Learned Query Optimizers

Suggested Citation: Bolin Ding, Rong Zhu and Jingren Zhou (2023), "Learned Query Optimizers", : Vol. xx, No. xx, pp 1–18. DOI: 10.1561/XXXXXXXXX.

Bolin Ding
Alibaba Group
bolin.ding@alibaba-inc.com

Rong Zhu
Alibaba Group
red.zr@alibaba-inc.com

Jingren Zhou
Alibaba Group
jingren.zhou@alibaba-inc.com

now
the essence of knowledge
Boston — Delft

Contents

Learned Query Optimizers

Bolin Ding[1], Rong Zhu[2] and Jingren Zhou[3]

[1]*Alibaba Group; bolin.ding@alibaba-inc.com*
[2]*Alibaba Group; red.zr@alibaba-inc.com*
[3]*Alibaba Group; jingren.zhou@alibaba-inc.com*

ABSTRACT

This survey presents recent progresses on using machine learning techniques to improve query optimizers in database systems. Centering around a generic paradigm of *learned query optimizers*, this survey covers several lines of efforts on rebuilding or aiding important components in query optimizers (*i.e.*, *cardinality estimators*, *cost models*, and *plan enumerators*) with machine learning. We introduce some important machine learning tools developed recently, which are useful for query optimization, and how they are adapted for sub-tasks of query optimization. This survey is for readers who are already familiar with query optimization and are eager to understand what machine learning techniques can be helpful and how to apply them with examples and necessary details, or for machine learning researchers who want to expand their research agendas to helping database systems with machine learning techniques. Some open research challenges are also discussed with the goal of making learned query optimizers truly applicable in production.

Bolin Ding, Rong Zhu and Jingren Zhou (2023), “Learned Query Optimizers”, : Vol. xx, No. xx, pp 1–18. DOI: 10.1561/XXXXXXXXX.

1

Introduction

1.1 Basics of Query Optimization

Query optimizer plays one of the most important roles in database systems. It aims to select an efficient execution plan for a query written in a declarative language, *e.g.*, SQL. Traditional cost-based query optimizers (Selinger *et al.*, 1979; Graefe and McKenna, 1993; Graefe, 1995) find the plan with the minimum estimated *cost* for the given query.

Let's start with some notations that will be used throughout this survey. A relational database $\mathbb{D}$ consists of a set of base relations (tables), $\{\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_{|\mathbb{D}|}\}$. A query q accesses and manipulates data in the database via relational operations, *e.g.*, *select*, *project*, *join*, and *aggregate*. There are usually a large number of ways to process a query q, called *physical query execution plans* (denoted as P) or *plans* for short in the rest of this survey, with different choices of *join ordering* (which relations are joined first), *join operators* (*e.g.*, hash join $\bowtie_{\mathtt{H}}$ and indexed nested loop join $\bowtie_{\mathtt{INL}}$), and *access paths* (different ways to retrieve tuples from relations, *e.g.*, index seek `IdxSeek` and sequential scan `SeqScan`). For example, to process the query $q = \mathbf{R} \bowtie \mathbf{S} \bowtie \mathbf{T}$,

$$P = (\mathtt{IdxSeek}(\mathbf{R}) \bowtie_{\mathtt{INL}} \mathtt{SeqScan}(\mathbf{S})) \bowtie_{\mathtt{H}} \mathtt{SeqScan}(\mathbf{T}) \tag{1.1}$$

is a plan which joins relations **R** and **S** first with an indexed nested loop join and then joins the result of $\mathbf{R} \bowtie \mathbf{S}$ with **T** with a hash join.

For a query q, let $\mathbb{P}(q)$ be the set of all valid plans. The goal of query optimization is to select the most "efficient" plan P^* from $\mathbb{P}(q)$.

A *cost model* in a cost-based query optimizer (*e.g.*, Selinger *et al.*, 1979) measures the "efficiency" of a plan in terms of the execution latency or other user-specified metrics about resource consumption for the plan to be executed. The cost estimates derived from cost models are in forms of formulas with cardinalities of sub-queries as variables as well as some magic constant numbers to approximate the actual execution latency of the plan. These formulas and magic constant numbers depend on the algorithmic complexities and implementations of physical operators (*e.g.*, various join algorithms). The cardinalities of sub-queries are the sizes of inputs to these physical operators and are unknown before a query is executed. Thus, their estimates are obtained with *cardinality estimators* and fed into the cost model.

A *plan enumerator* is a cost-based search algorithm that explores the plan space and aims to find the one with the minimal (estimated) cost based on, *e.g.*, transformation rules or dynamic programming.

Figure 1.1 (excluding the blue parts) gives an overview about how the three components, cost model, cardinality estimator, and plan enumerator, work together in a query optimizer.

While an obvious challenge in building a query optimizer is that the size of $\mathbb{P}(q)$ is exponential in the number of relations involved in q and the number of operator types, more uncertainty comes from the traditional cost model which depends on cardinality estimates for sub-queries, and quantitative models for costing query processing operators. Various heuristics and assumptions are essential in deriving these cardinality/cost estimates. For example, independence between attributes across relations is assumed and utilized for estimating cardinalities of joins of multiple relations (Tzoumas *et al.*, 2011; Leis *et al.*, 2015); magic constant numbers are prevalent in cost models, and they are often calibrated and tuned over years to ensure that the estimated cost matches the plan's performance well empirically, under certain system and hardware configurations though. It has been realized that such heuristics and assumptions are not always reliable for varying

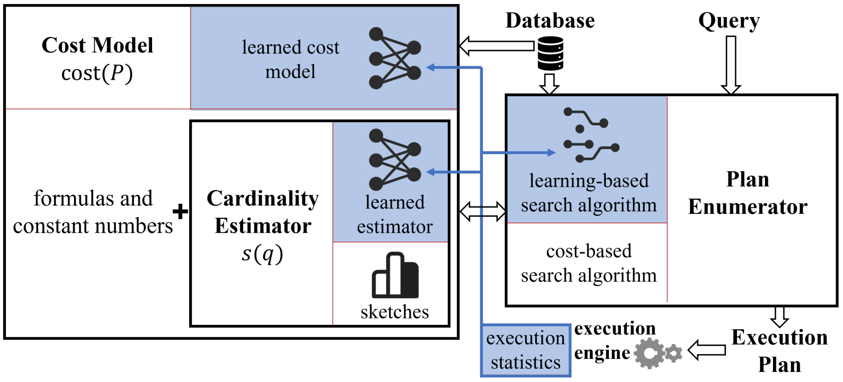

Figure 1.1: Overview of (learned) query optimizers.

data distributions (especially on skewed and correlated data) or system configurations. As a result, cost models may produce significant errors and the plan generated by the traditional query optimizer may have poor quality (Doraiswamy *et al.*, 2008; Han *et al.*, 2021).

1.2 Why a Learned Optimizer is Possible

There are a recent line of efforts to assist or rebuild these components in query optimizers with machine learning models, which are trained on a specific dataset and "previous experience" collected from executing queries in the same or historical workloads. Such attempts date back to 2000s, *e.g.*, DB2's LEarning Optimizer Leo (Stillger *et al.*, 2001).

From the perspective of machine learning and optimization, the tasks tackled by cardinality estimator and cost model are *regression problems* (predicting cardinalities and costs of sub-queries and plans, respectively) and the one by plan enumerator is a *decision-making problem* (finding the best execution plan). With the recent progresses on deep models (*e.g.*, Mou *et al.*, 2016; Vaswani *et al.*, 2017) and deep reinforcement learning (*e.g.*, Sutton and Barto, 2018), we have more powerful tools for these two types of tasks.

For example, an execution plan for a SQL query is a tree structure representing the join order with each node in the tree specifying a physical operator and its two children specifying the two input relations.

From the perspective of machine learning, it is non-trivial to map the plans with varying sizes into a regularized feature space while encoding both the plans' structural and node-wise information. The *tree convolution network* (Mou *et al.*, 2016) and the *attention mechanism* (Vaswani *et al.*, 2017) are two tools (though invented for different purposes) that are able to featurize such complex objects and judiciously utilize their structural information for the prediction task.

Specifically, two types of distributions are important for selecting efficient execution plans: i) data distributions over single and multiple relations (*e.g.*, deciding the join sizes), and ii) joint distribution over relations and query workloads (*e.g.*, deciding the selectivity of predicates). Traditional query optimizers rely on histograms and samples to approximate distributions in i) and ii) (refer to, *e.g.*, the survey by Cormode *et al.*, 2012) for the purposes of cardinality and cost estimation. Machine learning models trained on the targeting datasets and workloads may serve as their replacements, and indeed, the models need to be continuously updated when datasets and workloads are dynamic.

1.3 A Generic Paradigm of Learned Query Optimizers

Figure 1.1 illustrates how the three major components (*i.e.*, cost model, cardinality estimator, and plan enumerator) in a query optimizer can be replaced or enhanced with machine learning models (the blue parts). Modeling more complex and high-dimensional data-query distributions and utilizing feedback/statistics from query executions are where the opportunities lie for these machine-learned counterparts to further improve the performance of query optimizers. To this end, we need to collect training data for these models, from both the databases and the execution engine that processes the query workloads, and organize the training data according to the goals of different models (in *learned cardinality estimator*, *learned cost model*, and *learning-based search algorithm*). Most previous works on learning to optimize queries do not rebuild the whole optimizer. Instead, they focus on one or multiple of these machine-learned counterparts, without a clear separation between different components (especially in reinforcement learning), and integrate them into a traditional query optimizer in a holistic way.

• *From sketches to learned cardinality estimator.* For the task of cardinality estimation for (sub-)queries, there are two types of machine-learning based approaches, *data driven estimator* and *data-query jointly driven estimator*, both of which can be plugged into traditional cost models.

The former uses statistical and deep models (*e.g.*, deep autoregressive model) to approximate high-dimensional data distributions over database attributes and relations. The training and usage of such models can be analogous to how the traditional sketches (*e.g.*, histograms and samples) are constructed and used. They are trained on samples drawn from relations with the goal of minimizing the gap between the predicted data distribution and the seen distribution. Query workloads are assumed to be unknown when fitting these models. For a given query, these models are "invoked" to estimate its cardinality.

The latter trains models for a specific query workload for better accuracy. Queries are featurized as parts of the inputs to the model, and the model is trained to minimize the gap between the estimated cardinalities and the true cardinalities. Indeed, the model needs to be updated when the distribution of query workload shifts.

• *From traditional cost model to learned cost model.* The cost of a plan is the sum of costs of all operators in it. For each operator, a traditional cost model typically takes cardinality estimates of immediate sub-queries under the operator as the inputs in a formula to estimate its cost, since they are the numbers of tuples to be processed by this operator. The concrete form of this formula and the magic constants in it, depend on the operator's type and implementation, and are tuned with years of engineering efforts to ensure that the estimated cost matches the plan's performance well empirically. In this sense, traditional cost models are "human learning" models. It is thus a natural idea to develop machine learning models with execution statistics (for specific performance metrics) on different datasets and query workloads as the training data, to enable finer-grained characterization of various data distributions and system configurations, thus providing instance-level optimization of each query. The learned cost model can be plugged into the traditional cost-based search algorithm to cost (sub-)plans in the search procedure, and updated when more queries are processed.

• *Learning-based search algorithm.* Traditional query optimizers treat the task of finding the best execution plan as a combinatorial optimization problem. Thus, dynamic programming algorithms as well as heuristics (*e.g.*, based on transformation rules) are developed to find the best plan under certain cost models. If we treat query optimization as a machine learning task, we unlock other possibilities of designing the search algorithm. For example, we can model it as a *multi-armed bandit* problem, where each arm corresponds to a candidate plan and we want to select the best arm (execution plan) with more and more observations of their performance. We can also model it as a *deep reinforcement learning* problem, with learned cost models as value networks to guide the generative search for the best plan. Moreover, since what we essentially need for query optimization is an oracle that compares two plans and ranks a set of candidate query plans with respect to their execution efficiency, we can model the task as a *learning-to-rank* problem. These possibilities will be introduced and formalized later in this survey.

Technical questions. There are some key technical questions to be resolved in the above paradigm. First, the data-query-workload joint distribution is complex. We need to carefully featurize data and queries in such a way that we can effectively model their correlation and the marginal distributions via, *e.g.*, statistical or deep models. Second, we need to collect "training data" for these models. Cold start is always a challenge, especially when we train models to estimate and optimize the execution latency. Third, learning-based search algorithms need to be co-designed with the estimation models, so that they have consistent learning goals; meanwhile, when a search algorithm invokes learned estimation models with non-trivial inference costs (possibly many times), it needs to be designed to avoid prohibitive optimization cost.

Other possibilities and tasks. The generic paradigm in Figure 1.1 rules out some other possible ways to find better plans by learning from experience. For example, one can execute plans on samples of relations and use such experience to refine cardinality estimates and thus improve the final execution plans (Krauthgamer *et al.*, 2008; Wu *et al.*, 2016). Even during the processing of a specific query, one can use early-stage experience (*e.g.*, try different operator types and join orders on samples from intermediate results) to revise the remaining execution plan (Kabra and DeWitt, 1998; Markl *et al.*, 2004; Kader *et al.*, 2009). Detailed discussion about these works is beyond the scope of this survey, but one can refer to a recent benchmark paper by Zhang *et al.*, 2023 on such adaptive query processing algorithms.

Worst-case optimal join algorithms (refer to, *e.g.*, Ngo *et al.*, 2018) are set apart from traditional query processing algorithms with theoretical guarantees on their processing costs. Their practical performance also depends heavily on the order in which join attributes are processed, which is not reflected in the definition of worst-case optimality (w.r.t. worst-case assumptions about the database content) and the formal analysis by Ngo *et al.*, 2018. Wang *et al.*, 2023b introduces a query engine which selects the attribute orders via reinforcement learning.

While the paradigm in Figure 1.1 matters primarily for join ordering, access path, and operator selection in query optimization, there are other tasks that can effectively improve the execution performance of a SQL query. For example, *query rewriting* is to transform a poorly-written SQL query into one that executes more efficiently while maintaining the result set. Approaches for this task are based on, *e.g.*, rules (Begoli *et al.*, 2018; Wang *et al.*, 2022), program synthesis (Dong *et al.*, 2023), Monte Carlo tree search with deep estimation models (Zhou *et al.*, 2021; Zhou *et al.*, 2023), or, more recently, large language models (Liu and Mozafari, 2024). These query rewriting approaches are orthogonal to the majority of techniques discussed in this survey.

There are some specific scenarios of query optimization that can be aided by machine learning but are not covered by this survey. For example, *multi-query optimization* aims to select plans for a group of queries, considering opportunities to reduce the total execution cost by sharing redundant work to be done by an identical sub-query across plans of different queries. This problem can be tackled with, *e.g.*, reinforcement learning by Sioulas and Ailamaki, 2021. *Parametric query optimization*, addressed by, *e.g.*, Doshi *et al.*, 2023, is to generate a set of candidate plans for a single query template and decide which plan to use for each query instance. Learned query optimization for specialized types of data such as spatial data is also studied in Vu *et al.*, 2021.

1.4 Summary of the Survey

In a learned query optimizer, one or multiple core components are aided or rebuilt with machine learning techniques. Most state-of-the-art learned query optimizers can be regarded as concrete implementations of the aforementioned paradigm (Figure 1.1) or its variants. Chapter 2 will focus on representative techniques for the costing components (cardinality estimator and cost model). These two components are closely related, as in traditional query optimizers, cost models invoke cardinality estimators to cost plans. We will first discuss their relationship and how estimation error transfer from cardinality estimators to cost models. We will then introduce, purely data-driven as well as data-query jointly driven, machine learning techniques for cardinality estimation, followed by how to train machine learning models to cost plans directly. Chapter 3 will focus on plan enumerators. Several new types of search algorithms, empowered by machine learning models, are proposed recently. Chapter 3.1 introduces a multi-armed bandit modeling of the plan enumeration procedure. Chapter 3.2 introduces how to apply generative search in reinforcement learning for (bottom-up) plan construction, with the help of value networks which is adapted from learned cost models. Chapter 3.3 introduces a learning-to-rank scheme for plan enumeration and selection. We will also discuss interesting future research directions inspired by some more recent efforts in Chapter 4.

2

Learned Cost Models

Cost-based query optimizers rely on accurate estimations of the costs of plans to determine which one to execute. Each operator in a plan takes one or more input or intermediate relations and generates another intermediate relation or the final result. Thus, the estimated cost of a plan indeed depends on the estimation of sizes, or *cardinalities*, for intermediate relations of sub-plans. In particular, the cost of executing an operator in the plan can be estimated as a function of estimated sizes of inputs to this operator. While the functions for different operators, *e.g.*, I/O cost and index seek cost in different joins, can be accurately characterized or calibrated (Haas *et al.*, 1997) with magic constant numbers, the accuracy of cardinally estimations, to a large extent, decides the accuracy of cost estimations. We first present a formal characterization of the relationship between cardinality and cost in Chapter 2.1. We then introduce how machine learning models are designed to replace traditional cardinality estimators and cost models, in Chapter 2.2 and Chapter 2.3, respectively. These models, after properly trained, can be integrated into a database without changing the plan enumerator, and are expected to be helpful in selecting better plans.

