Data Science and Engineering (2021) 6:86–101
https://doi.org/10.1007/s41019-020-00149-7

A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration

Hai Lan[1] · **Zhifeng Bao**[1] · **Yuwei Peng**[2]

Received: 3 July 2020 / Revised: 21 November 2020 / Accepted: 26 November 2020 / Published online: 15 January 2021

Abstract
Query optimizer is at the heart of the database systems. Cost-based optimizer studied in this paper is adopted in almost all current database systems. A cost-based optimizer introduces a plan enumeration algorithm to find a (sub)plan, and then uses a cost model to obtain the cost of that plan, and selects the plan with the lowest cost. In the cost model, cardinality, the number of tuples through an operator, plays a crucial role. Due to the inaccuracy in cardinality estimation, errors in cost model, and the huge plan space, the optimizer cannot find the optimal execution plan for a complex query in a reasonable time. In this paper, we first deeply study the causes behind the limitations above. Next, we review the techniques used to improve the quality of the three key components in the cost-based optimizer, cardinality estimation, cost model, and plan enumeration. We also provide our insights on the future directions for each of the above aspects.

Keywords Query optimizer · Cardinality estimation · Cost model · Plan enumeration

1 Introduction

Query optimizer is at the heart of relational database management systems (RDBMSes) and some big data process engines, e.g., SCOPE [7]. Given a query written in a declarative language (e.g., SQL), the optimizer finds the most efficient execution plan (also called physical plan) and feeds it to the executor. Thus, most of the time, the users only think over how to transform their requirements to a valid query without the need to analyze how to run the query efficiently. Almost all systems adopt a cost-based optimizer based on the architecture of System R [79] or Volcano/Cascades [26, 27].

Figure 1 illustrates the three most important components in a cost-based optimizer: *cardinality estimation* (*CE*), *cost model* (*CM*), and *plan enumeration* (*PE*). *CE* uses statistics of data and some assumptions about data distribution, column correlation, and join relationship to get the number of tuples generated by an intermediate operator,[1] which is also crucial for other search problems, e.g., [101, 102]. *CM* can be regarded as a complex function that maps the current state of database and estimated cardinalities to the cost of executing a (sub)plan. *PE* is an algorithm to explore the space of semantically equivalent join orders and find the optimal orders with minimal cost. There are two principal approaches to find an optimal join order: bottom-up join enumeration via dynamic programming and top-down join enumeration through memorization.

Theoretically, provided that the estimated cardinality and cost are accurate, and plan enumeration component can efficiently walk through the huge search space, this architecture can obtain the optimal execution plan in a reasonable time. However, it fails in reality. Despite decades of work, cost-based query optimizers still make mistakes on "difficult" queries due to the error in *CE*, the difficulty in building an accurate *CM*, and the pain in finding the optimal *join orders* (*PE*) for complex queries. The details are presented in Sect. 2, i.e., why the existing optimizer is still far from satisfaction.

✉ Yuwei Peng
ywpeng@whu.edu.cn

Hai Lan
hai.lan@rmit.edu.au

Zhifeng Bao
zhifeng.bao@rmit.edu.au

[1] RMIT University, Melbourne, Australia

[2] Wuhan University, Wuhan, China

[1] In some context, the cardinality in the database area refers to distinct count [34].

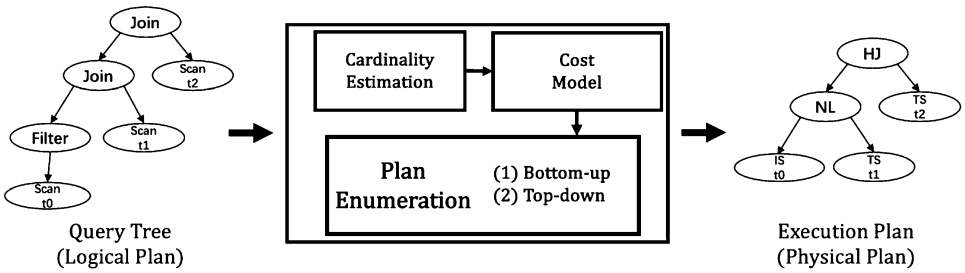

Fig. 1 Query optimizer architecture. IS, HJ, NL, and TS refer to index scan, hash join, nestloop join, and table scan

There are lots of research studies proposed to improve the capability of the optimizer. In this paper, we present a survey on them. Specifically, we review the publications which are proposed to improve the capabilities of the three key components in the optimizer, i.e., *CE*, *CM*, *PE*.

This paper makes the following contributions:

(1) We summarize the reasons why the *CE*, *CM*, and *PE* do not perform well (Sect. 2).
(2) We review the studies proposed to estimate cardinality more accurately. According to the techniques used, we categorize them into synopsis-based methods, sampling-based methods, and learning-based methods (Sect. 3).
(3) We review the work on improving the cost model. We classify them into three groups: improvement of the existing cost model, cost model alternatives, and performance prediction for a single query (Sect. 4).
(4) We review the techniques used in plan enumeration and study the non-learning methods used to handle large queries. Besides, we review recent proposed methods, which adopt reinforcement learning to select the join order (Sect. 5).
(5) In Sects. 3–5, we present our insights on the future directions, respectively.

There are two related surveys. In Chaudhuri [8] reviews the work with non-learning methods on query optimizer. In the last two decades, many methods are proposed to improve the capability of the optimizer. It is necessary to review the new work. Recently, Zhou et al. [107] investigate how AI is introduced in the different parts of DBMS, such as monitoring, tuning, and optimizer. In this paper, we focus on the query optimizer and give a comprehensive survey on the three key components of the optimizer. We summarize the learning-based and non-learning methods at the same time, review these work in details, and present possible future directions for each of them.

2 Why Key Components in Optimizer are Still Not Accurate?

In this section, we summarize the reasons why the cardinality estimation, cost model, and plan enumeration do not perform well, respectively. The studies reviewed in this paper try to improve the quality of the optimizer by handling these shortages.

2.1 Cardinality Estimation

Cardinality estimation is the ability to estimate the tuples generated by an operator and is used in the cost model to calculate the cost of that operator. Lohman [61] points out that the cost model can introduce errors of at most 30%, while the cardinality estimation can easily introduce errors of many orders of magnitude. Leis et al. [55] experimentally revisit the components, *CE*, *CM*, and *PE* in the classical optimizers with complex workloads. They focus on the quality of the physical plan on multi-join queries and get the same conclusion with Lohman.

The errors in cardinality estimation are mainly introduced in three cases:

(1) *Error in single table with predications* Database systems usually take histograms as the approximate distribution of data. Histograms are smaller than the original data. Thus, it cannot represent the true distribution entirely and some assumptions (e.g., uniformity on a single attribute, independence assumption among different attributes) are proposed. When those assumptions are not hold, estimation errors occur, leading to sub-optimal plans. The correlation among attributes in a table is not unusual. Multi-histograms have been proposed. However, it suffers from a large storage size.

(2) *Error in multi-join queries* Correlations possibly exist in columns from different tables. However, there is no efficient way to get synopses between two or more tables. Inclusion principle has been introduced for this case. The cardinality of a join operator is calculated using the inclusion principle with cardinalities of its children. It has large errors when the assumption is not held. Besides, for a complex query with multiple tables, the estimation errors can propagate and amplify from the leaves to root of the plan. The optimizers of commercial and open-source database systems still struggle in cardinality estimation for mult-join queries [55].

(3) *Error in user defined function* Most of database systems support the user-defined function (UDF). When a UDF exists in the condition, there is no general method to estimate how many tuples satisfying it [8].

2.2 Cost Model

Cost-based optimizers use a cost model to generate the estimate of cost for a (sub)query. The cost of (sub)plan is the sum of costs of all operators in it.

The cost of an operator depends on the hardware where the database is deployed, the operator's implementation, the number of tuples processed by the operator, and the current database state (e.g., data in the buffer, concurrent queries) [64]. Thus, a large number of magic numbers should be determined when combining all factors, and errors in cardinality estimation also affect the quality of the cost model. Furthermore, when the cost-based optimizer is deployed in a distributed or parallel database, the cloud environment, or the cross-platform query engines, the complexity of cost model is increasing dramatically. Moreover, even with the true cardinality, the cost estimation of a query is not linear to the running time, which may lead to a suboptimal execution plan [45, 81].

