estimator (§5.3.3). This cost model estimates the cost of a query plan simply by summing up the estimated result sizes of all operators.

**Tradeoffs of a minimal simulator.** We choose a minimal cost model to bake in as little prior knowledge as possible. The goal of simulation learning is to steer the agent away from definitively disastrous plans (when it starts the real execution phase), not to instill expert knowledge. It is also *generic*: by not modeling physical details, it can be used to bootstrap Balsa optimizing for any engine.

Due to its simplicity, the cost model is inherently inaccurate. Balsa will learn to fill in missing knowledge and correct inaccuracy when fine-tuning in the real execution phase (§5.4). As we will show in §5.8.3.1, while Balsa can leverage pre-engineered, more sophisticated

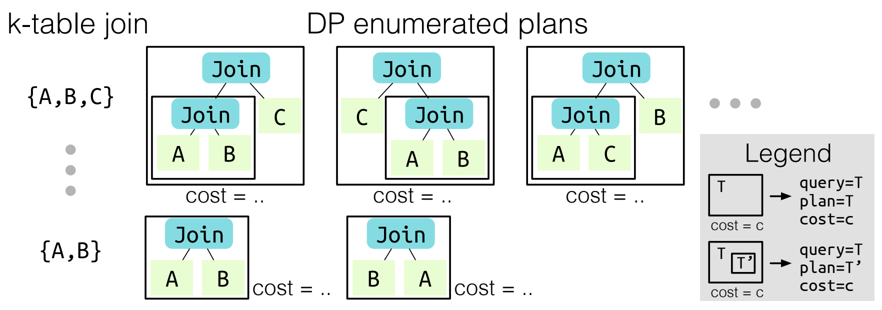


**Figure 5.2:** Simulation data collection and augmentation. For each $k$-table join in DP, Balsa collects and augments all its enumerated plans. Each bordered box yields a collected data point (see legend).

cost models to accelerate training, they are not required for Balsa to reach expert-level performance.

## 5.3.2 Simulation Data Collection

Given a simulator, we extract as much knowledge from it as possible by applying a batched data collection procedure. The output is the simulation dataset, $\mathcal{D}_{\text{sim}} = \{(\mathsf{query}, \mathsf{plan}, \mathsf{overall\ cost})\}$, which is used to train the value network $V_{\text{sim}}$. Specifically, we use dynamic programming (also used by DQ [60]) to collect data.

**Enumerating plans using dynamic programming.** For each query in Balsa's training workload, we run the classical Selinger [107] bottom-up DP with a bushy plan space. It starts by enumerating the best plans for all valid 2-table joins, composed out of base table scans, then enumerating 3-table joins, etc. Each enumerated plan $T$ will get a cost estimate $C$ from the cost model,[4] generating a data point ($\mathsf{query}$=$T$, $\mathsf{plan}$=$T$, $\mathsf{overall\ cost}$=$C$), where $\mathsf{query}$=$T$ denotes the original query restricted to the tables/filters of $T$. This data point undergoes a *data augmentation* procedure, described below, to yield a list of training data points to be added into $\mathcal{D}_{\text{sim}}$.

The data collection is *high-throughput*: data is generated from *all* enumerated plans, not just from the set of optimal plans in the final DP results. This means that some suboptimal plans (under the cost model) are included, which increase data variety and aid learning. Figure 5.2 illustrates the data collection procedure.

However, DP's runtime may become too large for queries joining many tables. Hence we skip collecting data from queries with $\geq n$ tables (we set $n = 12$). Alternative strategies can

[4]Balsa enumerates physical plans for $C_{out}$, which will ignore the differences between physical joins/scans and treat them as logical operators.

also be applied. For example, DQ proposes a partial DP scheme where the first $j$ levels of DP are run and the rest of the levels are planned greedily.

**Data augmentation.** Balsa employs a data augmentation technique proposed by DQ, where multiple data points are generated from a single enumerated plan. Specifically, given a $(\textsf{query}=T,\ \textsf{plan}=T,\ \textsf{overall cost}=C)$, each subplan $T'$ of $T$ will yield a distinct data point with the same "overall query" $T$ and the same cost: $\{(\textsf{query}=T,\ \textsf{plan}=T',\ \textsf{overall cost}=C) : \forall T' \subseteq T\}$. This technique significantly enriches the dataset $\mathcal{D}_{\text{sim}}$ in quantity and variety.

**Interpretation.** In RL terms, the augmentation reflects that all states (the subplans) in a trajectory (the overall query/final plan) share the same return, because intermediate rewards are defined to be 0 and terminal rewards are the negative costs of final plans.

### 5.3.3 Discussion

We found simulation learning to be highly effective. At the start of §5.3, we performed a simple experiment illustrating an up to 79× gap between randomly initialized (i.e., no bootstrapping) agents and an expert optimizer. Now, with simulation bootstrapping, agents significantly shorten this gap to only 5.8× slower than the expert at max—all without performing any real execution.

**Cardinality estimator.** The simulator needs a cardinality estimator. As mentioned in §5.1, we pick PostgreSQL's estimator for its simplicity (per-column histograms; heuristically assumes independence for joins; "magic constants" for complex filters) [64]. Balsa does *not* learn from PostgreSQL's optimizer (costs or plans).

We use an existing, textbook-style estimator for convenience, *not to rely on it for good performance*. In fact, most of Balsa's quality improvements are learned after the simulation stage (§5.8.2, §5.10).

**Alternative cost models.** While Balsa advocates for a minimal simulator, more prior knowledge can be plugged in by the user, if desired. Other cost models may include progressively more physical operator knowledge (e.g., the $C_{mm}$ cost model [64] for in-memory settings). New query engines optimizing for different objectives (e.g., lower memory footprint) may either bootstrap Balsa with $C_{out}$ (its fewer-tuples-are-better principle generally applies), or develop another minimal cost model tailored to the objective.

## 5.4 Learning from Real Execution

Simulation learning imparts basic knowledge to the agent. But no simulators can perfectly reflect the nuances of the real execution environment. Therefore, we fine-tune the agent through query executions in the real environment.

### 5.4.1 Reinforcement Learning of the Value Function

Balsa learns the real-execution value network, $V_{\text{real}}(\mathsf{query}, \mathsf{plan}) \rightarrow \mathsf{overall\ latency}$, using reinforcement learning. The basic idea is that the agent iteratively uses its current value network to optimize queries and runs them, then uses the latency feedback to improve itself. As this feedback loop runs, more execution data is collected, and the agent's $V_{\text{real}}$ becomes better at generating good plans.

Concretely, we start with $V_{\text{real}}$ initialized[5] from $V_{\text{sim}}$ and an empty real-execution dataset, $\mathcal{D}_{\text{real}} = \emptyset$. Each iteration of learning consists of an execute and an update phase.

***Execute.*** The agent uses the current $V_{\text{real}}$ to optimize each training query $q$, producing an execution plan $p$. (Planning will be described in §5.4.2.) Each plan is executed on the target engine with its latency $l$ measured. This results in one data point, $(\mathsf{query}{=}q,\ \mathsf{plan}{=}p,\ \mathsf{overall\ latency}{=}l)$, which then undergoes the same subplan data augmentation discussed in §5.3.2 to yield a list of data points:

$$\mathcal{D}_{\text{real}} \mathrel{+}= \{(\mathsf{query} = q, \mathsf{plan} = p', \mathsf{overall\ latency} = l) : \forall p' \subseteq p\}$$

***Update.*** Balsa uses the collected data to improve its $V_{\text{real}}$. We perform stochastic gradient descent (SGD) with an L2 loss between predicted and true latencies. Thus, mispredictions are corrected and good predictions are reinforced. Data points $(q, p, l)$ are sampled from $\mathcal{D}_{\text{real}}$. However, model outputs $V_{\text{real}}(q, p)$ are updated not towards $l$, but towards the *best latency obtained so far* of query $q$ that involves subplan $p$—a previously proposed technique [74]. The latency label correction is motivated as follows. Consider query $q$ joining tables $A, B, C, D$. Subplan $p = \mathsf{Join}(A, B)$ may have appeared in two executions, one with $C$ joined next and one with $D$ joined next. They may have wildly different latencies, say 1 vs. 100 seconds. As we wish to minimize latency, we define the lower latency $l = 1$ as the value of subplan $p$, because $p$ could have made $q$ run this fast. The best latencies so far are calculated from the entire $\mathcal{D}_{\text{real}}$.

Thus, data collection and value function improvement alternate. The algorithm can be thought of as either value iteration [120] or expert iteration [4], and variants of it have been recently applied in prior work in query optimization [74] (which, different from Balsa's updates, resets and retrains the value network across iterations) , theorem proving [93], and compute schedule optimization [1].

**On-policy learning.** Balsa employs a novel optimization on top of the algorithm above by using *on-policy learning*. Updates to $V_{\text{real}}$ are performed only on the data points generated by the current $V_{\text{real}}$. In other words, SGD is performed on data points $(q, p, \_)$ sampled from the most recent iteration of the dataset, $\mathcal{D}_{\text{real}}$, but not from its entirety. The latter would

[5]Predictions naturally change from the scales of costs to latencies through fine-tuning.

yield data from many iterations ago and is hence off-policy. Label correction still utilizes the entire dataset.

Intuitively, the most recent data points generally are the most surprising to the agent and have faster latency labels, so it should be beneficial to focus on them. Indeed, we find on-policy learning to significantly accelerate learning, by reducing the number of SGD steps per iteration, and improve the plan variety and performance of Balsa (§5.8.3.4). On-policy learning makes Balsa's training more than 9.6× faster when compared to Neo [74], a prior state-of-the-art method, which employs a full retraining scheme instead (§5.8.4). We hypothesize that this technique may also improve other applications of value functions that predict runtimes.

### 5.4.2 Plan Search

With the learned value network, Balsa uses a simple (best-first) beam search to produce execution plans for a given query.

Beam search operates on *search states*, each a set of partial plans for the query. The search starts with a root state that contains all tables (scans) in the query. A beam of size $b$ stores search states to be expanded, sorted by their predicted latencies.[6] At each step, the best search state is popped from the beam, and all available actions are applied to produce children states. Each action joins two eligible plans in the current state with a physical join operator assigned, as well as assigning scan operators if either side is a table. As a search state is a set of partial plans (joined relations and non-joined tables), applying actions to it will lead to at least one complete plan.

Then, all resulting children states are scored by the value network $V_{\text{real}}$ and added to the beam, which keeps the top $b$ states only. In this way, the learned value network *guides the search* to focus on the more promising regions of the plan space. Beam search terminates when $k$ complete plans are found. Balsa uses $b = 20$ and $k = 10$.

**Top-$k$ plans and exploration.** Beam search is not guaranteed to return globally optimal plans, and better plans may be found later in the search. We thus continue searching until $k$ complete plans are found. At test time, the best plan out of this list is emitted.

Interestingly, at training time, obtaining a list of plans enables a simple exploration technique on top. We treat all of these plans as having reasonable optimality—so that it should be safe to explore among them—and prioritize choosing the unseen plans as beam search outputs. This technique is discussed in §5.5.

[6] $V_{\text{real}}$ takes a $(\mathsf{query}, \mathsf{plan})$ as input, while a search state is a set of partial plans for the same query. To score the latter, we define $V(\mathsf{state}) \equiv \max_{\mathsf{plan} \in \mathsf{state}} V(\mathsf{query}, \mathsf{plan})$. Intuitively, it reflects that a state's latency is at least the maximum overall latency a subplan is predicted to take.