2.1 From Cardinality to Cost

In a typical cost model by Haas *et al.*, 1997, the cost of a join `J` is a polynomial function of s_1 and s_2 (the sizes of two input relations):

$$\begin{aligned}\texttt{cost}_{\texttt{J}}(s_1, s_2) = a_{2,0} \cdot s_1^2 + a_{0,2} \cdot s_2^2 + a_{1,1} \cdot s_1 s_2 + \\ a_{1,0} \cdot s_1 + a_{0,1} \cdot s_2 + a_{0,0}, \end{aligned} \tag{2.1}$$

with non-negative coefficients whose values depend on the join algorithm and the hardware. Intuitively, the degree-2 costing terms, *e.g.*, $a_{2,0} \cdot s_1^2$, are from the loop structure in the join algorithm, whose coefficients can be zero for some join types; and the degree-1 terms are from the costs of reading data or generating the output. It has been shown that coefficients in cost functions in the form of (2.1) can be estimated very accurately, which are data-independent and can be tuned for certain hardware and system configurations. It is the estimation for cardinalities s_1 and s_2 of intermediate results that tends to be more error prone and jeopardizes the accuracy of the cost model.

While the potential of replacing traditional cardinality estimators with learning-based models has been recognized (Lan *et al.*, 2021; Wang *et al.*, 2021b), a fundamental question is how the cardinality estimation error is translated into the deviation of the true execution costs. More importantly, how sensitive is the quality of plans produced by the optimizer with the accuracy of such estimates? The implication of answers to these questions is two-fold. First, is it worth the effort to train (possibly expensive) machine learning models to replace traditional cardinality estimators? Second, what loss function and error metric should be used when training and evaluating such models.

Moerkotte *et al.*, 2009 provides an analytical framework to answer the above questions. Assuming that the coefficients in (2.1) are accurate, instead of true cardinalities s_1 and s_2 of intermediate results, we can only obtain their estimates $\hat{s}_1$ and $\hat{s}_2$. Define the Q-*error* of an estimation $\hat{s}$ for s as (suppose both $s, \hat{s} \geq 0$)

$$\|s/\hat{s}\|_{\mathsf{Q}} = \max(s/\hat{s}, \hat{s}/s). \tag{2.2}$$

If $\|s/\hat{s}\|_{\mathsf{Q}} \leq \epsilon$ (for some $\epsilon \geq 1$), we have $(1/\epsilon) \cdot s \leq \hat{s} \leq \epsilon \cdot s$.

Proposition 2.1 (Moerkotte *et al.*, 2009)**.** Suppose the Q-errors of $\hat{s}_1$ and $\hat{s}_2$ for estimating s_1 and s_2 are bounded by ϵ, we have the Q-error of $\texttt{cost}_\texttt{J}(\hat{s}_1, \hat{s}_2)$ for estimating $\texttt{cost}_\texttt{J}(s_1, s_2)$ bounded by ϵ^2.

The proof is directly from that $\texttt{cost}_\texttt{J}(s_1, s_2)$ in (2.1) is a polynomial in s_1 and s_2 with degree at most 2 and positive coefficients.

Suppose the cost of a plan P is calculated by the sum over the costs for each join in the plan and the cost of a plan is in the form of (2.1). Let $\texttt{cost}(P; s)$ be the cost model calculated with true cardinalities, and $\texttt{cost}(P; \hat{s})$ be the one calculated with estimated cardinalities. For a given query q, let P^* be the optimal plan under $\texttt{cost}(\cdot; s)$ and $\hat{P}$ be the optimal one under $\texttt{cost}(\cdot; \hat{s})$, *i.e.*,

$$P^* = \operatorname*{arg\,min}_{P \in \mathbb{P}(q)} \texttt{cost}(P; s) \quad \text{and} \quad \hat{P} = \operatorname*{arg\,min}_{P \in \mathbb{P}(q)} \texttt{cost}(P; \hat{s}).$$

The following result establishes a relationship between the accuracy of cardinality estimation and the optimality of the plan derived.

Theorem 2.1 (Moerkotte *et al.*, 2009)**.** For a given query q, suppose the Q-error of cardinality estimation for intermediate result of any sub-query is bounded by ϵ, we have $\texttt{cost}(\hat{P}; s) \leq \epsilon^4 \texttt{cost}(P^*; s)$.

If the Q-error of any cardinality estimation is bounded ϵ, from Proposition 2.1, we sum up costs of joins and have $\|\texttt{cost}(P; s)/\texttt{cost}(P; \hat{s})\|_\mathsf{Q} \leq \epsilon^2$ for any plan P. Then from the optimality of P^* and $\hat{P}$ (under $\texttt{cost}(\cdot; s)$ and $\texttt{cost}(\cdot; \hat{s})$, respectively), we can complete the proof.

The above theorem shows that the error of cardinality estimations can be translated into the degree of sub-optimality of the plan selected. P^* can be interpreted as the truly optimal plan (assuming that the coefficients in (2.1) are tuned to be precise) and $\hat{P}$ as the one selected by the query optimizer with estimated cardinalities feed into the cost model. It says that a Q-error ϵ in cardinality estimation is amplified up to a factor of ϵ^4 in the plan cost, but is independent on the number of relations in the query q, with two implications. First, the cardinality estimation does not have to be perfect. Second, improvement on the estimation accuracy may lead to better plans (in the worst-case scenario); for a particular instance, however, it is possible to improve the cardinality estimation of some sub-query while making the overall query performance worse (Negi *et al.*, 2021c presents such an example).

Next, in Chapter 2.2, we will present some efforts to improve the cardinality estimation for (sub-)queries using machine learning techniques; in Chapter 2.3, we will introduce more holistic machine learning-based approaches to cost plans, with proper learning objectives (*e.g.*, *flow-loss* as is suggested in Negi *et al.*, 2021c), features, and models.

2.2 Learning to Estimate Cardinality

Traditional cardinality estimators reply on different ways to condense data into summaries, such as histograms (*e.g.*, Poosala and Ioannidis, 1997; Liu *et al.*, 2021b), samples (*e.g.*, Lipton *et al.*, 1990; Leis *et al.*, 2017; Li *et al.*, 2016; Zhao *et al.*, 2018), and sketches (*e.g.*, Atserias *et al.*, 2013; Cai *et al.*, 2019; Hertzschuch *et al.*, 2021; Izenov *et al.*, 2021). Let $\mathcal{H}_{\mathbb{D}}$ be the summary (*e.g.*, histograms) built on the database $\mathbb{D}$. The cardinality of a give query q, $s(q)$, can then be estimated as a function $\hat{s}(q, \mathcal{H}_{\mathbb{D}})$. With advances in deep learning models, a natural question is whether it is possible to replace the traditional data summary $\mathcal{H}_{\mathbb{D}}$ with a trained model $\mathcal{M}_{\mathbb{D}}(\cdot)$, which is invoked one or multiple times in an algorithm $\hat{s}(q, \mathcal{M}_{\mathbb{D}}(\cdot))$ to estimate $s(q)$ (*i.e.*, *data driven estimators*, introduced in Chapter 2.2.1). Moreover, is it possible to train a model $\mathcal{M}_{\mathbb{D}}(\cdot)$ on a database $\mathbb{D}$ and its query workload so that the cardinality of a given query q can be directly estimated as $\mathcal{M}_{\mathbb{D}}(q)$ (*i.e.*, *data-query jointly driven estimators*, introduced in Chapters 2.2.2 and 2.2.3)?

2.2.1 Data Driven Estimators

We now focus on the first line of efforts, *i.e.*, using a machine learning model $\mathcal{M}_{\mathbb{D}}(\cdot)$ to summarize the data in $\mathbb{D}$ and approximate the high-order joint distribution of tuples in $\mathbb{D}$.

Consider a relation $\mathbf{R}$ with attributes $\{A_1, A_2, \ldots, A_n\}$. A tuple $\mathbf{x} = (x_1, \ldots, x_n)$ in $\mathbf{R}$ is from the domain $\Delta(\mathbf{R}) = A_1 \times \ldots \times A_n$. Suppose a query q against $\mathbf{R}$ has a predicate $\theta : A_1 \times \ldots \times A_n \to \{0, 1\}$. The cardinality of q is $s(q) = |\{\mathbf{x} \in \mathbf{R} \mid \theta(\mathbf{x}) = 1\}|$. Define the joint distribution $\mathrm{P}_{\mathbf{R}}(\mathbf{x})$ as *the probability of seeing* $\mathbf{x}$ *if a tuple is drawn from* *the multi-set* $\mathbf{R}$ *uniformly at random.* $s(q)$ can be also written as:

$$s(q) = |\mathbf{R}| \cdot \sum_{x_1 \in A_1} \cdots \sum_{x_n \in A_n} \theta(\mathbf{x}) \cdot \mathrm{P}_{\mathbf{R}}(\mathbf{x}) \,, \tag{2.3}$$

where $|\mathbf{R}|$ is the number of rows in the relation $\mathbf{R}$.

There are three fundamental challenges in estimating $s(q)$ in the form of (2.3). First, $\mathrm{P}_{\mathbf{R}}$ is on a huge domain $\Delta(\mathbf{R})$ with size $\prod_{i=1}^{n} |A_i|$ which makes it difficult in practice, if not impossible, to estimate, store, and use the joint distribution $\mathrm{P}_{\mathbf{R}}$. Second, even with exact access to $\mathrm{P}_{\mathbf{R}}$, directly summing up $\theta(\mathbf{x}) \cdot \mathrm{P}_{\mathbf{R}}(\mathbf{x})$ as in (2.3) is too expensive. Third, $\mathbf{R}$ could be a join of multiple relations.

To tackle the first challenge, there are different ways of representing the joint distribution $\mathrm{P}_{\mathbf{R}}(\mathbf{x})$ in a compact manner. For example, Heimel *et al.*, 2015 and Kiefer *et al.*, 2017 use different kernel density functions around sample points to approximate $\mathrm{P}_{\mathbf{R}}(\mathbf{x})$. A systematical framework is to *factorize* (Grimmett and Stirzaker, 2001) $\mathrm{P}_{\mathbf{R}}(\mathbf{x})$ into some low-dimensional representation, which is introduced next.

Factorizing Joint Distribution

The simplest factorization approach assumes independence between attributes and has been used by database systems since 1970s (Selinger *et al.*, 1979). In the language of probabilistic models, it is assumed that

$$\mathrm{P}_{\mathbf{R}}(\mathbf{x}) \approx \prod_{i=1}^{n} \mathrm{P}_{A_i}(x_i) \,, \tag{2.4}$$

where $\mathrm{P}_{A_i}(x_i)$ is the probability of seeing x_i on attribute A_i if a tuple $\mathbf{x}$ is drawn from $\mathbf{R}$ uniformly at random and can be approximated with a 1D histogram on A_i. For a query with predicate "$A_1 = a_1$ and $A_2 = a_2$" on $\mathbf{R}$, its cardinality $s(q)$ can be estimated as $|\mathbf{R}| \cdot \mathrm{P}_{A_1}(a_1)\, \mathrm{P}_{A_2}(a_2)$.

Another natural way is to factorize the joint distribution $\mathrm{P}_{\mathbf{R}}(\mathbf{x})$ sequentially based on a specific order of attributes:

$$\begin{aligned}\mathrm{P}_{\mathbf{R}}(\mathbf{x}) &= \mathrm{P}_{\mathbf{R}}(x_1) \cdot \mathrm{P}_{\mathbf{R}}(x_2 \mid x_1) \cdot \mathrm{P}_{\mathbf{R}}(x_3 \mid x_1, x_2) \cdots \mathrm{P}_{\mathbf{R}}(x_n \mid x_1, \ldots, x_{n-1}) \\ &= \prod_{i=1}^{n} \mathrm{P}_{\mathbf{R}}(x_i \mid \mathbf{x}_{<i}) \,,\end{aligned} \tag{2.5}$$

where $\mathbf{x}_{<i} = (x_1, \ldots, x_{i-1})$ and $\mathbf{x}_{<1} = \emptyset$. Note that this factorization is exact, and $\mathrm{P}_{\mathbf{R}}(x_i \mid \mathbf{x}_{<i})$ can be interpreted as the likelihood of observing x_i in the attribute A_i of a tuple drawn from $\mathbf{R}$ under the condition that the last $i-1$ attributes have values $\mathbf{x}_{<i}$. Again, it is impossible to materialize the conditional distributions $\{\mathrm{P}_{\mathbf{R}}(x_i \mid \mathbf{x}_{<i})\}$ as the number of random variables involved is large (up to n). Yang *et al.*, 2019 and Yang *et al.*, 2020 show how to train deep autoregressive models to approximate $\{\mathrm{P}_{\mathbf{R}}(x_i \mid \mathbf{x}_{<i})\}$ and use them for cardinality estimation.

A general *probabilistic graphical model* can be used to factorize $\mathrm{P}_{\mathbf{R}}(\mathbf{x})$. Define $\texttt{parent}(i)$ to be the set of indices of A_i's *parent attributes*, whose values determine A_i's value in a probabilistic way. (2.5) corresponds to a graphical model where each attribute A_i has all attributes with indices $< i$ as its parents. In general, $\mathrm{P}_{\mathbf{R}}(\mathbf{x})$ can be approximated as

$$\mathrm{P}_{\mathbf{R}}(\mathbf{x}) = \prod_{i=1}^{n} \mathrm{P}_{\mathbf{R}}\Big(x_i \mid \mathbf{x}_{\texttt{parent}(i)}\Big) . \tag{2.6}$$

Getoor *et al.*, 2001 applies a specific probabilistic graphical model, namely Bayesian network, for cardinality estimation for the first time. There are a line of works (*e.g.*, Tzoumas *et al.*, 2011; Wu and Shaikhha, 2020; Hilprecht *et al.*, 2020; Zhu *et al.*, 2021; Wu *et al.*, 2023) based on different graphical models with improved training overhead and performance of inference.

Estimating Cardinality with Deep Autoregressive Model

We demonstrate how to train neural networks to approximate the factorized models in (2.5) and use them for cardinality estimation (Yang *et al.*, 2019; Yang *et al.*, 2020).

Model structure. A *deep autoregressive model* $\mathcal{M}_{\texttt{AR}}$ fits such factorization well. It takes a tuple $\mathbf{x}$ as the input and outputs a sequence of "predicted conditional probabilities" $\hat{\mathrm{P}}_{\mathbf{R}}(x_i \mid \mathbf{x}_{<i}) \approx \mathrm{P}_{\mathbf{R}}(x_i \mid \mathbf{x}_{<i})$:

$$\mathcal{M}_{\texttt{AR}}(\mathbf{x}) \to \Big[\hat{\mathrm{P}}_{\mathbf{R}}(x_1 \mid \mathbf{x}_{<1}), \hat{\mathrm{P}}_{\mathbf{R}}(x_2 \mid \mathbf{x}_{<2}), \ldots, \hat{\mathrm{P}}_{\mathbf{R}}(x_n \mid \mathbf{x}_{<n})\Big] . \tag{2.7}$$

Many network architectures can be used to instantiate $\mathcal{M}_{\texttt{AR}}$ in the above, such as DARN (Gregor *et al.*, 2014), MADE (Germain *et al.*, 2015), ResMADE (Durkan and Nash, 2019), and Transformer (Vaswani *et al.*, 2017). They commonly rely on a technique called *information masking* to fit the observed sequence of conditional probabilities. Namely, a neural net $\mathcal{M}_{A_i}$ per attribute A_i ($i = 1, \ldots, n$) takes properly encoded and aggregated (*e.g.*, vector concatenation) inputs from $x_1, \ldots, x_i$ and is trained to fit $\mathrm{P}_{\mathbf{R}}(x_i \mid \mathbf{x}_{<i})$. For example, for a table with attributes $\texttt{city}, \texttt{year}, \texttt{stars}$, the model is given the input tuple $\mathbf{x} = (\texttt{Portland}, 2017, 10)$ and outputs $\mathcal{M}_{\texttt{AR}}(\mathbf{x}) =$

$$\begin{bmatrix} \mathcal{M}_{\texttt{city}}(\texttt{Portland}) & \to \hat{\mathrm{P}}_{\mathbf{R}}(\texttt{city} = \texttt{Portland})\,, \\ \mathcal{M}_{\texttt{year}}(\texttt{Portland}, 2017) & \to \hat{\mathrm{P}}_{\mathbf{R}}(\texttt{year} = 2017 \mid \texttt{Portland})\,, \\ \mathcal{M}_{\texttt{stars}}(\texttt{Portland}, 2017, 10) & \to \hat{\mathrm{P}}_{\mathbf{R}}(\texttt{stars} = 10 \mid \texttt{Portland}, 2017) \end{bmatrix}.$$

While an arbitrary ordering of attributes $\{A_i\}$ can be used in $\mathcal{M}_{\texttt{AR}}$, Yang *et al.*, 2019 proposes several mutual-information-based heuristic orders that can empirical performance of the model.

