Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation

Yuxing Han[1,#], Ziniu Wu[1,#], Peizhi Wu[2], Rong Zhu[1,*], Jingyi Yang[2], Liang Wei Tan[2], Kai Zeng[1], Gao Cong[2], Yanzhao Qin[1,3], Andreas Pfadler[1], Zhengping Qian[1], Jingren Zhou[1], Jiangneng Li[1], Bin Cui[3]
[1]Alibaba Group, [2]Nanyang Technological University, [3]Peking University
[1]red.zr@alibaba-inc.com, [2]gaocong@ntu.edu.sg, [3]bin.cui@pku.edu.cn

ABSTRACT

Cardinality estimation (CardEst) plays a significant role in generating high-quality query plans for a query optimizer in DBMS. In the last decade, an increasing number of advanced CardEst methods (especially ML-based) have been proposed with outstanding estimation accuracy and inference latency. However, there exists no study that systematically evaluates the quality of these methods and answer the fundamental problem: *to what extent can these methods improve the performance of query optimizer in real-world settings, which is the ultimate goal of a CardEst method.*

In this paper, we comprehensively and systematically compare the effectiveness of CardEst methods in a real DBMS. We establish a new benchmark for CardEst, which contains a new complex real-world dataset STATS and a diverse query workload STATS-CEB. We integrate multiple most representative CardEst methods into an open-source database system PostgreSQL, and comprehensively evaluate their true effectiveness in improving query plan quality, and other important aspects affecting their applicability. We obtain a number of key findings for the CardEst methods, under different data and query settings. Furthermore, we find that the widely used estimation accuracy metric (Q-Error) cannot distinguish the importance of different sub-plan queries during query optimization and thus cannot truly reflect the generated query plan quality. Therefore, we propose a new metric P-Error to evaluate the performance of CardEst methods, which overcomes the limitation of Q-Error and is able to reflect the overall end-to-end performance of CardEst methods. It could serve as a better optimization objective for future CardEst methods. We have made all of the benchmark data and evaluation code publicly available at `https://github.com/Nathaniel-Han/End-to-End-CardEst-Benchmark`.

PVLDB Reference Format:
Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, Bin Cui. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. PVLDB, 15(4): XXX-XXX, 2022.
doi:10.14778/3503585.3503586

The first two authors contribute equally to this paper.
∗ Corresponding author.

Proceedings of the VLDB Endowment, Vol. 15, No. 4 ISSN 2150-8097.
doi:10.14778/3503585.3503586

1 INTRODUCTION

The query optimizer is an integral component in modern DBMSs. It is responsible for generating high-quality execution plans for the input SQL queries. *Cardinality estimation(CardEst)* plays a significant role in query optimization. It aims at estimating the result size of all sub-plans of each query and guiding the optimizer to select the optimal join operations. The performance of CardEst has a critical impact on the quality of the generated query plans.

Background: Due to its important role in DBMS, CardEst has been extensively studied, by both academic and industrial communities. Current open-source and commercial DBMSs mainly use two traditional CardEst methods, namely histogram [1, 10, 18, 38, 50, 59] in PostgreSQL[11] and SQL Server[34] and sampling [22, 27, 30, 31, 72] in MySQL [49] and MariaDB [51]. The core task of CardEst is to build a compact model capturing data and/or query information. With the prosperity of machine learning (ML), we witness a proliferation of learned methods for CardEst in the last three years [12, 21, 24, 28, 55, 63, 66, 69, 70, 75]. These methods could be categorized into two classes, namely query-driven and data-driven. Query-driven CardEst methods [12, 28] build discriminative models mapping featurized queries to their cardinalities while data-driven CardEst methods [16, 24, 58, 66, 69, 70, 75] directly model the joint distribution of all attributes. In comparison with the traditional methods, their estimation accuracy stands out as their models are more sophisticated and fine-grained [24, 66, 70, 75].

Motivation: Despite the recent advance of the CardEst methods, we notice that a fundamental problem has not yet been answered, which is *"to what extent can these advanced CardEst methods improve the performance of query optimizers in real-world settings?"* Although existing studies have conducted extensive experiments, they suffer from the following shortcomings:

1. *The data and query workloads used for evaluation may not well represent the real-world scenarios.* The widely adopted JOB-LIGHT query workload on IMDB benchmark data [29] touches at most 8 numerical or categorical attributes within six tables, whose schema forms a star-join. The recent benchmark work [60] only evaluate these methods in a single table scenario. Therefore, the existing works are not sufficient to reflect the behavior of CardEst methods on complex real-world data with high skewness and correlations and multi-table queries with various join forms and conditions.

2. *Most of the evaluations do not exhibit the end-to-end improvement of CardEst methods on the query optimizer.* Existing works usually evaluate CardEst methods on the algorithm-level metrics, such as estimation accuracy and inference latency. These metrics only evaluate the quality of the CardEst algorithm itself, but cannot

reflect how these methods behave in a real DBMS due to two reasons. First, the estimation accuracy does not directly equal to the query plan quality. As different sub-plan queries matters differently to the query plan [4, 46, 57], a more accurate method may produce a much worse query plan if they mistake a few very important estimations [42]. Second, the actual query time is affected by multiple factors, including both query plan quality and CardEst inference cost. Therefore, the "gold standard" to examine a CardEst method is to integrate it into the query optimizer of a real DBMS and record the *end-to-end query time*, including both query plan generation time and execution time. Unfortunately, this end-to-end evaluation has been ignored in most existing works.

To address these two problems, the DBMS community needs 1) *new benchmark datasets and query workloads that can represent the real-world settings* and 2) *an in-depth end-to-end evaluation to analyze performance of CardEst methods.*

Contributions and Findings: In this paper, we provide a systematic evaluation on representative CardEst methods and make the following contributions:

1. *We establish a new benchmark for CardEst that can represent real-world settings.* Our benchmark includes a real-world dataset STATS and a hand-picked query workload STATS-CEB. STATS has complex properties and STATS-CEB contains a number of diverse multi-table join queries. This benchmark pose challenges to better reveal the advantages and drawbacks of existing CardEst methods in real-world settings. (in Section 3)

2. We provide an end-to-end evaluation platform for CardEst and present a comprehensive evaluation and analysis of the representative CardEst methods. We provide an approach that could integrate any CardEst method in the built-in query optimizer of PostgreSQL, a well-known open-source DBMS. Based on this, we evaluate the performance of both traditional and ML-based CardEst methods in terms of the end-to-end query time and other important aspects affecting their applicability, including inference latency, model size, training time, update efficiency, and update accuracy. From the results, we make a dozen of key observations (O). Some key takeaway findings are listed as follows (in Sections 4–6):

K1. Improvement (O1): On numerical and categorical query workloads, the ML-based data-driven CardEst methods can achieve remarkable performance, whereas most of the other methods can hardly improve the PostgreSQL baseline.

K2. Method (O3, O8-10): Among the data-driven methods, probabilistic graphical models outperform deep models in terms of both end-to-end query time and other practicality aspects and are more applicable to deploy in real-world DBMS.

K3. Accuracy (O5-6, O11-13): Accurate estimation of some important queries, e.g., with large cardinality, is crucial to the overall query performance. The widely used accuracy metric Q-Error [37] cannot reflect a method's end-to-end query performance.

K4. Latency (O7): The inference latency of CardEst has a nontrivial impact on the end-to-end query time on OLTP workload.

3. We propose a new metric that can indicate the overall quality of CardEst methods. Previous CardEst quality metrics, such as Q-Error, can only reflect the estimation accuracy of each (sub-plan) query but not the overall end-to-end performance of CardEst methods. Therefore, inspired by the recent work [41, 42], we propose a new metric called P-Error, which directly relates the estimation accuracy of (sub-plan) queries to the ultimate query execution plan quality of the CardEst methods. Based on our analysis, P-Error is highly correlated with the end-to-end query time improvement. Thus, it could serve as a potential substitute for Q-Error and a better optimization objective for learned CardEst methods. (in Section 7)

4. We point out some future research directions for CardEst methods. On the application scope, future ML-based CardEst methods should enhance the ML models to support more types of queries. Moreover, it is also helpful to unify different approaches and/or models to adjust CardEst for different setting, i.e., OLTP and OLAP. On designing principles, we should optimize ML models towards the end-to-end performance metrics instead of purely accuracy metrics, with an emphasis on multi-table join queries. (in Section 8)

2 PRELIMINARIES AND BACKGROUND

In this section, we introduce some preliminaries and background, including a formal definition of the cardinality estimation (CardEst) problem, a brief review on representative CardEst algorithms and a short analysis on existing CardEst benchmarks.

CardEst Problem: In the literature, CardEst is usually defined as a statistical problem. Let T be a table with k attributes $A = \{A_1, A_2, \ldots, A_k\}$. T could either represent a single relational table or a joined table. In this paper, we assume each attribute A_i for each $1 \leq i \leq k$ to be either categorical (whose values can be mapped to integers) or continuous, whose domain (all unique values) is denoted as D_i. Thereafter, any selection query Q on T can be represented in a canonical form: $Q = \{A_1 \in R_1 \wedge A_2 \in R_2 \wedge \cdots \wedge A_n \in R_n\}$, where $R_i \subseteq D_i$ is the constraint region specified by Q over attribute A_i (i.e. filter predicates). Without loss of generality, we have $R_i = D_i$ if Q has no constraint on A_i. Let $\mathsf{Card}(T, Q)$ denote the *cardinality*, i.e., the exact number of records in T satisfying all constraints in Q. The CardEst problem requires estimating $\mathsf{Card}(T, Q)$ as accurately as possible without executing Q on T.

In this paper, we concentrate on evaluating these selection queries on numerical/categorical (n./c.) attributes. We do not consider 'LIKE" (or pattern matching) queries on string attributes due to two reasons: 1) practical CardEst methods for "LIKE" queries in DBMS often use magic numbers [11, 34, 49, 51], which are not meaningful to evaluate; and 2) CardEst solutions for n./c. queries mainly consider how to build statistical models summarizing attribute and/or query distribution information. Whereas, CardEst methods for "LIKE" queries [36, 52, 55] concern on applying NLP techniques to summarize semantic information in strings. Thus, they tackle with different technical challenges and are not comparable. Statistical CardEst methods can not well support "LIKE" queries.