### 5.4.3 Safe Execution via Timeouts

A unique challenge in query optimization is the proliferation of expensive plans in a vast search space, even when fast plans exist. When Balsa learns by trial and error from real executions, it can encounter long-running plans with unacceptably high latencies.

Balsa addresses this challenge by applying *timeouts*, a classical idea in distributed systems. Since training proceeds in iterations, earlier execution runtimes of the same training workload are known and can be used to bound future iterations.

Key to this mechanism is how to pick the initial timeout. Fortunately, simulation learning allows us to assume that when the real execution starts, the first ever plans produced for a set of training queries have reasonable (albeit suboptimal) latencies.

**Timeout policy.** During iteration 0's execute phase (just after simulation learning), the plans are allowed to finish execution in their entirety—simulation learning is assumed to yield a non-disastrous starting point. Let the maximum per-query runtime recorded be $T$.

For iteration $i > 0$, a timeout of $S \times T$ is applied for all agent-produced plans, where $S$ is a "slack factor". By definition of $T$, for any training query there exists a plan that can finish execution in time $T$. The slack's purpose is to give some extra room and account for runtime variance (Balsa uses $S = 2$).

If a plan has been executing longer than the current timeout, it is terminated early, since it would be slower than earlier found plans for the same query anyway. It gets assigned a large label[7] instead of its true, unknown latency. Such large labels serve to *discourage* and steer the agent away from similar plans in future iterations.

Timeouts are progressively tightened. If an iteration finishes with a maximum per-query runtime $T' < T$, then the next iteration's timeout is tightened to $S \times T'$. This progression ensures that the timeout is neither too small, which prevents progress, nor too large, which wastes efforts. It generates an *implicit learning curriculum* for the agent with just-about-right difficulties.

In sum, we found the timeout mechanism to significantly accelerate learning. It bounds the runtime of each iteration's execute phase and eliminates unexpected stalls, thereby achieving *safe execution*.

## 5.5 Safe Exploration in Real Execution

While an RL agent exploits its past experience for good performance, it must also explore new experience to escape local minima. To achieve this, an exploration strategy can be used.

[7]We use 4096 seconds throughout. It can also be set as some multiple of iteration 0's maximum per-query runtime.

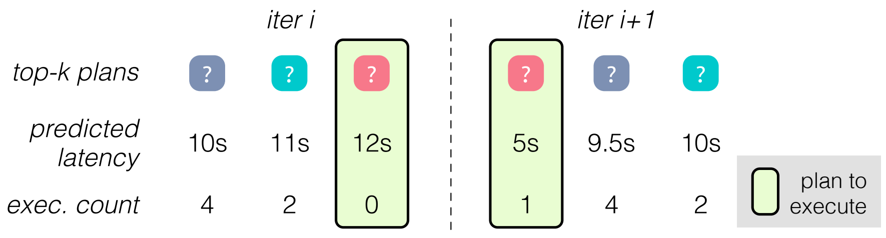


**Figure 5.3:** Safe exploration. For a training query, Balsa prioritizes running the unseen plans of the top-$k$ plans from tree search (exploration). If all seen, the predicted-best plan is chosen (exploitation).

However, the abundance of slow plans, a unique characteristic of query optimization, additionally requires *safe exploration*, i.e., disastrous plans be avoided. Random plans sampled from the search space are slow [64], and choosing to explore them would again stall learning. In our early experiments, a basic $\epsilon$-greedy strategy (for each training query, with a small $\epsilon$ probability a random plan is sampled, a la QuickPick [133]) often selected inferior plans that led to timeouts, slowing down the discovery of better plans and learning.

To achieve safe exploration, Balsa proposes a simple *count-based exploration* technique. In essence, this family of methods encourages an agent to explore a less-visited state or execute a less-chosen action. We instantiate this principle in the following way.

**Count-based exploration for beam search.** Our goal is to provide a "trust region" of reasonable plans for the agent to explore. To do so, beam search is asked to return top-$k$ plans, sorted by ascending predicted latencies, rather than the single best plan found. Instead of executing the best plan (i.e., with the lowest predicted latency), we execute *the best unseen plan* of this list. If all top-$k$ plans have been previously executed—indicating sufficient exploration—Balsa resorts to exploitation by executing the predicted-cheapest plan. The visit counts of plans are cached by a hash table, which adds low overheads, as past executions are already stored in $\mathcal{D}_{\text{real}}$. Figure 5.3 illustrates this technique using example statistics ($k = 3$).

Intuitively, all of the top-$k$ plans are *probably good* (since they are produced by value network-guided beam search), so they should not be chosen strictly by their predicted latencies (which are imperfect estimates). Therefore, executing novel, unseen plans in this "trust region" is both safe and exploratory.

## 5.6 Diversified Experiences

For learned query optimizers, robustly optimizing unseen queries is essential. To further enhance Balsa's generalization performance, we introduce a simple method, *diversified experiences*.

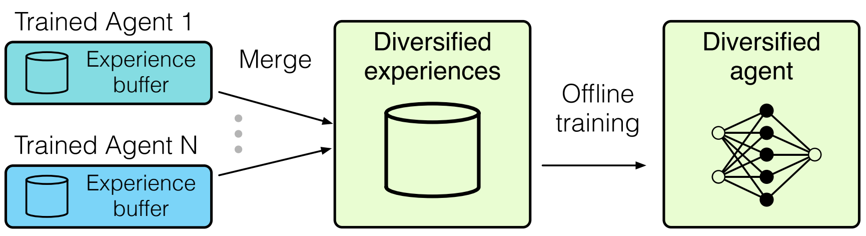


**Figure 5.4:** Diversified experiences. A more robust agent is produced by retraining on the experiences collected from different agents.

**Problem: mode diversity.** As a value network is used to guide plan search, an agent tends to only experience plans preferred by its value network, and may gradually converge to plans with similar characteristics, or a "mode". For example, if hash and loop joins are equally effective for a workload, an agent may learn to heavily use hash joins, while another may prefer loop joins. Either agent can output good plans, as both operators are effective, but they may lack the knowledge about plans that prefer alternative operators or shapes. (While exploration increases plan variety, the new plans are still relatively confined to a single agent's mode.) Low mode diversity can hinder an agent's generalization to highly distinct, unseen queries that require unfamiliar modes to be optimized well.

**Diversified experiences.** To enhance generalization, we propose simply merging the experiences ($\mathcal{D}_{\text{real}}$) collected by several independently trained agents (with different random seeds), and retraining a new agent on top without any real execution. Figure 5.4 illustrates this process. Our insight is that this *diversified experience covers multiple modes.* Thus, training on it produces a more robust value network that generalizes better.

**Table 5.1:** Diversifying experiences: number of data collection agents vs. number of unique plans after merging. Agents have highly diverse experiences. Trained on 113 JOB queries (details in §5.8.1).

| Num. Agents | 1 | 4 | 8 |
|---|---|---|---|
| Num. Unique Plans | 27K (1×) | 102K (3.8×) | 197K (7.3×) |

Table 5.1 confirms this insight: the number of unique plans grows almost linearly as the number of agents, showing that the plans experienced by different agents are indeed highly diverse. We find this simple method effective (§5.8.5), offering a way to trade more compute, when available, for better performance.

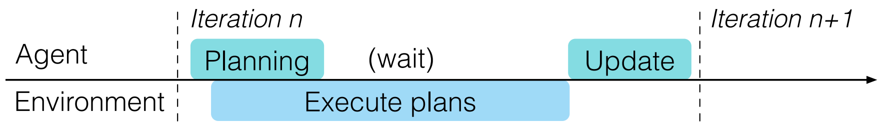


**Figure 5.5:** Pipelining agent planning and remote query execution.

## 5.7 Implementation

In this section we describe Balsa's detailed training setup. At a high level, to operate Balsa on a new engine it needs the following:

- An execution environment (executes plans; support for timeouts).
- Definition of the search space (the set of query operators and the rules to compose them).

**Optimizations.** We optimize training by parallel data collection, plan caching, and pipelining. Query executions are dispatched to a pool of identical virtual machines each running an instance of the target database, using Ray [79]. Each VM runs one query at a time to prevent interference. A plan cache is used so that reissued plans have their prior runtimes quickly looked up and can skip re-execution. Planning and remote query execution in each iteration are pipelined (Figure 5.5): as soon as tree search (run by the main agent thread) finishes planning a training query, the output plan is sent for remote execution, and then planning for the next query starts. The two stages thus overlap. The agent waits for all plans to finish before performing value network updates.

**Value network details.** The value networks, $V_{\text{sim}}$ and $V_{\text{real}}$, are implemented as simple tree convolution networks [74] (0.7M parameters, or 2.9MB). We also experimented with implementing them using a Transformer [128] early on; this was found to be similarly effective but had higher computational costs. When training or updating the value networks, we sample 10% of experience data as a validation set for early stopping. The inputs to the value network, query and plan, are encoded as follows. Each plan has the same encoding as Neo [74]. A query is featurized as a vector [table $\rightarrow$ selectivity] where each slot corresponds to a table and holds its estimated selectivity (§5.3.3). Absent tables' slots are filled with zeros. This encoding is simpler than both Neo and DQ [60].

## 5.8 Evaluation

We conduct an in-depth evaluation of Balsa. Our key findings are:

- Learning by trial and error, Balsa generates better execution plans that run up to 2.1–2.8× faster in workload runtime than two expert optimizers, PostgreSQL and "CommDB" [8] (§5.8.2).
- Balsa takes a few hours to surpass the experts and a few more hours to reach peak performance on the tested workloads (§5.8.2).
- Balsa outperforms learning from expert demonstrations [74], a prior state-of-the-art approach, despite not learning from an expert optimizer (§5.8.4). We also identify *poor generalization* as a potential failure mode in this prior method.
- Diversified experiences significantly enhance generalization, including to queries with highly distinct join templates (§5.8.5).
- Balsa learns *novel* preferences of operators and plan shapes (§5.8.6).

Additionally, we conduct detailed ablation studies to understand the effect of Balsa's design choices in §5.8.3.

### 5.8.1 Experimental Setup

We use the following workloads, in each of which Balsa is trained on a set of training queries and tested on a set of unseen queries:

**Join Order Benchmark (JOB)** contains 113 analytical queries designed by Leis et al. [64] to stress test query optimizers over a real-world dataset from the Internet Movie Database. The queries involve complex joins and predicates, ranging from 3-16 joins, averaging 8 joins per query. We benchmark against two train-test splits, each with 94 training and 19 test queries:

- Random Split (denoted as "JOB"): a randomly sampled split.
- Slow Split (denoted as "JOB Slow"): the test set consists of the 19 slowest-running queries when planned by an expert optimizer.

Random Split tests an average situation, while Slow Split evaluates when the test queries run maximally slower than the train queries.

[8] A leading commercial DBMS. We anonymize its name due to its licensing terms [99].

**TPC-H** is a standard analytical benchmark where data and queries are generated from uniform distributions. We use a scale factor of 10. We use 70 queries for training and 10 queries as the test set.[9]

**Expert baselines and engines.** We compare with the optimizers of two mature expert systems: PostgreSQL (12.5; open-source) and CommDB (a leading commercial DBMS; anonymized [99]). For each expert, we compare Balsa's plans with its optimizer's plans executed on that same engine. Balsa's plans are injected by hints [92].