Attributes of a relation may have different data types and thus need to be encoded properly at the beginning of neural nets, with attribute-specific encoders. For a categorical attribute in a small domain, one-hot encoding can be used. For example, if there are 4 cities $\{\texttt{A}, \texttt{Portland}, \texttt{C}, \texttt{D}\}$, the attribute-specific encoding $\texttt{Enc}_{\texttt{city}}(\texttt{Portland}) = [0, 1, 0, 0]$. For a categorical attribute in a large domain, a learnable embedding matrix of type $\mathbb{R}^{|A_i| \times h}$ is initialized randomly and updated during model training; the attribute-specific encoder $\texttt{Enc}_{A_i}$ maps a value in A_i to a h-dim row vector of the matrix. For a numerical attribute, we can first partition the continuous domain into a number of small bins; values in the same bin are rounded to the bin's index, and then the encoding method for a large-domain categorical attribute can be used. IAM (Meng *et al.*, 2022) integrates Gaussian mixture with an autoregressive model to fit the distribution of numerical attributes and reduce their domain size.

Training the model. From the model's output, we can calculate

$$\hat{\mathrm{P}}_{\mathbf{R}}(\mathbf{x}) = \prod_{i=1}^{n} \mathcal{M}_{A_i}(x_1, \ldots, x_i) = \prod_{i=1}^{n} \hat{\mathrm{P}}_{\mathbf{R}}(x_i \mid \mathbf{x}_{<i})\,, \tag{2.8}$$

which predicts the likelihood of seeing $\mathbf{x}$. Thus, the model can be trained to maximize, $\prod_{\mathbf{x}} \hat{\mathrm{P}}_{\mathbf{R}}(\mathbf{x})$, the likelihood estimation of seeing the training data $\mathbf{x} = \mathbf{R}_{\texttt{train}}$ (Yang *et al.*, 2019; Yang *et al.*, 2020):

$$\text{minimize} \quad \mathcal{L}^{\mathbb{D}} = -\frac{1}{|\mathbf{R}_{\texttt{train}}|} \sum_{\mathbf{x} \in \mathbf{R}_{\texttt{train}}} \log \hat{\mathrm{P}}_{\mathbf{R}}(\mathbf{x}) \,, \tag{2.9}$$

where $\mathbf{R}_{\texttt{train}}$ is $\mathbf{R}$ or a sample drawn from $\mathbf{R}$ uniformly at random.

Estimating cardinality by querying the model. After $\mathcal{M}_{\texttt{AR}}$ properly trained, we can construct a cardinality estimator $\hat{s}(q, \mathcal{M}_{\texttt{AR}})$ for a given query q. We consider two types of predicates in q here: i) point queries with conjunctions of equality constraints, and ii) range queries.

For a point query with predicate $A_1 = a_1 \wedge A_2 = a_2 \ldots A_n = a_n$, recall the relationship between the cardinality $s(q)$ and the joint distribution in $\mathrm{P}_{\mathbf{R}}(\mathbf{x})$ in (2.3), we can estimate $s(q)$ as

$$\hat{s}(q, \mathcal{M}_{\texttt{AR}}) = |\mathbf{R}| \cdot \hat{\mathrm{P}}_{\mathbf{R}}(a_1, a_2, \ldots, a_n) \,.$$

If not all the attributes are in the predicate, for example, $A_1 = a_1 \wedge A_3 = a_3$, we can train $\mathcal{M}_{\texttt{AR}}$ to handle wildcards $*$ by sampling tuples with a special token $*$ on one or more attributes, such as $(a_1, *, a_3)$ (meaning that it could any value on A_2), and inserting them into $\mathbf{R}_{\texttt{train}}$.

For a range query with predicate $A_1 \in R_1 \wedge A_2 \in R_2 \ldots A_n \in R_n$, from (2.3), we can calculate the following estimation:

$$|\mathbf{R}| \cdot \sum_{x_1 \in R_1} \cdots \sum_{x_n \in R_n} \hat{\mathrm{P}}_{\mathbf{R}}(\mathbf{x}) \,,$$

but it could be too expensive if the ranges R_i's are wide. A naive Monte Carlo method draws a set S of sample points uniformly and independently at random from $R_1 \times \cdots \times R_n$ and estimate

$$\hat{s}(q, \mathcal{M}_{\texttt{AR}}) = |\mathbf{R}| \cdot \frac{|R_1| \times \cdots \times |R_n|}{|S|} \sum_{\mathbf{x} \in S} \hat{\mathrm{P}}_{\mathbf{R}}(\mathbf{x}) \,.$$

The above estimator is unbiased but has a large variance since the values in $\{\hat{\mathrm{P}}_{\mathbf{R}}(\mathbf{x})\}$ are skewed for $\mathbf{x} \in S$. An importance sampling algorithm (called *progressive sampling*) is proposed by Yang *et al.*, 2019 to draw $\mathbf{x}$ with probability proportional to $\hat{\mathrm{P}}_{\mathbf{R}}(\mathbf{x})$ to reduce the variance.

Handling joins. In general, $\mathbf{R}$ could be a join of N relations $\mathbf{R} = \mathbf{T}_1 \bowtie \cdots \bowtie \mathbf{T}_N$. Note that in the above discussion, we do not have to access the whole relation $\mathbf{R}$; instead, it suffices to have a random sample $\mathbf{R}_{\texttt{train}}$ drawn from $\mathbf{R}$ in order to train $\mathcal{M}_{\texttt{AR}}$. Zhao *et al.*, 2018 provide an efficient algorithmic framework for drawing random samples from general multi-key joins, and Yang *et al.*, 2020 implements the *exact weight* algorithm from Zhao *et al.*, 2018 and adapts it for full outer joins. The estimator $\hat{s}(q, \mathcal{M}_{\texttt{AR}})$ also replies on the size of the full join, $|\mathbf{R}|$, which can be calculated during the sampling process as well.

Other Data Modeling Approaches

When the joint distribution on the relational $\mathbf{R}$ is too complicated to be factorized as a whole (*e.g.*, in the ways of (2.4)-(2.6)) and learned effectively, Sum-Product Networks (SPNs) (Poon and Domingos, 2011) are extended for relational data by Hilprecht *et al.*, 2020 and Zhu *et al.*, 2021. Relational Sum-Product Networks (RSPNs) in Hilprecht *et al.*, 2020 and Factorized Sum-Product Networks (FSPNs) Zhu *et al.*, 2021 both a data-splitting operation (*sum* node) to partition a relation $\mathbf{R}$ into two or more clusters of tuples in such a way that the distribution in each cluster is simpler and easier to be factorized. In each cluster, attributes are either partitioned into independent groups (*product* node) or factorized (*factorize* node). There two types of operations (*sum* v.s. *product*/*factorize*) can be applied recursively on $\mathbf{R}$ and its attributes. In practice, the sum node can be implemented by a standard clustering algorithm (Hilprecht *et al.*, 2020). If we treat the clustering result as a hidden random variable, RSPNs and FSPNs can be also interpreted as generalized probabilistic graphical models in (2.6).

To handle joins, we can either learn joint RSPNs/FSPNs over the full outer join, or build a set of small models, where each captures the joint distribution over several tables (Hilprecht *et al.*, 2020; Zhu *et al.*, 2021). FactorJoin (Wu *et al.*, 2023) builds single-table conditional distributions during an offline preprocessing phase; and these conditional distributions are then combined using a factor graph model (Loeliger, 2004). A query with joins can be translated into a factor graph over single-table data distributions for cardinality estimation.

The idea of using pre-trained models for relation data (analogous to pre-trained models for images and text) is firstly used in Iris (Lu *et al.*, 2021). Common frequency and correlation patterns of structured datasets are capture by pre-training on a large and diverse corpus of datasets. For different attribute sets, it pre-trains different summarization models, which can be efficiently updated when datasets change.

There are works aiming at handling more complex data types and schemes. FACE (Wang *et al.*, 2021a) leverages the Normalizing Flow based model to learn a continuous joint distribution for relational data. It transforms a complex distribution over continuous random variables into a simple distribution (*e.g.*, multivariate normal distribution), and use its probability density to estimate the cardinality of a given query. Astrid (Shetiya *et al.*, 2020) applies natural language processing techniques with deep models to learn cardinality of queries with string predicates; DREAM (Kwon *et al.*, 2022) builds deep models to learn cardinality of approximate substring queries; LMKG (Davitkova *et al.*, 2022) considers cardinality estimation on knowledge graphs.

2.2.2 Featurizing Queries as Vectors

The estimators developed by works introduced in Chapter 2.2.1 are all in the form of $\hat{s}(q, \mathcal{M}_{\mathbb{D}}(\cdot))$, where the model $\mathcal{M}_{\mathbb{D}}$ is trained purely on relational data. It is natural to train a model $\mathcal{M}_{\mathbb{D},\mathbb{Q}}$ on both datasets $\mathbb{D}$ and workloads $\mathbb{Q}$, where queries are represented as parts of features into $\mathcal{M}_{\mathbb{D},\mathbb{Q}}$, with the goal of predicting the cardinality $s(q)$.

There are several components in a SQL query to be featurized. We take the following class of counting queries as an example

$$\begin{aligned} q: \quad & \texttt{SELECT COUNT}(*)\ \texttt{FROM}\ \mathbf{R}_1, \ldots, \mathbf{R}_n \\ & \texttt{WHERE}\ \text{join predicate}\ \texttt{J}\ \texttt{AND}\ \text{filter predicate}\ \texttt{F}, \end{aligned} \tag{2.10}$$

where $\texttt{J}$ is a conjunction of join conditions "$\wedge_{i,j} \mathbf{R}_i.A^i_x = \mathbf{R}_j.A^j_y$" and $\texttt{F}$ a conjunction of filter conditions, to illustrate a typical featurization framework (Wang *et al.*, 2021b; Zhao *et al.*, 2022a; Li *et al.*, 2023).

Joins as vectors $\texttt{vec}_{\texttt{J}}(q)$. For a database with N relations $\mathbf{R}_1, \ldots, \mathbf{R}_N$, there are up to N^2 possible join patterns. One way to featurize joins specified as $\texttt{J}$ in (2.10) as a vector $\texttt{vec}_{\texttt{J}}(q)$ is to use one-hot encoding: we can use an N^2-dim 0-1 vector indexing all possible joins and an 1 bit on the ith position denoting that the ith join pattern appears in the query. Another way is to use $m_1 = \lceil \log_2(N+1) \rceil$ bits to encode the id of each relation and $m_2 = \lceil \log_2(M+1) \rceil$ bits (let M be the maximum number of attributes in a relation) to encode each attribute, for a join predicate in $\texttt{J}$. We can use up to $(2(m_1+m_2) \cdot N^2)$ bits to encode the join predicates $\texttt{J}$: if the ith join pattern $(1 \leq i \leq N^2)$ appears in the query, we use $2(m_1+m_2)$ bits to encode the pair of joining relations and attributes in the ith sub-vector; otherwise, we put 0's in this sub-vector.

The above two ways to encode joins are equivalent if there is at most one way to join every pair of relations. The second one contains some redundant information which may be helpful for training certain types of machine learning models. The set of relations in q are also encoded in $\texttt{vec}_{\texttt{J}}(q)$. Or, we can use a separate vector $\texttt{vec}_{\texttt{T}}(q)$ to encode the set of relations in q (*e.g.*, with one-hot encoding).

Filters as vectors $\texttt{vec}_{\texttt{F}}(q)$. For a numeric attribute, we can use at most two real numbers to encode either point constraints or range constraints. For example, a range constraint "$lb \leq A_k \leq ub$" can be encoded as (lb, ub). For a categorical attribute A_k in a large domain, we learn an embedding matrix of type $\mathbb{R}^{|A_k| \times h}$ and use h real numbers to represent a point constraint "$A_k = a$". The filter predicate $\texttt{F}$ can be represented as $\texttt{vec}_{\texttt{F}}(q)$, the concatenation of $(N \cdot M)$ 2-dim or h-dim vectors (depending on each attribute's type), where the ith one represents the filter condition on the ith attribute; for an attribute without a filter in $\texttt{F}$, we can leave a 0-vector correspondingly.

Putting them together. $\texttt{vec}_{\texttt{J}}(q)$ and $\texttt{vec}_{\texttt{F}}(q)$ can be concatenated as the feature vector $\texttt{vec}(q)$ for a query q. Figure 2.1 gives an example. Note that this featurization needs to be extended if we want to handle more complicated filter constraints such as $A_1 + A_2 \leq 10$, or more general join patterns such as multiple self joins on the same relation.

$$\text{query } q: \quad \begin{array}{l} \texttt{SELECT} * \texttt{FROM } \mathbf{R}_1, \mathbf{R}_2, \mathbf{R}_3, \cdots \texttt{ WHERE} \\ \mathbf{R}_1.A_1^1 = \mathbf{R}_2.A_1^2 \texttt{ AND } \mathbf{R}_2.A_2^2 = \mathbf{R}_3.A_3^3 \texttt{ AND } \cdots \\ \texttt{AND } 0.25 \le \mathbf{R}_1.A_1^1 < 0.5 \texttt{ AND } \cdots \end{array}$$

$$\texttt{vec}(q): \quad \overbrace{\underbrace{0101\ 1001}_{\mathbf{R}_1.A_1^1 = \mathbf{R}_2.A_1^2} \quad \underbrace{1010\ 1111}_{\mathbf{R}_2.A_2^2 = \mathbf{R}_3.A_3^3} \quad \cdots}^{\text{join featurization } \texttt{vec}_\texttt{J}(q)} \overbrace{\underbrace{0.25\ 0.5}_{0.25 \le A_1 < 0.5} \quad \underbrace{\cdots}_{l_k \le A_k < u_k}}^{\text{filter featurization } \texttt{vec}_\texttt{F}(q)}$$

Figure 2.1: An example of query featurization.

2.2.3 More on Models and Data-Query Jointly Driven Estimators

To estimate the cardinality $s(q)$, we can now train a model $\mathcal{M}_{\mathbb{D},\mathbb{Q}}(\texttt{vec}(q))$ whose parameters are optimized for datasets $\mathbb{D}$ and workloads $\mathbb{Q}$.

Traditional models. It is natural to model the cardinality estimation task as a regression problem, and use a low-overhead model as $\mathcal{M}_{\mathbb{D},\mathbb{Q}}$, such as linear regression (Malik *et al.*, 2007), tree-based ensembles, simple neural networks (Dutt *et al.*, 2019), and XGBoost (Dutt *et al.*, 2020), to fit the cardinalities of queries in the workload $\mathbb{Q}$. The labels, *i.e.*, true cardinalities s, are typically log-transformed, and the model is trained to minimize the mean-squared-error $\sum |\log s - \log \hat{s}|^2$, which is equivalent to minimizing the Q-error $\max(s/\hat{s}, \hat{s}/s)$ and shown to be helpful for the task of query optimization as in Theorem 2.1.

In general, the model $\mathcal{M}_{\mathbb{D},\mathbb{Q}}$ can be updated when more queries come over time (*i.e.*, $\mathbb{Q}$ becomes larger). For example, QuickSel (Park *et al.*, 2020) uses a mixture model with overlapping to approximate the probability density function. The model could be refined efficiently as more and more queries are observed, and yield increasingly more accurate selectivity estimates over time.

Deep models. We can directly apply a fully connected network for $\mathcal{M}_{\mathbb{D},\mathbb{Q}}$ by feeding the featurized query, $\texttt{vec}(q)$, into a unified input layer (Liu *et al.*, 2015; Kim *et al.*, 2022), and train the neural network to fit the true cardinalities of queries in $\mathbb{Q}$. A deep neural network is able to capture inter-relation, intra-relation, and query-relation correlation.

To reduce the number of parameters in $\mathcal{M}_{\mathbb{D},\mathbb{Q}}$, it can be noticed that different sub-vectors in $\texttt{vec}(q)$ encodes different types of information. Thus, sub-modules of $\mathcal{M}_{\mathbb{D},\mathbb{Q}}$ can be trained to only consume sub-vectors, respectively, with their outputs pooled (*i.e.*, averaged) to make the prediction. In particular, Kipf *et al.*, 2019 introduces a framework called *multi-set convolutional network* (MSCN):

$$\text{sub-modules: } w_{\mathtt{J}} \leftarrow \frac{1}{|\mathtt{J}|}\sum_{j\in\mathtt{J}}\mathcal{M}_{\mathtt{J}}(\texttt{vec}_{\mathtt{J}}(j)),\ \ w_{\mathtt{F}} \leftarrow \frac{1}{|\mathtt{F}|}\sum_{f\in\mathtt{F}}\mathcal{M}_{\mathtt{F}}(\texttt{vec}_{\mathtt{F}}(f));$$
$$\text{merge-output: } \hat{s}(q) \leftarrow \mathcal{M}_{\texttt{out}}(w_{\mathtt{J}}, w_{\mathtt{F}}). \tag{2.11}$$

$\texttt{vec}_{\mathtt{J}}(j)$ and $\texttt{vec}_{\mathtt{F}}(f)$ encode each join $j \in \mathtt{J}$ and each filter $f \in \mathtt{F}$, respectively, *e.g.*, in the way introduced in Chapter 2.2.2. $|\mathtt{J}|$ copies of the shared-parameter neural network $\mathcal{M}_{\mathtt{J}}$ take $\texttt{vec}_{\mathtt{J}}(j)$'s as the inputs and the outputs are averaged as $w_{\mathtt{J}}$; similarly, $\mathcal{M}_{\mathtt{F}}$ is trained for $\texttt{vec}_{\mathtt{F}}(f)$'s. $w_{\mathtt{J}}$ and $w_{\mathtt{F}}$ are concatenated and fed into a final output network $\mathcal{M}_{\texttt{out}}$.