CardEst Algorithms: There exist many CardEst methods in the literature, which can be classified into three classes as follows:

Traditional CardEst methods, such as histogram [50] and sampling [22, 27, 30], are widely applied in DBMS and generally based on simplified assumptions and heuristics. Lots of variants are proposed later to enhance their performance. Examples on histogram routine include multi-dimensional histogram based methods [10, 17, 18, 38, 48, 59], correcting and self-tuning histograms with query feedbacks [1, 13, 26, 53] and updating statistical summaries in DBMS [54, 61]. Examples on sampling routine include query-driven

kernel-based methods [22, 27], index based methods [30] and random walk based methods [31, 72]. Some other work, such as the sketch based method [3], explores a new routine for `CardEst`.

ML-based query-driven CardEst methods try to learn a model to map each featurized query Q to its cardinality $\text{Card}(T, Q)$ directly. Some ML-enhanced methods improve the performance of CardEst methods by using more complex models such as DNNs [28] or gradient boosted trees [12].

ML-based data-driven CardEst methods are independent of the queries. They regard each tuple in T as a point sampled according to the joint distribution $P_T(A) = P_T(A_1, A_2, \ldots, A_n)$. Let $P_T(Q) = P_T(A_1 \in R_1, A_2 \in R_2, \cdots, A_n \in R_n)$ be the probability specified by the region of Q. Then, we have $\text{Card}(T, Q) = P_T(Q) \cdot |T|$ so CardEst problem could be reduced to model the probability density function (PDF) $P_T(A)$ of table T. A variety of ML-based models have been used in existing work to represent $P_T(A)$, the most representative of which includes deep auto-regression model [69, 71] and probabilistic graphical models such as Bayesian networks (BN) [16, 58, 66], SPN [24], and FSPN [75]. They use different techniques to balance the estimation accuracy, inference efficiency, and model size.

CardEst Benchmark: Literature works have proposed some benchmark datasets and query workloads for CardEst evaluation. We analyze their pros and cons as follows:

1) The synthetic benchmarks such as TPC-H [8] and TPC-DS [7] and Star Schema benchmarks (SSB) [43] contain real-world data schemas and synthetic generated tuples. They are mainly used for evaluating query engines but not suitable for CardEst because their data generator makes oversimplified assumptions on the joint PDF of attributes, such as uniform distribution and independence. However, real-world datasets are often highly skewed and correlated [29], which are more difficult for CardEst.

2) IMDB dataset with its JOB workload [29] is a well-recognized benchmark, containing complex data and string "LIKE" queries. To evaluate statistical CardEst methods, most of the existing works [23, 28, 69, 71, 75] fetch the query workload subset JOB-LIGHT containing 70 realistic selection queries with varied number of joining tables. However, these selection queries touch only 8 n./c. attributes within six tables of IMDB and the join queries between these tables are only star joins centered at one table. Thus, this simplified IMDB dataset and its workload cannot comprehensively evaluate the performance of nowadays CardEst algorithms on more complex real-world data and varied join settings. On the other hand, some works [41, 69] generate queries on the IMDB dataset including "LIKE" queries which are not supported by most of recent statistical methods.

Apart from these well-established and general-purpose benchmarks, there also exist other benchmarks with specific purposes. For example, Wang [60] presents a series of real-world datasets to analyze whether existing CardEst algorithms are suitable to be deployed into real-world DBMS. However, it is only conducted on single-table datasets, which can not reflect the behavior of these models in more practical multi-table settings.

Summary: A surge of CardEst algorithms built on top of statistical models has been proposed in the literature, especially in the last decade. However, existing CardEst benchmarks are not sufficient to comprehensively evaluate their performance.

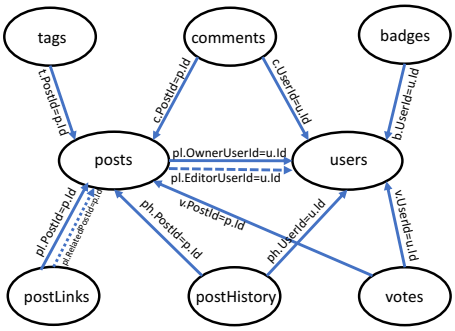

Figure 1: Join relations between tables in STATS.

3 OUR NEW BENCHMARK

In this section, we design a new benchmark with complex real-world data and diverse multi-table join query workload for evaluating CardEst algorithms. To simulate practical scenarios, the benchmark should attain the following properties:

1) Large scale with enough tables, attributes, and tuples in the full outer join;

2) Complex distribution with skewed and correlated attributes whose joint distribution can not be modeled in a straightforward manner (e.g. independent assumption);

3) Rich join schema containing joins of various number of tables and diverse join forms (e.g. star and chain);

4) Diverse workload with queries covering a wide range of true cardinalities and different number of filtering and join predicates.

To this end, we establish our benchmark on a new real-world dataset with a hand-picked query workload. It overcomes the drawbacks of existing CardEst benchmarks and fully fulfills the properties listed above. We describe the details on the data and workload settings in the following content.

Data Setting: We adopt the real-world dataset STATS[1] in our benchmark. It is an anonymized dump of user-contributed content on the Stats Stack Exchange network. STATS consumes 658MB storage space with 8 tables and 71 n./c. attributes on users, posts, comments, tags, and their relations. A comparison of the statistical information between STATS and IMDB (the simplified subset supporting JOB-LIGHT) is shown in Table 1. We argue that STATS more suitable for CardEst benchmark as follows:

1) Larger scale: STATS has more data tables and a larger number of n./c. attributes than the simplified IMDB. Moreover, its full outer join size is four orders of magnitude larger.

2) More complex data distribution: The distribution skewness of STATS and attribute correlation is more significant than the simplified IMDB. Moreover, STATS has 3× more attributes with a larger domain size, suggesting its PDF is much harder to model.

3) Larger query space: Each table in STATS has 1 to 8 n./c. attributes to be filtered while the simplified IMDB contains at most two in each table. Moreover, STATS's full outer join size is much larger than the simplified IMDB. These two aspects provide STATS a larger query space varying in cardinality and predicate numbers.

[1] `https://relational.fit.cvut.cz/dataset/Stats`

Table 1: Comparison of benchmark dataset.

Criteria	Item	IMDB	STATS
Scale	# of tables	6	8
	# of n./c. attributes	8	23
	# of n./c. attributes per table	1–2	1–8
	full outer join size	$2 \cdot 10^{12}$	$3 \cdot 10^{16}$
Data	total attribute domain size	369, 563	578, 341
	average distribution skewness	9.159	21.798
	average pairwise correlation	0.149	0.221
Schema	join forms	star	star/chain/mixed
	# of join relations	5	12

4) Richer join settings: The join relations between all tables in STATS are shown in Figure 1. The simplified IMDB contains only star joins between primary key and foreign keys (i.e. 5 join relations). Whereas, STATS has richer and more diverse join types varying in the number of joined tables (from 2 to 8), join forms (chain, star, and mixture of these two), and join keys (PK-FK and FK-FK).

Query Workload Setting: We generate and then carefully hand-pick a query workload STATS-CEB on STATS to fulfill both practicality and diversity. The generation process is done in two phases.

In the first phase, we generate 70 representative join templates based on the join schema in Figure 1, each of which specifies a distinct join pattern covering a set of tables. For these join templates, we do not consider: 1) cyclic joins as most of the ML-based CardEst algorithms [24, 66, 69, 75] do not support them; and 2) non-equal joins as they rarely occur in practice and many CardEst algorithms process them in the same way as many-to-many joins. We manually check and retain each join template if it has occurred in the log data of StackExchangeor has its real-world semantics. To reduce redundancy, we also ensure that these join templates are not very similar (e.g. only differ in inner or outer join conditions).

In the second phrase after deriving these 70 join templates, we generate 146 queries with $1-4$ queries for each template as the testing query workload STATS-CEB. We make sure all the generated filter predicates reflect real-world semantics and diversify in multiple perspectives. In comparison to JOB-LIGHT (illustrated in Table 2), we find the following advantages of STATS-CEB:

1) More diverse queries: STATS-CEB contains twice queries as JOB-LIGHT with 3× more join templates covering a wider range of the number of joined tables.

2) Richer join types: Unlike JOB-LIGHT benchmark with only star joins, STATS-CEB contains queries with chain joins and complex mixed join forms. Moreover, JOB-LIGHT only contains queries with one-to-many PK-FK joins, whereas STATS-CEB includes queries with many-to-many FK-FK joins.

3) More filter predicates: STATS-CEB contains queries with up to 16 distinct filter predicates, which is 4× larger than JOB-LIGHT.

4) Wider range of true cardinality: The cardinality range of STATS-CEB is an order of magnitude larger than JOB-LIGHT. The largest query in STATS-CEB has true cardinality of 20 billion, which is 3× larger than that of the JOB-LIGHT benchmark.

Summary: Our new benchmark with STATS dataset and STATS-CEB query workload are very comprehensive with more complex data, more skewed distribution, more diverse queries and more complicated join settings.

Table 2: Comparison of benchmark query workload.

Item	JOB-LIGHT	STATS-CEB
# of queries	70	146
# of joined tables	2–5	2–8
# of join templates	23	70
# of filtering n./c. predicates	1–4	1–16
join type	PK-FK	PK-FK/FK-FK
true cardinality range	$9 - 9 \cdot 10^9$	$200 - 2 \cdot 10^{10}$

4 EVALUATION PLAN

We aim to evaluate how CardEst algorithms behave in a real DBMS, including the end-to-end improvement on optimizing query plans and other practicality aspects, on our new benchmark. This section introduces the detailed evaluation plan. Section 4.1 presents all baseline CardEst algorithms chosen to be evaluated, Section 4.2 describes our implementation method and system settings, and Section 4.3 lists the evaluation metrics of our interests.

4.1 CardEst Algorithms

We identify and choose twelve representative CardEst algorithms across the three classes (traditional, ML-based query-driven, and ML-based data-driven) reviewed in Section 2. The selection principles and details of algorithms in each class are elaborated as follows.