We use Microsoft Azure VMs with 8 cores, 64GB RAM, and SSDs. Training is done on a NVIDIA Tesla M60 GPU. We configure PostgreSQL with 32GB shared buffers and cache size, 4GB work memory, and GEQO disabled—settings similar to Leis et al. [64]. We optimize CommDB extensively by following its tuning guides.

Balsa is trained for 500 iterations on the JOB workloads and 100 iterations on TPC-H due to its smaller search space. Balsa uses all components and default values discussed in prior sections.

**Expert performance.**[10] We follow the guidance in Leis et al. [64] to create all primary and foreign key indexes to make our baselines run JOB much faster than that of prior work [74, 122]. This also makes the search space more complex and challenging.

**Metrics.** We repeat each experiment 8 times and report the median metric, unless specified otherwise. In train/test curves, we show the *entire min/max ranges* in shaded areas. Workload runtime is defined as the sum of per-query latencies. When reporting normalized runtimes, they are calculated with respect to the expert's runtimes.

## 5.8.2 Balsa Performance

We begin with end-to-end results, answering the following:

- What is the performance of Balsa on training and test queries?
- How many hours (and executions) does Balsa need to surpass expert performance and reach its peak performance, respectively?

[9]For TPC-H, we use templates 3, 5, 7, 8, 12, 13, 14 for training and template 10 for testing, with 10 queries generated per template. We avoid the templates with advanced SQL features (views, sub-queries) due to a limitation in the pg_hint_plan extension.

[10]PostgreSQL runtimes (train/test): JOB 115s/24s; JOB Slow 44s/98s, TPC-H 452s/49s. We do not disable nested loop joins as suggested by Leis et al., because with indexes created, this change actually made the expert run JOB 60% slower.

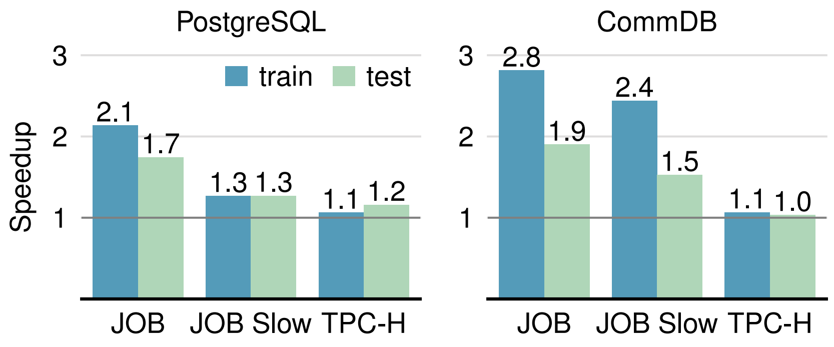


**Figure 5.6:** Balsa's performance on PostgreSQL (left) and CommDB (right): workload speedups achieved by Balsa plans over plans from the respective expert optimizer. Each bar is the median of 8 runs.

**Table 5.2:** Simulation learning efficiency: sizes of simulation datasets, time to collect data (in minutes), and time to train. Train times differ due to early stopping. Means ± standard deviations are shown.

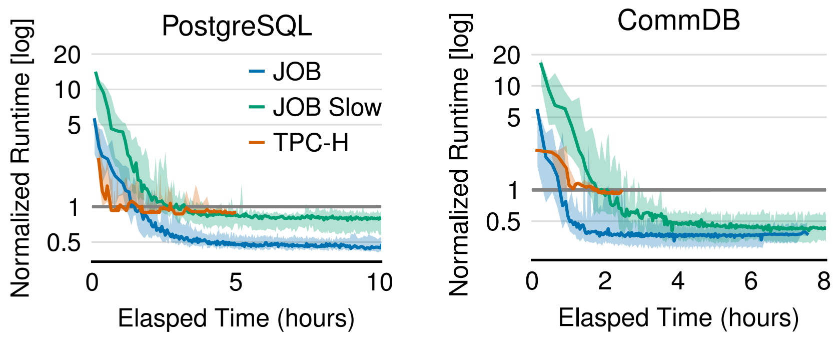
| Workload | Size | Collection time (min.) | Train time (min.) |
|---|---|---|---|
| JOB | 516K | $6.8 \pm 0.1$ | $24 \pm 8$ |
| JOB Slow | 551K | $7.6 \pm 0.1$ | $28 \pm 10$ |
| TPC-H | 12K | $1.1 \pm 0.01$ | $1.0 \pm 0.2$ |

**Performance.** Figure 5.6 summarizes Balsa's overall performance. On all workloads, Balsa is able to start from a minimal cost model and learn to surpass the expert optimizers by a sizable margin.

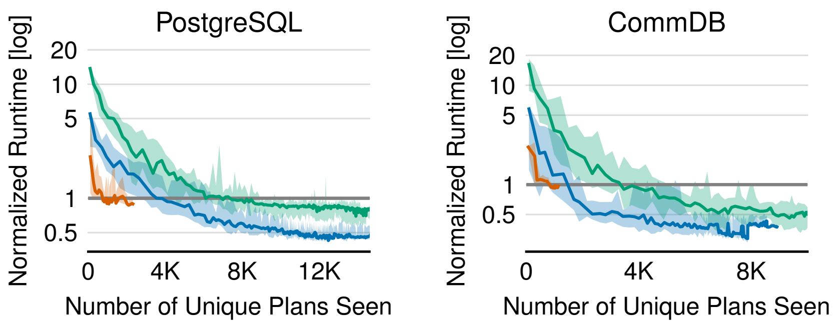
On PostgreSQL, Balsa achieves a 2.1× training-set speedup on JOB, 1.3× on JOB Slow, and 1.1× on TPC-H. While speedups on test sets slightly trail behind the training set speedups, Balsa can still produce faster execution plans than the expert (e.g., 1.7× faster on JOB). This shows that Balsa can generalize to unseen queries.

Balsa also outperforms CommDB's optimizer. The speedups are higher—1.1–2.8× for train and 1.0–1.9× for test sets—because CommDB allows a much smaller search space than PostgreSQL by not exposing bushy hints. (We estimate it to be 1000× smaller for an average-sized JOB query, counting plan shapes and operators.) Balsa thus explores the smaller search space more comprehensively.

**Runtime of simulation learning.** Table 5.2 shows simulation is data-rich and takes dozens of minutes. As it is a small fraction of real execution learning's duration, we focus



**(a)** Wall-clock efficiency



**(b)** Data efficiency

**Figure 5.7:** Learning efficiency of Balsa. Normalized runtime of training queries (log scale) vs. (a) elapsed time and (b) number of executed plans.

on the latter next.

**Learning efficiency.** Figure 5.7 shows the training performance of Balsa as a function of elapsed time and the number of distinct query plans executed. (The latter is called data/sample efficiency in RL terms, as each execution is an interaction with the environment.)

**Wall-clock efficiency.** Figure 5.7a shows Balsa's wall-clock efficiency during the real execution stage. Balsa starts off several times slower than the experts—this is the performance after bootstrapping from a simple simulator. With just a few hours of learning, Balsa matches the experts' performance (on PostgreSQL: 1.4 hours for JOB, 2.5 hours for JOB Slow, 1.5 hours for TPC-H; ∼0.5 hours faster on CommDB due to its smaller search space). Balsa continues to improve and reaches its peak performance after around 4–5 hours. TPC-H has less room for optimization—it has much fewer joins—so Balsa converges faster.

**Data efficiency.** Figure 5.7b shows data efficiency curves. It takes a few thousand exe-

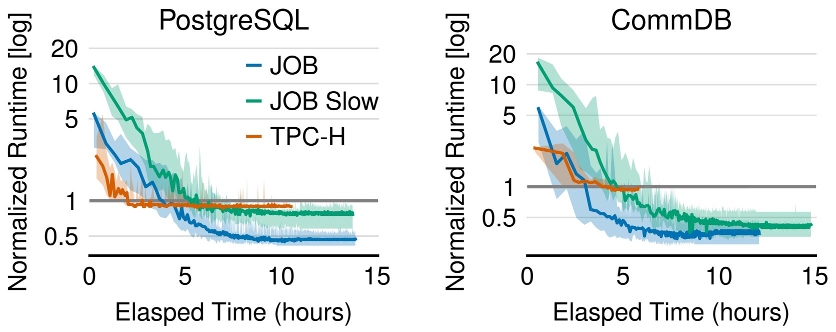
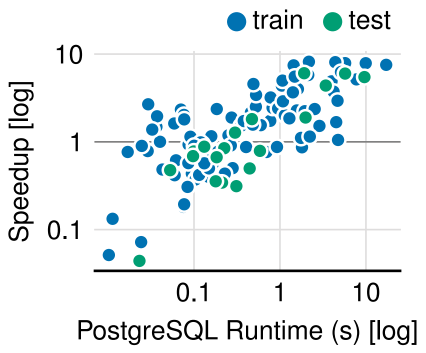
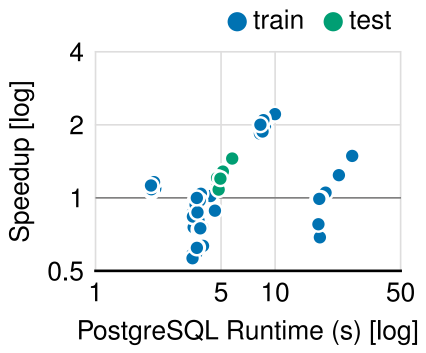


**Figure 5.8:** Wall-clock efficiency, non-parallel training mode.



**(a)** JOB



**(b)** TPC-H

**Figure 5.9:** Breakdown of Balsa's per-query speedups. Speedup of each query (log scale) vs. PostgreSQL expert runtime (log scale).

cutions to reach the experts' performance (on PostgreSQL: 3.2K for JOB, 7.4K for JOB Slow, 0.7K for TPC-H; on CommDB, ∼60% fewer plans are needed). The number of query plans required is higher for workloads where the agent starts with slower performance. Therefore, experiencing more plans helps Balsa improve performance by a greater amount.

**Non-parallel training wall-clock.** Throughout our evaluation, including the discussions above and Figure 5.7, we configure Balsa to use a few query execution nodes per run (average: 2.5 nodes/run) to speed up training. For completeness, Figure 5.8 shows non-parallel training times where each run uses one execution node. In all cases, peak performance is reached within single-digit hours, a comfortable "nightly maintenance" range. The time to match the experts is at most 3 hours slower than that for the parallel mode.

**Sources of speedup.** Figure 5.9 shows Balsa's per-query speedups over PostgreSQL plans. For JOB, Balsa produces better query plans for most queries in both training and testing.

Notably, Balsa considerably speeds up the slowest queries. Slowdowns mostly occur in the queries that are inherently fast to execute, and hence minimally affect the overall runtime. A similar trend holds for TPC-H.

**Summary.** Balsa can bootstrap from a minimal cost model and learn to surpass both an open-source and a commercial expert optimizer. Balsa is efficient to train, needing a few hours to match the experts and thousands of plans to reach its peak performance.

## 5.8.3 Analysis of Design Choices

Next, we analyze the design choices of each major component in Balsa: (1) the initial simulator, (2) the timeout mechanism, (3) exploration strategies, (4) the training scheme, and (5) beam search. In summary, we found all components to positively contribute to Balsa's performance and generalization.