2.3 Plan Enumeration

Plan enumeration algorithm is used to find the optimal join order from the space of semantically equivalent join orders such that the query cost is minimized. It has been proven to be an NP-hard problem [41]. Exhaustive query plan enumeration is a prohibitive task for large databases with multi-join queries. Thus, it is crucial to explore the right search space which should consist of the optimal join orders or approximately optimal join orders and design an efficient enumeration algorithm. The join trees in the search space could be zigzag trees, left-deep trees, right-deep trees, and bushy trees or the subset of them. Different systems consider different forms of join tree. There are three enumeration algorithms in traditional database systems: (1) bottom up join enumeration via dynamic programming (DP) (e.g., System R [79]), (2) top-down join enumeration through memorization (e.g., Volcano/Cascades [26, 27]), and (3) randomized algorithms (e.g., genetic algorithm in PostgreSQL [77] with numerous tables joining).

Plan enumeration suffers from three limitations: (1) the errors in cardinality estimation and cost model, (2) the rules used to prune the search space, and (3) dealing with the queries with large number of tables. When a query touches a large number of tables, optimizers have to sacrifice optimality and employ heuristics to keep optimization time reasonable, like genetic algorithm in PostgreSQL, greedy method in DB2, which usually generates poor plans.

We should notice the errors in cardinality will propagate to the cost model and lead to suboptimal join order. Eliminating or reducing the errors in cardinality is the first step to build a capable optimizer as Lohman [61] says *"The root of all evil, the Achilles Heel of query optimization, is the estimation of the size of intermediate results, known as cardinalities"*.

In the following three sections, we summarize the research efforts made to handle limitations in CE, CM, and PE, i.e., how to make the query optimizer good.

3 Cardinality Estimation

At present, there are three major strategies for cardinality estimation as shown in Fig. 2. We only list some representative work for each category. Every method tries to approximate the distribution of data well with less storage. Some proposed methods combine different techniques, e.g., [91, 92].

3.1 Synopsis-Based Methods

Synopsis-based methods introduce new data structures to record the statistics information. Histogram and sketch are the widely adopted forms. A survey on synopses has been proposed in 2012 [10], which focuses on distinguishing aspects of synopses that are pertinent to approximate query processing (AQP).

3.1.1 Histogram

There are two histogram types: *1*-dimensional and *d*-dimensional histograms, where $d \geq 2$. *d*-dimensional histograms can capture the correlation between different attributes.

A *1-dimensional histogram* on attribute a is constructed by partitioning the sorted tuples into $B(\geq 1)$ mutually disjoint subsets, called *buckets* and approximates the frequencies and values in each bucket in some common fashion, e.g., uniform distribution and continuous values. A *d-dimensional histogram* on an attribute group A is constructed by

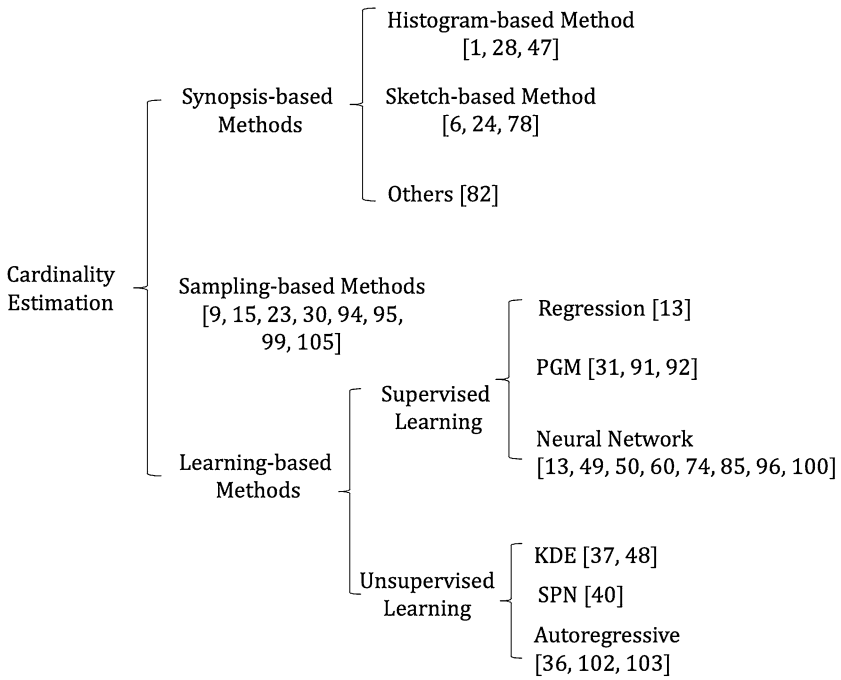

Fig. 2 A classification of cardinality estimation methods

partitioning the joint data distribution of A. Because there is no order between different attributes, the partition rule needs to be more intricate. Ioannidis [43] present a comprehensive survey on histograms following the classification method in [76]. Gunopulos et al. [29] also propose a survey in 2003, which focuses on the work used to estimate the selectivity over multiple attributes. They summarize the multi-dimensional histograms and kernel density estimators. After 2003, the work in histograms can be divided into three categories: (1) fast algorithm for histogram construction [1, 28, 32, 33, 42]; (2) new partition methods to divide the data into different buckets to achieve better accuracy [14, 58, 88]; (3) histogram construction based on query feedback [47, 57, 83]. Query feedback methods are also summarized in [10] (Section 3.5.1.2) and readers can refer to it for details.

Guha et al. [28] analyze the previous algorithm, VODP [44] and find some calculations on the minimal sum-of-squared-errors (SSE) can be reduced. They design an efficient algorithm AHistL-Δ with time complexity $O(n + B^3(\lg n + \epsilon^{-2}))$ while VODP takes $O(n^2B)$, where n is the domain size, B is the number of buckets, and ϵ is a precision parameter. Halim et al. [32, 33] propose GDY, a fast histogram construction algorithm based on greedy local search. GDY generates good sample boundaries, which then are used to construct B final partitions optimally using VODP. This study compares GDY variants with AHistL-Δ [28] in minimizing the total errors of all the buckets and shows its superiority in resolving the efficiency-quality trade-off. Instead of scanning the whole dataset [28, 42] design a greedy algorithm to construct the histogram on the random samples from dataset with time complexity $O((B^5/\epsilon^8)\log^2 n)$ and sample complexity $O((B/\epsilon)^2 \log n)$. [1] study the same problem with [42] and propose a merging algorithm with time complexity $O(1/\epsilon^2)$. Methods in [1, 28, 42] can be extended to approximate distributions by piecewise polynomials.

Considering the tree-based indexes divide the data into different segments (nodes), which is quite similar with buckets in the histogram, Eavis and Lopez [14] build the multi-dimensional histogram based on R-tree. They first build a native R-tree histogram on the Hilbert sort of data and then propose a sliding window algorithm to enhance the naive histogram under a new proposed metric, which seeks to minimize the dead space between bucket points. Lin et al. [58] design a two-level histogram for one attribute, which is quite similar to the idea of the B-tree index. The first level is used to locate which leaf histograms to be used, and the leaf histograms store the statistics information. To et al. [88] construct a histogram based on the principle of minimizing the entropy reduction of the histogram. They design two different histograms for the equality queries and an incremental algorithm to construct the histogram. However, it only considers the one-dimensional histogram and does not handle range queries well.

3.1.2 Sketch

Sketch models a column as a vector or matrix to calculate the distinct count (e.g., HyperLogLog [21]) or frequency of tuples (e.g., Count Min [11]) on a value. Rusu and Dobra

[78] summarize how to use different sketches to estimate the join size. This work considers the case of two tables (or data streams) without filters. The basic idea of them is: (1) building the sketch (a vector or matrix) on the join attribute, while ignoring all the other attributes, (2) estimating the join size based on the multiplication of the vectors or matrices. These methods only support the equi-join and join on single column. As shown in [94], a possible method introducing one filter in sketch is to build an imaginary table which only consists of the join value of tuples which satisfy the filter. However, this makes the estimation drastically worse. Skimmed sketch [24] is based on the idea of bifocal sampling [23] to estimate the join size. However, it requires knowing frequencies of the most frequent join attribute values. Recent work [6] on join size estimation introduces the sketch to record the degree of a value.

3.1.3 Other Techniques

TuG [82] is a graph-based synopsis. The node of TuG represents a set of tuples from the same table or a set of values for the same attribute. The edge represents the join relationship between different tables or between attributes and values. The authors adopt a three-step algorithm to construct TuG and introduce the histogram to summarize the value distribution in a node. When a new query comes, the selectivity is estimated by traversing TuG. The construction process is quite time-consuming and cannot be used in a large dataset. Without the relationship between different tables, TuG cannot be built.

3.2 Sampling-Based Methods

Synopsis-based methods are quite difficult to capture the correlation between different tables. Some researchers try to use a specific sampling strategy to collect a set of samples (tuples) from tables, and then run the (sub)query over samples to estimate the cardinality. As long as the distribution of the obtained samples is close to the original data, the cardinality estimation is believable. Thus, lots of work have been proposed to design a good sampling approach, from independent sampling to correlated sampling technique. Sampling-based methods also are summarized in [10]. After 2011, there are numerous studies that utilize the sampling techniques. Different with [10], we mainly summarize the new work. Moreover, we review the work according to their publishing time and present the relationship between them, i.e., which shortages of the previous work the later work tries to overcome.