Traditional CardEst Algorithms: In this class, we choose five algorithms along the technical directions: 1) for histogram-based methods, we evaluate PostgreSQL and MultiHist, which applies the one-dimensional and multi-dimensional histograms for `CardEst`; 2) for sampling-based methods, we evaluate the uniformly random sampling UniSample method and the more advanced WJSample method for join sampling; and 3) for other methods, we evaluate PessEst, a recent proposed method that exhibit state-of-the-art performance in some aspects. The details are as follows:

1) PostgreSQL [11] refers to the histogram-based CardEst method used in the well-known DBMS PostgreSQL. It assumes that all attributes are mutually independent and maintains a 1-D (cumulative) histogram to represent $P_T(A_i)$ for each attribute A_i. The probability $P_T(Q)$ can then be easily obtained by multiplying all $P_T(R_i)$ together. In addition, optimization strategies such as collecting the most common value and industrial implementation are used to enhance the performance.

2) MultiHist [48] identifies subsets of correlated attributes and model them as multi-dimensional histograms. We use the implementation provided in the repository [71]. We do not compare with the variant methods DBHist [10], GenHist [17, 18] and VIHist [59] over [48] since their improvements are not very significant and their open-sourced implementations are not provided.

3) UniSample [30, 72] makes no assumption but randomly fetches records from T on-the-fly according to $P_T(A)$ to estimate the probability $P_T(Q)$. It is also widely used in DBMS such as MySQL [49] and MariaDB [51]. We set the sampling size to 10^4.

4) WJSample [31] designs a random walk based method called wander join to sample tuples from multiple tables. It has been integrated into DBMSs [32] and exhibits favorable performance in a recent study [44]. We use the implementation in [74] provided by authors in their XDB system and also set the sampling size to

10^4 for comparison. The method [72] then improves from biased to unbiased sampling. We do not compare with it to avoid redundancy.

5) PessEst [3] leverages randomized hashing and data sketching to tighten the bound for multi-join queries. It is a new class of estimator as it never underestimates the cardinality. Meanwhile, it has been verified to perform well in real world DBMS [44]. We use the implementation provided by authors in [2] to generate the sketches and partition the data with 4096 buckets. Different from [44], we apply the same setting as [2] to populate the bound sketches for queries with selection predicates, which would improve the estimation accuracy and generate better plans.

We do not compare with the other variants of traditional methods [1, 13, 22, 26, 27, 30, 53, 54, 61] as they do not exhibit significantly better performance or provide open-source implementation.

ML-Based CardEst Algorithms: In our evaluation, we choose four query-driven (MSCN, LW-XGB, LW-NN and UAE-Q) and four data-driven (NeuroCard, BayesCard, DeepDB and FLAT) CardEst methods. They are representative as they apply different statistical models, and exhibit state-of-the-art performance using each model. Specifically, for query-driven methods, MSCN, LW-XGB/LW-NN and UAE-Q use deep neural networks, classic lightweight regression models and deep auto-regression models to learn the mapping functions, respectively. For the data-driven methods, they build the data distribution utilizing deep auto-regression models and three probabilistic graphical models: BN, SPN, and FSPN, respectively. They use different techniques to balance estimation accuracy, inference efficiency, and model size. Besides, we also evaluate UAE, an extension of UAE-Q using both query and data information. The details are as follows:

6) MSCN [28] is a deep learning method built upon the multi-set convolutional network model. The features of attributes in table T, join relations in query Q, and predicates of query Q are firstly fed into three separate modules, where each is comprised of a two-layer neural network. Then, their outputs are averaged, concatenated, and fed into a final neural network for estimation. We use the implementation provided in the repository [71].

7) LW-XGB and 8) LW-NN [12] formulate the CardEst mapping function as a regression problem and apply gradient boosted trees and neural networks for regression, respectively. Specifically, LW-XGB applies the XGBoost [5] as it attains equal or better accuracy and estimation time than both LightGBM [25] and random forests [56] for a given model size. As the original models only support single table queries, we extend them to support joins with an additional neural network to combine single-table information. We implement them by ourselves since they are not open-sourced.

9) UAE-Q [67] applies the deep auto regression models to learn the mapping function. It proposes differentiable progressive sampling via the Gumbel-Sotfmax trick to enables deep auto-regression models to learn from queries.

For above query-driven CardEst methods, we automatically generate 10^5 queries as the training examples to train these models.

10) NeuroCard [69], the multi-table extension of Naru [70], is built upon a deep auto-regression model. It decomposes the joint PDF $P_T(A) = P_T(A_1) \cdot \prod_{i=2}^{k} P_T(A_i|A_1, A_2, \ldots, A_{i-1})$ according to the chain rule and model each (conditional) PDF parametrically by a 4-layer DNN (4×128 neuron units). All tables can be learned together using a single masked auto-encoder [15]. Meanwhile, a progressive sampling technique [33] is provided to sample points from the region of query Q to estimate its probability. We set the sampling size to 8,000. We omit a very similar method in [20] as it has slightly worse performance than NeuroCard. Worth noticing that the original NeuroCard method is only designed for datasets with tree-structured join schema. On our STATS benchmark with cyclic join schema, we partition the schema into multiple tree-structured join schemas and build one NeuroCard model for each schema. To avoid ambiguity, we denote this extension method as NeuroCard^E in the following content. We use the implementation provided by authors in the repository [35].

11) BayesCard [66] is fundamentally based on BN, which models the dependence relations among all attributes as a directed acyclic graph. Each attribute A_i is assumed to be conditionally independent of the remaining ones given its parent attributes $A_{\mathrm{pa}(i)}$ so the joint PDF $\Pr_T(A) = \prod_{i=1}^{k} \Pr_T(A_i|A_{\mathrm{pa}(i)})$. BayesCard revitalizes BN using probabilistic programming to improve its inference and model construction speed (i.e., learning the dependence graph and the corresponding probability parameters). Moreover, it adopts the advanced ML techniques to process the multi-table join queries, which significantly increases its estimation accuracy over previous BN-based CardEst methods [16, 19, 58], which will not be evaluated in this paper. We use the Chow-Liu Tree [6] based method to build the structure of BayesCard and apply the complied variable elimination algorithm for inference. We use the implementation provided by authors in the repository [64].

12) DeepDB [24], based on sum-product networks (SPN) [9, 47], approximates $P_T(A)$ by recursively decomposing it into local and simpler PDFs. Specifically, the tree-structured SPN contains sum node to split $P_T(A)$ to multiple $P_{T'}(A)$ on tuple subset $T' \subseteq T$, product node to decompose $P_T(A)$ to $\prod_S P_T(S)$ for independent set of attributes S and leaf node if $P_T(A)$ is a univariate PDF. The SPN structure can be learned by splitting table T in a top-down manner. Meanwhile, the probability of $\Pr_T(Q)$ can be obtained in a bottom-up manner with time cost linear in the SPN's node size.

13) FLAT [75], based on factorize-split-sum-product networks (FSPN) [68], improves over SPN by adaptively decomposing $P_T(A)$ according to the attribute dependence level. It adds the factorize node to split P_T as $P_T(W) \cdot P_T(H|W)$ where H and W are highly and weakly correlated attributes in T. $P_T(W)$ is modeled in the same way as SPN. $P_T(H|W)$ is decomposed into small PDFs by the split nodes until H is locally independent of W. Then, the multi-leaf node is used to model the multivariate PDF $P_T(H)$ directly. Similar to SPN, the FSPN structure and query probability can be recursively obtained in a top-down and bottom-up fashion, respectively.

For both DeepDB and FLAT, we set the RDC thresholds to 0.3 and 0.7 for filtering independent and highly correlated attributes, respectively. Meanwhile, we do not split a node when it contains less than 1% of the input data. We use the implementations provided by authors in the repository [23] and [65], respectively.

14) UAE [67] extends the UAE-Q method by unifiying both query and data information using the auto-regression model. It is a representative work aiming at closing the gap between data-driven and query-driven CardEst methods. Both UAE and UAE-Q are implemented by authors in the repository [62].

Remarks: For other hyper-parameters, if they are known to be a trade-off of some metrics, we choose the default values recommended in the original paper. Otherwise, we run a grid search to explore the combination of value that largely improves the end-to-end performance on a validation set of queries. For algorithms with randomness (UniSample, WJSample, NeuroCard^E and UAE-Q/UAE), we ran each test 10 times and report the average results. Notice that, there have also been proposed some CardEst modules [55, 67] that are optimized together with other components in a query optimizer in an end-to-end manner. We do not compare with them as they do not fit our evaluation framework.

4.2 Implementation and System Settings

To make our evaluation more realistic and convincing, we integrate each CardEst algorithm into the query optimizer of PostgreSQL [11], a well-recognized open-source DBMS. Then, the quality of each CardEst method can be directly reflected by the end-to-end query runtime with their injected cardinality estimation.

Before introducing the details of our integration strategy, we introduce an important concept called *sub-plan query*. For each SQL query Q, each *sub-plan query* is a query touching only a subset of tables in Q. The set of all these queries is called *sub-plan query space*. For the example query $A \bowtie B \bowtie C$, its sub-plan query space contains queries on $A \bowtie B$, $A \bowtie C$, $B \bowtie C$, A, B, and C with the corresponding filtering predicates. The built-in planner in DBMS will generate the sub-plan query space, estimate their cardinalities, and determine the optimal execution plan. For example, the sub-plan queries A, B, and C only touch a single table, their CardEst results may affect the selection of table-scan methods, i.e. index-scan or seq-scan. The sub-plan queries $A \bowtie B$, $A \bowtie C$, and $B \bowtie C$ touch two tables. Their cardinalities may affect the join order, i.e. joining $A \bowtie B$ with C or $A \bowtie C$ with B, and the join method, i.e. nested-loop-join, merge-join, or hash-join. Therefore, the effects of a CardEst method on the final query execution plan are entirely decided by its estimation results over the sub-plan query space.