In each experiment, we change one component at a time and hold all other configurations fixed at default values. We then measure each variant's performance on the JOB (random split) workload on PostgreSQL. Default choices are highlighted in bold in each figure.

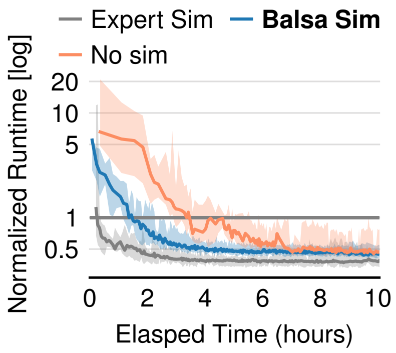
### 5.8.3.1 Impact of the initial simulator

Balsa bootstraps from a minimal simulator. We can consider two alternatives that differ the most from this choice in terms of the amount of prior knowledge:

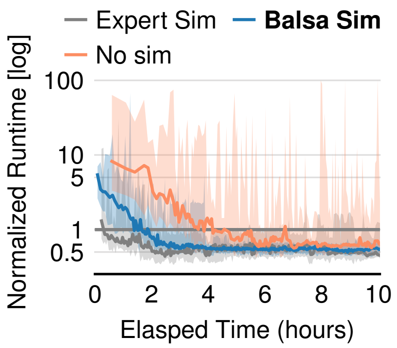
- **Expert Simulator**: the cost model from an expert optimizer, PostgreSQL, which has sophisticated modeling of all physical operators and captures the nuances of its execution engine. (Note that this variant means Balsa uses this cost model as the simulator; it does not represent PostgreSQL's own plans.)
- **Balsa Simulator** (§5.3; $C_{out}$): a minimal cost model that sums up the estimated result sizes of all operators. It has no knowledge about physical operators or the execution engine.
- **No simulator**: skip bootstrapping altogether and initialize the agent from random weights.

Figure 5.10 shows the simulator's impact. We make four observations:

First, simulators with more prior knowledge shorten the time to reach expert performance on training queries (Figure 5.10a). Balsa with an expert simulator needs only ∼0.3 hours of learning to match the expert. Balsa's default simple simulator takes ∼1.4 hours to match, while agents without simulation learning take ∼3.8 hours.



**(a)** Training performance



**(b)** Test performance

**Figure 5.10:** Impact of the initial simulator. (a) Better simulators accelerate learning. (b) Simulation is essential for generalization.

Second, more prior knowledge also leads to slightly better final performance at the end of training (Figure 5.10a). The gap, however, is relatively small. Agents using a minimal simulator mostly catch up with those using an expert simulator.

Third, it is a pleasant surprise that the agents without simulation ("No sim") can finish training. This is enabled by the use of timeouts and safe exploration, which keep the bulk of the learning safe.

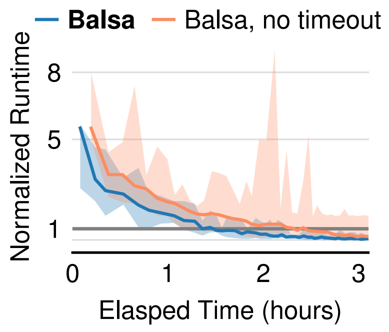
Fourth, *simulation is essential for generalization.* Agents without simulation learning can fail at test time (note the high variance of "No sim" in Figure 5.10b). The unstable performance on test queries occurs despite good training performance, rendering this choice impractical. The instability is caused by randomly initialized agents overfitting the experience collected during the real execution phase, which is limited in quantity (∼700 subplans per iteration, so it takes at least ∼700 iterations to catch up to the 0.5M-plan simulation dataset, assuming each iteration's data is unique).

In summary, bootstrapping from a minimal simulator gives good train and test time performance. Since new execution engines may not have an expert-developed cost model, this approach has the additional benefit of potentially generalizing to new systems and alleviating the human development cost.

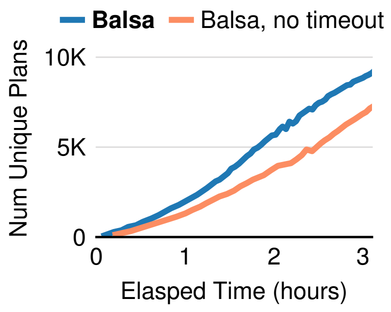
#### 5.8.3.2 Impact of the timeout mechanism

We study the impact of timeouts (§5.4.3), a mechanism critical for real execution learning:

- **Timeout**: early-terminate query plans that have been executing for longer than the current iteration's timeout.



(a) Initial training performance



(b) Number of executed plans

**Figure 5.11:** Impact of the timeout mechanism. (a) Timeouts accelerate learning and prevent spikes. (b) With the same wall-clock time, agents with timeouts execute more plans, improving plan variety.

- **No timeout**: the mechanism is turned off.

With timeouts, agents are expected to save wall-clock time on unpromising plans and potentially learn faster.

Results are presented in Figure 5.11. Timeout agents reach expert performance about 35% faster than no-timeout agents (Figure 5.11a). While both choices lead to similar final performance, there is a pronounced difference in the initial phase of learning. Agents without timeouts may execute expensive query plans, leading to significant spikes. Such regressions are unpredictable: they can happen after the no-timeout agents reaching expert performance.

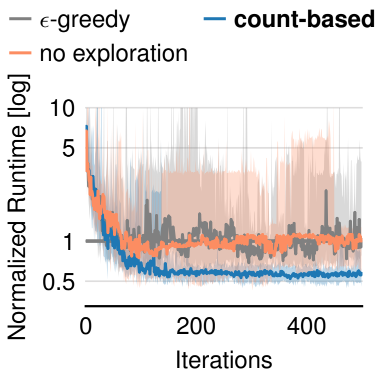
In contrast, agents achieve safe execution when timeout is enabled. The early-terminated plans "nudge" the agents in a different direction to look for more promising plans. Figure 5.11b shows how the saved time is more judiciously spent: with the same wall-clock time, agents with timeouts run more plans, speeding up learning.

Overall, these results show that the timeout mechanism accelerates learning and improves Balsa's plan variety.

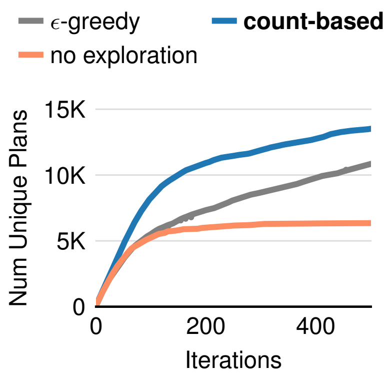
#### 5.8.3.3 Impact of exploration

Exploration exposes RL agents to diverse states, boosting performance and generalization. We compare:

- **Count-based exploration** (§5.5): Balsa's safe exploration method, which chooses the best unseen plan from beam search outputs.



(a) Test performance



(b) Number of unique plans seen

**Figure 5.12:** Impact of exploration. Balsa's count-based safe exploration improves generalization to unseen test queries.

- **$\epsilon$-greedy beam search**: at each step of the search, with a small probability $\epsilon$ the beam is "collapsed" into one state, discarding the rest. The search continues as usual. We chose $\epsilon$ such that about 10% of training queries have random joins injected.
- **No exploration**: no exploration algorithms are used.

Figure 5.12a shows that agents with count-based safe exploration generalize to test queries much better than the other two variants. The better generalization is a result of the higher number of distinct plans experienced (Figure 5.12b). Training performance is omitted for space reasons, where count-based is around 8% and 14% faster than no-exploration and $\epsilon$-greedy beam at convergence, respectively.

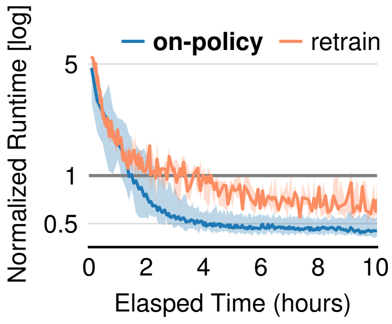
Interestingly, although $\epsilon$-greedy beam search has similar plan diversity to count-based, it is less stable. This is because it contains random joins, which may only lead to low-quality complete plans even when a value network is used to guide the remaining search.

In summary, these results show that safe exploration is non-trivial, and Balsa's count-based method is both simple and effective.

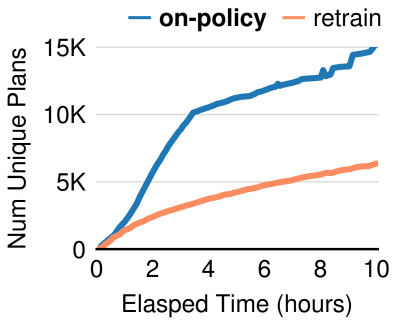
#### 5.8.3.4 Impact of the training scheme

We compare Balsa's on-policy learning to a full retrain scheme used by prior work, Neo [74]:

- **On-policy learning** (§5.4.1): Balsa's training scheme which uses the latest iteration's data to update $V_{\text{real}}$.
- **Retrain**: re-initialize $V_{\text{real}}$ and retrain on the entire experience ($\mathcal{D}_{\text{real}}$) at every iteration. Last iteration's $V_{\text{real}}$ is discarded.

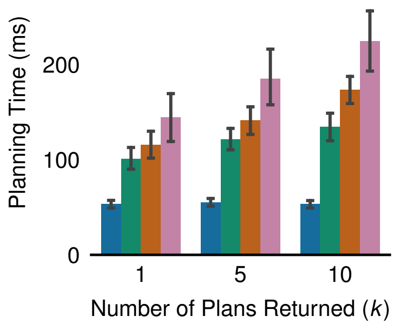


**(a)** Training performance

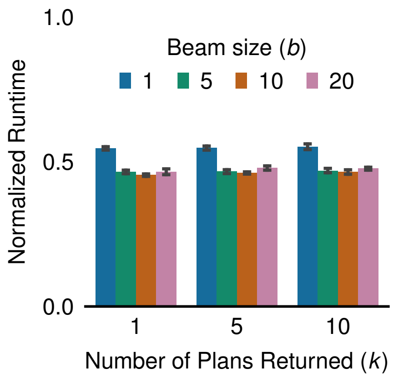


**(b)** Number of unique plans seen

**Figure 5.13:** Impact of the training scheme. (a) On-policy learning accelerates training. (b) Time saved is used towards more exploration.



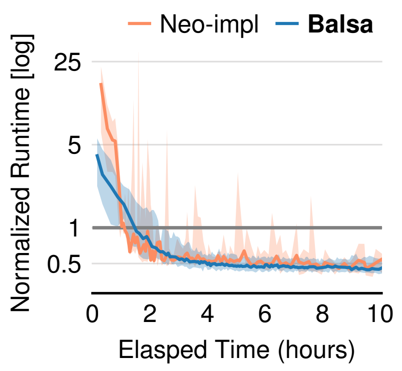
**(a)** Per-query planning time



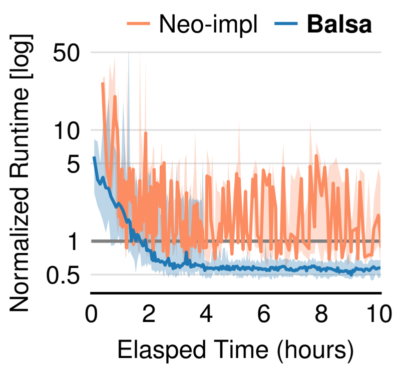
**(b)** Workload runtime

**Figure 5.14:** Impact of search parameters on planning time and performance on JOB test set. Means and standard deviations are shown.