Hayek and Shmueli, 2020 introduces a CRN network, using the separation-pooling idea similar to MSCN, to learn the *containment rate* between pairs of queries (q_1, q_2), *i.e.*, the percentage of tuples in the output of q_1 that are also in the output of q_2. The predicted containment rate can be used to estimate the cardinality of q_2 (if q_1's output is a super-set of q_1's output and has a known size). Negi *et al.*, 2023 proposes Robust-MSCN using the query masking technique to adapt to workload changes. Kim *et al.*, 2022 conducts in-depth analysis and evaluation on query-driven learned estimators. It also proposes a tradeoff solution between the simple fully connected network, which could potentially have too many parameters to be optimized, and MSCN, which may loose some intra-table correlation. The idea is to use one copy of a neural network to consume features from each relation; parameters of these network copies are shared across relations and their outputs are pooled (*e.g.*, averaged) before the output module.

Fauce (Liu *et al.*, 2021a) uses an ensemble of deep models to estimate the cardinality as well as the underlying uncertainty of the prediction. Zhao *et al.*, 2022a employs Bayesian deep learning (BDL) to bridge between Bayesian inference and deep learning. The prediction distribution by BDL makes it possible to calibrate uncertainty for the estimated cardinality. This algorithm, known as Neural Network Gaussian Process

(NNGP), inherits the advantages of Bayesian approach while keeping the ability of universal approximation from neural networks.

Deep models are also developed for progressive cardinality estimation in query re-optimization. Specifically, during query execution, re-optimization is triggered if the cardinality estimations are found to have large errors. To this end, LPCE (Wang *et al.*, 2023a) is introduced with an initial model (LPCE-I) and a refinement model (LPCE-R). LPCE-I is trained to estimate cardinalities before query execution, and LPCE-R progressively refines the cardinality estimations using the actual cardinalities of sub-queries observed during query execution.

Estimation Models Jointly Driven by Data and Query

A learned cardinality estimator $\mathcal{M}_{\mathbb{D},\mathbb{Q}}(\texttt{vec}(q))$ that is purely driven by featurized queries can be accurately tuned for a specific database $\mathbb{D}$ and a workload $\mathbb{Q}$. We hope that it can generalize for unseen queries as long as they are properly featurized; however, when the database $\mathbb{D}$ is updated (*e.g.*, more rows are inserted into some relations), the estimation accuracy can easily deteriorate.

Several ways to model data and queries jointly have been proposed to handle dynamic data. For example, UAE introduced by (Wu and Cong, 2021) extends the deep autoregressive model in Chapter 2.2.1 with a loss function to measure how well the model performs on a query workload $\mathbb{Q}$. Specifically, the loss function $\mathcal{L}^{\mathbb{D}}$ in (2.9) about how data fits the model is augmented with the query loss function

$$\mathcal{L}^{\mathbb{Q}} = \sum_{q \in \mathbb{Q}} \max(s(q)/\hat{s}(q, \mathcal{M}_{\texttt{AR}}), \hat{s}(q, \mathcal{M}_{\texttt{AR}})/s(q)), \tag{2.12}$$

where $\hat{s}(q, \mathcal{M}_{\texttt{AR}})$ is the estimated cardinality produced by invoking the autoregressive model $\mathcal{M}_{\texttt{AR}}$ using, *e.g.*, progressive sampling, as introduced in Chapter 2.2.1. $\mathcal{L}^{\mathbb{Q}}$ takes queries as supervised information, and the model can then be trained under a hybrid loss $\mathcal{L} = \mathcal{L}^{\mathbb{D}} + \lambda\mathcal{L}^{\mathbb{Q}}$ with hyper-parameter λ, to capture both the data and query workload information simultaneously to learn the joint data distribution.

A different approach is to encode statistical information of the relations in $\mathbb{D}$ into the input feature $\texttt{vec}(\mathbb{D})$. A model $\mathcal{M}_{\mathbb{D},\mathbb{Q}}(\texttt{vec}(q), \texttt{vec}(\mathbb{D}))$

is trained with both query featurization and data featurization to estimate the cardinality $s(q)$. One obvious advantage of training $\mathcal{M}_{\mathbb{D},\mathbb{Q}}$ with $\mathtt{vec}(\mathbb{D})$ as part of the features is that its model parameters are not fully determined by a specific instance of database $\mathbb{D}$, and thus it allows mild drifts of data distribution or incremental updates in $\mathbb{D}$ (reflected in the values of $\mathtt{vec}(\mathbb{D})$) while maintaining the accuracy of estimating $s(q)$. Li *et al.*, 2023 proposes to use a *d-bin histogram* (*i.e.*, merging values from the domain of a relation into d bins) to summarize value frequencies for each attribute of each relation in $\mathbb{D}$, and encode the histogram as a d-dim vector. These vectors for all the attributes are stacked together to form a $T \times d$ matrix $\mathtt{vec}(\mathbb{D}) \in \mathbb{R}^{T\times d}$, where T is the total number of attributes of all relations in $\mathbb{D}$.

The model $\mathcal{M}_{\mathbb{D},\mathbb{Q}}$ designed by Li *et al.*, 2023 applies the attention mechanism (Vaswani *et al.*, 2017) and learns two attention functions:

$$\textit{data encoder}: \quad \mathbf{Z} \leftarrow \mathcal{M}_{\mathtt{enc}}(\mathtt{vec}(\mathbb{D})), \quad \mathbf{Z} \in \mathbb{R}^{T\times d_q}; \tag{2.13}$$

$$\textit{query analyzer}: \quad \mathbf{y} \leftarrow \mathcal{M}_{\mathtt{ana}}(\mathbf{Z}, \mathtt{vec}(q)), \quad \mathtt{vec}(q) \in \mathbb{R}^{d_q}; \tag{2.14}$$

$$\text{ALECE estimator}: \quad \hat{s}(q) \leftarrow \mathcal{M}_{\mathtt{regression}}(\mathbf{y}). \tag{2.15}$$

The data encoder $\mathcal{M}_{\mathtt{enc}}$ learns the inter-attribute correlation and represents the data compactly in $\mathbf{Z}$; the query analyzer can be interpreted as a soft-lookup in $\mathbf{Z}$ with $\mathtt{vec}(q)$ as the lookup key; and a regression model can be learned on the output $\mathbf{y}$ to estimate the cardinality. The loss function to train ALECE can be either (weighted) mean-squared-error or Q-error of $\hat{s}(q)$ on $\mathbb{Q}$. The queries and underlying data are de-coupled in ALECE's feature space, and thus, ALECE can be continuously trained and updated when one or both of $\mathbb{D}$ and $\mathbb{Q}$ change.

2.3 Learning to Cost a Plan

Recall that the traditional cost model $\mathtt{cost}(P; s)$ relies on the cardinalities of sub-queries $s(\cdot)$'s in the plan P. We can plug in the cardinality estimates $\hat{s}(\cdot)$'s derived from the machine models introduced in Chapter 2.2 and estimate the cost as $\mathtt{cost}(P; \hat{s})$. Alternatively, we can develop machine learning models (similar to those query-driven models introduced in Chapter 2.2.3) to directly predict the *cost* of a plan P, as long as i) we can obtain *true labels/costs* (feedback obtained from query execution, *e.g.*, *execution latency* or *resource consumption*) for the models to fit, and ii) we can encode the *plan's structure* (*e.g.*, join orders and physical operators) in the input features and/or models.

In the following, we use $\texttt{cost}(P)$ to denote the true cost (some measurable metric such as execution latency or resource consumption) of executing the plan P. A model $\mathcal{M}_{\mathbb{D},\mathbb{P}}(P)$ is trained after observing a set of plans, $\mathbb{P}$, in a database $\mathbb{D}$ to predict $\texttt{cost}(P)$.

It is not difficult to encode the types of physical operators used in a plan, as they can be considered enumerable attributes associated with nodes in the plan. Meanwhile, there different ways to encode join tree or join order so that the model can utilize it to cost the plan.

2.3.1 Plan as a Sequence and Vectors

We can establish a one-to-one mapping between binary trees (*e.g.*, join trees in our context) and sequences. For example, one can reconstruct a binary tree from its pre-order and post-order traversals together, or from a parenthetic string representation of the tree (Goodrich *et al.*, 2013). Thus, for the purpose of query optimization, there different lossless ways to vectorize the tree structure of a plan.

Marcus and Papaemmanouil, 2018 encodes a plan as a sequence of matrices, where each matrix represents an "intermediate state" of the plan with each row vector specifying the heights of relations in a partial join result. For example, following are the two steps to process $q = \mathbf{R} \bowtie \mathbf{S} \bowtie \mathbf{T}$ in a plan ($\mathbf{R} \bowtie \mathbf{S}$ first and then join $\mathbf{T}$):

$$[\mathbf{R} \bowtie \mathbf{S}, \mathbf{T}] \rightarrow \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad [(\mathbf{R} \bowtie \mathbf{S}) \bowtie \mathbf{T}] \rightarrow \begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{2} \end{bmatrix}.$$

Each column corresponds to a relation in all the matrices, and the value is 1/(height of the relation in the partial join tree), or 0 if the relation is not in the partial join tree corresponding to that row. For example, in the first matrix, the first row represents the partial join tree $\mathbf{R} \bowtie \mathbf{S}$. The sequence of matrices and values in the row vectors uniquely decides the join order, and matrices can be used as "states" to train a reinforcement learning model (Marcus and Papaemmanouil, 2018).

Yuan *et al.*, 2020 represents a plan explicitly as a sequence of operators to be executed in the specified order, and each operator is represented a vector specifying its type, related relations and attributes, and parameters. The whole sequences and associated vectors can be fed into a wide-deep model to predict the cost.

Regarding each operator, there statistical information which can be obtained from the native query optimizer and used to predict the cost. For example, *the estimated number of I/Os*, *the estimated input/output size*, and *the average width of input/output tuples* from the native cost model and the optimizer can be used as features, and we can train a model for each operator to predict its cost (Akdere *et al.*, 2012).

For the purpose of estimation, we do not have to represent a plan in a lossless way. There are also "lossy" ways to featurize query plans. Akdere *et al.*, 2012 proposes a plan-level modeling approach which aggregates the above-mentioned statistical information for the whole plan as the input feature to a model. Ding *et al.*, 2019 aggregates such information (called *feature channels*) over different combinations of physical operator, execution mode, and parallelism (*i.e.*, operator_mode_parallelism as the group-by key) as features. While structural information such as join order is lost in such representations, the feature domain has a fixed width and could be sufficient for certain database tuning tasks.

2.3.2 Modeling Structures of Plans

We now present several ways to take the plan's structural information into consideration when designing machine learning models. For examples, Tree LSTM (Tai *et al.*, 2015) and Tree Convolution Networks (Mou *et al.*, 2016) are inspired by LSTM and CNN, respectively. Both can be applied to represent the tree structures of query plans in lossless ways, with different ways to encode other information, *e.g.*, operator types, relations involved, and data distributions.

Tree LSTM

The LSTM (Long Short Term Memory) networks have been shown to be effective in making prediction after observing a sequence, even with noisy inputs (*e.g.*, Yan *et al.*, 2021). Meanwhile, a plan can be naturally represented as a sequence as discussed in Chapter 2.3.1; thus, it is possible to adopt LSTM to estimate the cost of this sequence.

The native LSTM cannot be directly adopted because the sequence representing a plan has complex structure in each node, *e.g.*, specifying child-parent relationship. Sun and Li, 2019 and Yu *et al.*, 2020 propose to use Tree-LSTM model (Tai *et al.*, 2015) to run a tree structured input (a plan). Recall that an LSTM network has a functional form:

$$(\mathbf{h}_t, \mathbf{m}_t) \leftarrow \mathcal{M}_{\texttt{LSTM}}(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{m}_{t-1}),$$

where $\{\mathbf{x}_t\}$ is the input sequence, $\mathbf{h}_t$ is the output, and $\mathbf{m}_t$ is the memory at time stamp t. Parameters in $\mathcal{M}_{\texttt{LSTM}}$ are learned as more sequences are observed. Conceptually, a 2-ary Tree-LSTM can be adopted to run and represent a plan recursively in the following way:

$$(\mathbf{h}_P, \mathbf{m}_P) \leftarrow \mathcal{M}_{\texttt{LSTM}}(\mathbf{x}_P, \mathbf{h}_{P_L}, \mathbf{m}_{P_L}, \mathbf{h}_{P_R}, \mathbf{m}_{P_R}),$$

where $P = P_L \bowtie P_R$ is a plan joining two sub-plans P_L and P_R, $\mathbf{x}_P$ is the input vector that featurizes the root operator (*e.g.*, encoding its physical type, parameters, and statistics) in P, and $(\mathbf{h}_P, \mathbf{m}_P)$ is the representation of P (output and memory). $(\mathbf{h}_{P_L}, \mathbf{m}_{P_L})$ and $(\mathbf{h}_{P_R}, \mathbf{m}_{P_R})$ for sub-plans are derived recursively till the leaf nodes (single relations/attributes). $\mathbf{h}_P$ is fed into the output layers to predict $\texttt{cost}(P)$.

Tree Convolution Networks

CNN (Convolutional Neural Networks) is another way to model tree structures in plans, considering that plans usually have bounded sizes.

An early attempt to explicitly represent the structure of a query or plan as a tree is Ortiz *et al.*, 2018. It represents each sub-query as a learnable latent vector, starting from the whole query with an initial vector that captures the properties of the database. These vectors are learned in a recursive model layer by layer in the tree to predict observed metrics for (sub-)queries or plans, such as cardinalities or latencies.

A neural network architecture, QPPNet, introduced by Marcus and Papaemmanouil, 2019 for query performance prediction learns a neural unit for each type of operators (instead of one recursive model). For example, a join unit $\texttt{NN}_{\bowtie}(\mathbf{x}_1, \mathbf{x}_{\texttt{op}}, \mathbf{x}_2)$ takes representations $\mathbf{x}_1$ and

$\mathbf{x}_2$ of two partial results and features $\mathbf{x}_{\texttt{op}}$ about this operator (*e.g.*, its join type) as the inputs; a scan unit $\texttt{NN}_{\texttt{scan}}(\mathbf{x}_{\texttt{data}}, \mathbf{x}_{\texttt{op}})$ takes the representation $\mathbf{x}_{\texttt{data}}$ of data to be scanned and features $\mathbf{x}_{\texttt{op}}$ about the scan as the inputs. To make prediction for a plan $P = (\mathbf{R} \bowtie \mathbf{S}) \bowtie \mathbf{T}$, two join units are used: $\mathbf{y} \leftarrow \texttt{NN}_{\bowtie}(\texttt{NN}_{\bowtie}(\mathbf{x}_{\mathbf{R}}, \mathbf{x}_{\texttt{join1}}, \mathbf{x}_{\mathbf{S}}), \mathbf{x}_{\texttt{join2}}, \mathbf{x}_{\mathbf{T}})$; the output $\mathbf{y}$ is fed into additional layers to predict the performance. Note that a neural unit is not bound to a specific operator; instead, its parameters are shared across all operators in the same type. In the above example, the two join units (one for $\mathbf{R} \bowtie \mathbf{S}$ and one for $\bowtie \mathbf{T}$) share the same learnable parameters. Thus, during the processes of training and inference, for a plan in any shape, proper neural units can be assembled on-the-fly to fit the true cost or to make prediction.

The learned cost models in more recent works (*e.g.*, Marcus *et al.*, 2019; Marcus *et al.*, 2021; Kang *et al.*, 2021; Yang *et al.*, 2022; Zhu *et al.*, 2023) follow a scheme called *tree convolution networks* (Mou *et al.*, 2016) to make the model understand tree structures. Since a query plan could be in any shape (in the class of binary trees), we rely on an operation called *tree-based convolution* to aggregate (3-way pooling) information from an operator in the plan tree and its two children (for an operator with only one child, we can always add a dummy child under it). A tree-based convolution operation with learnable weights is always applied on the whole plan to propagate information from all children to their parents; and multiple tree-based convolution operators with different dimensionality and weights can be applied to propagate information from leaves to the root. At a very high level, a neural unit in QPPNet can be analogous to a tree-based convolution matrix in a tree convolution network, for each type of operators.

More concretely, each node v in the plan tree P could be either a relation in the database $\mathbb{D}$ if it is a leaf, or an operator (*e.g.*, join or scan). Initially, we encode the following information in the representation (embedding vector) $\mathbf{x}_v$ of each node v: i) the operator's type; ii) parameters of the operator; iii) relations involved in the operator; iv) data distribution of relations. One-hot encoding can be used for i)-iii), and iv) can be encoded as histograms directly or with learnable embeddings. $\mathbf{x}_v$ has the same size for all nodes; for leaf node, i) and ii) empty and we can pad 0's on the corresponding positions. A tree-based convolution operation is applied on all nodes at one time:

$$\mathbf{x}'_v = \sigma\left(W_{\texttt{op}} \cdot \mathbf{x}_v + W_{\texttt{left}} \cdot \mathbf{x}_{\texttt{left}(v)} + W_{\texttt{right}} \cdot \mathbf{x}_{\texttt{right}(v)} + \mathbf{b}\right).$$

For every node v, $\mathbf{x}_v \in \mathbb{R}^d$, and $\texttt{left}(v)$ and $\texttt{right}(v)$ are the two children of v. $W_{\texttt{op}}$, $W_{\texttt{left}}$, $W_{\texttt{right}} \in \in \mathbb{R}^{d' \times d}$ are learnable convolution matrices and $\mathbf{b} \in \mathbb{R}^d$ is a learnable bias. $\sigma(\cdot)$ is a non-linear transformation (*e.g.*, ReLU) which outputs $\mathbf{x}'_v \in \mathbb{R}^{d'}$. This convolution operation is parameterized by $(W_{\texttt{op}}, W_{\texttt{left}}, W_{\texttt{right}}, \mathbf{b})$. Multiple tree-based convolution operations can be applied one by one on the initial representation $\mathbf{x}_v$'s (the output of one operation is the input to the next one). The final output is sent to additional layers to make the prediction.