To this end, in our implementation, we overwrite the function "calc_joinrel_size_estimate" in the planner of PostgreSQL to derive the sub-plan query space for each query in the workload. Specifically, every time the planner needs a cardinality estimation of a sub-plan query, the modified function "calc_joinrel_size_estimate" will immediately capture it. Then, we call each CardEst method to estimate the cardinalities of the sub-plan queries and inject the estimations back into PostgreSQL. Afterward, we run the compiler of PostgreSQL on Q to generate the plan. It will directly read the injected cardinalities produced by each method. Finally, we execute the query with the generated plan. In this way, we can support any CardEst method without a large modification on the source code of PostgreSQL. We can report the total time (except the sub-plan space generation time) as the end-to-end time cost of running a SQL query using any CardEst method.

For the environment, we run all of our experiments in two different Linux Servers. The first one with 32 Intel(R) Xeon(R) Platinum 8163 CPUs @ 2.50GHz, one Tesla V100 SXM2 GPU and 64 GB available memory is used for model training. The other one with 64 Intel(R) Xeon(R) E5-2682 CPUs @ 2.50GHz is used for the end-to-end evaluation on PostgreSQL.

4.3 Evaluation Metrics

Our evaluation mainly focuses on *quantitative* metrics that directly reflect the performance of CardEst algorithms from different aspects. We list them as follows:

1) End-to-end time of the query workload, including both the query plan generation time and physical plan execution time. It serves as the "gold-standard" for CardEst algorithm, since improving the end-to-end time is the ultimate goal for optimizing CardEst in query optimizers. We report the end-to-end time of TrueCard, which injects the true cardinalities of all sub-plan queries into PostgreSQL. Ideally if the cost model is very accurate, TrueCard can obtain the optimal plan with shortest time. For a fixed PostgreSQL cost model, we find TrueCard can obtain the optimal query plan for most of the time. Thus, this could serve as a good baseline.

2) Inference latency reflects the time cost for CardEst, which directly relates to the query plan generation time. It is crucial as CardEst needs to be done numerous times in optimizing the plan of each query. Specifically, an accurate CardEst method may be very time-costly in inference. Despite the fast execution time of the plans generated by this method, the end-to-end query performance can be poor because of its long plan generation time.

3) Space cost refers to the CardEst model size. A lightweight model is also desired as it is convenient to transfer and deploy.

4) Training cost refers to the models' offline training time.

5) Updating speed reflects the time cost for CardEst models update to fit the data changes. For real-world settings, this metric plays an important role as its underlying data always updates with tuples insertions and deletions.

Besides these metrics, [60] proposed some *qualitative metrics* related to the stability, usage, and deployment of CardEst algorithms and made a comprehensive analysis. Thus, we do not consider them in this paper. In the following, we first evaluate the overall end-to-end performance of all methods in Section 5. Then, we analyze the other practicality aspects in Section 6. At last, we point out the drawbacks of existing evaluation metric and propose a new metric as its potential substitution in Section 7.

5 HOW GOOD ARE CARDEST METHODS?

In this section, we first thoroughly investigate the true effectiveness of the aforementioned CardEst methods in improving query plan quality. Our evaluation focuses on a static environment where data in the system has read-only access. This setting is ubiquitous and critical for commercial DBMS, especially in OLAP workloads of data warehouses[14, 39, 45, 73]. We organize the experimental results as follows: Section 5.1 reports the overall evaluation results, Section 5.2 provides detailed analysis of the method's performance on various query types and an in-depth case study on the performance of some representative methods.

5.1 Overall End-to-End Performance

We evaluate the end-to-end performance (query execution time plus planning time) on both JOB-LIGHT and STATS-CEB benchmarks for all CardEst methods including two baselines PostgreSQL and TrueCard shown in Table 3. We also report their relative improvement over the PostgreSQL baseline as an indicator of their end-to-end performance. In the following, we will first summarize

Table 3: Overall performance of CardEst algorithms.

Category	Method	Data / Workload					
		IMDB / JOB-LIGHT			STATS / STATS-CEB		
		End-to-End Time	Exec. + Plan Time	Improvement	End-to-End Time	Exec. + Plan Time	Improvement
Baseline	PostgreSQL	3.67h	3.67h + 3s	0.0%	11.34h	11.34h + 25s	0.0%
	TrueCard	**3.15h**	**3.15h + 3s**	**14.2%**	**5.69h**	**5.69h + 25s**	**49.8%**
Traditional	MultiHist	3.92h	3.92h + 30s	−6.8%	14.55h	14.53h + 79s	−28.3%
	UniSample	4.87h	4.84h + 96s	−32.6%	> 25h	−−	−−
	WJSample	4.15h	4.15h + 23s	−13.1%	19.86h	19.85h + 45s	−75.0%
	PessEst	3.47h	3.38h + 324s	5.4%	6.42h	6.10h + 1,135s	43.4%
Query-driven	MSCN	3.50h	3.50h + 12s	4.6%	8.13h	8.11h + 46s	28.3%
	LW-XGB	4.31h	4.31h + 8s	−17.4%	> 25h	−−	−−
	LW-NN	3.63h	3.63h + 9s	1.1%	11.33h	11.33h + 34s	0.0%
	UAE-Q	3.65h	3.55h+356s	−1.9%	11.21h	11.03h+645s	1.1%
Data-driven	NeuroCard^E	3.41h	3.29h + 423s	6.8%	12.05h	11.85h + 709s	−6.2%
	BayesCard	3.18h	3.18h + 10s	13.3%	7.16h	7.15h + 35s	36.9%
	DeepDB	3.29h	3.28h + 33s	10.3%	6.51h	6.46h + 168s	42.6%
	FLAT	3.21h	3.21h + 15s	12.9%	5.92h	5.80h + 437s	47.8%
Query + Data	UAE	3.71h	3.60h + 412s	−2.7%	11.65h	11.46h + 710s	−0.02%

several overall observations (O) regarding Table 3, and then provide detailed analysis w.r.t. each of these CardEst methods.

O1: Most of the ML-based data-driven CardEst methods can achieve remarkable performance, whereas most of the traditional and ML-based query-driven CardEst methods do not have much improvement over PostgreSQL. The astonishing performance of these ML-based data-driven CardEst methods (BayesCard, DeepDB, and FLAT) come from their accurate characterization of data distributions and reasonable independence assumption over joined tables. Traditional histogram and sampling based methods (MultiHist, UniSample, and WJSample) have worse performance than PostgreSQL whereas the new traditional approach (PessEst) is significantly better. The query-driven CardEst methods' performance is not stable. They rely on a large amount of executed queries as training data and the testing query workload should follow the same distribution as the training workload to produce an accurate estimation [24].

O2: The differences among the CardEst methods' improvements over PostgreSQL are much more drastic on datasets with more complicated data distributions and join schemas. We observe that the execution time for CardEst method that can outperform PostgreSQL (PessEst, NeuroCard^E, BayesCard, DeepDB, and FLAT) on JOB-LIGHT are all roughly $3.2h$, which is very close to the minimal execution time of TrueCard($3.15h$). As explained in Section 3, the data distributions in the simplified IMDB dataset and the JOB-LIGHT queries are relatively simple. Specifically, the table title in the IMDB dataset plays a central role in the join schema that other tables are all joined with its primary key, so the joint distribution could be easily learned. However, their performance differences on STATS are very drastic because the STATS dataset is much more challenging with high attribute correlations and various join types. Therefore, the STATS-CEB benchmark can help expose the advantages and drawbacks of these methods.

Analysis of Traditional CardEst Methods: Histogram and sampling based methods perform significantly worse than PostgreSQL on both benchmarks because of their inaccurate estimation. MultiHist and UniSample use the join uniformity assumption to estimate join queries, whose estimation error grows rapidly for queries joining more tables. WJSample makes a random walk based sample across the join of multiple tables. However, as the cardinality increases with the number of joined tables, the relatively small sample size can not effectively capture the data distribution, leading to large estimation error. These queries joining larger number of tables are generally more important in determining a good execution join order [29]. Therefore, these methods tend to yield poor join orders and long-running query plans. The PostgreSQL produces more accurate estimations because of its high-quality implementation and fine-grained optimizations on join queries. The new traditional method PessEst has a significant improvement over the PostgreSQL because it can compute the upper bound on estimated cardinalities to avoid expensive physical join plans. Notice that, we observe that PessEst is more effective than WJSample, which is different from [44]. This is because our bound sketches in PessEst are built with selection predicates while [44] builds them on rough plain join schema. As a result, the estimation accuracy, so as the execution time, of PessEst largely improves. However, the plan (inference) time of PessEst with sketch construction is much longer than others, sometimes even longer than the execution time of OLTP queries. This degrades the practicality of PessEst.

Analysis of ML-based Query-driven CardEst Methods: Overall the query-driven methods have comparable performance to the PostgreSQL baseline. Specifically, MSCN can slightly outperform the PostgreSQL (4.6% faster runtime on JOB-LIGHT and 19.7% faster on STATS-CEB), LW-XGB has much slower query runtime, and LW-NN has comparable performance. The unsatisfactory performance of these methods could be due to the following reasons.

- These methods are essentially trying to fit the probability distributions of all possible joins in the schema, which has super-exponential complexity. Specifically, there can exist an exponential number possible joins in a schema, and for each joining table, the complexity of its distribution is exponential w.r.t. its attribute number and domain size [66]. Thus, these models themselves are not complex enough to fully understand all these distributions.
- Similarly, these methods would require an enormous amount of executed queries as training data to accurately characterize these

Table 4: End-to-end time improvement ratio of CardEst algorithms on queries with different number of join tables.

# tables	# queries	PessEst	MSCN	BayesCard	DeepDB	FLAT	TrueCard
2 – 3	38	2.62%	2.04%	2.07%	1.98%	2.48%	**3.66%**
4	50	53.1%	−12.3%	55.8%	48.0%	55.7%	**55.9%**
5	28	31.7%	29.8%	36.55%	32.90%	35.4%	**37.0%**
6 – 8	34	29.6%	−4.06%	2.51%	26.3%	32.0%	**34.6%**

complex distributions. In our experiment, our computing resources can only afford to generate 10^5 queries (executing 146 queries in STATS-CEB takes 10 hours), which may not be enough for this task. Besides, it is unreasonable to assume that a CardEst method can have access to this amount of executed queries in reality.