On-policy learning significantly accelerates training, reaching the expert's performance 2.1× faster than retrain agents (Figure 5.13a). Its lead is consistent throughout training. The faster learning is due to on-policy saving time by updating $V_{\text{real}}$ on a constant-size dataset, rather than retraining it on an increasingly larger dataset. The time saved is used towards exploration, i.e., executing more unique plans (Figure 5.13b). Better exploration thus further accelerates learning. On-policy has slightly higher variance due to performing SGD on much less data. However, the slowest on-policy agent (the upper edge of the shading) is still mostly faster than retrain agents.



**(a)** Training performance



**(b)** Test performance

**Figure 5.15:** Comparison with learning from expert demonstrations.

#### 5.8.3.5 Impact of planning time

Balsa performs beam search with beam size $b$ using the value network to generate $k$ complete query plans, and then picks the best plan to execute (during training, the best unexplored plan is picked). Figure 5.14 studies Balsa's planning time and performance of the JOB test queries using various combinations of $b$ and $k$ on a trained checkpoint.

For all settings, the mean per-query planning time is below 250ms. The planner is implemented in Python and thus leaves room for optimization. Using $b = 1$ (where beam search degenerates into greedy search) slightly hurts performance; all other settings produce plans with similar runtime. Hence, Balsa's performance is insensitive to these parameters, and we can flexibly reduce planning time for deployment by using lower values (e.g., $b = 5, k = 1$ speeds up planning time by $2\times$ with no performance drop). We use $b = 20, k = 10$ during training as larger values can help exploration.

### 5.8.4 Comparison with Learning from Expert Demonstrations

We compare Balsa with Neo [74], a recently proposed learned optimizer that relies on PostgreSQL-generated plans—i.e., learning from expert demonstrations. This experiment uses the same setup as §5.8.3 (JOB workload on PostgreSQL). As Neo is not open source, we implement our best-effort reproduction, denoted as "Neo-impl". We make both approaches use identical modeling choices (e.g., architecture, featurizations, beam search), and turn off Balsa's algorithmic components for Neo-impl (bootstrapping from simulation; on-policy learning; exploration; timeout mechanism). One notable difference is that Neo completely resets its model to random weights in each iteration and retrains it on the entire collected experience.

Figure 5.15a shows training performance. At initialization, Balsa is $5\times$ faster than Neo-impl, since simulation learning provides a high state coverage (Table 5.2) as opposed to a limited number of expert demonstrations (one complete plan per query). Balsa remains stable throughout training, as it employs timeouts. Neo-impl experiences performance spikes (note the variance) as it has no mechanism to deal with disastrous plans. These regressions are unpredictable and can occur after hours of training. In terms of training efficiency, Neo-impl's retraining scheme makes it progress increasingly slower as the amount of experience accumulates. Neo-impl spent about 25 hours to finish 100 iterations, whereas Balsa only spent 2.6 hours.

Surprisingly, despite reaching a relatively stable training performance with 5 hours of learning, Neo-impl is still not robust enough to generalize to unseen test queries and suffers from high variance (Figure 5.15b). Its median workload runtime fluctuates between 1–5× slower than the expert and its maximum is up to 10× worse. This failure mode may prohibit this approach from producing reliable models for practical deployment.

In contrast, Balsa is much more robust. Balsa consistently generates faster plans than the expert for unseen queries, with a 2× maximum speedup. Balsa's better generalization is due to a broader state coverage offered by simulation, on-policy learning, and safe exploration (see Figures 5.12 and 5.13).

In sum, Balsa learns faster, achieves safe execution, generalizes better due to simulation and better exploration, while refuting the previously held belief that expert demonstrations are needed [74].

#### 5.8.4.1 Comparison with Bao

Bao [73] is a related approach that assumes an expert optimizer is available. Like Neo, it requires expert demonstrations to train its model. Bao learns to provide a set of hints (e.g., disable hash join) for each query, "steering" the expert optimizer to produce better plans. This is different from Balsa which learns to produce physical plans by itself. Nevertheless, we compare the performance of the query plans generated by Balsa with those by Bao on top of PostgreSQL.

We substantially optimize the Bao source code [6] as follows. First, we turn on an optimization that bootstraps its model from PostgreSQL's expert plans, rather than from a random state. Second, its paper specifies that it trains on the most recent $k = 2000$ experiences, which we found led to highly unstable performance. We thus train Bao on all past experiences, stabilizing convergence.

Table 5.3 shows that Balsa generally matches or outperforms Bao. These results are not surprising: they confirm the finding in the Bao paper that a learned optimizer with *higher degrees of freedom* (action space) can outperform Bao in plan quality on stable workloads.

**Table 5.3:** Balsa vs. Bao: speedups with respect to PostgreSQL.

| | JOB, train | JOB, test | JOB Slow, train | JOB Slow, test |
|---|---|---|---|---|
| Balsa | 2.1× | 1.7× | 1.3× | 1.3× |
| Bao | 1.6× | 1.8× | 1.2× | 1.1× |

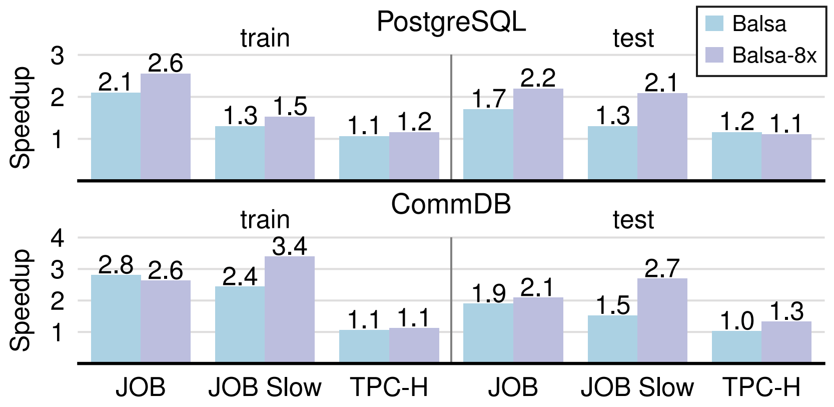


**Figure 5.16:** Enhancing generalization using diversified experiences.

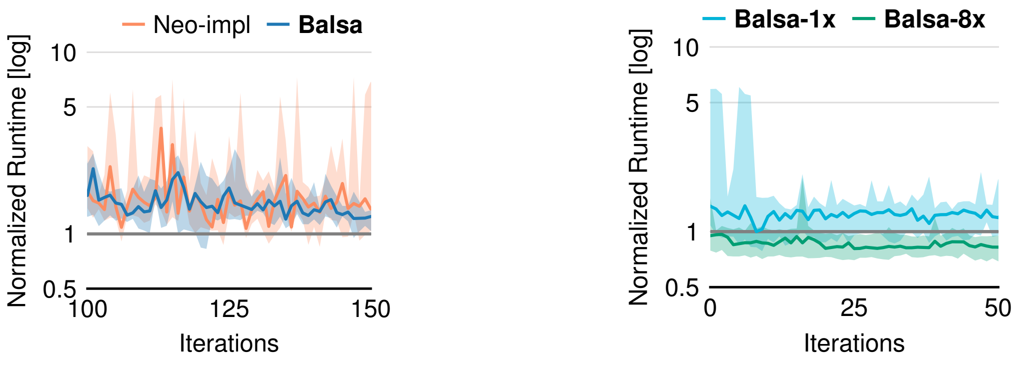
### 5.8.5 Enhancing Generalization

Figure 5.6 already shows that Balsa can generalize to unseen test queries quite well, outperforming experts without ever seeing the test queries. Here, we study *(i)* the benefit of diversified experiences (§5.6), and *(ii)* generalizing to entirely distinct join templates/filters.

**Diversified experiences.** We build diversified experiences for all workloads/engines in Figure 5.6, by merging the data of each main experiment's eight agents. We retrain a new agent on top, referred to as "Balsa-8x"; this process is repeated eight times to control for training variance. (Training is efficient as no query executions are performed.) Figure 5.16 shows the median performance: we observe that Balsa-8x *improves speedups on both training and test queries* in almost all cases, sometimes even by 60–80% (JOB Slow, test).

Improving training speedups is not surprising: a retrained agent can mix-and-match the best plans found by the base agents. Importantly, *test queries see large speedups too without ever being executed* (e.g., on both engines, both JOB splits now have $> 2\times$ test speedups). This is because diversified experiences have highly diverse plans, so more generalizable value networks can be trained on top.

**Queries with entirely new join templates.** We further examine Balsa's generalization



**(a)** Single-agent experiences **(b)** Diversified experiences

**Figure 5.17:** Generalizing to highly distinct join templates: test performance on Ext-JOB, with JOB as the training set. On PostgreSQL.

to difficult unseen queries. First, we split JOB using 4 slowest *templates* (17, 16, 6, 19) as the test set (20 queries) and the rest as the train set. On this new split, Balsa achieves good train and test speedups (1.4×, 1.5×), further confirming its robustness.

Second, we evaluate on Extended JOB (Ext-JOB), a hard generalization workload [74]. It has 24 new queries on the same IMDb dataset, having 2–10 joins and averaging 5 joins per query. These queries are challenging and "out-of-distribution" since they contain *entirely different join templates and predicates* from the original JOB.

Figure 5.17a shows the test performance of Neo-Impl and Balsa on Ext-JOB with the entire 113 JOB queries as the training set. While Balsa is more stable than Neo-impl, neither surpasses the expert on the Ext-JOB test set (although they come close). This confirms that Ext-JOB is a highly challenging generalization workload.

Next, we compare Balsa-8x as described above, with Balsa-1x that retrains on only one agent's data. Surprisingly, in iteration 0, Balsa-8x already matches the expert on the test set (Figure 5.17b). We then allow these agents to learn for 50 more iterations on the training set. Throughout the process, *the agents never train on the Ext-JOB test queries*. Balsa-8x reaches significantly better test set performance on Ext-JOB (20% faster than the expert) than Balsa-1x (which still fails to match the expert). The gain is also consistent. These results show that *diversified experiences* and *further exploration* are valuable strategies to improve generalization to out-of-distribution queries.

## 5.8.6 Behaviors Learned by Balsa

To gain intuition on the behaviors learned by Balsa, we visualize the operator and shape compositions of agent-produced plans over the course of training. Results are shown in

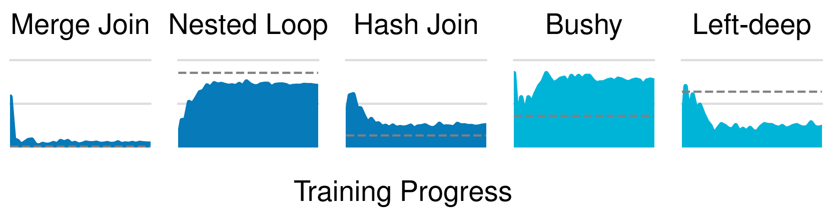


**Figure 5.18:** Balsa's use of join operators (dark blue) and plan shapes (light blue) on JOB. Dashed lines are counts from PostgreSQL plans.

Figure 5.18.