Other Approaches

Zhao *et al.*, 2022b extends the Transformer architecture (Vaswani *et al.*, 2017) with *height encoding* (inspired by the Positional Encoding method) and *tree-bias attention* to model tree structures in query plans. The extended model, called QueryFormer, can be used for cost estimation as well as other estimation tasks. Hilprecht and Binnig, 2022 introduces zero-shot cost models, which enable learned cost estimation that generalizes to unseen databases. Saturn (Liu *et al.*, 2022) encodes each query plan tree into a compressed vector by using a traversal-based query plan auto-encoder to cope with diverse plan structures. The compressed vectors can be leveraged to distinguish different query types, which is highly useful for downstream tasks.

2.3.3 Modeling Concurrent Queries and Cloud Workloads

The task of cost estimation in a cloud environment with complex workload of concurrent and correlated queries could be more complicated. Duggan *et al.*, 2011 started this line of research by proposing a modeling approach to estimate the impact of concurrency on query performance, with *query latency indicators* identified. These indicators are metrics that capture the resource contention in the underlying hardware and can be used to predict the latency of concurrent queries.

The implications of such complexity in the cloud environment are two-fold. On one hand, the performance of query execution depends on the amount of allocated resource. For example, Li *et al.*, 2022 proposes a resource-aware deep learning model that can predict the execution time of query plans based on historical execution statistics and features related the allocated resource. To train their model, it featurizes the query execution plans as well as the allocated resources. A deep learning model with adaptive attention mechanisms is then trained to predict the execution time of query plans. Most learned cost models introduced in Chapter 2.3 for single queries can be also extended for handling concurrent queries by incorporating resource-based features into the original models (*e.g.*, as in Marcus and Papaemmanouil, 2019).

On the other hand, more information can be retrieved from such an environment to train machine learning models. For example, Wu *et al.*, 2018 extracts overlapping sub-query templates that appear across multiple queries in the workload, and trains a cardinality estimation model with data collected from these templates with varying parameters and inputs. GPredictor (Zhou *et al.*, 2020) utilizes the graph neural network to capture the correlation between plans of different queries, and the calibrates the prediction across different plans; thus, the performance of a specific plan can be estimated more accurately.

Siddiqui *et al.*, 2020 investigates two key questions about cost models for big-data query optimization and processing: i) whether we can learn accurate cost models for big data systems; ii) whether we can integrate the learned models within the query optimizer.

3

Exploring Plan Space

A native query optimizer, without additional changes, can use a learned cost model $\hat{\texttt{cost}}(P)$, which is derived by either plugging a learned cardinality estimator (introduced in Chapter 2.2) in the traditional cost model or directly invoking a machine learning model for predicting plan cost (introduced in Chapter 2.3), to improve its performance. This is also one way how previous works on learning-based cardinality/cost estimation evaluate the accuracy of their approaches. This straightforward integration of learned models into plan enumerators, however, may not be an effective way to explore the plan space for two reasons.

First, learned cardinality or cost estimation models usually have non-trivial inference costs. If a plan enumerator invoke such modes too many times, its optimization cost can be prohibitive.

Second, recall that a model for predicting plan cost, $\hat{\texttt{cost}}(P) \leftarrow \mathcal{M}_{\mathbb{D},\mathbb{P}}(P)$, is trained and updated after observing a historical collection $\mathbb{P}$ of plans. $\mathbb{P}$ can be considered as the plan space explored by the plan enumerator in the query optimizer on a workload of queries so far. There are opportunities to co-design the plan enumerator for selecting $\mathbb{P}$ and exploring the plan space more actively, so that $\mathcal{M}_{\mathbb{D},\mathbb{P}}$ can be continuously tuned as more queries and plans are observed, and high-quality plans are not missed during the search procedure.

We will introduce three types of approaches to explore the plan space. The first one directly utilizes the model $\mathcal{M}_{\mathbb{D},\mathbb{P}}$ for cost prediction, and actively generates multiple candidate plans for a given query to balance exploration and exploit (Chapter 3.1). The second one twists $\mathcal{M}_{\mathbb{D},\mathbb{P}}$ to be a value network (by twisting the loss function a bit) and conducts generative search to build up plans (Chapter 3.2). The third one is based on the idea of learning to rank (Chapter 3.3).

3.1 Generating Candidate Plans: Exploration and Exploit

The purpose of generating more than one candidate plans for a given query q is two-fold. First (*exploit*), we want to select the best candidate plan for execution using the trained model $\mathcal{M}_{\mathbb{D},\mathbb{P}}$. Second (*exploration*), since it is expensive to obtain true labels for $\mathcal{M}_{\mathbb{D},\mathbb{P}}$ by executing plans, we want to select a candidate plan that helps build up more experience after being added to $\mathbb{P}$ and improve the accuracy of $\mathcal{M}_{\mathbb{D},\mathbb{P}}$.

A formulation based on multi-armed bandits. Marcus *et al.*, 2021 provides a formulation based on contextual multi-armed bandits.

Consider a workload $\mathbb{Q}$ of queries. For each query $q \in \mathbb{Q}$, suppose we can ask an oracle $\mathbb{C}(q)$, *i.e.*, *candidate generator*, to generate a set of candidate query plans for q. The number of candidate plans $|\mathbb{C}(q)|$ is typically much smaller than the total number of valid plans $|\mathbb{P}(q)|$. We will introduce several ways to implement the candidate generator later.

An algorithm $\mathcal{A}_{\mathbb{C}}(q)$ selects from $\mathbb{C}(q)$ a plan $\bar{P}$ to be executed. After its execution, $\bar{P}$ is added to $\mathbb{P}$ and $\mathcal{M}_{\mathbb{D},\mathbb{P}}$ is updated. The *regret* for a query q is the difference between the performance of $\mathcal{A}_{\mathbb{C}}(q)$'s selection and the best of candidates in $\mathbb{C}(q)$. Formally, the regret is

$$R_q = \texttt{cost}(\mathcal{A}_{\mathbb{C}}(q)) - \min_{P \in \mathbb{C}(q)} \texttt{cost}(P).$$

Each "arm" here is a choice from $\mathbb{C}(q)$. By utilizing and updating $\mathcal{M}_{\mathbb{D},\mathbb{P}}$ over time, the algorithm $\mathcal{A}_{\mathbb{C}}$ aims to improve its selection and get closer to choosing optimally from $\mathbb{C}(q)$ (*i.e.*, minimizing regret). Note that the goal is not too aggressive, as we are not aiming at choosing the best possible plan among all valid plans $\mathbb{P}(q)$. In the long run, we may want to minimize the regret for the workload $\mathbb{Q}$, $R(\mathbb{Q}) = \mathbf{E}[\sum_q R_q]$, and thus balance the trade-off between exploration and exploitation.

Utilizing and Updating Model

Assuming that the model $\mathcal{M}_{\mathbb{D},\mathbb{P}}$ does not have significant bias, it is always a wise choice for $\mathcal{A}_{\mathbb{C}}$ to select the plan with the minimum estimated cost for execution, *i.e.*, $\bar{P} \leftarrow \arg\min_{P\in\mathbb{C}(q)} \mathcal{M}_{\mathbb{D},\mathbb{P}}(P)$. After being executed, $\bar{P}$ is added into $\mathbb{P}$ with the performance statistics (*i.e.*, true cost) collected. Marcus *et al.*, 2021 adapts the Thompson sampling algorithm (Thompson, 1933) to update $\mathcal{M}_{\mathbb{D},\mathbb{P}}$. Suppose $\mathcal{M}_{\mathbb{D},\mathbb{P}}$ has model parameters (weights) $\boldsymbol{\theta}_{\mathcal{M}}$. According to the scheme of Thompson sampling, theoretically, whenever a new plan is executed and added to $\mathbb{P}$, the model $\mathcal{M}_{\mathbb{D},\mathbb{P}}$ should be re-trained with $\boldsymbol{\theta}_{\mathcal{M}}$ sampled from the distribution $\mathrm{P}(\boldsymbol{\theta}_{\mathcal{M}} \mid \mathbb{D}, \mathbb{P})$. Two questions remain: i) how to ensure the sampling properties about $\boldsymbol{\theta}_{\mathcal{M}}$; ii) in the context of query optimization, it is not practical to re-train the model whenever $\mathbb{P}$ is updated.

$\mathcal{M}_{\mathbb{D},\mathbb{P}}$ is usually a neural network (*e.g.*, tree LSTM network or tree convolution network introduced in Chapter 2.3.2) and normally a training algorithm finds the most likely model weights, instead of sampling its weights from $\mathrm{P}(\boldsymbol{\theta}_{\mathcal{M}} \mid \mathbb{D}, \mathbb{P})$. Marcus *et al.*, 2021 proposes to adopt an alternative implementation of Thompson sampling (Osband and Roy, 2015): the model is trained as usual, but on a “bootstrap” of the training data: when the network needs to be trained, we draw $|\mathbb{P}|$ random samples *with replacement* from $\mathbb{P}$ as the training data, inducing the desired properties about the weights $\boldsymbol{\theta}_{\mathcal{M}}$.

In practice, Marcus *et al.*, 2021 re-train $\mathcal{M}_{\mathbb{D},\mathbb{P}}$ when every k new plans are added to the observation $\mathbb{P}$ (instead of every plan). When $\mathbb{P}$ becomes too large, we can focus on the most recent m plans observed.

Implementing Candidate Generator

A straightforward implementation of the candidate generator is to draw random samples from all valid plans $\mathbb{P}(q)$ for a query q. The obvious drawback is that, with high likelihood, high-quality plans could be missing in a random sample; otherwise, the sample size has to be too large for the algorithm $\mathcal{A}_{\mathbb{C}}$ to pick a good plan with reasonable cost.

There are two requirements for the candidates in $\mathbb{C}(q)$. i) For the purpose of *query optimization*, the candidate list must include some truly good plans for consideration. ii) For the purpose of *plan exploration*, we want to catch past optimization mistakes and adjust the model $\mathcal{M}_{\mathbb{D},\mathbb{P}}$ timely using the newly observed runtime information, and thus candidate plans in $\mathbb{C}(q)$ needs to be diversified sufficiently. To this end, there are two ways to design and implement the candidate generator without intrusively modify the plan enumerator in query optimizers.

Tuning hint-set for candidate generation. Bao (Marcus *et al.*, 2021) proposes to correct the native query optimizer and enforce it to explore different possibilities via hint-set tuning. There are binary hints (boolean flags) available in RDBMS for the DBA to disable/force certain types of optimization rules. For example, disable/force index usage and disable/force merge join. Let $\mathcal{H}$ be the set of all such hints. Any subset of $\mathcal{H}$ can be used to enforce the native query optimizer to generate a potentially different plan with higher quality. For a query q, we can enumerate all hint sets $H \in 2^{\mathcal{H}}$, invoke the native optimizer for each hint set, and keep the unique resulting plans as $\mathbb{C}(q)$

The worst-case choice by the search algorithm $\mathcal{A}_{\mathbb{C}}$ is bounded by the worst plan in $\mathbb{C}(q)$, while some truly good plans could be generated by $\mathbb{C}(q)$ under certain hint sets. For example, for a highly selective query, a left-deep loop join plan could be a good candidate in $\mathbb{C}(q)$ by disabling hash and merge joins even if the cardinality estimation is erroneous.

There are also drawbacks of this approach. First, modern databases may expose up to hundreds of hints. That is, $\mathcal{H}$ could be too large for us to try all hint sets $H \in 2^{\mathcal{H}}$, especially in industrial deployments (Negi *et al.*, 2021b; Zhang *et al.*, 2022), even though there are a lot of duplicates among the resulting plans. Second, when $\mathcal{H}$ is large, the selection of hint sets for $\mathbb{C}(q)$ to generate candidates requires good knowledge about the query optimizer and the workload. Anneser *et al.*, 2023 proposes a greedy hint-set search algorithm to generate promising candidates with reasonable overhead, and a generic hint-set steering framework that can be connected to various databases. Another alternative proposed by

Woltmann *et al.*, 2023 is to model the hint-set selection problem as a classification task and train a query-context-aware classification model to help pick the best hint set.

Tuning cardinality estimator for candidate generation. Lero (Zhu *et al.*, 2023) uses the cardinality estimator as a tuning knob to generate candidate plans $\mathbb{C}(q)$ for each query q. Recall the relationship between cardinality and cost (in Chapter 2.1). The cost model used in the native query optimizer, $\texttt{cost}(P; \hat{s})$, invokes the cardinality estimator $\hat{s}(\cdot)$ for sub-queries in P to calculate the cost estimate. Theorem 2.1 says that, in order to find high-quality plans, the cardinality estimates for sub-queries just needs to be close enough to their true cardinalities. Thus, we can purposely scaled up/down the cardinality estimates for sub-queries before feeding them into the cost model in the native query optimizer. The optimizer may either generate a better plan for a query q (if tuning towards the true cost of a sub-query which was incorrectly estimated), or a worse plan (tuning in the opposite direction).

Plans with various quality also increase the diversity among candidates. In particular, it is shown that diversity can be introduced in the generated plans by tuning selectivities (cardinalities) of predicates in a query Dey *et al.*, 2008 and Doraiswamy *et al.*, 2008.

There are still too many ways to tune the cardinality estimates for sub-queries. Thus, for a query q, at one time, we can focus on its size-k sub-queries and scale their cardinality estimates all by a factor of α (when the native optimizer asks for them), while keeping the estimates for other sub-queries unchanged). For each choice of (k, α), the native optimizer can be invoked and generate a candidate plan to be added into $\mathbb{C}(q)$ (note that we do not execute it at this point). One advantage of this candidate generator is that it replies only on a fundamental component, the cardinality estimator, of cost-based query optimizers, but is not bound to any specific version or brand of RDBMS.

3.2 Generative Search with Value Network

In generative search, we build up a plan from scratch (bottom-up). For a given query, initially no operator is executed. We call it the *initial state* or the *empty subplan*. A subplan can *transit* to another subplan if one more operator is specified: i) specifying a scan to be a table scan or an index scan on a relation; or ii) specifying two relations or intermediate results to be joined and the join type. Consider a simplified example with a query joining four relations $q = \mathbf{A} \bowtie \mathbf{B} \bowtie \mathbf{C} \bowtie \mathbf{D}$:

$$\overbrace{[\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}]}^{P_0} \Rightarrow \overbrace{[\mathbf{A} \bowtie \mathbf{B}, \mathbf{C}, \mathbf{D}]}^{P_1} \Rightarrow \overbrace{[\mathbf{A} \bowtie \mathbf{B}, \mathbf{C} \bowtie \mathbf{D}]}^{P_2} \Rightarrow \overbrace{[(\mathbf{A} \bowtie \mathbf{B}) \bowtie (\mathbf{C} \bowtie \mathbf{D})]}^{P_3}.$$

P_0 is the empty subplan; in subplan P_1, **A** and **B** are joined; then **C** and **D** are joined in P_3; and eventually P_3 is the complete plan.

Starting from the empty subplan P_0, there are a lot of ways to *reach* a complete plan after multiple transition steps. The goal of generative search is to find a transition path to reach the best complete plan. Different from dynamic programming algorithms in traditional query optimizers (*e.g.*, Moerkotte and Neumann, 2008) and approaches based on candidate generation (introduced in Chapter 3.1), instead of costing a plan or a subplan, a generative search algorithm (Marcus *et al.*, 2019; Yang *et al.*, 2022) relies on a *value network* $\mathtt{V}(P, q)$ to predict the performance of the best-possible complete plan that a subplan P could reach for a query q. We denote $P \preceq P'$ iff a subplan P can *reach* P' after one or more transition steps. Formally, the value network estimates

$$\mathtt{V}(P, q) \approx \min\{\mathtt{cost}(P') \mid P \preceq P' \wedge P' \in \mathbb{P}(q)\}.^{1} \tag{3.1}$$

Recall that $\mathbb{P}(q)$ is the set of valid plans (complete plan) for q.

Assuming the value network $\mathtt{V}(P, q)$ has been trained, there are many generative search algorithms with different complexity-optimality tradeoffs, ranging from greedy search, to best-first heap or beam search (Marcus *et al.*, 2019; Yang *et al.*, 2022) and Monte Carlo tree search.

For a given query q, the best-first heap search works as follows. A min heap is initialized with only one element, the empty subplan P_0. The heap is ordered by the value network's output for each subplan (*i.e.*, the estimated cost of the best complete plan that a subplan could reach). In each iteration, the top of the min heap, denoted by $P_{\mathtt{top}}$, is removed. Every subplan P' that $P_{\mathtt{top}}$ can transit to (with one more operator

[1]Note that it is different from the convention of value networks in reinforcement learning here: the smaller $\mathtt{V}(P, q)$, is the better.

specified) is inserted into the heap. As more and more operators are specified, complete plans for the query q will be added to the heap. The first complete plan that is removed from the heap is expected to be the best plan, if the value network's estimation is accurate.