Haas et al. [30] analyze the six different fixed-step (a predefined sample size) sampling methods for the equi-join queries. They conclude that if there are some indexes built on join keys, page-level sampling combining the index is the best way. Otherwise, the page-level cross-product sampling is the most efficient way. Then, the authors extend the fixed-step methods to fixed-precision procedures.

Ganguly et al. [23] introduce bifocal sampling to estimate the size of an equi-join. They classify values of the join attribute in each relation into two groups, sparse (s) and dense (d) based on their frequencies. Thus, the join type between tuples can be s–s, s–d, d–s, and d–d. The authors first adopt *t_cross* sampling [30] to estimate the join size of d–d, then adopt *t_index* to estimate the join size of the remaining cases, and finally add all the estimation as the join size estimation. However, it needs an extra pass to determine the frequencies of different values and needs indexes to estimate the join size for s–s, s–d, and d–s. Without indexes, the process is time-consuming.

End-biased sampling [15] stores the (v,f_v) if $f_v \geq T$, where v is a value in the join attribute domain, f_v is the number of tuples with value v, and T is a defined threshold. It applies a hash function $h(v) : v \mapsto [0, 1]$. If $h(v) \leq \frac{f_v}{T}$, it stores (v,f_v) or not. Different tables adopt the same hash function to correlate their sampling decisions for tuples with low frequencies. Then, the join size can be estimated using stored (v,f_v) pairs. However, it only supports equi-join on two tables and cannot handle other filter conditions. Notice, end-bias sampling is quite similar to bifocal sampling. The difference is: the former uses a hash function to sample correlated tuples and the latter uses the indexes. Both of them require an extra pass through the data to compute the frequencies of the join attribute values.

Yu et al. [105] introduce correlated sampling as a part of CS2 algorithm. They (1) choose one of the tables in a join graph as the source table R_1, (2) use a random sampling method to obtain sample set S_1 for R_1 (mark R_1 as visited), (3) follow an unvisited edge $< R_i, R_j >$ (R_i is visited) in the join graph and collect the tuples from R_j which are joinable with tuples in S_i as S_j, and (4) estimate the join size over the samples. To support the query without source tables, they propose a reverse estimator, which tracks to the source tables to estimate the join size. However, due to the walking through the join graph many times, it is time-consuming without indexes. Furthermore, it requires an unpredictable large space to store the samples.

Vengerov et al. [94] propose a correlated sampling method without the prior knowledge of frequencies of join attributes, like in [15, 23]. A tuple with join value v is included in the sample set if $h(v) < p$, where $p = \frac{n}{T}$, $h(v)$ is a hash function similar in [15], n is the sample size, and T is the table size. Then, we can use obtained samples to estimate the join size and handle specified filter conditions. Furthermore, the authors extend the method into more tables join and complex join conditions. In most cases, the correlated sampling has lower variance than independent Bernoulli sampling (*t_cross*), but when the

Table 1 Learning-based methods for cardinality estimation

References	Model	Model count	Encoding	Multi-columns	Multi-tables	UDF	Workload shift
[63]	LR	1 Model/1 Template	Predicates, arguments	✓	✓	✓	×
[75]	MixModel	1 Model	Predicates	✓	×	×	✓
[91, 92]	BN	1 Model	predicates	✓	✓	×	×
[31]	BN	1 Model/1 Table	Predicates	✓	×	×	×
[53]	NN	1 Model/1 UDF	Arguments	×	×	✓	×
[60]	NN	1 Model	Predicates	✓	×	×	×
[100]	NN/PR/MLR	1 Model/1 Subquery	Predicates, input cardinalities	✓	✓	✓	×
[49]	MSCN	1 Model	Predicates, tables, joins	✓	✓	×	×
[13]	Tree-Ensemble/NN	1 Model	Predicates	✓	×	×	×
[96, 97]	NN	1 Model/1 Template	Predicates	✓	✓	×	×
[74]	DNN/RNN/Tree	1 Model	Predicates, tables, joins	✓	✓	×	×
[85]	tree-LSTM	1 Model	Predicate, operator, metadata	✓	✓	×	×
[38]	KDE	1 Model	Samples	✓	×	×	✓
[48]	KDE	1 Model 1 Model/1 Table	Samples	✓	✓	×	✓
[40]	SPN	1 Model	Tuples; predicates	✓	✓	×	✓
[35, 104]	Autoregression	1 Model	Tuples; predicates	✓	×	×	✓
[103]	Autoregression	1 Model	Tuples; predicates	✓	✓	×	✓

values of many join attributes occur with large frequencies, the Bernoulli sampling is better. One possible solution the authors propose is to adopt a one-pass algorithm to detect the values with high frequencies, which is back to the method in [15].

Through experiments, Chen and Yi [9] conclude that there does not exist one sampling method suitable for all cases. They propose a two-level sampling method, which is based on the independent Bernoulli sampling, end-bias sampling [15], and correlated sampling [94]. Level-one sampling samples a value v from join attribute domain into value set (V), if $h(v) < p_v$. h is a hash function similar to [15], p_v is a defined probability for value v. Before level-two sampling, they sample a random tuple, called the sentry, for every v in V into tuple set. Level-two sampling samples tuples with value v ($v \in V$) with probability q. Then, we can estimate the join size by using the tuple samples. Obviously, the first level is a correlated sampling and the second level is independent Bernoulli sampling. The authors analyze how to set the p_v and q according to different join types and the frequencies of values in join attributes.

Wang and Chan [95] extend [9] to a more general framework in terms of five parameters. Based on the new framework, they propose a new class of correlated sampling methods, called CSDL, which is based on the discrete learning algorithm. A variant of CSDL, CSDL-Opt has outperformed [9] when the samples are small or join value density is small.

Wu et al. [99] adopt the online sampling to correct the possible errors in the plan generated by the optimizers.

3.3 Learning-Based Methods

Due to the capability of the learning-based methods, many researchers have introduced a learning-based model to capture the distribution and correlations of data. We classify them into: (1) supervised methods, (2) unsupervised methods (Table 1).

3.3.1 Supervised Methods

Malik et al. [63] group queries into templates and adopt machine learning techniques (e.g., linear regression model, tree models) to learn the distribution of query result sizes for each family. The features used in it include query attributes, constants, operators, aggregates, and arguments to UDFs.

Park et al. [75] propose a model, QuickSel, in query-driven paradigm, which is similar to [47, 57, 83], to estimate the selectivity of one query. Instead of adopting the histograms, QuickSel introduces the uniform mixture models to represent the distribution of the data. They train the model by minimizing the mean squared error between the mixture model and a uniform distribution.

Tzoumas et al. [91, 92] build a Bayesian network and decompose the complex statistics over multiple attributes into small one-/two-dimensional statistics, which means the model captures dependencies between two relations at most. They build the histograms for these small dimensional statistics and adopt a dynamic programming to calculate the selectivity for the new queries. Different with previous method [25], it can handle more general joins and has a

more efficient construction algorithm because of capturing smaller dependencies. However, the authors do not verify their method with multiple tables join and in large dataset. Moreover, constructing the two-dimensional statistics with attributes from different tables needs the join operation. Halford et al. [31] also introduce a method based on Bayesian network. To construct the model quickly, they only factorize the distribution of attributes inside each relation and use the previous assumptions for joins. However, they do not present how well their method compared with [91, 92].

Lakshmi and Zhou [53] first introduce the NN into the cardinality estimation of user defined function (UDF), which the histograms and other statistics cannot support. They design a two-layer neural network (NN) and employ the back propagation to update the model.

Liu et al. [60] formalize a selectivity function, $Sel : R^{2N} \mapsto R, (l_1, u_1, \ldots, l_n, u_n) \mapsto c$, where N is the number of attributes, l_i and u_i is the lower and upper bound on ith attribute for a query. They employ a 3-layer NN to learn the selectivity function. To support $>$ and $<$, they add $2N$ small NNs to produce l_i and u_i.

Wu et al. [100] use a learning-based method for workload in shared clouds, where the queries are often recurring and overlapping in nature. They first extract overlapping subgraph templates in multiple query graphs. Then, they learn the cardinality models for those sub-graph templates.

Kipf et al. [49] introduce the multi-set convolutional network (MSCN) model to estimate the cardinality of correlated joins. They represent a query as a collection of a set of tables T, joins J, and predicts P and build the separate 2-layer NN for each of them. Then, the outputs of three NNs are concatenated after the averaging operation and fed into the final output network. Deep sketch [50] is built on [49] and is a wrapper of it.