In early stages of training, Balsa quickly learns to reduce the use of operators and shapes that incur high runtimes in the current environment. For example, after 25 iterations, the use of merge joins is kept below 10%. Meanwhile, Balsa starts to prefer more efficient choices. Nested loop joins are preferred since a large portion (85% across iterations) are the efficient indexed variant.

Balsa's preference is novel when compared to the expert, a difference especially pronounced in the plan shapes. This is due to the expert optimizer being one-size-fits-all, while Balsa learns to tailor to the given workload and hardware.

## 5.9 Related Work

**Learned query optimizers.** Balsa is most related to DQ [60] and Neo [74]. DQ offers the insight that the classical components of query optimization—cost estimation and plan enumeration—can be cast as long-term value estimation and planning. All three work follow this formulation by using a learned value network and plan search. Balsa also adopts DQ's use of batched data collection on top of a cost model in our simulation learning. Unlike DQ, Balsa demonstrates fine-tuning entire workloads in real execution.

Neo requires learning from expert demonstrations (PostgreSQL plans) followed by fine-tuning. In contrast, Balsa does *not* learn from an expert optimizer. Lifting this restrictive assumption opens the possibility to automatically learn to optimize in future environments. Balsa differs in three more aspects with important consequences. *(i)* Learning from a simulator fundamentally differs from expert demonstrations. While the latter are inherently limited in quantity and variety (one expert plan per query), simulation allows us to extract a maximal amount of experience, boosting generalization. *(ii)* Balsa addresses the challenge of disastrous and slow plans. *(iii)* Balsa introduces novel techniques (e.g., on-policy learning, timeout as a learning curriculum, safe exploration, diversified experiences), all of which lead to higher efficiency, performance, or robustness. In §5.8.4, we showed that Balsa outperforms the approach of learning from expert demonstrations and is more robust on unseen queries, despite not learning from an expert optimizer.

SkinnerDB [122] is an execution algorithm that learns by trying many left-deep join orders during a query's execution. Both Balsa and SkinnerDB use timeouts to mitigate bad plans but propose substantially different timeout policies. While SkinnerDB must iterate over a set of pre-defined timeouts unrelated to prior executions, Balsa directly uses past plans' latencies as timeouts. Balsa also offers more general capabilities, as it can build bushy plans and assign physical operators, both of which are not supported in SkinnerDB.

**Optimizer assistants.** Many recent proposals use ML to *assist or improve existing optimizers*. Since Leis et al. [64] showed that inaccurate cardinality estimates are most responsible for poor plans, many projects have used ML to improve cardinality estimation [135, 136, 56, 26, 44, 142, 143, 110, 119], thus helping today's optimizers find better plans. The recent work Bao [73] also assists expert optimizers by learning what optimizer flags to set for each query. Different from this line of work, Balsa does not assist an existing optimizer, and tackles learning to optimize precisely assuming no expert optimizers.

**Sim-to-real, timeouts, and caching.** These are general techniques applicable to a range of systems problems. Hilprecht et al. [43] have proposed using sim-to-real to learn high-quality data partitionings and applying timeouts and caching to optimize training. Balsa applies these methods in learned query optimization instead and offers the novel finding that simulation learning improves generalization.

## 5.10 Lessons Learned and Discussions

During the development of Balsa, we have learned a few lessons. We discuss them below.

**Simulation learning boosts generalization.** To our surprise, while Balsa generalizes well to unseen queries, we find that agents without a simulation phase—including those that learn from expert demonstrations—become unstable on new queries (§5.8.3.1, §5.8.4). At first glance, it might be counterintuitive why simulation improves generalization. After all, the simulator we use is a minimal, logical-only cost model that is agnostic to the execution environment. It imparts inaccurate knowledge to the agent that must be corrected.

We believe the reason is the simulation enables Balsa to achieve *a high coverage of the plan space*. During bootstrapping, Balsa trains on thousands of plans per query (Table 5.2), much more than the experiences collected in real execution. Then, in real execution, a bootstrapped agent can update its belief to simultaneously correct much of the simulated knowledge, which can improve generalization. In contrast, agents that learn only from real executions will only see a small set of query plans, which can lead to overfitting.

**Using inaccurate cardinality estimates.** In traditional optimizers, cardinality estimates are known to be highly inaccurate [64], which can lead to poor plans. In Balsa, however, we find an effective use of inaccurate estimates: use them in the simulator. We find that inaccurate estimates can still provide effective simulation.[11] Importantly, Balsa's performance is not overly tied to the simulator—most learning occurs *after* simulation, when Balsa uses real execution to vastly improve over the simulated knowledge (e.g., initial vs. final performance have a 4–40× gap in Figure 5.7). Consistent with prior work [64], we expect better estimates to lead to a better simulator, which would accelerate learning (e.g., "Expert Sim" in Figure 5.10).

**How to better leverage an expert optimizer, if available?** For learning to optimize in a new system, even if a *compatible* expert optimizer (i.e., all operators of the expert are supported by the target engine) exists, prior state-of-the-art [74] proposes bootstrapping only from the expert optimizer's *plans*. We show that this can lead to poor generalization due to the limited amount of demonstrations (§5.8.4). In contrast, Balsa can *better leverage the expert* by bootstrapping from the expert optimizer's *cost model*—a data-rich simulator (see the "Expert Sim" Balsa variant in Figure 5.10). We show that bootstrapping from a cost model significantly improves generalization to new queries (§5.8.3.1), which is a novel finding of this chapter.

## 5.11 Summary

To our knowledge, Balsa is the first approach to show that learning an optimizer without expert demonstrations is both possible and efficient. Balsa learns by iteratively planning a given set of queries, executing them, and learning from their latencies to build better execution plans in the future. To make learning practical, Balsa must avoid disastrous plans that can dramatically hinder learning. We address this key challenge with three simple techniques: bootstrapping from a simulator, safe execution, and safe exploration.

Balsa paves the road towards automatically learning a query optimizer tailored to a workload and a compute environment. New data systems may have execution models [91] or objectives [83] that go beyond our knowledge of query optimization. By learning on its own and not learning from an expert system, Balsa may alleviate the significant optimizer development cost for systems yet to be developed.

[11] We use PostgreSQL's estimates, which have $\sim$100× median errors and up to $10^6\times$ tail errors on JOB [64]. We tried making them even more inaccurate, by dividing them by *random noises* (a median noise factor of 5×), and saw little impact on Balsa's plans.

# Chapter 6

# Conclusion

This dissertation has demonstrated the promise of using modern machine learning to tame the complexity of query optimization. We have shown that ML advances, when enhanced with new systems and ML techniques, can be used to achieve previously unattainable goals:

- Removing decades-old, accuracy-impacting heuristics in cardinality estimation, thereby significantly improving its accuracy compared to prior state of the art;
- Automatically learning to optimize queries by trial-and-error and without expert supervision, to a level that matches or outperforms mature real-world optimizers.

In the context of data systems, the first direction improves query optimizers, while the second opens the possibility of alleviating the complex optimization in future environments and engines. More generally, our results on machine learning for query optimization—a complex yet critical component in data systems—add a compelling proof point to the nascent field of applying machine learning to solve computer systems problems ("ML for Systems").

We conclude this dissertation by reflecting on lessons learned and sketching some potential areas for future work.

## 6.1 Lessons Learned

**When and how should we apply ML to systems problems?** While ML has achieved remarkable successes in many domains in the past ten years (§2.2), ML for systems is a nascent area. Here, we reflect on the question: why is ML effective for the long-standing data systems challenges studied in this dissertation? We offer two points in the hope of forming some amount of guidance for related ML-for-systems problems in the future.

First, the problem nature and the ML method should "fit". In Chapter 1, we discussed that one reason ML is a promising tool for us is its rapid progress in the past ten years, mostly in the form of deep learning. However, the more important contributor to its effectiveness, as we have learned, is to ensure the problem is essentially an ML problem or the problem's nature calls for ML. For example, cardinality estimation has long relied on statistical data summaries, and deep autoregressive modeling is one such new tool that is more powerful than prior methods. In addition, in optimizing queries we inherently would like to pick intermediate subplans to maximize the *long-term goodness* of the final plan, the true metric, rather than the current practice of picking them according to their immediate short-term *costs*, a proxy metric. Optimizing for the long-term goodness *is* the central goal in reinforcement learning, making RL a suitable tool. Thus, one lesson we have learned in this work is to apply ML to systems problems where there is a natural fit.

Second, directly applying a new ML method is often not enough, and new techniques to adapt or enhance it are often required. This has been a recurring theme in this dissertation. In Naru, for example, we developed a new approximate inference algorithm (progressive sampling) and an inference optimization (wildcard skipping) to *enhance* a deep autoregressive model to handle estimating the cardinalities of range predicates. Without these enhancements, the estimator would not be usable. In NeuroCard, we built on top of Naru but had to add systems techniques (join sampling; column factorization) and ML techniques (inference algorithms for joins) to handle join estimation. In Balsa, we saw that using value network-guided tree search—a new RL advance—out-of-the-box is insufficient, as it does not address *safety*: mitigating extremely slow plans when learning in real execution. We thus combined it with simulation-to-reality learning (another RL advance), safe execution via a timeout policy (a classic systems technique that we added), and safe exploration (an ML technique we proposed in this work) to make learning feasible. These examples suggest that ML advances alone may not be the panacea: to make the ML part feasible or efficient in systems problems, expect new techniques to be required.

**RL agents that learn on a real system need to be "safe".** An RL agent must make mistakes to improve, but these mistakes can be highly costly if executions on real systems are involved. For example, an extremely costly query plan produced by an agent may take too long to complete, which means the agent will not get a reward signal to improve for a long time. Meanwhile, the real system busying running the plan may lock up or even increase its use of hardware resources (e.g., a plan that keeps spilling intermediate buffers to the disk), further slowing down the learning. In such settings, therefore, enabling fast data collection (getting a reward signal for agent actions) is a key challenge.

When developing Balsa, we were initially faced with many of these issues, which prompted us to propose and apply several safety mechanisms (sim-to-real; safe execution via a timeout policy; safe exploration). These safety techniques are general and not overly specific to Balsa; thus, we expect they can be applied to and benefit other RL-for-systems problems.

**Act vs. search: using RL to search for interpretable and deployable solutions.** While RL commonly involves an agent *acting* in an environment, we found using RL as an *intelligent search procedure* to be a promising approach in systems problems. Under this alternative view, we can use RL to search (i.e., optimize) for interpretable *solutions*, such as query plans, that can be inspected, modified if desired, and deployed to the downstream system in a white-box manner. For example, Balsa can be used to search for the best plans that it can find for a set of high-value queries, by treating those queries as "training" queries and recording the best plans encountered in training. If the queries are recurring, these high-quality agent plans can then be deployed and reused. In other words, this alternative use seeks the optimal outputs of the agent, and less so the trained agent itself.

Using RL as a search procedure has enjoyed successes in other applications. Examples include searching for the best data structures that perform packet classification in networking [67], the best partitioning layout for a dataset that maximizes the current workload's performance [144], or the best hardware layout for designing a computer chip [77]. In these examples, the RL agent's outputs are interpretable solutions to the underlying problem or canonical inputs to the system (i.e., they have the same form as existing hand- or machine-designed solutions). Deploying such *static* solutions, rather than the agent itself which involves neural network computations, also reduces runtime overheads. Thus, this use of RL may build more trust with systems developers and operators, leading to a promising path to increasingly adopt RL in systems problems.