There are a lot of variants of the above algorithm. If the value network's estimation is accurate, the greedy search suffices (Krishnan *et al.*, 2018), *i.e.*, for each subplan removed from the heap, we only add the best subplan it can transit to back. Since the value network's estimate is usually inaccurate, we normally add the best k subplans back to the heap, which is called *k-beam search*; meanwhile, we may continue the search procedure till c complete plans are removed from the top, and the best among them is used. We can also restrict the heap to keep only the best b subplans to speed up the search procedure (Yang *et al.*, 2022). Chen *et al.*, 2023a proposes a fusion of ϵ-greedy algorithm (Watkins, 1989) and beam search, called ϵ-beam search: among the k subplans in beam search to be inserted into the heap, a random number (whose distribution depends on ϵ) of subplans are generated randomly, for the purpose of exploring more diversified plan space.

The class of heap search algorithms have the benefit of being "anytime algorithms"–they can be run for as long as needed (under a budget of optimization cost), getting better results (in expectation) the longer they search. The greedy search has lower complexity than the traditional dynamic programming search, assuming an accurate value network.

We can also learn a policy function π : subplan $\rightarrow$ operator to guide the search, *i.e.*, for a subplan (*state*) which operator to be added (*action*), as in RTOS (Yu *et al.*, 2020) and BASE (Chen *et al.*, 2023b).

Training the value network. A machine learning model $\mathcal{M}_{\mathbb{D},\mathbb{P}}(P)$ introduced in Chapter 2.3 can be trained as the value network $\mathtt{V}(P, q)$. For most model architectures introduced in Chapter 2.3, features about the query q can be easily encoded; however, it is not straightforward to obtain the *true label*, *i.e.*, the R.H.S. of (3.1), the performance of the best-possible complete plan that the subplan P can read for answering q. Thus, Marcus *et al.*, 2019 and Yang *et al.*, 2022 approximate the true label of $\mathtt{V}(P, q)$ with the best performance of q's complete plans that the subplan P can reach and have been seen so far by the system.

The model is trained to minimize the L2 loss between predicted and such approximate true label. With different goals of selecting execution plans, other labels (*e.g.*, relative execution latency) and loss functions are also introduced by Marcus *et al.*, 2019 and Chen *et al.*, 2023a,

Note that the label for a data point (P, q) obtained in the above way can be updated, if a better complete plan for the query q is found after the system is running for a while. Thus, when a data point is used again later for updating the model, we rely on its most recent label to alleviate the impact of inaccurate labels in previous training iterations.

Cold-start and model updating. There is a lack of training data to train the value network $\mathtt{V}$ before sufficient plans are observed and executed for the query workload. Yang *et al.*, 2022 proposes to use the native cost model in the query optimizer for cold-start. That is, a simulated value network $\mathtt{V}_{\mathtt{sim}}$ is firstly trained on a synthetic query workload without executing any plan; the label of a data point (P, q), *i.e.*, R.H.S. of (3.2), is generated from the native cost model $\hat{\mathtt{cost}}$:

$$\mathtt{V}_{\mathtt{sim}}(P, q) \approx \min\{\hat{\mathtt{cost}}(P') \mid P \preceq P' \wedge P' \in \mathbb{P}(q)\}. \tag{3.2}$$

The labels for training $\mathtt{V}_{\mathtt{sim}}$ can be calculated using dynamic programming (similar to how the native query optimizer search for the best plan using $\hat{\mathtt{cost}}$). After fully trained, model parameters of $\mathtt{V}_{\mathtt{sim}}$ are copied to the real value network $\mathtt{V}$ (with the same network architecture), and $\mathtt{V}$ is used in generative search and fine-tuned using real execution feedback.

At the beginning, the value network $\mathtt{V}$ can be fine-tuned with a sample workload. As more and more execution statistics are collected, $\mathtt{V}$ is updated periodically: we can either sample a batch of training data points from the whole historical observation (Marcus *et al.*, 2019) or sample training data points from iterations since $\mathtt{V}$'s last update, *i.e.*, so-called *on-policy learning* (Yang *et al.*, 2022).

3.3 Learning to Rank Plans

Approaches introduced in Chapter 3.1 and Chapter 3.2 reply on cost prediction models and value networks (potential prediction models) to explore the plan space. While it is natural to develop regression models for such prediction tasks, Zhu *et al.*, 2023 asks: for the purpose of query optimization, do we really need to estimate/predict the execution latency or any other performance metric in the form of cost model?

In essence, what we need for query optimization is a *learned* oracle that is able to *rank* a set of valid query plans with respect to their execution efficiency. Looking back, traditional cost models are essentially "human learning" models, whose parameters, *i.e.*, magic constant numbers, have been tuned with decades of engineering efforts. Under the *learning-to-rank* paradigm, traditional and learned cost models can be regarded as *pointwise* approaches (Liu, 2009), which outputs an ordinal score (i.e., estimated cost) for each plan to rank them. Lero (Zhu *et al.*, 2023) adopts an *pairwise* learning-to-rank approach.

Zhu *et al.*, 2023 explores the plan space via candidate generation (as introduced in Chapter 3.1) and pairwise plan comparison.

A *comparator model* is trained to compare two plans and predict the better one (binary answer). Compared to other *pointwise* approaches, training such a binary classifier is often easier (Johannes and Eyke, 2011). Note that a pairwise comparison model is also used for index tuning by Ding *et al.*, 2019. More formally, a *comparator* generates a binary answer for any two plans P_1 and P_2

$$\texttt{cmp}(P_1, P_2) = \begin{cases} 0 & \text{if } \texttt{latency}(P_1) \leq \texttt{latency}(P_2) \\ 1 & \text{if } \texttt{latency}(P_1) > \texttt{latency}(P_2) \end{cases}, \tag{3.3}$$

based on specified performance metric, such as latency. A comparator model $\mathcal{M}^{\texttt{cmp}}_{\mathbb{D},\mathbb{P}}(\cdot,\cdot) \rightarrow (0,1)$ can be trained to approximate the above oracle cmp, such that if $\mathcal{M}^{\texttt{cmp}}_{\mathbb{D},\mathbb{P}}(P_1, P_2) \leq 0.5$, we predict $\texttt{cmp}(P_1, P_2) = 0$, and if otherwise, $\texttt{cmp}(P_1, P_2) = 1$ (as in a classification task).

For a given query q, a set of candidate plans $\mathbb{C}(q)$ are generated with the approach introduced in Chapter 3.1. The plan comparator is invoked $|\mathbb{C}(q)| - 1$ times to select the best execution plan.

In the above selection process, the comparator model needs to preserve two properties (otherwise, different orders of plans in $\mathbb{C}(q)$ would lead to different selection results). The first one is *commutativity*:

$$\mathcal{M}^{\texttt{cmp}}_{\mathbb{D},\mathbb{P}}(P_1, P_2) = 1 - \mathcal{M}^{\texttt{cmp}}_{\mathbb{D},\mathbb{P}}(P_2, P_1),$$

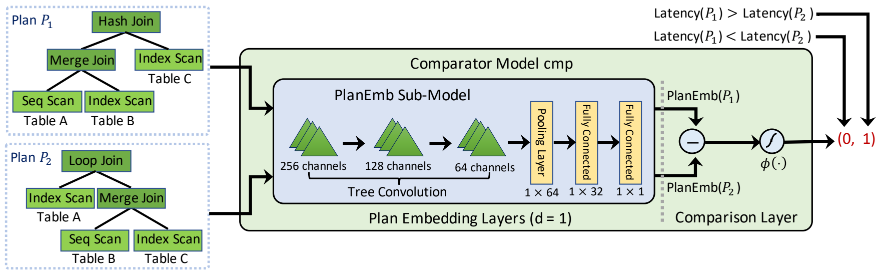

Figure 3.1: A plan comparator model with $d = 1$ (Zhu *et al.*, 2023).

that is, exchanging the order of input plans does not affect the comparison result. The second is *transitivity*:

$$\mathcal{M}^{\mathtt{cmp}}_{\mathbb{D},\mathbb{P}}(P_1, P_2) < 0.5 \wedge \mathcal{M}^{\mathtt{cmp}}_{\mathbb{D},\mathbb{P}}(P_2, P_3) < 0.5 \Rightarrow \mathcal{M}^{\mathtt{cmp}}_{\mathbb{D},\mathbb{P}}(P_1, P_3) < 0.5,$$

that is, P_1 is better than P_2 and P_2 is better than P_3 imply that P_1 is better than P_3. We will introduce how to train such a comparator next.

A comparator model. Zhu *et al.*, 2023 introduce a simple but effective way to extend single-plan costing models (*e.g.*, those introduced in Chapter 2.3) for comparing pairs of plans. The comparator model $\mathcal{M}^{\mathtt{cmp}}_{\mathbb{D},\mathbb{P}}(P_1, P_2)$ (outlined in Figure 3.1) has two copies of a *plan embedding sub-model* `PlanEmb`$(\cdot)$, which share the same model architecture and values of learnable parameters. `PlanEmb`(P_1) and `PlanEmb`(P_2) map plans P_1 and P_2 from the original feature space to a d-dim embedding space, in order to learn differences between plans. Either tree LSTM or tree convolution networks introduced in Chapter 2.3.2 can be used as the sub-model `PlanEmb`, and its two copies in the comparator are trained and updated together. The plan embeddings of P_1 and P_2 are compared with the following *comparison layers* to generate an output in $(0, 1)$, indicating which one is better. For $d = 1$, we can simply feed their difference $x = \mathtt{PlanEmb}(P_1) - \mathtt{PlanEmb}(P_2)$ into an activation function $\phi(x) = (1 + \exp(-x))^{-1}$ to generate the final output.

It can be shown that no matter whatever values of parameters are learned, the above comparator model satisfies both commutativity and transitivity for $d = 1$ (Zhu *et al.*, 2023).

Training and updating the comparator. The comparator is trained to maximize the likelihood of outputting the right order of any two plans. Lero (Zhu *et al.*, 2023) introduces a loss function to this end.

Instead of training from scratch, Lero pre-trains the comparator model offline first on synthetic workloads to inherit the wisdom of the native query optimizer (similar to how to cold start the training of value network in Yang *et al.*, 2022). For each query in the workload, its candidate plans are compared under the native cost model (without being executed). The comparison results of pairs of plans are used as labels to pre-train the comparator model. As the native cost models are often in a class of functions with simple structures, the model pre-training converges very fast, and after that, the comparator model performs at least as good as the native cost model.

The comparator model is then continuously trained and updated online on real workloads with training data collected during actual executions of plans. Suppose the system has executed h plans (of possibly different queries) and a new plan is selected and executed, we can compare the latency of the new plan with that of the previously executed plans and form h plan pairs for training. Meanwhile, since idle workers commonly exists in big data platforms (Zhou *et al.*, 2012; Negi *et al.*, 2021a) for, *e.g.*, performance testing, we can use them to execute $k > 1$ candidate plans for each query (when idle workers are available), and form $k(k-1)/2$ pairs for training. The comparator model is updated periodically with batches of such training data points.

4

Open Research Challenges

There are still open challenges for both researchers and engineers to tackle in order to deploy the algorithmic and system innovations for learned query optimizers in production. We conclude this survey with the discussion of some open research challenges.

Pre-training models. The very first few questions a database engineer would raise when being asked to integrate a learned query optimizer into a commercial database are how expensive it is to train the machine learning models and how much training data we need to collect. While it is important to ensure that the models used in learned query optimizers are not too large, and can be cold-started with little or no execution statistics as in, *e.g.*, Balsa (Yang *et al.*, 2022) and Lero (Zhu *et al.*, 2023), pre-training models is one way to alleviate such concerns. We can pre-train the models in a learned query optimizer with a large and diverse corpus of datasets, and fine-tune them (if necessary) with only a little resource when the optimizer is used on a specific workload. We want the pre-trained models to capture two types of information.

The first type of information is some common frequency and correlation patterns in structured datasets, especially for those generated in the same vertical domain (*e.g.*, finance or sports). This is analogous to pre-trained models that are vastly used for images and text. For the task of cardinality estimation, Iris (Lu *et al.*, 2021) is the first attempt to pre-train encoders for data summarization and pre-train decoders to generate cardinality estimates with these summaries. The summarization model can be incrementally updated for a specific workload.

The second type of information is the universal and workload-invariant logic behind a task. For example, some traditional cardinality estimators are closed-form formulas on data sketches such as histograms; these formulas or estimation algorithms are derived under certain assumptions about the contents of databases (*e.g.*, the independence assumption) and can be vulnerable when the assumptions are violated. Meanwhile, for some statistics such as *number of distinct values* (NDV), it is inherently difficult to derive accurate estimations from data sketches (Charikar *et al.*, 2000) with worst-case guarantees. Via pre-training on a diverse corpus, the ability of universal approximation from neural networks (Sonoda and Murata, 2017) may help explore the function space for more robust or instance-optimal estimations (especially those that cannot be represented as statistical estimators). Wu *et al.*, 2021 is such an attempt for estimating NDV with sample sketches.

System infrastructure. The deployment of learned query optimizers in databases requires close cooperation between machine learning and database developers and heavy engineering cost. These two communities have very different programming paradigms (*e.g.*, PyTorch v.s. C). Although we expect that the gap is closing, it is rare that one developer masters both. Thus, it is important to have an abstraction of the learned query optimizer to be deployed so that machine learning and database developers can focus their own components, and facilitate the resource management and isolation for model training and inference.

Ideally, we want to build learned query optimizers as middlewares that can be plugged into different database systems (*e.g.*, from PostgreSQL to Spark) with minimal efforts of modifying the implementations of certain interfaces, while machine learning developers can focus on iterating the algorithms and models. Moreover, for a fair benchmark of different solutions, it is better for them to share as many atomic operators as possible in such a middleware (*e.g.*, how plans are fetched from and pushed into the database for execution). AutoSteer (Anneser *et al.*, 2023) and PilotScope (Zhu *et al.*, 2024) are two early attempts on designing and implementing such a system infrastructure.

Quality control. Quality control of plans generated by a learned query optimizer, or a new version of an optimizer, is not a new challenge. For example, Apollo (Jung *et al.*, 2019) can take in two versions of a query optimizer, automatically detect performance regressions, locate the root cause of regressions, and produce performance regression reports.

According to the experimental studies in, *e.g.*, Marcus *et al.*, 2021, Yang *et al.*, 2022, and Zhu *et al.*, 2023, after properly trained, learned query optimizers are able to generate better plans than the native query optimizers in database systems do for a majority of queries in standard benchmarks such as TPC-H and TPC-DS, and thus provide non-trivial performance improvement on average. However, performance regression seems inevitable for some queries due to the inherent difficulty in model generalization in these learned query optimizers.

Negi *et al.*, 2021b devises an offline pipeline which utilizes SCOPE's compiler, flags, and A/B testing infrastructure to analyze historical jobs, and extracts effective configurations to learn models for future recurring jobs. It can be also interpreted as a framework which uses idle resources to run A/B testings for quality control in recurring workloads.

Quality control can be formulated as a classification task, which determines whether the plan generated by the learned query optimizer is better than the plan generated by the native optimizer. If not, we can always execute the latter instead. Ideally, if we can train a classification model that is sufficiently accurate, the deployment of a learned query optimizer can bring us robust online performance gains. Eraser (Weng *et al.*, 2024) is an early attempt in this direction, but we expect more efforts to make learned query optimizers truly applicable.

References

Akdere, M., U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. (2012). "Learning-based Query Performance Modeling and Prediction". In: *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*. IEEE Computer Society. 390–401. DOI: 10.1109/ICDE.2012.64. URL: https://doi.org/10.1109/ICDE.2012.64.

Anneser, C., N. Tatbul, D. E. Cohen, Z. Xu, P. Pandian, N. Laptev, and R. Marcus. (2023). "AutoSteer: Learned Query Optimization for Any SQL Database". *Proc. VLDB Endow.* 16(12): 3515–3527. DOI: 10.14778/3611540.3611544. URL: https://www.vldb.org/pvldb/vol16/p3515-anneser.pdf.

Atserias, A., M. Grohe, and D. Marx. (2013). "Size Bounds and Query Plans for Relational Joins". *SIAM J. Comput.* 42(4): 1737–1767. DOI: 10.1137/110859440. URL: https://doi.org/10.1137/110859440.

Begoli, E., J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire. (2018). "Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM. 221–230. DOI: 10.1145/3183713.3190662. URL: https://doi.org/10.1145/3183713.3190662.

Cai, W., M. Balazinska, and D. Suciu. (2019). "Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities". In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM. 18–35. DOI: 10.1145/3299869.3319894. URL: https://doi.org/10.1145/3299869.3319894.

Charikar, M., S. Chaudhuri, R. Motwani, and V. R. Narasayya. (2000). "Towards Estimation Error Guarantees for Distinct Values". In: *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 15-17, 2000, Dallas, Texas, USA*. ACM. 268–279. DOI: 10.1145/335168.335230. URL: https://doi.org/10.1145/335168.335230.