• The well-known workload shift issue states that query-driven methods trained one query workload will not likely produce an accurate prediction on a different workload [24]. In our experiment, the training query workload is automatically generated whereas the JOB-LIGHT and STATS-CEB testing query workload is hand-picked. Therefore, the training and testing workload of these methods have different distributions.

Analysis of ML-based Data-driven Methods: Data-driven ML methods (BayesCard, NeuroCard^E, DeepDB, and FLAT), do consistently outperform PostgreSQL by 7 – 13% on JOB-LIGHT. Except for NeuroCard^E, the other three improve the PostgreSQL by 37–48% on STATS-CEB.Their performance indicates that data-driven methods could serve as a practical counterpart of the PostgreSQL CardEst component. Through detailed analysis of NeuroCard^E method, we derive the following observation:

O3: Learning one model on the (sample of) full outer join of all tables in a DB may lead to poor scalability. We conjecture that an effective CardEst method should make appropriate independent assumptions for large datasets. The advantages of NeuroCard^E over PostgreSQL disappear when shifting from JOB-LIGHT to STATS-CEB benchmark for the following reasons. First, the STATS dataset contains significantly more attributes with larger domain size, which can be detrimental to NeuroCard^E's underlying deep auto-regressive models [60, 66]. Second, the full outer join size of STATS is significantly larger than the simplified IMDB, making the sampling procedure of NeuroCard^E less effective. Specifically, the full outer join size can get up to 3×10^{16} and an affordable training data sample size would be no larger than 3×10^8. Therefore, the NeuroCard^E model trained on this sample only contains 1×10^{-8} of the information as the original dataset. Third, the join keys in STATS dataset have very skewed distribution. E.g. there exist key values of a table that can match with zero or one as well as hundreds of tuples in another table. This complicated distribution of join keys makes NeuroCard^E's full outer join sample less effective. Therefore, NeuroCard^E can hardly capture the correct data distributions especially for join tables with small cardinalities. Specifically, we find that for queries on the joins of a small set of tables, NeuroCard^E's prediction deviates significantly from the true cardinality because its training sample does not contain much not-null tuples for this particular set of join tables.

All other three data-driven CardEst methods can significantly outperform the PostgreSQL baseline because their models are not constructed on the full outer join of all tables. Specifically, they all use the "divide and conquer" idea to divide the large join schema into several smaller subsets with each representing a join of multiple tables. In this way, they can capture the rich correlation within each subset of tables; simultaneously, they avoid constructing the full outer join of all tables by assuming some independence between tables with low correlations. Then, BayesCard, DeepDB, and FLAT build a model (BN, SPN, and FSPN respectively) to represent the distribution of the corresponding small part. This approach solves the drawback of NeuroCard^E, yields relatively accurate estimation, and produces effective query execution plans. Among them, FLAT achieves the best performance (47.8% improvement), which is very close to the improvement 49.8% for TrueCard. It can outperform DeepDB mostly because the STATS dataset is highly correlated, so the FSPN in FLAT has a more accurate representation of the data distribution than the SPN in DeepDB. On the other hand, BayesCard has an even more accurate representation of data distribution and yields the best end-to-end time for most queries in STATS-CEB. It does not outperform FLAT most because of one extremely long-run query, which we will study in detail in Section 5.2.

5.2 Analysis of Different Query Settings

In this section, we further examine to what extent the CardEst methods improve over PostgreSQL on various query types, i.e. different number of join tables (*#tables*) and different intervals of true cardinalities. Since JOB-LIGHT workload does not contain queries with very diverse types and the ML-based data-driven methods do not show significant difference on these queries, we only investigate queries on STATS-CEB. Worth noticing that we only examine the methods with clear improvements over PostgreSQL on STATS-CEB: MSCN, BayesCard, DeepDB, and FLAT.

Number of Join Tables: Table 4 shows performance improvement of different ML-based methods over the PostgreSQL baseline and we derive the following observation:

O4: The improvement gaps between these methods and the performance of TrueCard increase with the number of join tables. Specifically, the BayesCard achieves near-optimal improvement for queries joining no more than 5 tables, but it barely has much improvement for queries joining 6 tables and more. This observation suggests that the estimation qualities of these SOTA methods decline for queries joining more tables. In fact, the fanout join estimation approach adopted by all these methods sacrifices accuracy for efficiency by assuming some tables are independent of others. This estimation error may accumulate for queries joining a large number of tables, leading to sub-optimal query plans.

Size of Cardinality: We choose Q57 (in Figure 2) of STATS-CEB as a representative query to study the effect of estimation accuracy w.r.t. different cardinalities and investigate when a CardEst method could go wrong. The execution time of Q57 for TrueCard and FLAT is $1.90h$ and $1.92h$, while the time for BayesCard is $3.23h$. We derive two important observations from this query, which are verified to be generalizable to other queries in JOB-LIGHT and STATS-CEB.

O5: Accurate estimation of (sub-plan) queries with large cardinalities is sometimes more important than the small ones. When choosing the join method in the root node of execution plans for Q57, BayesCard underestimates the final join size and chooses the "merge join" physical operation. Alternatively, FLAT produces

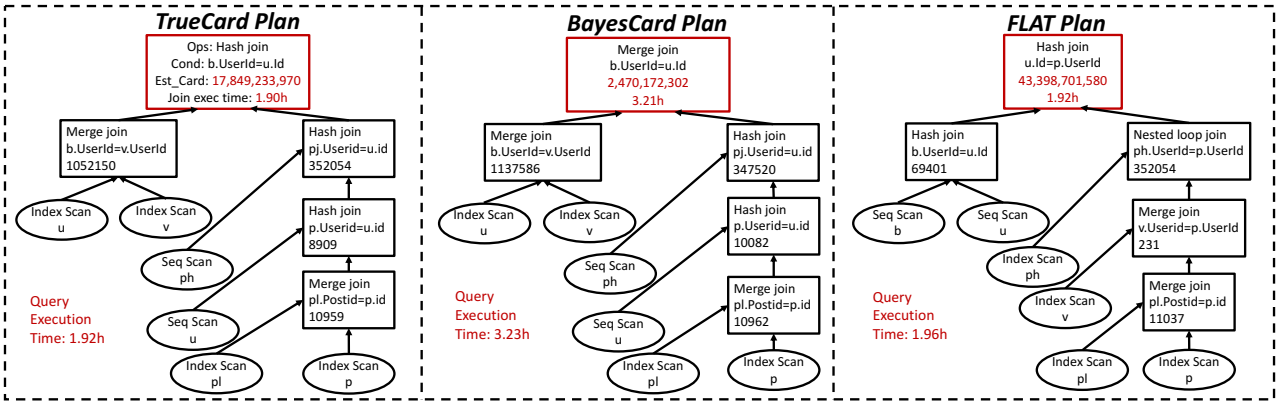

Figure 2: Case study of STATS Q57.

a more accurate estimation for the final join size and chooses the "hash join" operation, which is twice as faster as the "merge join". Since the final join operation takes up 99% of the total execution time, FLAT significantly outperforms BayesCard on this query.

Generally, the (sub-plan) query with larger true cardinality requires a longer time to execute. It is very common that the join size of two intermediate tables is much larger than both of them. Therefore, some sub-plans can take a significantly longer time to execute than other sub-plans. A bad estimation on these large sub-plan queries can have a detrimental result on the overall runtime, whereas a series of good estimations on small sub-plan queries will not influence the runtime as much. Therefore, the estimation accuracy of sub-plan queries with very large true cardinalities dominate the overall quality of the query plan.

O6: Choosing the correct physical operations sometimes is more important than selecting the optimal join order. As shown in Figure 2 BayesCard can generate the optimal join order of Q57 because of its near-perfect estimation of all sub-plan queries except for the one at the root node. The join order selected by FLAT is very different from the optimal one. Surprisingly, FLAT's plan is roughly twice faster to execute than BayesCard's plan due to the dominant large sub-plan query at the root node. For Q57, a sub-optimal query plan is only 1% slower to execute but only one sub-optimal physical operation is 68% slower.

These aforementioned two observations also hold for other queries in these two benchmarks, so we conjecture that they might be generalizable to all queries.

6 WHAT OTHER ASPECTS OF CARDEST METHODS MATTER?

In addition to CardEst's improvement in execution time, we discuss model practicality aspects in this section: inference latency (in Section 6.1), model size and training time (in Section 6.2), and model update speed and accuracy (in Section 6.3). We only compare the recently proposed CardEst methods, which have been proved to significantly improve the PostgreSQL baseline, namely PessEst, MSCN, NeuroCard^E, BayesCard, DeepDB, and FLAT.

Table 5: OLTP/OLAP Performance on STATS-CEB.

Methods	TP E2E Time	TP Plan Time	AP E2E Time	AP Plan Time
PostgreSQL	49.5s	4.8s (9.7%)	11.33h	20.3s (0.05%)
TrueCard	13s	4.8s (36.9%)	5.69h	20.3s (0.1%)
PessEst	27.7s	8.4s (30.3%)	6.10h	35.4s (0.16%)
MSCN	23.9s	8.2s (34.3%)	8.12h	38.0s (0.13%)
NeuroCard^E	99.3s	73s (73.5%)	11.94h	350s (0.81%)
BayesCard	18s	7.3s (40.6%)	7.16h	27.4s (0.11%)
DeepDB	45.1s	33.6s (74.5%)	6.50h	135s (0.58%)
FLAT	55.8s	41.5s (74.4%)	5.91h	396s (1.86%)

6.1 Inference Latency

The end-to-end query time is comprised of query execution and planning time, the latter of which is determined by the CardEst method's inference latency. Commercial DBMS normally has a negligible planning time due to their simplified cardinality estimator and engineering effort to accelerate the inference speed. However, the inference latency of some ML-based data-driven methods can approach one second per sub-plan query, which slows down the end-to-end query execution time. To further illustrate the importance of inference latency, we divide the STATS-CEB queries into OLTP and OLAP two workloads based on the query execution time. We report the results in Table 5 and derive the following observation.