**Leveraging a powerful neural summary of data.** For the first time, deep autoregressive models—combining next-token prediction and a modern neural network architecture—give us the ability to approximate high-dimensional data distributions with high accuracy. Naru and NeuroCard, with their new techniques discussed above, successfully transform these models into powerful *neural summaries of tabular data*, which can then be used to, for example, perform cardinality estimation.

Since the summary condenses information, we can extract approximate information out of it. We thus believe this new capability developed in this work can be leveraged in other data problems too, especially if an accurate understanding of tables and columns—and their correlations—is required. For example, ReStore [42] extends Naru to schema-structured autoregressive models to support *data imputation and completion*. SAM [139] adapts NeuroCard's modeling and inference ideas to perform *synthetic database generation*, where the neural summary is sampled to produce simulated, high-fidelity databases, which can be used to accelerate the development and testing of data systems. Other possible tasks include using a neural summary to run approximate query processing or to model non-tabular data [132].

## 6.2 Future Work

**Extensions to Balsa and learning to optimize queries.** Here, we sketch several possible ideas of extensions to Balsa, and, more generally, to the direction of automatically learning to optimize queries:

- **Pushing the scale.** Since Balsa was evaluated in single-machine settings, an interesting next step is to test how far we can push Balsa to work on a larger data scale, possibly running on a distributed engine. Such an engine may have new operators, e.g., distributed joins or shuffles, enriching the action space and the complexity of the learning task. As data size grows and the engine uses more costly operators, bad plans would be more expensive to run, so Balsa's safety mechanisms may become even more important. Another possibility to handle a larger database is to first train on a smaller, sampled version of the data, which speeds up the execution of agent plans, and then transfer the trained agent to the full database.

- **Using self-supervised cardinality estimators in Balsa.** Balsa uses basic cardinality estimates in two places: in query features where estimates represent a query's base-table predicates, and in the simulator whose cost model needs estimates as inputs.[1] It would be interesting to use a self-supervised cardinality estimator—such as NeuroCard—for these purposes, replacing our current choice of histogram-based estimates from a real system.

  This can have two benefits. First, NeuroCard should produce much more accurate estimates, which would make the simulator more accurate; as we showed in §5.8, better simulation accelerates learning. Second, more fundamentally, removing the reliance on estimates produced from a real system would be a major next step towards learning to optimize queries *entirely from scratch*.

- **Making Balsa more robust.** Balsa can be made more robust in two aspects. First, there is some variance in performance across random seeds (see Figure 5.7 and other performance curves in §5.8). A certain amount of variance may be unavoidable and is perhaps even desirable—indeed, variance implies that the agent is sufficiently exploratory, and differently initialized agents can accumulate diverse experiences, which, as we showed, can be leveraged. However, it would be important to control or reduce the worst-case performance across random seeds, which would enable any training run to produce a reasonably high-quality agent.

  The second aspect of robustness is generalization, i.e., how well the agent optimizes new queries unseen in training. While our ablation studies in §5.8 show that many mechanisms in Balsa improve generalization, it would be interesting to see how much we can improve it further: e.g., can the agent optimize any plausible query well?

- **Using Balsa for a new engine that has no optimizer or has only a heuristic optimizer.** For new data systems that currently have no optimizer or only have a heuristic one, it would be interesting to train Balsa on that engine to produce plans, essentially using it as a learned cost-based optimizer. One question to overcome here, if

[1] §5.8.3.1 presents evidence that entirely removing the simulation *could* work, at the cost of slower learning and higher variance in generalization performance. If these issues were mitigated with new techniques, then Balsa's reliance on simulation, and thus the cardinality estimates used there, could be removed.

the system truly has no optimizer built, is evaluation: how do we know Balsa-produced plans are good? Potential choices of new data systems to carry out this study include, as cited in Chapter 1, a streaming DBMS [76] and dataframe systems [91], as well as engines that process graph data [100] or engines that run SQL on GPUs [20, 39].

- **Using Balsa to search for optimal plans for recurring workloads.** As discussed above (§6.1), Balsa as an RL agent can be used to search for optimal plans for high-value, perhaps recurring, queries. These plans can then be deployed for repeated execution. In this use case, we can presumably trade longer training time for better final plans. The idea of diversified experiences (§5.6) would be useful here to achieve higher quality by performing more computations.

- **Leveraging historical execution plans.** If an organization has collected historical execution plans and associated information (e.g., total runtime of each plan, per-operator runtimes), one can train a value network on this data. This value network would be *limited*, as it does not cover the plan space well, lacking knowledge about non-executed plans. Thus, the question here is how to integrate this offline, off-policy training with Balsa's RL training. The availability of historical plans implies there is an existing expert (or at least baseline) optimizer. Therefore, one possibility is to perform simulation training first using the expert's cost model (recall the result that a better simulator accelerates learning), then train on off-policy historical logs, followed by an optional phase of learning in real execution. The second phase may already correct much of the knowledge learned in simulation (e.g., cost-to-latency corrections) to produce a reasonably competent agent. The optional real execution learning would cover the plan space better, enhancing the agent.

**Extensions to self-supervised cardinality estimation.** Naru and NeuroCard have demonstrated the promise of autoregressive models as cardinality estimators. Their capability of learning the entire joint data distribution, with all conditionals learned, is unique among modern generative models. We here list a few extensions to this line of work that can improve their capabilities, space efficiency, runtime efficiency, and beyond.

- **Handling large domains with lower space and higher precision.** Naru and NeuroCard use embeddings to discretize column values into tokens (§3.4.2), and the trainable embedding matrix for each column takes space linear in the column's domain size. Memory usage is therefore quite high for large domains, especially for categorical columns with a large number of distinct values. While column factorization (§4.5) mitigates this issue by trading off some accuracy, there may be other ideas to investigate. The NLP community, for example, commonly utilizes a small fixed-size vocabulary [108] to tokenize text inputs. It may be interesting to come up with a similar "universal column vocabulary" for tabular data, which would reduce memory usage as well as potentially generalize to rare or new values better.

- **Optimizing runtime efficiency.** To ease deployment of this approach, further optimizing inference runtime is important. Fortunately, there are several mature techniques to leverage here, such as quantization (use a lower number of bits in model weights, rather than 32-bit floating points), optimizing the inference algorithm (progressive sampling) by, e.g., implementing it entirely in a native language or in CUDA rather than in Python, using just-in-time compilation, etc. While important, training time is less of a concern and can be controlled by tuning many existing knobs (e.g., training epochs, in conjunction with learning rate schedules).

- **Pushing the boundary of supported schemas and predicates.** Another direction is to extend the capability of Naru and NeuroCard to handle more complex scenarios. For example, better handling of schema graphs with a variety of shapes [37], a larger number of tables in the schema, cyclic join queries, and better support for string columns and string predicates [68] are interesting areas of future work.

- **Investigating other self-supervised generative models.** Naru and NeuroCard pioneered the use of modern generative models in cardinality estimation, using deep autoregressive models as the key tool. Our estimators convincingly show that they capture data correlations much better than (query-)supervised learning and older density models. The field of generative modeling has been growing rapidly in recent years (§2.2), however, which means newer tools from the deep learning community may further improve our results in model quality, runtime efficiency, and model size. For example, normalizing flows [88] have been recently applied in cardinality estimation with good results on datasets with a few columns [134]. Normalizing flows, if shown to better handle tabular data of larger dimensionality, can potentially be combined with NeuroCard's join support to further push the boundary of join cardinality estimation.

# Bibliography

[1] Andrew Adams et al. “Learning to optimize halide with tree search and random programs”. In: *ACM Transactions on Graphics (TOG)* 38.4 (2019), pp. 1–12.

[2] Ilge Akkaya et al. “Solving rubik’s cube with a robot hand”. In: *arXiv preprint arXiv:1910.07113* (2019).

[3] OpenAI: Marcin Andrychowicz et al. “Learning dexterous in-hand manipulation”. In: *The International Journal of Robotics Research* 39.1 (2020), pp. 3–20.

[4] Thomas Anthony, Zheng Tian, and David Barber. “Thinking Fast and Slow with Deep Learning and Tree Search”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS’17. Long Beach, California, USA, 2017, pp. 5366–5376. ISBN: 9781510860964.

[5] Peter Bailis, Joseph M Hellerstein, and Michael Stonebraker. *Readings in Database Systems, 5th Edition*. http://www.redbook.io/.

[6] *Bao source code*. https://github.com/learnedsystems/BaoForPostgreSQL. 2020.

[7] Christopher Berner et al. “Dota 2 with large scale deep reinforcement learning”. In: *arXiv preprint arXiv:1912.06680* (2019).

[8] Tom Brown et al. “Language models are few-shot learners”. In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[9] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. “STHoles: A Multidimensional Workload-aware Histogram”. In: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’01. Santa Barbara, California, USA: ACM, 2001, pp. 211–222. ISBN: 1-58113-332-4.

[10] Kaushik Chakrabarti et al. “Approximate query processing using wavelets”. In: *The VLDB Journal* 10.2-3 (2001), pp. 199–223.

[11] Surajit Chaudhuri. “An overview of query optimization in relational systems”. In: *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 1998, pp. 34–43.

[12] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. “On random sampling over joins”. In: *ACM SIGMOD Record*. Vol. 28. 2. ACM. 1999, pp. 263–274.

[13] Chungmin Melvin Chen and Nick Roussopoulos. "Adaptive Selectivity Estimation Using Query Feedback". In: *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*. SIGMOD '94. Minneapolis, Minnesota, USA: ACM, 1994, pp. 161–172. ISBN: 0-89791-639-5.

[14] Mark Chen et al. "Generative pretraining from pixels". In: *International conference on machine learning*. PMLR. 2020, pp. 1691–1703.

[15] C Chow and Cong Liu. "Approximating discrete probability distributions with dependence trees". In: *IEEE transactions on Information Theory* 14.3 (1968), pp. 462–467.

[16] Sophie Cluet and Guido Moerkotte. "On the complexity of generating optimal left-deep processing trees with cross products". In: *International Conference on Database Theory*. Springer. 1995, pp. 54–67.

[17] *Commit history of the PostgreSQL optimizer*. https://github.com/postgres/postgres/commits/master/src/backend/optimizer/.

[18] Graham Cormode et al. "Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches". In: *Foundations and Trends in Databases* 4.1-3 (2011), pp. 1–294. ISSN: 1931-7883.

[19] Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[20] *dask-sql*. https://dask-sql.readthedocs.io/en/latest/.

[21] *DeepDB source code*. https://github.com/DataManagementLab/deepdb-public.

[22] 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.

[23] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186.

[24] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. "Density estimation using Real NVP". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. 2017.

[25] Conor Durkan and Charlie Nash. "Autoregressive Energy Machines". In: *Proceedings of the 36th International Conference on Machine Learning*. Vol. 97. Proceedings of Machine Learning Research. Long Beach, California, USA: PMLR, June 2019, pp. 1735–1744.

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

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

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

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

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

[31] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in neural information processing systems* 27 (2014).

[32] Goetz Graefe. "The cascades framework for query optimization". In: *IEEE Data Eng. Bull.* 18.3 (1995), pp. 19–29.

[33] Goetz Graefe and David J DeWitt. "The EXODUS optimizer generator". In: *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*. 1987, pp. 160–172.

[34] Goetz Graefe and William J McKenna. "The volcano optimizer generator: Extensibility and efficient search". In: *Proceedings of IEEE 9th international conference on data engineering*. IEEE. 1993, pp. 209–218.