Chen, T., J. Gao, H. Chen, and Y. Tu. (2023a). "LOGER: A Learned Optimizer towards Generating Efficient and Robust Query Execution Plans". *Proc. VLDB Endow.* 16(7): 1777–1789. URL: https://www.vldb.org/pvldb/vol16/p1777-gao.pdf.

Chen, X., Z. Wang, S. Liu, Y. Li, K. Zeng, B. Ding, J. Zhou, H. Su, and K. Zheng. (2023b). "BASE: Bridging the Gap between Cost and Latency for Query Optimization". *Proc. VLDB Endow.* 16(8): 1958–1966. URL: https://www.vldb.org/pvldb/vol16/p1958-chen.pdf.

Cormode, G., M. N. Garofalakis, P. J. Haas, and C. Jermaine. (2012). "Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches". *Found. Trends Databases.* 4(1-3): 1–294. DOI: 10.1561/1900000004. URL: https://doi.org/10.1561/1900000004.

Davitkova, A., D. Gjurovski, and S. Michel. (2022). "LMKG: Learned Models for Cardinality Estimation in Knowledge Graphs". In: *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*. OpenProceedings.org. 2:169–2:182. DOI: 10.48786/edbt.2022.07. URL: https://doi.org/10.48786/edbt.2022.07.

Dey, A., S. Bhaumik, H. Doraiswamy, and J. R. Haritsa. (2008). "Efficiently approximating query optimizer plan diagrams". *Proc. VLDB Endow.* 1(2): 1325–1336. DOI: 10.14778/1454159.1454173. URL: http://www.vldb.org/pvldb/vol1/1454173.pdf.

Ding, B., S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. R. Narasayya. (2019). "AI Meets AI: Leveraging Query Executions to Improve Index Recommendations". In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM. 1241–1258. DOI: 10.1145/3299869.3324957. URL: https://doi.org/10.1145/3299869.3324957.

Dong, R., J. Liu, Y. Zhu, C. Yan, B. Mozafari, and X. Wang. (2023). "SlabCity: Whole-Query Optimization using Program Synthesis". *Proc. VLDB Endow.* 16(11): 3151–3164. DOI: 10.14778/3611479.3611515. URL: https://www.vldb.org/pvldb/vol16/p3151-dong.pdf.

Doraiswamy, H., P. N. Darera, and J. R. Haritsa. (2008). "Identifying robust plans through plan diagram reduction". *Proc. VLDB Endow.* 1(1): 1124–1140. DOI: 10.14778/1453856.1453976. URL: http://www.vldb.org/pvldb/vol1/1453976.pdf.

Doshi, L., V. Zhuang, G. Jain, R. Marcus, H. Huang, D. Altinbüken, E. Brevdo, and C. Fraser. (2023). "Kepler: Robust Learning for Parametric Query Optimization". *Proc. ACM Manag. Data.* 1(1): 109:1–109:25. DOI: 10.1145/3588963. URL: https://doi.org/10.1145/3588963.

Duggan, J., U. Çetintemel, O. Papaemmanouil, and E. Upfal. (2011). "Performance prediction for concurrent database workloads". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*. ACM. 337–348. DOI: 10.1145/1989323.1989359. URL: https://doi.org/10.1145/1989323.1989359.

Durkan, C. and C. Nash. (2019). "Autoregressive Energy Machines". In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Vol. 97. *Proceedings of Machine Learning Research*. PMLR. 1735–1744. URL: http://proceedings.mlr.press/v97/durkan19a.html.

Dutt, A., C. Wang, V. R. Narasayya, and S. Chaudhuri. (2020). "Efficiently Approximating Selectivity Functions using Low Overhead Regression Models". *Proc. VLDB Endow.* 13(11): 2215–2228. URL: http://www.vldb.org/pvldb/vol13/p2215-dutt.pdf.

Dutt, A., C. Wang, A. Nazi, S. Kandula, V. R. Narasayya, and S. Chaudhuri. (2019). “Selectivity Estimation for Range Predicates using Lightweight Models”. *Proc. VLDB Endow.* 12(9): 1044–1057. DOI: 10.14778/3329772.3329780. URL: http://www.vldb.org/pvldb/vol12/p1044-dutt.pdf.

Germain, M., K. Gregor, I. Murray, and H. Larochelle. (2015). “MADE: Masked Autoencoder for Distribution Estimation”. In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015.* Vol. 37. *JMLR Workshop and Conference Proceedings.* JMLR.org. 881–889. URL: http://proceedings.mlr.press/v37/germain15.html.

Getoor, L., B. Taskar, and D. Koller. (2001). “Selectivity Estimation using Probabilistic Models”. In: *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001.* ACM. 461–472. DOI: 10.1145/375663.375727. URL: https://doi.org/10.1145/375663.375727.

Goodrich, M. T., R. Tamassia, and M. H. Goldwasser. (2013). *Data Structures and Algorithms in Python.* Wiley.

Graefe, G. (1995). “The Cascades Framework for Query Optimization”. *IEEE Data Eng. Bull.* 18(3): 19–29. URL: http://sites.computer.org/debull/95SEP-CD.pdf.

Graefe, G. and W. J. McKenna. (1993). “The Volcano Optimizer Generator: Extensibility and Efficient Search”. In: *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria.* IEEE Computer Society. 209–218. DOI: 10.1109/ICDE.1993.344061. URL: https://doi.org/10.1109/ICDE.1993.344061.

Gregor, K., I. Danihelka, A. Mnih, C. Blundell, and D. Wierstra. (2014). “Deep AutoRegressive Networks”. In: *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014.* Vol. 32. *JMLR Workshop and Conference Proceedings.* JMLR.org. 1242–1250. URL: http://proceedings.mlr.press/v32/gregor14.html.

Grimmett, G. R. and D. R. Stirzaker. (2001). *Probability and Random Processes.* Oxford University Press.

Haas, L. M., M. J. Carey, M. Livny, and A. Shukla. (1997). "Seeking the Truth About ad hoc Join Costs". *VLDB J.* 6(3): 241–256. DOI: 10.1007/S007780050043. URL: https://doi.org/10.1007/s007780050043.

Han, Y., Z. Wu, P. Wu, R. Zhu, J. Yang, L. W. Tan, K. Zeng, G. Cong, Y. Qin, A. Pfadler, Z. Qian, J. Zhou, J. Li, and B. Cui. (2021). "Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation". *Proc. VLDB Endow.* 15(4): 752–765. DOI: 10.14778/3503585.3503586. URL: https://www.vldb.org/pvldb/vol15/p752-zhu.pdf.

Hayek, R. and O. Shmueli. (2020). "Improved Cardinality Estimation by Learning Queries Containment Rates". In: *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020.* OpenProceedings.org. 157–168. DOI: 10.5441/002/edbt.2020.15. URL: https://doi.org/10.5441/002/edbt.2020.15.

Heimel, M., M. Kiefer, and V. Markl. (2015). "Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015.* ACM. 1477–1492. DOI: 10.1145/2723372.2749438. URL: https://doi.org/10.1145/2723372.2749438.

Hertzschuch, A., C. Hartmann, D. Habich, and W. Lehner. (2021). "Simplicity Done Right for Join Ordering". In: *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings.* www.cidrdb.org. URL: http://cidrdb.org/cidr2021/papers/cidr2021%5C_paper01.pdf.

Hilprecht, B. and C. Binnig. (2022). "Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction". *Proc. VLDB Endow.* 15(11): 2361–2374. URL: https://www.vldb.org/pvldb/vol15/p2361-hilprecht.pdf.

Hilprecht, B., A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig. (2020). "DeepDB: Learn from Data, not from Queries!" *Proc. VLDB Endow.* 13(7): 992–1005. DOI: 10.14778/3384345.3384349. URL: http://www.vldb.org/pvldb/vol13/p992-hilprecht.pdf.

Izenov, Y., A. Datta, F. Rusu, and J. H. Shin. (2021). "COMPASS: Online Sketch-based Query Optimization for In-Memory Databases". In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM. 804–816. DOI: 10.1145/3448016.3452840. URL: https://doi.org/10.1145/3448016.3452840.

Johannes, F. and H. Eyke. (2011). *Preference Learning*. Springer.

Jung, J., H. Hu, J. Arulraj, T. Kim, and W. Kang. (2019). "APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems". *Proc. VLDB Endow.* 13(1): 57–70. DOI: 10.14778/3357377.3357382. URL: http://www.vldb.org/pvldb/vol13/p57-jung.pdf.

Kabra, N. and D. J. DeWitt. (1998). "Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans". In: *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*. ACM Press. 106–117. DOI: 10.1145/276304.276315. URL: https://doi.org/10.1145/276304.276315.

Kader, R. A., P. A. Boncz, S. Manegold, and M. van Keulen. (2009). "ROX: run-time optimization of XQueries". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*. ACM. 615–626. DOI: 10.1145/1559845.1559910. URL: https://doi.org/10.1145/1559845.1559910.

Kang, J. K. Z., Gaurav, S. Y. Tan, F. Cheng, S. Sun, and B. He. (2021). "Efficient Deep Learning Pipelines for Accurate Cost Estimations Over Large Scale Query Workload". In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by G. Li, Z. Li, S. Idreos, and D. Srivastava. ACM. 1014–1022. DOI: 10.1145/3448016.3457546. URL: https://doi.org/10.1145/3448016.3457546.

Kiefer, M., M. Heimel, S. Breß, and V. Markl. (2017). "Estimating Join Selectivities using Bandwidth-Optimized Kernel Density Models". *Proc. VLDB Endow.* 10(13): 2085–2096. DOI: 10.14778/3151106.3151112. URL: http://www.vldb.org/pvldb/vol10/p2085-kiefer.pdf.

Kim, K., J. Jung, I. Seo, W. Han, K. Choi, and J. Chong. (2022). "Learned Cardinality Estimation: An In-depth Study". In: *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM. 1214–1227. DOI: 10.1145/3514221.3526154. URL: https://doi.org/10.1145/3514221.3526154.

Kipf, A., T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper. (2019). "Learned Cardinalities: Estimating Correlated Joins with Deep Learning". In: *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. URL: http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf.

Krauthgamer, R., A. Mehta, V. Raman, and A. Rudra. (2008). "Greedy List Intersection". In: *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*. IEEE Computer Society. 1033–1042. DOI: 10.1109/ICDE.2008.4497512. URL: https://doi.org/10.1109/ICDE.2008.4497512.

Krishnan, S., Z. Yang, K. Goldberg, J. M. Hellerstein, and I. Stoica. (2018). "Learning to Optimize Join Queries With Deep Reinforcement Learning". *CoRR*. abs/1808.03196. arXiv: 1808.03196. URL: http://arxiv.org/abs/1808.03196.

Kwon, S., W. Jung, and K. Shim. (2022). "Cardinality Estimation of Approximate Substring Queries using Deep Learning". *Proc. VLDB Endow.* 15(11): 3145–3157. URL: https://www.vldb.org/pvldb/vol15/p3145-jung.pdf.

Lan, H., Z. Bao, and Y. Peng. (2021). "A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration". *Data Sci. Eng.* 6(1): 86–101. DOI: 10.1007/S41019-020-00149-7. URL: https://doi.org/10.1007/s41019-020-00149-7.

Leis, V., A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. (2015). "How Good Are Query Optimizers, Really?" *Proc. VLDB Endow.* 9(3): 204–215. DOI: 10.14778/2850583.2850594. URL: http://www.vldb.org/pvldb/vol9/p204-leis.pdf.

Leis, V., B. Radke, A. Gubichev, A. Kemper, and T. Neumann. (2017). "Cardinality Estimation Done Right: Index-Based Join Sampling". In: *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org. URL: http://cidrdb.org/cidr2017/papers/p9-leis-cidr17.pdf.

Li, F., B. Wu, K. Yi, and Z. Zhao. (2016). "Wander Join: Online Aggregation via Random Walks". In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM. 615–629. DOI: 10.1145/2882903.2915235. URL: https://doi.org/10.1145/2882903.2915235.

Li, P., W. Wei, R. Zhu, B. Ding, J. Zhou, and H. Lu. (2023). "ALECE: An Attention-based Learned Cardinality Estimator for SPJ Queries on Dynamic Workloads". *Proc. VLDB Endow.* 17(2): 197–210. URL: https://www.vldb.org/pvldb/vol17/p197-li.pdf.

Li, Y., L. Wang, S. Wang, Y. Sun, and Z. Peng. (2022). "A Resource-Aware Deep Cost Model for Big Data Query Processing". In: *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE. 885–897. DOI: 10.1109/ICDE53745.2022.00071. URL: https://doi.org/10.1109/ICDE53745.2022.00071.

Lipton, R. J., J. F. Naughton, and D. A. Schneider. (1990). "Practical Selectivity Estimation through Adaptive Sampling". In: *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*. ACM Press. 1–11. DOI: 10.1145/93597.93611. URL: https://doi.org/10.1145/93597.93611.

Liu, H., M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte. (2015). "Cardinality estimation using neural networks". In: *Proceedings of 25th Annual International Conference on Computer Science and Software Engineering, CASCON 2015, Markham, Ontario, Canada, 2-4 November, 2015*. IBM / ACM. 53–59. URL: http://dl.acm.org/citation.cfm?id=2886453.

Liu, J., W. Dong, D. Li, and Q. Zhou. (2021a). "Fauce: Fast and Accurate Deep Ensembles with Uncertainty for Cardinality Estimation". *Proc. VLDB Endow.* 14(11): 1950–1963. DOI: 10.14778/3476249.3476254. URL: http://www.vldb.org/pvldb/vol14/p1950-liu.pdf.

Liu, J. and B. Mozafari. (2024). "Query Rewriting via Large Language Models". *CoRR*. abs/2403.09060. DOI: 10.48550/ARXIV.2403.09060. arXiv: 2403.09060. URL: https://doi.org/10.48550/arXiv.2403.09060.

Liu, Q., Y. Shen, and L. Chen. (2021b). "LHist: Towards Learning Multi-dimensional Histogram for Massive Spatial Data". In: *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE. 1188–1199. DOI: 10.1109/ICDE51399.2021.00107. URL: https://doi.org/10.1109/ICDE51399.2021.00107.

Liu, S., X. Chen, Y. Zhao, J. Chen, R. Zhou, and K. Zheng. (2022). "Efficient Learning with Pseudo Labels for Query Cost Estimation". In: *Proceedings of the 31st ACM International Conference on Information & Knowledge Management, Atlanta, GA, USA, October 17-21, 2022*. Ed. by M. A. Hasan and L. Xiong. ACM. 1309–1318. DOI: 10.1145/3511808.3557305. URL: https://doi.org/10.1145/3511808.3557305.

Liu, T.-Y. (2009). "Learning to Rank for Information Retrieval". *Foundations and Trends in Information Retrieval*. 3(3): 225–331.

Loeliger, H. (2004). "An introduction to factor graphs". *IEEE Signal Process. Mag.* 21(1): 28–41. DOI: 10.1109/MSP.2004.1267047. URL: https://doi.org/10.1109/MSP.2004.1267047.

Lu, Y., S. Kandula, A. C. König, and S. Chaudhuri. (2021). "Pre-training Summarization Models of Structured Datasets for Cardinality Estimation". *Proc. VLDB Endow.* 15(3): 414–426. DOI: 10.14778/3494124.3494127. URL: http://www.vldb.org/pvldb/vol15/p414-lu.pdf.

Malik, T., R. C. Burns, and N. V. Chawla. (2007). "A Black-Box Approach to Query Cardinality Estimation". In: *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org. 56–67. URL: http://cidrdb.org/cidr2007/papers/cidr07p06.pdf.

Marcus, R., P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. (2021). "Bao: Making Learned Query Optimization Practical". In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM. 1275–1288. DOI: 10.1145/3448016.3452838.

Marcus, R. and O. Papaemmanouil. (2018). "Deep Reinforcement Learning for Join Order Enumeration". In: *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2018, Houston, TX, USA, June 10, 2018*. ACM. 3:1–3:4. DOI: 10.1145/3211954.3211957. URL: https://doi.org/10.1145/3211954.3211957.

Marcus, R. C., P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. (2019). "Neo: A Learned Query Optimizer". *Proc. VLDB Endow.* 12(11): 1705–1718. DOI: 10.14778/3342263.3342644. URL: http://www.vldb.org/pvldb/vol12/p1705-marcus.pdf.

Marcus, R. C. and O. Papaemmanouil. (2019). "Plan-Structured Deep Neural Network Models for Query Performance Prediction". *Proc. VLDB Endow.* 12(11): 1733–1746. DOI: 10.14778/3342263.3342646. URL: http://www.vldb.org/pvldb/vol12/p1733-marcus.pdf.

Markl, V., V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. (2004). "Robust Query Processing through Progressive Optimization". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*. ACM. 659–670. DOI: 10.1145/1007568.1007642. URL: https://doi.org/10.1145/1007568.1007642.

Meng, Z., P. Wu, G. Cong, R. Zhu, and S. Ma. (2022). "Unsupervised Selectivity Estimation by Integrating Gaussian Mixture Models and an Autoregressive Model". In: *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*. OpenProceedings.org. 2:247–2:259. DOI: 10.48786/edbt.2022.13. URL: https://doi.org/10.48786/edbt.2022.13.