Dutt et al. [13] formalize the estimation as a function similar to [60], and they consider it as a regression problem. They adopt two different approaches for the regression problem, NN-based methods and tree-based ensembles. Different with [60], the authors also use histograms and domain knowledge (e.g., AVI, EBO, and MinSel) as the extra features in the models, which improve the estimation accuracy. Due to the domain knowledge quickly updated when the data distribution changes, the model is robust to the updates on the datasets.

Woltmann et al. [96] think building a single NN, called global model, over the entire database schema has the sparse encoding and needs numerous samples to train the model. Thus, they build different models, called local models, for different query templates. Every local model adopts multi-layer perceptrons (MLP) to produce the cardinality estimation. To collect the true cardinality, many sample queries are issued during the training process, which is time-consuming. Furthermore, Woltmann et al. [97] introduce the method of pre-aggregating the base data using the data cube concept and execute the example queries over this pre-aggregated data.

Ortiz et al. [74] empirically analyze various of deep learning approaches used in cardinality estimation, including deep neural network (DNN) and recurrent neural network (RNN). The DNN model is similar with [96]. To adopt RNN model, the authors focus on left-deep plans and model a query as a series of actions. Every action represents an operation (i.e., selection or join). In each timestamp t, the model receives two inputs: x_t, the encoding of tth of operation, and h_{t-1}, the generated hidden state from timestamp $t-1$, which can be regarded as the encoding of a subquery and captures the important details about the intermediate results.

Sun and Li [85] introduce a tree-LSTM model to learn a representation of an operator and add an estimation layer upon the tree-LSTM model to estimate the cardinality and cost simultaneously.

3.3.2 Unsupervised Methods

Heimel et al. [38] introduce the Kernel Density Estimator (KDE) into estimating the selectivity on single table with multiple predicates. They first adopt the Gaussian Kernel and the bandwidth obtained by a certain rule to construct the initial KDE, and then, they use the history queries to choose the optimal bandwidth by minimize the estimation error using initial KDE. To support the shifts in workload and dataset, they update the bandwidth after each incoming query and design the new sample maintenance method for insert-only workload and updates/deletions workload. Furthermore, in Kiefer et al. [48] extend the method into estimating the selectivity of join. They design two different models: single model over the join samples and the models over the base tables, which does not need the join operation and estimates the selectivity of join with the independent assumption.

Yang et al. [104] propose a model called Naru, which adopts the deep autoregressive model to produce n conditional densities $\hat{P}(x_i|x_{<i})$ on a set of n-dimensional tuples. Then, they estimate the selectivity using the product rule:

$$\begin{aligned}\hat{P}(\mathbf{x}) &= \hat{P}(x_1, x_2, \ldots, x_n) \\ &= \hat{P}(x_n|x_1, \ldots, x_{n-1})\hat{P}(x_{n-1}|x_1, \ldots, x_{n-2}) \ldots \\ &\quad \hat{P}(x_2|x_1)\hat{P}(x_1)\end{aligned} \tag{1}$$

To support range conditions, they introduce a progressive sampling method by sampling points from more meaningful region according to the trained autoregressive model, which is robust to the skewed data. Furthermore, they adopt the wildcard-skipping to handle wildcard condition.

Table 2 A preliminary comparison in different methods for cardinality estimation

Methods	Workload shift	Data change	Update time	Storage usage	Multi-columns	Multi-tables
1-histogram [43]	×	×	Short	Small	×	×
d-histogram [29]	×	×	Short	Large	✓	×
Sampling [9, 95, 99]	✓	×	Medium	Large	✓	✓
Supervised learning [49, 85]	×	×	Long	Small	✓	✓
Unsupervised learning [40, 103]	✓	×	Long	Small	✓	✓

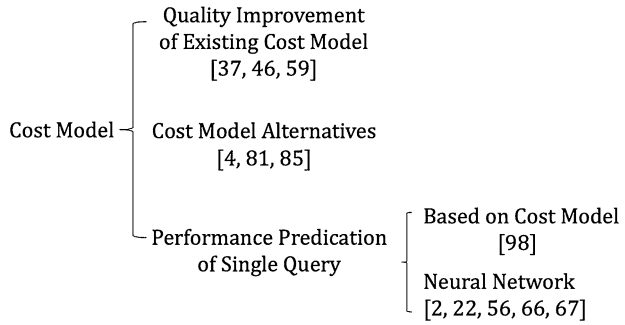

Fig. 3 A classification of cost estimation methods

Hasan et al. [35] also adopt the deep autoregressive models and introduce an adaptive sampling method to support range queries. Compared with the Naru, the authors adopt the binary encoding method and the sampling process runs parallelly, which leads the model is smaller than Naru and makes the inference faster. Besides, it can incorporate with the workload by assigning the tuples with weights according to the workload when defining the cross-entropy loss function.

Hilprecht et al. [40] introduce the Relational Sum Product Network (RSPN) to capture the distribution of single attributes and the joint probability distribution. They focus on Tree-SPNs, where one leaf is the approximation of a single attribute, and the internal node is Sum node (splitting the rows into clusters) or Product node (splitting the columns of one cluster). To support cardinality estimation of join, they build the RSPN over the join results.

Yang et al. [103] extend their previous work, Naru, to support joins. They build an autoregressive model over the full outer join of all tables. They introduce the lossless column factorization for large-cardinality columns and employ the join count table to support any queries on the subset of tables.

3.4 Our Insights

3.4.1 Summaries

The basic histogram types (e.g., equi-width, equi-depth, *d*-dimensional) have been introduced before 2000. Recent studies mainly focus on how to quickly construct the histograms and to improve the accuracy of them. Updating the histograms by query feedback is a good approach to improve the quality of histograms. However, there are still two limitations in the histograms: (1) the storage size increases dramatically when building a *d*-dimensional histograms; (2) histograms cannot capture the correlation between attributes from different tables. If building a histogram for the attributes from different tables, the join operation is required, like in [91, 92] and it is difficult to update this histogram. Sketch can be used to estimate the distinct count of an attribute or the cardinality of equi-join results. However, it cannot support more general cases well, e.g., join with filters. Synopsis-based methods cannot estimate the size of final or intermediate relations when one or both of the child relations is an intermediate relation.

Sampling is a good approach to capture the correlations between different tables. However, when the tuples in tables have been updated, the samples may become out-of-date. Sampling-based methods also suffer from the storage used to store the samples and the time used to retrieve the samples, especially when the original data is numerous. Furthermore, current sampling methods only support the equi-join.

The supervised learning methods are mostly query-driven, which means the model is trained for a specific workload. If the workload shifts, the model needs to be retrained. Thus, the data-driven (unsupervised learning) approaches come out, which still can estimate the cardinality even if the workload shifts. As shown in [104] (Section 6.3), Naru is robust to workload shift, while MSCN and KDE are sensitive to the training queries. Moreover, both of the supervised and unsupervised learning methods suffer from the data change. As presented in [103] (Section 7.6) and [85] (Section 7.5), both of them are sensitive to data change and the models will be updated in an incremental mode or retrained from scratch. However, they only consider that new tuples are appended into one table and there does not exist delete or update operation.

Due to the difference in the experiment settings, we only present a preliminary comparison between the methods for cardinality estimation as shown in Table 2. Sometimes, the size of learning-based methods is still not small

as presented in [103]. The state-of-the-art method is [103], which shows superiority compared with other methods in their experiments.

3.4.2 Possible Future Directions

There are several possible directions as follows:

(1) *Learning-based methods* Many studies on cardinality estimation are learning-based methods in last two years. The learning-based models currently integrated a real system are the light model or one model for one (sub-)query graph [13, 100], which can be trained and updated quickly. However, the accuracy and generality of these models are limited. More complex models (achieve a better accuracy) still suffer from the long training time and update time. Training time is influenced by the hardware. For example, it only takes several minutes in [103], while it is 13 h in [85] using a GPU with relatively poor performance. A database instance, especially in the cloud environment, is in a resource-constrained environment. How to train the model efficiently should be considered. The interaction between the models and the optimizer also needs to be considered, which should not be with too much overhead on the database systems [13]. As presented above, current proposed methods for data change cannot handle delete or update operation. A possible method is to adopt the idea of active learning to update the model [62].

(2) *Hybrid methods* Query-driven methods are sensitive to workload. Although data-driven methods can support more queries, it may not achieve the best accuracy for all queries. How to combine two methods in these two different catalogs is a possible direction. Actually, the previous query-feedback histogram is an instance of this case. Another interesting thing is that utilizing the query feedback information will help the model be aware of the data change.