O7: Inference latency can have a significant impact on the OLTP workload but a trivial impact on the OLAP workload. On OLTP workload of STATS-CEB, we observe that the planning time composes a large proportion of total end-to-end time. Specifically, some ML-based methods' (NeuroCard^E, DeepDB, and FLAT) inference speeds are relatively slow. Although their execution time on OLTP workload is faster than PostgreSQL, they have worse end-to-end performance because of the long planning time. For OLAP workload of STATS-CEB, the CardEst methods' planning time is much shorter than their execution time because OLAP workload contains extremely long-run queries. In this case, the quality of the generated query plans overshadows the slow inference latency. Therefore, we believe that CardEst methods targeting different workloads should fulfill different objectives. For OLTP workload, a desirable method should have fast inference speed, whereas the methods targeting OLAP workload can have high inference latency as long as they can produce high-quality query plans.

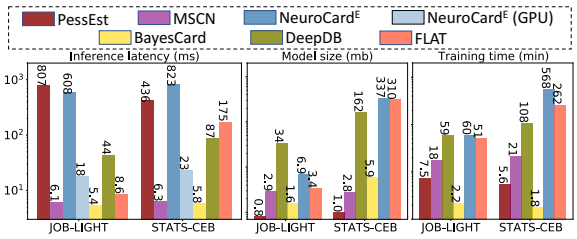

Figure 3: Practicality aspects of CardEst algorithms.

Table 6: Update performance of CardEst algorithms.

Criteria	NeuroCard^E	BayesCard	DeepDB	FLAT
Update time	5,569s	**12s**	248s	360s
Original E2E time	11.85h	7.16h	6.46h	5.80h
E2E time after update	13.94h	7.16h	6.72h	7.04h

Figure 3 reports the average inference latencies of all sub-queries in the workload for each method. Their inference speed can be ranked as BayesCard > NeuroCard^E(GPU) > FLAT/DeepDB >> NeuroCard^E. The newly proposed inference algorithms on BN provide BayesCard with a very fast and stable inference speed on both benchmarks. However, the inference speeds of FLAT and DeepDB are not as stable because they tend to build much larger models with more computation circuits for the more complicated database STATS. The inference process of NeuroCard requires a large number of progressive samples and its underlying DNN is computationally demanding on CPUs. Therefore, we observe that the inference speed is greatly improved by running it on GPUs.

6.2 Model Deployment

Figure 3 reports the model size and training time of all aforementioned methods. Based on the results of STATS-CEB query, we derive the following observation.

O8: BN-based CardEst approach is very friendly for system deployment. First of all, the BN-based approaches, such as BayesCard, are generally interpretable and predictable, thus easy to debug for DBMS analytics. More importantly, a CardEst method friendly for system deployment should have faster training time and lightweight model size and BayesCard has the dominant advantage over the other ML-based data-driven methods in these two aspects because of its underlying Bayesian model. Specifically, from both training time and model size aspects, these methods can be ranked as BayesCard << FLAT/DeepDB < NeuroCard^E. We provide the detailed reasoning as follows.

BayesCard proposes an accelerated model construction process of BN using probabilistic programming. Its model training time is roughly 100 times faster than the other three methods. Moreover, the BNs in BayesCard, which utilize the attribute conditional independence to reduce the model redundancy, are naturally compact and lightweight.

FLAT and DeepDB recursively learn the underlying FSPN and SPN models. Their training time is less stable and varies greatly with the number of highly correlated attributes in the datasets. Thus, we observe a much longer training time on STATS than on the IMDB dataset for these two methods. The SPNs in DeepDB iteratively split the datasets into small regions, aiming to find local independence between attributes within each region. However, in presence of highly correlated attributes (e.g. STATS), the SPNs tend to generate a long chain of dataset splitting operation, leading to long training time and a very large model size. The FSPNs in FLAT effectively address this drawback of SPNs by introducing the factorize operation but their training time and model size suffer greatly for datasets with a large number of attributes (e.g. STATS) because of the recursive factorize operations.

The training of NeuroCard^E is particularly long and its size is also the largest on STATS because its join schema does not form a tree. As mentioned in Section 4.1, the original NeuroCard only supports tree-structured schemas. Thus, NeuroCard^E extracts 16 tree sub-structures from STATS schema graph and train one model for each tree. Therefore, we argue that extending NeuroCard for non-tree-structured schemas can greatly improve its practicality.

6.3 Model Update

Model update is a crucial aspect when deploying a CardEst method in OLTP databases. Frequent data updates in these DBs require the underlying CardEst method to swiftly update itself and adjust to the new data accurately. In the following, we first provide our observation regarding the updatability of ML-based query-driven methods and then provide the update experimental settings and results for ML-based data-driven methods on the STATS dataset.

O9: Existing query-driven CardEst methods are impractical for dynamic DBs. The query-driven models require a large amount of executed queries to train their model, which might be unavailable for a new DB and very time-consuming to generate (e.g. 146 STATS-CEB queries take more than ten hours to execute). More importantly, they need to recollect and execute the queries whenever datasets change or query workload shifts. Therefore, they can not keep up with the frequent data update in dynamic DBs.

Experimental Settings: To simulate a practical dynamic environment, we split the STATS data into two parts based on the timestamps of tuples. We first train a stale model for each method on the data created before 2014 (roughly 50%) and insert the rest of the data to update these models. We only test the data insertion scenario since some methods (NeuroCard^E and DeepDB) do not support data deletion. We use the update algorithm in these methods' publicly available source code.

Experimental Results: As shown in Table 6, we record the time these methods take to update their stale models and evaluate the end-to-end query performance of the updated models on STATS-CEB queries. We also cite the original model performance from Table 3 as comparison baselines. We first summarize the most important observation based on this table and then provide detailed reasoning from the update speed and accuracy perspectives.

O10: Data-driven CardEst methods have the potential to keep up with fast data update and can be applied in dynamic DBs. Specifically, BayesCard takes 12*s* to update itself for an insertion of millions of tuples on multiple tables in the DB. More important, its end-to-end performance is unaffected by this massive update, thus very suitable for dynamic DBs.

Update speed of these methods can be ranked as BayesCard >> DeepDB > FLAT > NeuroCard^E. BayesCard preserves its underlying BN's structure and only incrementally updates the model parameters. Since its model size is relatively small, the update speed is more than 20 times faster than others. DeepDB and FLAT also preserves their underlying structure of SPN and FSPN but as their structures are significantly larger, the incrementally updating their model parameters still take a large amount of time.

Update accuracy can be ranked as BayesCard > DeepDB > FLAT > NeuroCard^E. BayesCard's underlying BN's structure captures the inherent causality, which is unlikely to change when data changes. Therefore, BayesCard can preserve its original accuracy after model update (i.e. same as its comparison baseline). The structures of SPN in DeepDB and FSPN in FLAT are learned to fit the data before the update and cannot extrapolate well to the newly inserted data. Therefore, only updating the model parameters will cause modeling inaccuracy (i.e. we observe a drop in their end-to-end performance when compared with their baselines).

7 IS CURRENT METRIC GOOD ENOUGH?

Most of the existing works [12, 24, 28, 69, 75] use Q-Error [37] to evaluate the quality of their CardEst methods. However, the ultimate goal of CardEst is to generate query plans with faster execution time. Therefore, we explore whether Q-Error is a good metric to fulfill this goal in this section. We first analyze the correlations between Q-Error and query execution time in Section 7.1. The results show that smaller Q-Error does not necessarily lead to shorter execution time. Thus, we identify the limitations of Q-Error and propose another metric called P-Error in Section 7.2. We show that P-Error has better correspondence to query execution time and advocate it to be a potential substitution of Q-Error.

7.1 Problems with Q-Error

Q-Error is a well-known metric to evaluate the quality of different CardEst methods. It measures the relative multiplicative error of the estimated cardinality from the actual one as:

$$\textbf{Q-Error} = \max\left(\frac{\text{Estimated Cardinality}}{\text{True Cardinality}}, \frac{\text{True Cardinality}}{\text{Estimated Cardinality}}\right).$$

Q-Error penalizes both overestimation and underestimation of the true cardinality. However, existing works have not investigated whether Q-Error is good evaluation metric for `CardEst`. I.e, would CardEst methods with smaller Q-Errors definitely generate query plans with shorter execution time and vice versa? To answer this question, we revisit the experimental results. Table 7 reports the distributions (50%, 90% and 99% percentiles) of all sub-plan queries' Q-Errors generated by different CardEst methods on both JOB-LIGHT and STATS-CEB benchmarks. Please note that we report all but the PessEst method because it is a bound-based method whose Q-Error is not fairly comparable with other methods. We sort all CardEst methods in a descending order of their execution time. From a first glance, we derive the following observation:

O11: The Q-Error metric can not serve as a good indicator for query execution performance. This observation is supported by a large amount of evidence from Table 7. We list three typical examples on STATS-CEB as follows: 1) NeuroCard^E has the worst Q-Errors in all methods, but its execution time is comparable to PostgreSQL and much better than histogram and sampling based methods and LW-XGB; 2) BayesCard has the best Q-Errors, yet execution time is 1.4h slower than FLAT; and 3) the Q-Errors of MSCN are significantly worse than PostgreSQL, but the execution time of MSCN largely outperforms it.

Table 7: Comparison between Q-Error and P-Error.