Moerkotte, G. and T. Neumann. (2008). “Dynamic programming strikes back”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*. ACM. 539–552. DOI: 10.1145/1376616.1376672. URL: https://doi.org/10.1145/1376616.1376672.

Moerkotte, G., T. Neumann, and G. Steidl. (2009). “Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors”. *Proc. VLDB Endow.* 2(1): 982–993. DOI: 10.14778/1687627.1687738. URL: http://www.vldb.org/pvldb/vol2/vldb09-657.pdf.

Mou, L., G. Li, L. Zhang, T. Wang, and Z. Jin. (2016). “Convolutional Neural Networks over Tree Structures for Programming Language Processing”. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*. AAAI Press. 1287–1293. DOI: 10.1609/AAAI.V30I1.10139. URL: https://doi.org/10.1609/aaai.v30i1.10139.

Negi, P., M. Interlandi, R. Marcus, M. Alizadeh, T. Kraska, M. Friedman, and A. Jindal. (2021a). “Steering Query Optimizers: A Practical Take on Big Data Workloads”. In: *SIGMOD*. 2557–2569.

Negi, P., M. Interlandi, R. Marcus, M. Alizadeh, T. Kraska, M. T. Friedman, and A. Jindal. (2021b). “Steering Query Optimizers: A Practical Take on Big Data Workloads”. In: *SIGMOD ’21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM. 2557–2569. DOI: 10.1145/3448016.3457568. URL: https://doi.org/10.1145/3448016.3457568.

Negi, P., R. C. Marcus, A. Kipf, H. Mao, N. Tatbul, T. Kraska, and M. Alizadeh. (2021c). “Flow-Loss: Learning Cardinality Estimates That Matter”. *Proc. VLDB Endow.* 14(11): 2019–2032. DOI: 10.14778/3476249.3476259. URL: http://www.vldb.org/pvldb/vol14/p2019-negi.pdf.

Negi, P., Z. Wu, A. Kipf, N. Tatbul, R. Marcus, S. Madden, T. Kraska, and M. Alizadeh. (2023). “Robust Query Driven Cardinality Estimation under Changing Workloads”. *Proc. VLDB Endow.* 16(6): 1520–1533. URL: https://www.vldb.org/pvldb/vol16/p1520-negi.pdf.

Ngo, H. Q., E. Porat, C. Ré, and A. Rudra. (2018). “Worst-case Optimal Join Algorithms”. *J. ACM.* 65(3): 16:1–16:40. DOI: 10.1145/3180143. URL: https://doi.org/10.1145/3180143.

Ortiz, J., M. Balazinska, J. Gehrke, and S. S. Keerthi. (2018). “Learning State Representations for Query Optimization with Deep Reinforcement Learning”. In: *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD 2018, Houston, TX, USA, June 15, 2018*. ACM. 4:1–4:4. DOI: 10.1145/3209889.3209890. URL: https://doi.org/10.1145/3209889.3209890.

Osband, I. and B. V. Roy. (2015). “Bootstrapped Thompson Sampling and Deep Exploration”. *CoRR*. abs/1507.00300. arXiv: 1507.00300. URL: http://arxiv.org/abs/1507.00300.

Park, Y., S. Zhong, and B. Mozafari. (2020). “QuickSel: Quick Selectivity Learning with Mixture Models”. In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM. 1017–1033. DOI: 10.1145/3318464.3389727. URL: https://doi.org/10.1145/3318464.3389727.

Poon, H. and P. M. Domingos. (2011). “Sum-Product Networks: A New Deep Architecture”. In: *UAI 2011, Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, Barcelona, Spain, July 14-17, 2011*. AUAI Press. 337–346.

Poosala, V. and Y. E. Ioannidis. (1997). “Selectivity Estimation Without the Attribute Value Independence Assumption”. In: *VLDB’97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*. Morgan Kaufmann. 486–495. URL: http://www.vldb.org/conf/1997/P486.PDF.

Selinger, P. G., M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. (1979). “Access Path Selection in a Relational Database Management System”. In: *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*. ACM. 23–34. DOI: 10.1145/582095.582099. URL: https://doi.org/10.1145/582095.582099.

Shetiya, S., S. Thirumuruganathan, N. Koudas, and G. Das. (2020). “Astrid: Accurate Selectivity Estimation for String Predicates using Deep Learning”. *Proc. VLDB Endow.* 14(4): 471–484. DOI: 10.14778/3436905.3436907. URL: http://www.vldb.org/pvldb/vol14/p471-shetiya.pdf.

Siddiqui, T., A. Jindal, S. Qiao, H. Patel, and W. Le. (2020). “Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings”. In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM. 99–113. DOI: 10.1145/3318464.3380584. URL: https://doi.org/10.1145/3318464.3380584.

Sioulas, P. and A. Ailamaki. (2021). “Scalable Multi-Query Execution using Reinforcement Learning”. In: *SIGMOD ’21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM. 1651–1663. DOI: 10.1145/3448016.3452799. URL: https://doi.org/10.1145/3448016.3452799.

Sonoda, S. and N. Murata. (2017). “Neural Network with Unbounded Activation Functions is Universal Approximator”. *Applied and Computational Harmonic Analysis*. 43(2): 233–268.

Stillger, M., G. M. Lohman, V. Markl, and M. Kandil. (2001). “LEO - DB2’s LEarning Optimizer”. In: *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. Morgan Kaufmann. 19–28. URL: http://www.vldb.org/conf/2001/P019.pdf.

Sun, J. and G. Li. (2019). “An End-to-End Learning-based Cost Estimator”. *Proc. VLDB Endow.* 13(3): 307–319. DOI: 10.14778/3368289.3368296. URL: http://www.vldb.org/pvldb/vol13/p307-sun.pdf.

Sutton, R. S. and A. G. Barto. (2018). *Reinforcement Learning*. The MIT Press.

Tai, K. S., R. Socher, and C. D. Manning. (2015). “Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks”. In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*. The Association for Computer Linguistics. 1556–1566. DOI: 10.3115/V1/P15-1150. URL: https://doi.org/10.3115/v1/p15-1150.

Thompson, W. R. (1933). “On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples”. *Biometrika*. 25: 285–294.

Tzoumas, K., A. Deshpande, and C. S. Jensen. (2011). “Lightweight Graphical Models for Selectivity Estimation Without Independence Assumptions”. *Proc. VLDB Endow.* 4(11): 852–863. URL: http://www.vldb.org/pvldb/vol4/p852-tzoumas.pdf.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. (2017). “Attention is All you Need”. In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 5998–6008. URL: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Vu, T., A. Belussi, S. Migliorini, and A. Eldawy. (2021). “A Learned Query Optimizer for Spatial Join”. In: *SIGSPATIAL ’21: 29th International Conference on Advances in Geographic Information Systems, Virtual Event / Beijing, China, November 2-5, 2021*. ACM. 458–467. DOI: 10.1145/3474717.3484217. URL: https://doi.org/10.1145/3474717.3484217.

Wang, F., X. Yan, M. L. Yiu, S. LI, Z. Mao, and B. Tang. (2023a). “Speeding Up End-to-end Query Execution via Learning-based Progressive Cardinality Estimation”. *Proc. ACM Manag. Data.* 1(1): 28:1–28:25. DOI: 10.1145/3588708. URL: https://doi.org/10.1145/3588708.

Wang, J., C. Chai, J. Liu, and G. Li. (2021a). “FACE: A Normalizing Flow based Cardinality Estimator”. *Proc. VLDB Endow.* 15(1): 72–84. DOI: 10.14778/3485450.3485458. URL: http://www.vldb.org/pvldb/vol15/p72-li.pdf.

Wang, J., I. Trummer, A. Kara, and D. Olteanu. (2023b). “ADOPT: Adaptively Optimizing Attribute Orders for Worst-Case Optimal Join Algorithms via Reinforcement Learning”. *Proc. VLDB Endow.* 16(11): 2805–2817. DOI: 10.14778/3611479.3611489. URL: https://www.vldb.org/pvldb/vol16/p2805-wang.pdf.

Wang, X., C. Qu, W. Wu, J. Wang, and Q. Zhou. (2021b). “Are We Ready For Learned Cardinality Estimation?” *Proc. VLDB Endow.* 14(9): 1640–1654. DOI: 10.14778/3461535.3461552. URL: http://www.vldb.org/pvldb/vol14/p1640-wang.pdf.

Wang, Z., Z. Zhou, Y. Yang, H. Ding, G. Hu, D. Ding, C. Tang, H. Chen, and J. Li. (2022). “WeTune: Automatic Discovery and Verification of Query Rewrite Rules”. In: *SIGMOD ’22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM. 94–107. DOI: 10.1145/3514221.3526125. URL: https://doi.org/10.1145/3514221.3526125.

Watkins, C. (1989). “Learning From Delayed Rewards”. May.

Weng, L., R. Zhu, D. Wu, B. Ding, B. Zheng, and J. Zhou. (2024). “Eraser: Eliminating Performance Regression on Learned Query Optimizer”. *Proc. VLDB Endow.* 17(5): 926–938. URL: https://www.vldb.org/pvldb/vol17/p926-zhu.pdf.

Woltmann, L., J. Thiessat, C. Hartmann, D. Habich, and W. Lehner. (2023). “FASTgres: Making Learned Query Optimizer Hinting Effective”. *Proc. VLDB Endow.* 16(11): 3310–3322. DOI: 10.14778/3611479.3611528. URL: https://www.vldb.org/pvldb/vol16/p3310-habich.pdf.

Wu, C., A. Jindal, S. Amizadeh, H. Patel, W. Le, S. Qiao, and S. Rao. (2018). “Towards a Learning Optimizer for Shared Clouds”. *Proc. VLDB Endow.* 12(3): 210–222. DOI: 10.14778/3291264.3291267. URL: http://www.vldb.org/pvldb/vol12/p210-wu.pdf.

Wu, P. and G. Cong. (2021). “A Unified Deep Model of Learning from both Data and Queries for Cardinality Estimation”. In: *SIGMOD ’21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM. 2009–2022. DOI: 10.1145/3448016.3452830. URL: https://doi.org/10.1145/3448016.3452830.

Wu, R., B. Ding, X. Chu, Z. Wei, X. Dai, T. Guan, and J. Zhou. (2021). “Learning to be a Statistician: Learned Estimator for Number of Distinct Values”. *Proc. VLDB Endow.* 15(2): 272–284. DOI: 10.14778/3489496.3489508. URL: http://www.vldb.org/pvldb/vol15/p272-wu.pdf.

Wu, W., J. F. Naughton, and H. Singh. (2016). “Sampling-Based Query Re-Optimization”. In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM. 1721–1736. DOI: 10.1145/2882903.2882914. URL: https://doi.org/10.1145/2882903.2882914.

Wu, Z., P. Negi, M. Alizadeh, T. Kraska, and S. Madden. (2023). “FactorJoin: A New Cardinality Estimation Framework for Join Queries”. *Proc. ACM Manag. Data*. 1(1): 41:1–41:27. DOI: 10.1145/3588721. URL: https://doi.org/10.1145/3588721.

Wu, Z. and A. Shaikhha. (2020). “BayesCard: A Unified Bayesian Framework for Cardinality Estimation”. *CoRR*. abs/2012.14743. arXiv: 2012.14743. URL: https://arxiv.org/abs/2012.14743.

Yan, S., B. Ding, W. Guo, J. Zhou, Z. Wei, X. Jiang, and S. Xu. (2021). “FlashP: An Analytical Pipeline for Real-time Forecasting of Time-Series Relational Data”. *Proc. VLDB Endow.* 14(5): 721–729. DOI: 10.14778/3446095.3446096. URL: http://www.vldb.org/pvldb/vol14/p721-ding.pdf.

Yang, Z., W. Chiang, S. Luan, G. Mittal, M. Luo, and I. Stoica. (2022). “Balsa: Learning a Query Optimizer Without Expert Demonstrations”. In: *SIGMOD ’22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Z. G. Ives, A. Bonifati, and A. E. Abbadi. ACM. 931–944. DOI: 10.1145/3514221.3517885. URL: https://doi.org/10.1145/3514221.3517885.

Yang, Z., A. Kamsetty, S. Luan, E. Liang, Y. Duan, X. Chen, and I. Stoica. (2020). “NeuroCard: One Cardinality Estimator for All Tables”. *Proc. VLDB Endow.* 14(1): 61–73. DOI: 10.14778/3421424.3421432. URL: http://www.vldb.org/pvldb/vol14/p61-yang.pdf.

Yang, Z., E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. (2019). “Deep Unsupervised Cardinality Estimation”. *Proc. VLDB Endow.* 13(3): 279–292. DOI: 10.14778/3368289.3368294. URL: http://www.vldb.org/pvldb/vol13/p279-yang.pdf.

Yu, X., G. Li, C. Chai, and N. Tang. (2020). “Reinforcement Learning with Tree-LSTM for Join Order Selection”. In: *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE. 1297–1308. DOI: 10.1109/ICDE48307.2020.00116. URL: https://doi.org/10.1109/ICDE48307.2020.00116.

Yuan, H., G. Li, L. Feng, J. Sun, and Y. Han. (2020). “Automatic View Generation with Deep Learning and Reinforcement Learning”. In: *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE. 1501–1512. DOI: 10.1109/ICDE48307.2020.00133. URL: https://doi.org/10.1109/ICDE48307.2020.00133.

Zhang, W., M. Interlandi, P. Mineiro, S. Qiao, N. Ghazanfari, K. Lie, M. T. Friedman, R. Hosn, H. Patel, and A. Jindal. (2022). “Deploying a Steered Query Optimizer in Production at Microsoft”. In: *SIGMOD ’22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM. 2299–2311. DOI: 10.1145/3514221.3526052. URL: https://doi.org/10.1145/3514221.3526052.

Zhang, Y., Y. Chronis, J. M. Patel, and T. Rekatsinas. (2023). “Simple Adaptive Query Processing vs. Learned Query Optimizers: Observations and Analysis”. *Proc. VLDB Endow.* 16(11): 2962–2975. DOI: 10.14778/3611479.3611501. URL: https://www.vldb.org/pvldb/vol16/p2962-zhang.pdf.

Zhao, K., J. X. Yu, Z. He, R. Li, and H. Zhang. (2022a). “Lightweight and Accurate Cardinality Estimation by Neural Network Gaussian Process”. In: *SIGMOD ’22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM. 973–987. DOI: 10.1145/3514221.3526156. URL: https://doi.org/10.1145/3514221.3526156.

Zhao, Y., G. Cong, J. Shi, and C. Miao. (2022b). “QueryFormer: A Tree Transformer Model for Query Plan Representation”. *Proc. VLDB Endow.* 15(8): 1658–1670. URL: https://www.vldb.org/pvldb/vol15/p1658-zhao.pdf.

Zhao, Z., R. Christensen, F. Li, X. Hu, and K. Yi. (2018). “Random Sampling over Joins Revisited”. In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM. 1525–1539. DOI: 10.1145/3183713.3183739. URL: https://doi.org/10.1145/3183713.3183739.

Zhou, J., N. Bruno, M. Wu, P. Larson, R. Chaiken, and D. Shakib. (2012). “SCOPE: parallel databases meet MapReduce”. *VLDB J.* 21(5): 611–636. DOI: 10.1007/S00778-012-0280-Z. URL: https://doi.org/10.1007/s00778-012-0280-z.

Zhou, X., G. Li, C. Chai, and J. Feng. (2021). “A Learned Query Rewrite System using Monte Carlo Tree Search”. *Proc. VLDB Endow.* 15(1): 46–58. DOI: 10.14778/3485450.3485456. URL: http://www.vldb.org/pvldb/vol15/p46-li.pdf.

Zhou, X., G. Li, J. Wu, J. Liu, Z. Sun, and X. Zhang. (2023). “A Learned Query Rewrite System”. *Proc. VLDB Endow.* 16(12): 4110–4113. DOI: 10.14778/3611540.3611633. URL: https://www.vldb.org/pvldb/vol16/p4110-li.pdf.

Zhou, X., J. Sun, G. Li, and J. Feng. (2020). “Query Performance Prediction for Concurrent Queries using Graph Embedding”. *Proc. VLDB Endow.* 13(9): 1416–1428. DOI: 10.14778/3397230.3397238. URL: http://www.vldb.org/pvldb/vol13/p1416-zhou.pdf.

Zhu, R., W. Chen, B. Ding, X. Chen, A. Pfadler, Z. Wu, and J. Zhou. (2023). “Lero: A Learning-to-Rank Query Optimizer”. *Proc. VLDB Endow.* 16(6): 1466–1479. URL: https://www.vldb.org/pvldb/vol16/p1466-zhu.pdf.

Zhu, R., L. Weng, W. Wei, D. Wu, J. Peng, Y. Wang, B. Ding, D. Lian, B. Zheng, and J. Zhou. (2024). “PilotScope: Steering Databases with Machine Learning Drivers”. *Proc. VLDB Endow.* 17(5): 980–993. URL: https://www.vldb.org/pvldb/vol17/p980-zhu.pdf.

Zhu, R., Z. Wu, Y. Han, K. Zeng, A. Pfadler, Z. Qian, J. Zhou, and B. Cui. (2021). “FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation”. *Proc. VLDB Endow.* 14(9): 1489–1502. DOI: 10.14778/3461535.3461539. URL: http://www.vldb.org/pvldb/vol14/p1489-zhu.pdf.