(3) *Experimental Study* Although many methods have been proposed, it lacks of experimental studies to verify these methods. Different methods have different characteristics as shown in Table 2. It is crucial to conduct a comprehensive experimental study for proposed methods. We think the following aspects should be included: (1) is it easy to integrate the method into a real database system; (2) what is the performance of the method under different workload patterns (e.g., static or dynamic workload, OLTP or OLAP) and different data scales and distributions; (3) researchers should pay attention to the trade-off between storage usage and accuracy of candidate estimation and the trade-off between efficiency of model update and the accuracy.

4 Cost Model

In this section, we present the researches proposed to solve the limitations in the cost model. We classify the methods into three groups: (1) improving the capability of the existed cost model, (2) building a new cost model, and (3) predicting the query performance. We include the work on the single query performance prediction, because the cost used in the optimizer is the metric for performance. These methods are possibly integrated into the cost-based optimizer and replace the cost model to estimation the cost of a (sub)plan, like in [67]. However, we do not consider the query performance prediction under concurrent context (e.g., [108]). On the one hand, the concurrent queries existing during the optimization may be quite different with queries during the execution process. On the other hand, it also needs to collect more information than the model predicting the performance of a single query. We list the representative work of in the cost model in Fig. 3.

4.1 Quality Improvement of Existing Cost Model

Several studies try to estimate the cost of UDF [3, 36, 37]. Boulos and Ono [3] execute the UDF several times with different input values, collect the different costs, and then use these costs to build a multi-dimensional histogram. The histogram is stored in a tree structure. When estimating the UDF with specific parameters, traverse the tree top-down to get the estimated cost to locate the leaf with similar parameters with inputs. However, this method needs to know the upper and lower bounds of every parameter and it cannot solve the complex relation between input parameters and the costs. Unlike the static histogram used in [3], He et al. [36] introduce a dynamic quadtree-based approach to store the UDF execution information. When a query is executed, the actual cost of executing the UDF is used to update the cost model. He et al. [37] introduce a memory-limited K-nearest neighbors (MLKNN) method. They design a data structure, called PData, to store the execution cost and a multidimensional index used for fast retrieval k nearest PData for a given query point (parameter in UDF) and fast insertion of new PData.

Liu and Blanas [59] introduce a cost model for hash-based join for main-memory database. They model the response time of a query as being proportional to the number of operations weighted by the costs of four basic access patterns. They first adopt the microbenchmarks to get the cost of each access pattern and then model the cost of sequential scan, hash join, hash join with different orders by the basic access patterns.

Most of the previous cost models only consider the execution cost, which may be not reasonable in the cloud

environments. The users of the cloud database systems care about the economic cost. Karampaglis et al. [46] first propose a bi-objective query cost model, which is used to derive running time and monetary cost together in the multi-cloud environment. They model the execution time based on the method in [98]. For economic cost estimation, they first model the charging policies and estimate the monetary cost by combining the policy and time estimation.

4.2 Cost Model Alternatives

The cost model is a function mapping the (sub)plan with annotated information to a scalar (cost). Because a neural network on data primarily approximates the unknown underlying mapping function from inputs to outputs, most of the methods used to replace the origin cost model are learning-based, especially NN-based.

Boulos et al. [4] firstly introduce the neural network for cost evaluation. They design two different models: a single large neural network for every query type and a single small neural network for every operator. In the first model, they also train another model to classify a query in a certain type. The output of the first model is the cost of a (sub)plan, while the second model needs to add up the outputs from small models to get the cost.

Sun and Li [85] adopt a tree-LSTM model to learn the presentation of an operator and add an estimation layer upon the tree-LSTM model to estimate the cost of the query plan.

Due to the difficulty in collecting statistics and the needs of picking the resources in big data systems, particularly in modern cloud data services, Siddiqui et al. [81] propose a learning-based cost model and integrate it into the optimizer of SCOPE [7]. They build large number of small models to predict the costs of common (sub)queries, which are extracted from the workload history. The features encoded into the models are quite similar with [100]. Moreover, to support resource-aware query planning, they add number of partitions allocated to the operator into the features. In order to improve the coverage of the models, they introduce operator-input models and operator-subgraphApprox models and employ a meta-ensemble model to combine the models above as the final model.

4.3 Query Performance Prediction

The performance of the one query mainly refers to the latency. Wu et al. [98] adopt an offline profiling to calibrate the coefficients in the cost model under a specific hardware and software conditions. Then, they adopt the sampling method to obtain the true cardinalities of the physical operators to predict the execution times.

Ganapathi et al. [22] adopt the kernel canonical correlation analysis (KCCA) into the resource estimation, e.g., CPU time. They only model the plan level information, e.g., the number of each physical operator type and their cardinality, which is too vulnerable.

To estimate the resources (CPU time and logical I/O times), Li et al. [56] train a boosted regression tree for every operator in the database and the consumption of the plan is the sum of the operators'. To make the model more robust, they train a separate scaling function for every operator and combine scaling functions with the regression models to handle the cases when the data distribution, size, or queries' parameters are quite different with the training data. Different with [22], this is an operator-level model.

Akdere et al. [2] propose the learning-based models to predict the query performance. They first design a plan-level model if the workload is known in advance and an operator-level model. Considering the plan-level model makes highly accurate prediction and the operator-level generalizes well, for queries with low operator-level prediction accuracy, they train models for specific query subplans using plan-level modeling and compose both types of models to predict the performance of the entire plan. However, the models adopted are linear.

Marcus and Papaemmanouil [66] introduce a plan-structure neural network to predict the query's latency. They design a small neural network, called neural unit, for every logic operator and any instance of the same logic operator shares the same network. Then, these neural units are combined into a tree shape according to the plan tree. The output of one neural unit consists of two parts, the latency of current operator and the information sent to its parent node. The latency of the root neural unit of a plan is the plan's latency.

Neo [67] is a learning-based query optimizer, which introduces a neural network, called value network, to estimate the latency of (sub)plan. The query-level encoding (join graph and columns with predicts) is fed through several full-connected layers and then concatenated with the plan level encoding, which is a tree vector to represent the physical plan. Next, the concatenated vector is fed into a tree convolution and another several full-connected layers to predict the latency of the input physical plan.

4.4 Our Insights

4.4.1 Summaries

The methods trying to improve the existing cost model focus on different aspects, e.g., UDFs, hash join in main memory. These studies leave us an important lesson: when introducing a new logical or physical operator, or re-implementing the existing physical operators, we should consider how to add them into the optimization process and design the

corresponding cost estimation formulas for them (e.g., [54, 70]).

Leaning-based methods adopt the model to capture the complex relationship between cost and the factors, while the traditional cost model is defined as a certain formula by the database experts. The NN-based methods used to predict the performance, estimate cost, and estimate cardinality in Sect. 3.3.1 are quite similar in the features and models selection. For example, Sun and Li [85] use the same model to estimate the cost and cardinality and Neo [67] uses the latency (performance) of (sub)plan as the cost. A model, which is able to capture the data itself, operator level information, and subplan information, can predict the cost accurately. For example, the work [85], one of the state-of-the-art methods, adopts the tree-LSTM model to capture the information mentioned above. However, all of them are supervised methods. If the workload shifts or the data is updated the models need to be retrained from the scratch.

4.4.2 Possible Future Directions

There are two possible directions as follows:

(1) *Cloud database systems* The users of the cloud database systems need to meet their latency or throughput at the lowest price. Integrating the economic cost of running queries into the cost model is a possible direction. It is interesting to consider these related information into the cost model. For example, Siddiqui et al. [81] consider the number of container into their cost model.
(2) *Learning-based methods* Learning-based methods to estimate the cost also suffer from the same problems with methods in cardinality estimation (Sect. 3.4.2). The model that has been adopted in a real system is a light model [81]. The trade-off between accuracy and training time is still a problem. The possible solutions adopted in cardinality estimation also can be used in the cost model.

5 Plan Enumeration

In this section, we present the researches published to handle the problems in plan enumeration. We classify the work on plan enumeration into two groups: non-learning methods and learning-based methods.

5.1 Non-Learning Methods

Steinbrunn et al. [84] proposed a representative survey for selecting an optimal join orders. Thus, we mainly focus on the researches after 1997.