Method	Execution Time	Q-Error			P-Error		
	(Descending Order)	50%	90%	99%	50%	90%	99%
UniSample	> 25h	3.259	135.4	$6 \cdot 10^4$	1.000	1.345	3.593
LW-XGB	> 25h	5.652	453.2	$3 \cdot 10^5$	1.742	6.627	526.2
WJSample	19.86h	3.452	351.4	$8 \cdot 10^4$	1.103	4.954	501.6
NeuroCard	11.85h	951.4	$9 \cdot 10^5$	$6 \cdot 10^8$	1.193	2.844	$1 \cdot 10^3$
UAE	11.46h	3.239	130.8	$1.1 \cdot 10^4$	1.187	2.420	7.873
PostgreSQL	**11.34h**	**1.439**	**11.08**	$\mathbf{2 \cdot 10^3}$	**1.000**	**1.595**	**12.35**
LW-NN	11.33h	15.73	832.9	$2 \cdot 10^4$	1.159	4.651	18.02
UAE-Q	11.03h	2.875	20.613	$1 \cdot 10^4$	1.145	4.001	13.14
MSCN	8.11h	20.92	392.3	$1 \cdot 10^4$	1.138	4.031	11.11
BayesCard	7.16h	1.182	50.40	156.4	1.000	1.582	6.843
DeepDB	6.46h	2.451	22.37	$1 \cdot 10^3$	1.030	1.833	6.819
FLAT	5.80h	1.675	10.44	768.8	1.000	1.346	5.546

Next, we analyze the underlying reasons behind $O11$. This is particularly important as the DB communities have made great efforts in purely optimizing the Q-Error of CardEst methods, but sometimes neglect the ultimate goal of CardEst in DBMS. As shown in Section 4.2, the CardEst method would be invoked for multiple sub-plan queries to decide the query plan. The estimation errors of different sub-plan query have different impact on the final query plan performance. However, the Q-Error metric could not distinguish this difference and regard the estimation errors of all queries equally. This would cause the phenomenon that a more accurate estimation measured by Q-Error may lead to a worse query execution plan in reality. We list two typical scenarios in the benchmark where Q-Error fails to distinguish the difference as follow:

O12: Q-Error does not distinguish queries with small and large cardinality that have the same Q-Error value but matter differently to the query plan. For Q-Error, an estimation 1 for true cardinality of 10 has the same Q-Error as an estimation 10^{11} for true cardinality 10^{12}. The previous case may barely affect the overall query plan, whereas the latter one can be catastrophic since the (sub-plan) queries with large cardinalities dominate the overall effectiveness of the query plan (shown in O5). For example, in Figure 2, the overall Q-Error of BayesCard over all sub-plan queries of Q57 is better than FLAT. However, only for the root query which matters most importantly to the query execution time, BayesCard fails to correctly estimate and leads to a much slower plan.

O13: Q-Error can not distinguish between query underestimation and overestimation that have the same Q-Error value but matter differently to the query plan. For Q-Error, an underestimation 10^9 for true cardinality 10^{10} is the same as an overestimation of 10^{11}. These two estimations are very likely to lead to different plans with drastically different execution time. Recall the Q57 example, BayesCard underestimates the cardinality of the root query by 7× and selects a "merge join" operation. We test this query but injecting a 7× overestimation for this sub-plan query, and it then selects the "hash join" operation with twice faster time.

As a result, Q-Error does not consider the importance of different sub-plan queries and may mislead the query plan generation..

7.2 An Alternative Metric: P-Error

Obviously, the best way to evaluate the quality of a CardEst method is to directly record its query execution time on some benchmark datasets and query workloads (e.g. JOB-LIGHT and STATS-CEB). However, this is time consuming and not suitable for the situations where fast evaluation is needed, e.g., hyper-parameter tuning. A desirable metric should be fast to compute and simultaneously correlated with the query execution time. In the following, we propose the P-Error metric to fulfill this goal and quantitatively demonstrate that P-Error can be a possible substitute for Q-Error.

P-Error metric for CardEst: Although obtaining the actual query execution time is expensive, we could approximate it using the built-in component in DBMS. Note that, given a query plan, the cost model of a DBMS could output an estimated cost, which is designed to directly reflect the actual execution time. Inspired by the recent research [41], we believe that the estimated cost could serve as a good metric for evaluating the CardEst methods.

Specifically, given a query Q and a CardEst method A, let C^{T} and C^{E} denote the set of true and estimated cardinality of all sub-plan queries of Q. When C^{E} is fed into the query optimizer, it would generate a query plan $P(C^{\mathrm{E}})$ of Q. During the actual execution of this query plan, the true cardinalities of all sub-plan queries along this plan will be instantiated. Therefore, to estimate the execution cost of $P(C^{\mathrm{E}})$, we inject the true cardinality of sub-plan queries C^{T} into the DBMS. The DBMS will output an estimated cost based on this query plan, which is highly correlated to the actual time for an accurate cost model. Following prior work [41], we choose PostgreSQL to calculate this estimated cost, which is denoted as $PPC(P(C^{\mathrm{E}}), C^{\mathrm{T}})$. Ideally, if the cost model is accurate, the query plan $P(C^{\mathrm{T}})$ found by the true cardinality C^{T} should be optimal, i.e. $PPC(P(C^{\mathrm{T}}), C^{\mathrm{T}}) = \min_C PPC(P(C), C^{\mathrm{T}})$. Therefore, we define

$$\textbf{P-Error} = PPC(P(C^{\mathrm{E}}), C^{\mathrm{T}}) / PPC(P(C^{\mathrm{T}}), C^{\mathrm{T}})$$

as our CardEst metric. The P-Error for an existing workload of queries can be computed instantaneously using the modified plugin pg_hint_plan provided in [40] as long as we pre-compute and store the true cardinalities of all sub-plan queries.

In P-Error, the effectiveness of a CardEst method's estimation C^{E} is measured on the plan cost level. The impact on the estimation error of each sub-plan query is reflected by its importance in generating the query plan $P(C^{\mathrm{E}})$ (e.g. small or large cardinality, underestimation or overestimation, etc.).

Notice that, in real-world DBMS, the cost model can sometimes be inaccurate [29], which may lead to worse query plans with better estimated cost, i.e., $PPC(P(C^{\mathrm{T}}), C^{\mathrm{T}})$ may be not the minimal cost over all query plans. However, this is not an issue as $PPC(P(C^{\mathrm{T}}), C^{\mathrm{T}})$ is identical to different CardEst methods, we could always compare their relative performance using P-Error no matter $P(C^{\mathrm{T}})$ is optimal or not. Meanwhile, we find that $P(C^{\mathrm{T}})$ is optimal in most cases. On our STATS benchmark, the query plan generated by the true cardinality is optimal on more than 98% queries using the default cost model of PostgreSQL.

Advantages of P-Error Metric: In Table 7, we report the P-Error distributions (50%, 90%, 99% percentiles) over the STATS-CEB workload of all CardEst methods and derive the following observation:

O14: P-Error is more highly correlated to the query execution time than Q-Error. We can roughly see that methods with better runtime tend to have smaller P-Error (e.g. FLAT has the best P-Error). We also compute the correlation coefficients between the query execution time and Q-Error/P-Error. On the STATS-CEB query workload, the value between 50% and 90% percentiles of Q-Error distribution w.r.t. query time is 0.036 and 0.037. Whereas, the value between 50% and 90% percentiles of P-Error distribution w.r.t. query time is 0.810 and 0.838. This indicates that P-Error is a better correspondence to the query execution time than Q-Error.

In addition, P-Error is more convenient as it outputs a single value on the plan cost level whereas Q-Error outputs a value for each sub-plan query of Q. Therefore, P-Error makes an attempt to overcome the limitations of Q-Error and is shown to be more suitable to measure the actual performance of CardEst methods.

8 DISCUSSIONS AND CONCLUSIONS

In this paper, we establish a new benchmark for CardEst, which contains the complex real-world dataset STATS and the diverse query workload STATS-CEB. This new benchmark helps to clearly identify the pros and cons of different CardEst methods. In addition, we propose the new metric P-Error as a potential substitute for the well-known Q-Error. Based on the exhaustive experimental analysis, we derive a series of important observations that will provide the DBMS community with a holistic view of the CardEst problem and help researchers design more effective and efficient CardEst methods. At last, we summarize the following key takeaway messages and future research opportunities:

• **Overall performance:** ML-based data-driven CardEst methods can achieve near-optimal performance, whereas the other methods barely have any improvement over PostgreSQL. Admittedly, the query-driven methods are more general because they can handle complex string "LIKE" queries.

• **Importance of different queries:** Accurate estimation of queries with large cardinalities is much important than the small ones. Therefore, researchers should develop CardEst methods that can produce accurate estimation for queries with large cardinalities instead of fine-grained estimation on extremely small ones.

• **Estimation of multi-table join queries:** The ML-based methods exhibit degrading performance for queries with an increasing number of join tables. Since learning one large data-driven model on the (sample of) full outer join of all tables has poor scalability, we believe an effective CardEst method should make appropriate independent assumptions and advocate researchers follow and improve the fanout methods first proposed by DeepDB [24].

• **Practicality aspects:** The inference latency of CardEst methods has a non-trivial impact on the end-to-end query time. Existing ML-based query-driven methods are inherently impractical for dynamic DBs with frequent data updates. Therefore, designing CardEst methods with fast inference speed and effective update algorithms is also very important.

• **Problems of Q-Error:** The well-recognized Q-Error metric does not reflect a CardEst method's end-to-end query performance. Alternatively, the newly proposed P-Error metric has better correspondence to the query performance and could serve as a better optimization objective for future researches.

REFERENCES

[1] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. 2001. STHoles: a multidimensional workload-aware histogram. In *SIGMOD*. 211–222.

[2] Walter Cai. 2021. Github repository: pqo-open source. *https://github.com/waltercai* (2021).

[3] Walter Cai, Magdalena Balazinska, and Dan Suciu. 2019. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In *SIGMOD*. 18–35.

[4] Surajit Chaudhuri, Vivek Narasayya, and Ravi Ramamurthy. 2009. Exact cardinality query optimization for optimizer testing. *Proceedings of the VLDB Endowment* 2, 1 (2009), 994–1005.

[5] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[6] C. Chow and Cong Liu. 1968. Approximating discrete probability distributions with dependence trees. *IEEE transactions on Information Theory* 14, 3 (1968), 462–467.

[7] Transaction Processing Performance Council(TPC). 2021. TPC-DS Vesion 2 and Version 3. *http://www.tpc.org/tpcds/* (2021).

[8] Transaction Processing Performance Council(TPC). 2021. TPC-H Vesion 2 and Version 3. *http://www.tpc.org/tpch/* (2021).