[35] Geoffrey Grimmett, David Stirzaker, et al. *Probability and random processes*. Oxford university press, 2001.

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

[37] Yuxing Han et al. "Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation". In: *VLDB* 15.4 (2022). ISSN: 2150-8097.

[38] Trevor Hastie, Robert Tibshirani, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer, 2009.

[39] *HeavyDB (formerly OmniSciDB)*. https://heavyai.github.io/heavydb/.

[40] 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.

[41] Joseph M Hellerstein, Michael Stonebraker, and James Hamilton. "Architecture of a database system". In: *Foundations and Trends® in Databases* 1.2 (2007), pp. 141–259.

[42] Benjamin Hilprecht and Carsten Binnig. "ReStore-Neural Data Completion for Relational Databases". In: *Proceedings of the 2021 International Conference on Management of Data*. 2021, pp. 710–722.

[43] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. "Learning a Partitioning Advisor for Cloud Databases". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 143–157. ISBN: 9781450367356.

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

[45] Jonathan Ho et al. "Flow++: Improving flow-based generative models with variational dequantization and architecture design". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 2722–2730.

[46] *How We Built a Cost-Based SQL Optimizer*. https://www.cockroachlabs.com/blog/building-cost-based-sql-optimizer/.

[47] Chin-Wei Huang et al. "Neural autoregressive flows". In: *International Conference on Machine Learning*. PMLR. 2018, pp. 2078–2087.

[48] Yannis E Ioannidis and Stavros Christodoulakis. "On the propagation of errors in the size of join results". In: *Proceedings of the 1991 ACM SIGMOD International Conference on Management of data*. 1991, pp. 268–277.

[49] Matthias Jarke and Jurgen Koch. "Query optimization in database systems". In: *ACM Computing surveys (CsUR)* 16.2 (1984), pp. 111–152.

[50] Raghav Kaushik et al. "Synopses for query optimization: A space-complexity perspective". In: *ACM Transactions on Database Systems (TODS)* 30.4 (2005), pp. 1102–1127.

[51] Martin Kiefer et al. "Estimating join selectivities using bandwidth-optimized kernel density models". In: *PVLDB* 10.13 (2017), pp. 2085–2096.

[52] 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.

[53] Diederik P. Kingma and Prafulla Dhariwal. "Glow: Generative flow with invertible 1x1 convolutions". In: *Advances in Neural Information Processing Systems*. 2018, pp. 10215–10224.

[54] Diederik P. Kingma and Max Welling. "Auto-Encoding Variational Bayes". In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. 2014.

[55] Diederik P Kingma, Max Welling, et al. "An introduction to variational autoencoders". In: *Foundations and Trends® in Machine Learning* 12.4 (2019), pp. 307–392.

[56] 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.

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

[58] 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.

[59] 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.

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

[61] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[62] Yann LeCun et al. "Backpropagation applied to handwritten zip code recognition". In: *Neural computation* 1.4 (1989), pp. 541–551.

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

[64] Viktor Leis et al. "How good are query optimizers, really?" In: *PVLDB* 9.3 (2015), pp. 204–215.

[65] 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.

[66] 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.

[67] Eric Liang et al. "Neural packet classification". In: *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 256–269.

[68] Eric Liang et al. "Variable skipping for autoregressive range density estimation". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 6040–6049.

[69] Henry Liu et al. "Cardinality estimation using neural networks". In: *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*. IBM Corp. 2015, pp. 53–59.

[70] Guy Lohman. *Is query optimization a "solved" problem?* `https://wp.sigmod.org/?p=1075`. 2014.

[71] M. Heimel. *Bitbucket repository, feedback-kde*. `https://bitbucket.org/mheimel/feedback-kde`. [Online; accessed March, 2019]. 2019.

[72] 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.

[73] Ryan Marcus et al. "Bao: Making Learned Query Optimization Practical". In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD/PODS '21. Virtual Event, China: Association for Computing Machinery, 2021, pp. 1275–1288. ISBN: 9781450383431.

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

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

[76] *Materialize*. https://materialize.com/.

[77] Azalia Mirhoseini et al. "A graph placement methodology for fast chip design". In: *Nature* 594.7862 (2021), pp. 207–212.

[78] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[79] Philipp Moritz et al. "Ray: A distributed framework for emerging AI applications". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 561–577.

[80] *MSCN source code*. https://github.com/andreaskipf/learnedcardinalities.

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

[82] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[83] Arjun Narayan. *Materialize: Roadmap to Building a Streaming Database on Timely Dataflow*. https://materialize.com/blog-roadmap/. 2020.

[84] *Naru source code*. github.com/naru-project/naru.

[85] 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.

[86] Aaron van den Oord, Oriol Vinyals, and koray kavukcuoglu koray. "Neural Discrete Representation Learning". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017.

[87] Jennifer Ortiz et al. "Learning State Representations for Query Optimization with Deep Reinforcement Learning". In: *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*. DEEM'18. Houston, TX, USA: ACM, 2018, 4:1–4:4. ISBN: 978-1-4503-5828-6.

[88] George Papamakarios et al. "Normalizing Flows for Probabilistic Modeling and Inference." In: *J. Mach. Learn. Res.* 22.57 (2021), pp. 1–64.

[89] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. "QuickSel: Quick Selectivity Learning with Mixture Models". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 1017–1033. ISBN: 9781450367356.

[90] Matthew Perron et al. "How I Learned to Stop Worrying and Love Re-optimization". In: *35th IEEE International Conference on Data Engineering, ICDE 2019*. 2019.

[91] Devin Petersohn et al. "Towards Scalable Dataframe Systems". In: *Proc. VLDB Endow.* 13.12 (July 2020), pp. 2033–2046. ISSN: 2150-8097.

[92] *pg_hint_plan*. https://github.com/ossc-db/pg_hint_plan.

[93] Stanislas Polu and Ilya Sutskever. "Generative language modeling for automated theorem proving". In: *arXiv preprint arXiv:2009.03393* (2020).

[94] 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.

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

[96] 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.

[97] Alec Radford et al. *Language models are unsupervised multitask learners*. https://openai.com/blog/better-language-models. 2019.

[98] Ali Razavi, Aaron Van den Oord, and Oriol Vinyals. "Generating diverse high-fidelity images with vq-vae-2". In: *Advances in neural information processing systems* 32 (2019).

[99] Anthony G Read. "DeWitt clauses: Can we protect purchasers without hurting Microsoft". In: *Rev. Litig.* 25 (2006), p. 387.

[100] *RelationalAI*. https://relational.ai/.

[101] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[102] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[103] 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.

[104] Julian Schrittwieser et al. "Mastering atari, go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (2020), pp. 604–609.

[105] David W Scott. *Multivariate Density Estimation: Theory, Practice, and Visualization.* John Wiley & Sons, Inc., 1992.

[106] Terrence J Sejnowski. *The deep learning revolution.* MIT press, 2018.

[107] 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.

[108] 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.

[109] Srinath Shankar et al. "Query optimization in microsoft SQL server PDW". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data.* 2012, pp. 767–776.

[110] Suraj Shetiya et al. "Astrid: accurate selectivity estimation for string predicates using deep learning". In: *Proceedings of the VLDB Endowment* 14.4 (2020), pp. 471–484.

[111] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144.

[112] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. ISSN: 1476-4687.

[113] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550.7676 (Oct. 2017), pp. 354–359. ISSN: 1476-4687.

[114] Mohamed A Soliman et al. "Orca: a modular query optimizer architecture for big data". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data.* 2014, pp. 337–348.

[115] *Spark SQL.* https://spark.apache.org/sql/.

[116] Utkarsh Srivastava et al. "ISOMER: Consistent histogram construction using query feedback". In: *22nd International Conference on Data Engineering (ICDE'06).* IEEE. 2006, pp. 39–39.

[117] State of New York. *Vehicle, snowmobile, and boat registrations.* https://catalog.data.gov/dataset/vehicle-snowmobile-and-boat-registrations. [Online; accessed March 1st, 2019]. 2019.

[118] Michael Stillger et al. "LEO-DB2's learning optimizer". In: *VLDB.* Vol. 1. 2001, pp. 19–28.

[119] 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.

[120] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[121] Josh Tobin et al. "Domain randomization for transferring deep neural networks from simulation to the real world". In: *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE. 2017, pp. 23–30.

[122] Immanuel Trummer et al. "SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning". In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. New York, NY, USA: ACM, 2019, pp. 1153–1170. ISBN: 978-1-4503-5643-5.

[123] 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.

[124] Kostas Tzoumas, Amol Deshpande, and Christian S Jensen. "Lightweight graphical models for selectivity estimation without independence assumptions". In: *PVLDB* 4.11 (2011), pp. 852–863.

[125] Benigno Uria, Iain Murray, and Hugo Larochelle. "A Deep and Tractable Density Estimator". In: *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*. ICML'14. Beijing, China: JMLR.org, 2014, pp. I-467–I-475.

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

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

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

[129] Todd L. Veldhuizen. "Triejoin: A Simple, Worst-Case Optimal Join Algorithm". In: *ICDT*. 2014.

[130] Stratis D Viglas, Jeffrey F Naughton, and Josef Burger. "Maximizing the output rate of multi-way join queries over streaming information sources". In: *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment. 2003, pp. 285–296.

[131] Oriol Vinyals et al. "Grandmaster level in StarCraft II using multi-agent reinforcement learning". In: *Nature* 575.7782 (2019), pp. 350–354.

[132] Dimitri Vorona et al. "DeepSPACE: Approximate geospatial query processing with deep learning". In: *Proceedings of the 27th ACM SIGSPATIAL international conference on advances in geographic information systems*. 2019, pp. 500–503.

[133] Florian Waas and Arjan Pellenkoft. "Join order selection (good enough is easy)". In: *British National Conference on Databases*. Springer. 2000, pp. 51–67.

[134] Jiayi Wang et al. "FACE: a normalizing flow based cardinality estimator". In: *Proceedings of the VLDB Endowment* 15.1 (2021), pp. 72–84.

[135] Xiaoying Wang et al. "Are We Ready for Learned Cardinality Estimation?" In: *Proc. VLDB Endow.* 14.9 (May 2021), pp. 1640–1654. ISSN: 2150-8097.

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

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

[138] 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.

[139] Jingyi Yang et al. "SAM: Database Generation from Query Workloads with Supervised Autoregressive Models". In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, pp. 1542–1555.

[140] Zhilin Yang et al. "XLNet: Generalized Autoregressive Pretraining for Language Understanding". In: *Advances in Neural Information Processing Systems*. Vol. 32. 2019.

[141] Zongheng Yang et al. "Balsa: Learning a Query Optimizer Without Expert Demonstrations". In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD/PODS '22. Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 931–944. ISBN: 9781450392495.

[142] Zongheng Yang et al. "Deep Unsupervised Cardinality Estimation". In: *Proceedings of the VLDB Endowment*. Vol. 13. 3. VLDB Endowment, 2019, pp. 279–292.

[143] Zongheng Yang et al. "NeuroCard: One Cardinality Estimator for All Tables". In: *Proceedings of the VLDB Endowment*. Vol. 14. 1. VLDB Endowment, 2021, pp. 61–73.

[144] 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.

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

[146] Jingren Zhou et al. "SCOPE: parallel databases meet MapReduce". In: *The VLDB Journal* 21.5 (2012), pp. 611–636.