5.1.1 Dynamic Programming

Selinger et al. [79] propose a dynamic programming algorithm to select the optimal join order for a given conjunctive query. They generate the plan in the order of increasing size and restrict the search space to left-deep trees, which significantly speeds up the optimization. Vance and Maier [93] propose a dynamic programming algorithm to find the optimal join order by considering different partial table sets. They use it to generate the optimal bushy tree join trees containing cross-products. Selinger et al. and Vance and Maier [79, 93] are generate-and-test paradigm and most of the operations are used to check whether the subgraphs are connected and two subgraphs are combinative. Thus, none of them meet the lower bound in [73]. Moerkotte and Neumann [68] propose a graph-based Dynamic programming algorithm. They first introduce a graph-based method to generate the connected subgraph. Thus, it does not need to check out the connection and combinations and perform more efficiently. Then, they adopt DP over them for the generation of optimal bushy tree without cross-products. Moerkotte and Neumann [69] extend the method in [68] to deal with non-inner joins and a more generalized graph, hyper graph, where join predicates can involve more than two relations.

5.1.2 Top-Down Strategies

TDMinCutLazy is the first efficient top-down join enumeration algorithm proposed by DeHaan and Tompa [12]. They utilize the idea of minimal cuts to partition a join graph and introduce two different pruning strategies, predicted cost bounding and accumulated cost bounding into top-down partitioning search, which can avoid exhaustive enumeration. Top-down method is almost as efficient as dynamic programming and has other tempting properties, e.g., pruning and interesting order. Fender and Moerkotte [17] propose an alternative top-down join enumeration strategy (TDMinCutBranch). TDMinCutBranch introduces a graph-based enumeration strategy which only generates the valid join, i.e., cross-product free partitions, unlike TDMinCutLazy which adopts a generate-and-test approach. In the following year, Fender et al. [20] propose another top-down enumeration strategy TDMinCutConservative which is easier to implement and gives better runtime performance in comparison to TDMinCutBranch. Furthermore, Fender and Moerkotte [18, 19] present a general framework to handle non-inner joins and a more generalized graph, hyper graph for top-down join enumeration. RS-Graph, a new join transformation rule based on top-down join enumeration, is presented in [80] to efficiently generate the space of cross-product free join trees.

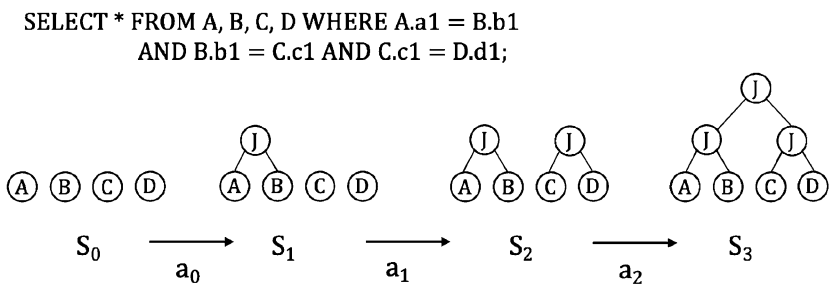

Fig. 4 One possible join order episode

5.1.3 Large Queries

For large queries, Greedy Operator Ordering [16] builds the bushy join trees bottom-up by adding the most profitable (with the smallest intermediate join size) joins first. To support large queries, Kossmann and Stocker [51] propose two different incremental dynamic programming methods, IDP-1 and IDP-2. With a given size k, IDP-1 runs the algorithm in [79] to construct the cheapest plan with that size and then regards it as a base relation, and repeats the process. With a given size k, in every iteration, IDP-2 first performs a greedy algorithm to construct the join tree with k tables and then runs DP on the generated join tree to produce the optimal plan, regards the plan as a base relation, and then repeats the process. Neumann [71] proposes a two-stage algorithm: first, it performs the query simplification to restrict the query graph by the greedy heuristic until the graph becomes tractable for DP, and then, it runs a DP algorithm to find the optimal join orders for the simplified the join graph. Bruno et al. [5] introduce enumerate-rank-merge framework which generalizes and extends the previous heuristics [16, 86]. The enumeration step considers the bushy trees. The ranking step is used to evaluate the best join pair each step, which adopts the min-size metric. The merging step constructs the selected join pair. Neumann and Radke [72] divide the queries into three types: small queries, medium, and large queries according to their query graph type and the number of tables. Then, they adopt DP to solve small queries, restrict the DP by linearizing the search space for medium queries, and use the idea in [51] for large queries.

5.1.4 Others

Trummer and Koch [89] transform the join ordering problem into a mixed integer linear program to minimize the cost of the plan and adopt the existing the MILP solvers to obtain a linear join tree (left-deep tree). To satisfy the linear properties in MILP, they approximate the cost of scan and join operations via linear functions.

Most of existed OLAP systems mainly focus on start/snowfake join queries (PK-FK join relation) and generate the left-deep binary join tree. When handling FK-FK joins, like in TPC-DS (snowstorm schema), they inccur a large number of intermediate results. Nam et al. [70] introduce a new n-ary join operator, which extends the plan space. They define the core graph to represent the FK-FK joins in the join graph and adopt the n-ary join operator to process it. They design a new cost model for this operator and integrate it into an existed OLAP systems.

5.2 Learning-Based Methods

All learning-based methods adopt the reinforcement learning (DL). In RL, an agent interacts with environment by actions and rewards. At each step t, the agent uses a policy π to choose an action a_t according to the current state s_t and transitions to a new state s_{t+1}. Then, the environment applies the action a_t and returns a reward r_t to the agent. The goal of RL is to learn a policy π, a function that automatically takes an action based on the current state, with the maximum long-term reward. In join order selection, state is the current sub-trees, and action is to combine two sub-trees, like in Fig. 4. The reward of intermediate action is 0, and the reward of the last action is the cost or latency of the query.

ReJoin [65] adopts the deep reinforcement learning (DRL), which has widely been adopted in other areas, e.g., influence maximization [87], to identify the optimal join orders. State in DRL represents the current subtrees. Each action will combine two subtrees together into a single tree. It uses cost obtained from the cost model in optimizer as the reward. ReJoin encodes the tree structure of (sub)plan, join predicates, and selection predicates in state. Different with [65], Heitz and Stockinger [39] create a matrix to represent a table or a subquery in each row and adopt the cost model in [55] to quickly obtain the cost of one query. DQ [52] is also a DRL-based method. It uses one-hot vectors to encode the visible attributes in the (sub)query. DQ also encodes the choice of physical operator by adding another one-hot vector. When training the model, DQ first uses the cost observed from the cost model of the optimizer and then fine-tunes the model with true running time. Yu et al. [106] adopt DRL and tree-LSTM together for join order selection. Different with the previous methods [39, 52, 65], tree-LSTM can capture more the structure information of the query tree. Similar with [52], they also use cost to train the model and then switch to running time as feedback for fine-tuning.

Notice, they also discuss how to handle the changes in the database schema, e.g., adding/deleting the tables/columns. SkinnerDB [90] adopts the UCT, a reinforcement learning algorithm, and learns from the current query, while the previous learning-based join order methods are learning from previous queries. It divides the execution of one query into many small slices where different small slices may choose the different join order and learn from the previous execution slices.

5.3 Our Insights

5.3.1 Summaries

The non-learning based studies focus on improving the efficiency and the ability (to handle the more general join cases) of the existing approaches. Compared with dynamic programming approach, the top-down strategy is tempting due to the better extensibility, e.g., adding new transformation rules, branch-and-bound pruning. Both of them have been implemented in many database systems.

Compared with the non-leaning methods, learning-based approaches have a fast planning time. All learning-based methods employ reinforcement learning. The main differences between them are: (1) choosing which information as the state and how to encode them, (2) adopting which models. A more complicated model with more related information can achieve better performance. The state-of-the-art method [106] adopts a tree-LSTM model similar with [85] to generate the representation of a subplan. Due to the inaccuracy in the cost model, it can improve the quality of model by using the latency to fine-tune the model. Although current state-of-the-art method [106] outperforms the non-learning based methods as shown in their experiments, how to integrate the learning-based method into the real system must be solved.

5.3.2 Possible Future Directions

There are two possible directions as follows:

(1) *Handle large queries* All methods proposed to handle large queries are DP-based methods in the bottom-up manner. A question is remaining: how to make the top-down search strategy support the large queries. Besides, the state-of-art method for large queries [72] cannot support the general join cases.

(2) *Learning-based methods* Current leaned methods only focus on the PK-FK join and the join type is inner join. How to handle the other join cases is a possible direction. None of the proposed methods have discussed how to integrate them into a real system. In their experiments, they implement the method as a separate component to get the right join order, and then send to the database. The database still needs to optimize it to get the final physical plan. If a query has subquery, they may interact multiple times. The reinforcement learning methods are trained in a certain environment, which refers to a certain database in the join order selection problem. How to handle the changes in the table schemas and data is also a possible direction.

6 Conclusion

Cardinality estimation, cost model, and plan enumeration play critical roles to generate an optimal execution plan in a cost-based optimizer. In this paper, we review the work proposed to improve their qualities, including the traditional and learning-based methods. Besides, we provide possible future directions, respectively.