[9] Mattia Desana and Christoph Schnörr. 2020. Sum-product graphical models. *Machine Learning* 109, 1 (2020), 135–173.

[10] Amol Deshpande, Minos Garofalakis, and Rajeev Rastogi. 2001. Independence is good: Dependency-based histogram synopses for high-dimensional data. *ACM SIGMOD Record* 30, 2 (2001), 199–210.

[11] Postgresql Documentation 12. 2020. Chapter 70.1. Row Estimation Examples. *https://www.postgresql.org/docs/current/row-estimation-examples.html* (2020).

[12] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. *PVLDB* 12, 9 (2019), 1044–1057.

[13] Dennis Fuchs, Zhen He, and Byung Suk Lee. 2007. Compressed histograms with arbitrary bucket layouts for selectivity estimation. *Information Sciences* 177, 3 (2007), 680–702.

[14] Hector Garcia-Molina and Gio Wiederhold. 1982. Read-only transactions in a distributed database. *ACM Transactions on Database Systems (TODS)* 7, 2 (1982), 209–234.

[15] Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. 2015. MADE: Masked autoencoder for distribution estimation. *International Conference on Machine Learning* (2015), 881–889.

[16] Lise Getoor, Benjamin Taskar, and Daphne Koller. 2001. Selectivity estimation using probabilistic models. In *SIGMOD*. 461–472.

[17] Dimitrios Gunopulos, George Kollios, Vassilis J Tsotras, and Carlotta Domeniconi. 2000. Approximating multi-dimensional aggregate range queries over real attributes. In *SIGMOD*. 463–474.

[18] Dimitrios Gunopulos, George Kollios, Vassilis J Tsotras, and Carlotta Domeniconi. 2005. Selectivity estimators for multidimensional range queries over real attributes. *The VLDB Journal* 14, 2 (2005), 137–154.

[19] Max Halford, Philippe Saint-Pierre, and Franck Morvan. 2019. An approach based on bayesian networks for query selectivity estimation. *DASFAA* 2 (2019).

[20] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2019. Multi-attribute selectivity estimation using deep learning. In *SIGMOD*.

[21] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1035–1050.

[22] Max Heimel, Martin Kiefer, and Volker Markl. 2015. Self-tuning, gpu-accelerated kernel density models for multidimensional selectivity estimation. In *SIGMOD*. 1477–1492.

[23] Benjamin Hilprecht. 2019. Github repository: deepdb public. *https://github.com/DataManagementLab/deepdb-public* (2019).

[24] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2019. DeepDB: learn from data, not from queries!. In *PVLDB*.

[25] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017), 3146–3154.

[26] Andranik Khachatryan, Emmanuel Müller, Christian Stier, and Klemens Böhm. 2015. Improving accuracy and robustness of self-tuning histograms by subspace clustering. *IEEE TKDE* 27, 9 (2015), 2377–2389.

[27] Martin Kiefer, Max Heimel, Sebastian Breß, and Volker Markl. 2017. Estimating join selectivities using bandwidth-optimized kernel density models. *PVLDB* 10, 13 (2017), 2085–2096.

[28] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2019. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR*.

[29] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *PVLDB* 9, 3 (2015), 204–215.

[30] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *CIDR*.

[31] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander join: Online aggregation via random walks. In *SIGMOD*. 615–629.

[32] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2019. Wander join and XDB: online aggregation via random walks. *ACM Transactions on Database Systems* 44, 1 (2019), 1–41.

[33] Eric Liang, Zongheng Yang, Ion Stoica, Pieter Abbeel, Yan Duan, and Peter Chen. 2020. Variable Skipping for Autoregressive Range Density Estimation. In *ICML*. 6040–6049.

[34] Pedro Lopes, Craig Guyer, and Milener Gene. 2019. Sql docs: cardinality estimation (SQL Server). *https://docs.microsoft.com/en-us/sql/relational-databases/performance/cardinality-estimation-sql-server?view=sql-server-ver15* (2019).

[35] Frank Luan, Amog Kamsetty, Eric Liang, and Zongheng Yang. 2020. Github repository: neurocard project. *https://github.com/neurocard/neurocard* (2020).

[36] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed representations of words and phrases and their compositionality. *NIPS* (2013).

[37] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proceedings of the VLDB Endowment* 2, 1 (2009), 982–993.

[38] M Muralikrishna and David J DeWitt. 1988. Equi-depth multidimensional histograms. In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*. 28–36.

[39] Yoon-Min Nam Nam, Donghyoung Han Han, and Min-Soo Kim Kim. 2020. SPRINTER: a fast n-ary join query processing method for complex OLAP queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2055–2070.

[40] Parimarjan Negi. 2021. Github repository: pg_hint_plan. *https://github.com/parimarjan/pg_hint_plan* (2021).

[41] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *arXiv preprint arXiv:2101.04964* (2021).

[42] Parimarjan Negi, Ryan Marcus, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2020. Cost-guided cardinality estimation: Focus where it matters. In *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 154–157.

[43] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 237–252.

[44] Yeonsu Park, Seongyun Ko, Sourav S Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. 2020. G-CARE: A framework for performance benchmarking of cardinality estimation techniques for subgraph matching. In *SIGMOD*. 1099–1114.

[45] Pedro Pedreira, Yinghai Lu, Sergey Pershin, Amit Dutta, and Chris Croswhite. 2018. Rethinking concurrency control for in-memory OLAP dbmss. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1453–1464.

[46] Matthew Perron, Zeyuan Shang, Tim Kraska, and Michael Stonebraker. 2019. How I learned to stop worrying and love re-optimization. In *ICDE*. 1758–1761.

[47] Hoifung Poon and Pedro Domingos. 2011. Sum-product networks: A new deep architecture. In *ICCV Workshops*. 689–690.

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

[49] MySQL 8.0 Reference Manual. 2020. Chapter 15.8.10.2 Configuring Non-Persistent Optimizer Statistics Parameters. *https://dev.mysql.com/doc/refman/8.0/en/innodb-statistics-estimation.html* (2020).

[50] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *SIGMOD*. 23–34.

[51] MariaDB Server Documentation. 2020. Statistics for optimizing queries: InnoDB persistent statistics. *https://mariadb.com/kb/en/innodb-persistent-statistics/* (2020).

[52] Suraj Shetiya, Saravanan Thirumuruganathan, Nick Koudas, and Gautam Das. 2020. Astrid: accurate selectivity estimation for string predicates using deep learning. *Proceedings of the VLDB Endowment* 14, 4 (2020), 471–484.

[53] Utkarsh Srivastava, Peter J Haas, Volker Markl, Marcel Kutsch, and Tam Minh Tran. 2006. Isomer: Consistent histogram construction using query feedback. In *ICDE*. 39–39.

[54] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO-DB2's learning optimizer. In *PVLDB*, Vol. 1. 19–28.

[55] Ji Sun and Guoliang Li. 2019. An end-to-end learning-based cost estimator. *VLDB* (2019).

[56] Vladimir Svetnik, Andy Liaw, Christopher Tong, J Christopher Culberson, Robert P Sheridan, and Bradley P Feuston. 2003. Random forest: a classification and regression tool for compound classification and QSAR modeling. *Journal of chemical information and computer sciences* 43, 6 (2003), 1947–1958.

[57] Immanuel Trummer. 2019. Exact cardinality query optimization with bounded execution cost. In *Proceedings of the 2019 International Conference on Management of Data*. 2–17.

[58] Kostas Tzoumas, Amol Deshpande, and Christian S Jensen. 2011. Lightweight graphical models for selectivity estimation without independence assumptions. *PVLDB* 4, 11 (2011), 852–863.

[59] Hai Wang and Kenneth C Sevcik. 2003. A multi-dimensional histogram for selectivity estimation and fast approximate query answering. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*. 328–342.

[60] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready For Learned Cardinality Estimation? *VLDB* 14, 9 (2021), 1640–1654.

[61] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a learning optimizer for shared clouds. *PVLDB* 12, 3 (2018), 210–222.

[62] Peizhi Wu. 2021. Github repository: UAE/UEA-Q. *https://github.com/pagegitss/UAE* (2021).

[63] Peizhi Wu and Gao Cong. 2021. A Unified Deep Model of Learning from both Data and Queries for Cardinality Estimation. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*.

[64] Ziniu Wu. 2021. Github repository: BayesCard. *https://github.com/wuziniu/BayesCard* (2021).

[65] Ziniu Wu. 2021. Github repository: FSPN. *https://github.com/wuziniu/FSPN* (2021).

[66] Ziniu Wu and Amir Shaikhha. 2020. BayesCard: A Unified Bayesian Framework for Cardinality Estimation. *arXiv preprint arXiv:2012.14743* (2020).

[67] Ziniu Wu, Peilun Yang, Pei Yu, Rong Zhu, Yuxing Han, Yaliang Li, Defu Lian, Kai Zeng, and Jingren Zhou. 2021. A Unified Transferable Model for ML-Enhanced DBMS. *arXiv preprint arXiv:2105.02418* (2021).

[68] Ziniu Wu, Rong Zhu, Andreas Pfadler, Yuxing Han, Jiangneng Li, Zhengping Qian, Kai Zeng, and Jingren Zhou. 2020. FSPN: A New Class of Probabilistic Graphical Model. *arXiv preprint arXiv:2011.09020* (2020).

[69] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2021. NeuroCard: One Cardinality Estimator for All Tables. *PVLDB* 14, 1 (2021), 61–73.

[70] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep unsupervised cardinality estimation. *PVLDB* (2019).

[71] Zongheng Yang and Chenggang Wu. 2019. Github repository: naru project. *https://github.com/naru-project/naru* (2019).

[72] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random sampling over joins revisited. In *SIGMOD*. 1525–1539.

[73] Zhuoyue Zhao, Feifei Li, and Yuxi Liu. 2020. Efficient join synopsis maintenance for data warehouse. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2027–2042.

[74] Zhuoyue Zhao, Bin Wu, Feifei Li, and Ke Yi. 2021. Github repository: InitialDLLab/XDB. *https://github.com/InitialDLab/XDB* (2021).

[75] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2021. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation. *VLDB* 14, 9 (2021), 1489–1502.