We observe that more and more learning-based methods are introduced and outperform traditional methods. However, they suffer from long training and updating time. How to make the models robust to workload shifts and data changes or to update models quickly is still an open question. Traditional methods with theoretical guarantees are widely adopted in real systems. There is a great possibility of improving traditional methods with new algorithms and data structures. Moreover, We believe the ideas behind the traditional methods can be used to enhance the learning-based methods.

Acknowledgements Zhifeng Bao is supported in part by ARC DP200102611, DP180102050, and a Google Faculty Award.

References

1. Acharya J, Diakonikolas I, Hegde C, Li JZ, Schmidt L (2015) Fast and near-optimal algorithms for approximating distributions by histograms. In: PODS, pp 249–263
2. Akdere M, Çetintemel U, Riondato M, Upfal E, Zdonik SB (2012) Learning-based query performance modeling and prediction. In: ICDE, pp 390–401
3. Boulos J, Ono K (1999) Cost estimation of user-defined methods in object-relational database systems. SIGMOD Rec 28(3):22–28

4. Boulos J, Viemont Y, Ono K (1997) A neural networks approach for query cost evaluation. Trans Inf Process Soc Jpn 38(12):2566–2575
5. Bruno N, Galindo-Legaria C, Joshi M (2010) Polynomial heuristics for query optimization. In: ICDE, pp 589–600
6. Cai W, Balazinska M, Suciu D (2019) Pessimistic cardinality estimation: tighter upper bounds for intermediate join cardinalities. In: SIGMOD, pp 18–35
7. Chaiken R, Jenkins B, Larson PÅ, Ramsey B, Shakib D, Weaver S, Zhou J (2008) SCOPE: easy and efficient parallel processing of massive data sets. VLDB 1(2):1265–1276
8. Chaudhuri S (1998) An overview of query optimization in relational systems. ACM Press, New York, pp 34–43
9. Chen Yu, Yi K (2017) Two-level sampling for join size estimation. In: SIGMOD, pp 759–774
10. Cormode G, Garofalakis MN, Haas PJ, Jermaine C (2012) Synopses for massive data: samples, histograms, wavelets, sketches. Found Trends Databases 4(1–3):1–294
11. Cormode G, Muthukrishnan S (2004) An improved data stream summary: the count-min sketch and its applications. In: LATIN 2004: theoretical informatics, 6th Latin American symposium, Buenos Aires, Argentina, April 5–8, 2004, Proceedings, pp 29–38
12. DeHaan D, Tompa FW (2007) Optimal top-down join enumeration. In: SIGMOD, pp 785–796
13. Dutt A, Wang C, Nazi A, Kandula S, Narasayya VR, Chaudhuri S (2019) Selectivity estimation for range predicates using lightweight models. VLDB 12(9):1044–1057
14. Eavis T, Lopez A (2007) Rk-hist: an r-tree based histogram for multi-dimensional selectivity estimation. In: CIKM, pp 475–484
15. Estan C, Naughton FJ (2006) End-biased Samples for join cardinality estimation. In: ICDE, p 20
16. Fegaras L (1998) A new heuristic for optimizing large queries. In: DEXA, pp 726–735
17. Fender P, Moerkotte G (2011) A new, highly efficient, and easy to implement top-down join enumeration algorithm. In: ICDE, pp 864–875
18. Fender P, Moerkotte G (2013) Counter strike: generic top-down join enumeration for hypergraphs. VLDB 6(14):1822–1833
19. Fender P, Moerkotte G (2013) Top down plan generation: from theory to practice. In: ICDE, pp 1105–1116
20. Fender P, Moerkotte G, Neumann T, Leis V (2012) Effective and robust pruning for top-down join enumeration algorithms. In: ICDE, pp 414–425
21. Flajolet P, Fusy E, Gandouet O, Meunier F (2007) HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In: Discrete mathematics and theoretical computer science, pp 137–156
22. Ganapathi A, Kuno HA, Dayal U, Wiener JL, Fox A, Jordan MI, Patterson DA (2009) Predicting multiple metrics for queries: better decisions enabled by machine learning. In: ICDE, pp 592–603
23. Ganguly S, Gibbons PB, Matias Y, Silberschatz A (1996) Bifocal sampling for skew-resistant join size estimation. In: SIGMOD, pp 271–281
24. Ganguly S, Garofalakis MN, Rastogi R (2004) EDB Processing data-stream join aggregates using skimmed sketches. In: EDBT, pp 569–586
25. Getoor L, Taskar B, Koller D (2001) Selectivity estimation using probabilistic models. In: SIGMOD, pp 461–472
26. Graefe G (1995) The cascades framework for query optimization. IEEE Data Eng. Bull. 18(3):19–29
27. Graefe G, McKenna WJ (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, pp 209–218
28. Guha S, Koudas N, Shim K (2006) Approximation and streaming algorithms for histogram construction problems. ACM Trans. Database Syst. 31(1):396–438
29. Gunopulos D, Kollios G, Tsotras VJ, Domeniconi C (2005) Selectivity estimators for multidimensional range queries over real attributes. VLDB J. 14(2):137–154
30. Haas PJ, Naughton JF, Seshadri S, Swami AN (1993) Fixed-precision estimation of join selectivity. In: PODS, pp 190–201
31. Halford M, Saint-Pierre P, Morvan F (2019) An approach based on Bayesian networks for query selectivity estimation. In: DASFAA, pp 3–19
32. Halim F, Karras P, Yap RHC (2009) Fast and effective histogram construction. In: CIKM, pp 1167–1176
33. Halim F, Karras P, Yap RHC (2010) Local search in histogram construction. In: AAAI
34. Harmouch H, Naumann F (2017) Cardinality estimation: an experimental survey. Proc VLDB Endow 11(4):499–512
35. Hasan S, Thirumuruganathan S, Augustine J, Koudas N, Das G (2020) Deep learning models for selectivity estimation of multi-attribute queries. In: SIGMOD, pp 1035–1050
36. He Z, Lee BS, Snapp RR (2004) Self-tuning UDF cost modeling using the memory-limited quadtree. EDBT 2992:513–531
37. He Z, Lee BS, Snapp RR (2005) Self-tuning cost modeling of user-defined functions in an object-relational DBMS. TODS 30(3):812–853
38. Heimel M, Kiefer M, Markl V (2015) Self-tuning, GPU-accelerated kernel density models for multidimensional selectivity estimation. In: SIGMOD, pp 1477–1492
39. Heitz J, Stockinger K (2019) Join query optimization with deep reinforcement learning algorithms. CoRR. arXiv:abs/1911.11689
40. Hilprecht B, Schmidt A, Kulessa M, Molina A, Kersting K, Binnig C (2020) DeepDB: learn from data, not from queries!. VLDB 13(7):992–1005
41. Ibaraki T, Kameda T (1984) On the optimal nesting order for computing N-relational joins. ACM Trans Database Syst 9(3):482–502
42. Indyk P, Levi R, Rubinfeld R (2012) Approximating and testing k-histogram distributions in sub-linear time. In: PODS, pp 15–22
43. Ioannidis YE (2003) The history of histograms (abridged). In: VLDB. Morgan Kaufmann, Los Altos, pp 19–30
44. Jagadish HV, Koudas N, Muthukrishnan S, Poosala V, Sevcik KC, Suel T (1998) Optimal histograms with quality guarantees. In: VLDB, pp 275–286
45. Kaoudi Z, Quiané-Ruiz J-A, Contreras-Rojas B, Pardo-Meza R, Troudi A, Chawla S (2020) ML-based cross-platform query optimization. In: ICDE, pp 1489–1500
46. Karampaglis Z, Gounaris A, Manolopoulos Y (2014) A bi-objective cost model for database queries in a multi-cloud environment. In: MEDES, pp 109–116
47. Kaushik R, Suciu D (2009) Consistent histograms in the presence of distinct value counts. VLDB 2(1):850–861
48. Kiefer M, Heimel M, Breß S, Markl V (2017) Estimating join selectivities using bandwidth-optimized kernel density models. VLDB 10(13):2085–2096
49. Kipf A, Kipf T, Radke B, Leis V, Boncz PA, Alfons K (2019) Learned cardinalities, estimating correlated joins with deep learning. In: CIDR
50. Kipf A, Vorona D, Müller J, Kipf T, Radke B, Leis V, Boncz P, Neumann T, Kemper A (2019) Estimating cardinalities with deep sketches. J CoRR. arXiv:abs/1904.08223
51. Kossmann D (2000) Iterative dynamic programming: a new class of query optimization algorithms. TODS 25(1):43–82
52. Krishnan S, Yang Z, Goldberg K, Hellerstein JM, Stoica I (2018) Learning to optimize join queries with deep reinforcement learning. CoRR. arXiv:abs/1808.03196

53. Lakshmi SM, Zhou S (1998) Selectivity estimation in extensible databases—a neural network approach. In: VLDB, pp 623–627
54. Leeka J, Rajan K (2019) Incorporating super-operators in big-data query optimizers. VLDB 13(3):348–361
55. Leis V, Gubichev A, Mirchev A, Boncz PA, Kemper A, Neumann T (2015) How good are query optimizers, really? VLDB 9(3):204–215
56. Li J, König AC, Narasayya VR, Chaudhuri S (2012) Robust estimation of resource consumption for SQL queries using statistical techniques. VLDB 5(11):1555–1566
57. Lim L, Wang M, Vitter JS (2003) SASH: a self-adaptive histogram set for dynamically changing workloads. In: VLDB, pp 369–380
58. Lin X, Zeng X, Xiaowei P, Sun Y (2015) A cardinality estimation approach based on two level histograms. J Inf Sci Eng 31(5):1733–1756
59. Liu F, Blanas S (2015) Forecasting the cost of processing multi-join queries via hashing for main-memory databases. In: Socc, pp 153–166
60. Liu H, Mingbin X, Ziting Yu, Corvinelli V, Zuzarte C (2015) Cardinality estimation using neural networks. In: CASCON, pp 53–59
61. Lohman G (2014) Is query optimization a "solved" problem?. http://wp.sigmod.org/?p=1075 Accessed 10 June 2020
62. Ma L, Ding B, Das S, Swaminathan A (2020) Active learning for ML enhanced database systems. In: SIGMOD, pp 175–191
63. Malik T, Burns RC, Chawla NV (2007) A black-box approach to query cardinality estimation. In: CIDR, pp 56–67
64. Manegold S, Boncz PA, Kersten ML (2002) Generic database cost models for hierarchical memory systems. In: VLDB, pp 191–202
65. Marcus R, Papaemmanouil O (2018) Deep reinforcement learning for join order enumeration. In: aiDM@SIGMOD, pp 3:1–3:4
66. Marcus RC, Papaemmanouil O (2019) Plan-structured deep neural network models for query performance prediction. VLDB 12(11):1733–1746
67. Marcus RC, Negi P, Mao H, Zhang C, Alizadeh M, Kraska T, Papaemmanouil O, Tatbul N (2019) Neo: a learned query optimizer. VLDB 12(11):1705–1718
68. Moerkotte G, Neumann T (2006) Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In: VLDB, pp 930–941
69. Moerkotte G, Neumann T (2008) Dynamic programming strikes back. In: SIGMOD, pp 539–552
70. Nam Y-M, Han D, Kim M-S (2020) SPRINTER: a fast n-ary join query processing method for complex OLAP queries. In: SIGMOD, pp 2055–2070
71. Neumann T (2009) Query simplification: graceful degradation for join-order optimization. In: SIGMOD, pp 403–414
72. Neumann T, Radke B (2018) Adaptive optimization of very large join queries. In: SIGMOD, pp 677–692
73. Ono K, Lohman GM (1990) Measuring the complexity of join enumeration in query optimization. In: VLDB, pp 314–325
74. Ortiz J, Balazinska M, Gehrke J, Keerthi SS (2019) An empirical analysis of deep learning for cardinality estimation. CoRR. arXiv:abs/1905.06425
75. Park Y, Zhong S, Mozafari B (2020) QuickSel: quick selectivity learning with mixture models. In: SIGMOD, pp 1017–1033
76. Poosala V, Ioannidis YE, Haas PJ, Shekita EJ (1996) Improved histograms for selectivity estimation of range predicates. In: SIGMOD, pp 294–305
77. PostgreSQL Database (2020) howpublished. http://www.postgresql.org/
78. Rusu F, Dobra A (2008) Sketches for size of join estimation. ACM Trans Database Syst 33(3):15:1–15:46
79. Selinger PG, Astrahan MM, Chamberlin DD, Lorie RA, Price TG (1979) Access path selection in a relational database management system. In: SIGMOD, pp 23–34
80. Shanbhag A, Sudarshan S (2014) Optimizing join enumeration in transformation-based query optimizers. VLDB 7(12):1243–1254
81. Siddiqui T, Jindal A, Qiao S, Patel H, Le W (2020) Cost models for big data query processing: learning, retrofitting, and our findings. In: SIGMOD, pp 99–113
82. Spiegel J, Polyzotis N (2006) Graph-based synopses for relational selectivity estimation. In: SIGMOD, pp 205–216
83. Srivastava U, Haas PJ, Markl V, Kutsch M, Tran TM (2006) ISOMER: consistent histogram construction using query feedback. In: ICDE, p 39
84. Steinbrunn M, Moerkotte G, Kemper A (1997) Heuristic and randomized optimization for the join ordering problem. VLDB J 6(3):191–208
85. Sun J, Li G (2020) An end-to-end learning-based cost estimator. VLDB 13(3):307–319
86. Swami AN (1989) Optimization of large join queries: combining heuristic and combinatorial techniques. In: SIGMOD, pp 367–376
87. Tian S, Mo S, Wang L, Peng Z (2020) Deep reinforcement learning-based approach to tackle topic-aware influence maximization. Data Sci Eng 5(1):1–11
88. To H, Chiang K, Shahabi C (2013) Entropy-based histograms for selectivity estimation. In: CIKM, pp 1939–1948
89. Trummer I, Koch C (2017) Solving the join ordering problem via mixed integer linear programming. In: SIGMOD, pp 1025–1040
90. Trummer I, Wang J, Maram D, Moseley S, Jo S, Antonakakis J (2019) SkinnerDB: regret-bounded query evaluation via reinforcement learning. In: SIGMOD, pp 1153–1170
91. Tzoumas K, Deshpande A, Jensen CS (2011) Lightweight graphical models for selectivity estimation without independence assumptions. VLDB 4(11):852–863
92. Tzoumas K, Deshpande A, Jensen CS (2013) Efficiently adapting graphical models for selectivity estimation. VLDB J 22(1):3–27
93. Vance B, Maier D (1996) Rapid bushy join-order optimization with Cartesian products. In: SIGMOD, pp 35–46
94. Vengerov D, Menck AC, Zaït M, Chakkappen S (2015) Join size estimation subject to filter conditions. VLDB 8(12):1530–1541
95. Wang TN, Chan C-Y (2020) Improved correlated sampling for join size estimation. In: ICDE, pp 325–336
96. Woltmann L, Hartmann C, Thiele M, Habich D, Lehner W (2019) Cardinality estimation with local deep learning models. In: aiDM@SIGMOD, pp 5:1–5:8
97. Woltmann L, Hartmann C, Habich D, Lehner W (2020) Machine learning-based cardinality estimation in DBMS on pre-aggregated data. CoRR. arXiv:abs/2005.09367
98. Wentao W, Chi Y, Zhu S, Tatemura J, Hacigümüs H, Naughton JF (2013) Predicting query execution time: are optimizer cost models really unusable?. In: ICDE, pp 1081–1092
99. Wentao W, Naughton JF, Singh H (2016) Sampling-based query re-optimization. In: SGMOD, pp 1721–1736
100. Wu C, Jindal A, Amizadeh S, Patel H, Le W, Qiao S, Rao S (2018) Towards a learning optimizer for shared clouds. VLDB 12(3):210–222
101. Yang Y, Zhang W, Zhang Y, Lin X, Wang L (2019) Selectivity estimation on set containment search. Data Sci Eng 4(3):254–268
102. Yang Y, Zhang W, Zhang Y, Lin X, Wang L (2019) Selectivity estimation on set containment search. In: DASFAA, Part I, pp 330–349
103. Yang Z, Kamsetty A, Luan S, Liang E, Duan Y, Chen X, Stoica I (2020) NeuroCard: one cardinality estimator for all tables. CoRR. arXiv:abs/2006.08109

104. Yang Z, Liang E, Kamsetty A, Chenggang W, Duan Y, Chen P, Abbeel P, Hellerstein JM, Krishnan S, Stoica I (2019) Deep unsupervised cardinality estimation. VLDB 13(3):279–292
105. Yu F, Hou W-C, Luo C, Che D, Zhu M (2013) CS2: a new database synopsis for query estimation. In: SIGMOD, pp 469–480
106. Yu X, Li G, Chai C, Tang N (2020) Reinforcement learning with tree-LSTM for join order selection. In: ICDE, pp 1297–1308
107. Zhou X, Chai C, Li G, SUN J (2020) Database meets artificial intelligence a: survey. In: TKDE, pp 1–1
108. Zhou X, Sun J, Li G, Feng J (2020) Query performance prediction for concurrent queries using graph embedding. VLDB 13(9):1416–1428