FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation

Rong Zhu[1,#], Ziniu Wu[1,#], Yuxing Han[1], Kai Zeng[1,*], Andreas Pfadler[1], Zhengping Qian[1], Jingren Zhou[1], Bin Cui[2]

[1]*Alibaba Group,* [2]*Peking University*

[1]{red.zr, ziniu.wzn, yuxing.hyx, zengkai.zk, andreaswernerrober, zhengping.qzp, jingren.zhou}@alibaba-inc.com [2]bin.cui@pku.edu.cn

ABSTRACT

Query optimizers rely on accurate cardinality estimation (CardEst) to produce good execution plans. The core problem of CardEst is how to model the rich joint distribution of attributes in an accurate and compact manner. Despite decades of research, existing methods either over-simplify the models only using independent factorization which leads to inaccurate estimates, or over-complicate them by lossless conditional factorization without any independent assumption which results in slow probability computation. In this paper, we propose FLAT, a CardEst method that is simultaneously *fast* in probability computation, *lightweight* in model size and *accurate* in estimation quality. The key idea of FLAT is a novel unsupervised graphical model, called FSPN. It utilizes both independent and conditional factorization to adaptively model different levels of attributes correlations, and thus combines their advantages. FLAT supports efficient online probability computation in near linear time on the underlying FSPN model, provides effective offline model construction and enables incremental model updates. It can estimate cardinality for both single table queries and multi-table join queries. Extensive experimental study demonstrates the superiority of FLAT over existing CardEst methods: FLAT achieves 1–5 orders of magnitude better accuracy, 1–3 orders of magnitude faster probability computation speed and 1–2 orders of magnitude lower storage cost. We also integrate FLAT into Postgres to perform an end-to-end test. It improves the query execution time by 12.9% on the well-known IMDB benchmark workload, which is very close to the optimal result 14.2% using the true cardinality.

PVLDB Reference Format:
Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Bin Cui. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation. PVLDB, 14(9): 1489 - 1502, 2021.
doi:10.14778/3461535.3461539

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

Proceedings of the VLDB Endowment, Vol. 14, No. 9 ISSN 2150-8097.
doi:10.14778/3461535.3461539

1 INTRODUCTION

Cardinality estimation (CardEst) is a key component of query optimizers in modern database management systems (DBMS) and analytic engines [1, 53]. Its purpose is to estimate the result size of a SQL query before its actual execution, thus playing a central role in generating high-quality query plans.

Given a table T and a query Q, estimating the cardinality of Q is equivalent to computing P—the probability of records in T satisfying Q. Therefore, the core task of CardEst is to condense T into a model M to compute P. In general, such models could be obtained in two ways: *query-driven* and *data-driven*. Query-driven approaches learn functions mapping a query Q to its predicted probability P, so they require large amounts of executed queries as training samples. They only perform well if future queries follow the same distribution as the training workload. Data-driven approaches learn unsupervised models of $\Pr(T)$—the joint probability density function (PDF) of attributes in T. As they can generalize to unseen query workload, data-driven approaches receive more attention and are widely used for CardEst.

Challenge and Status of CardEst. An effectual CardEst method should satisfy three criteria [13, 21, 57, 60], namely high estimation accuracy, fast inference time and lightweight storage overhead, at the same time. Existing methods have made some efforts in finding trade-offs between the them. However, they still suffer from one or more deficiencies when modeling real-world complex data.

In a nutshell, there exist three major strategies for building unsupervised models of $\Pr(T)$ on data table T. The first strategy directly compresses and stores all entries in $\Pr(T)$ [15, 46], whose storage overhead is intractable and the lossy compression may significantly impact estimation accuracy. The second strategy utilizes sampling [29, 65] or kernel density based methods [18, 23], where samples from T are fetched on-the-fly to estimate probabilities. For high-dimensional data, they may be either inaccurate without enough samples or inefficient due to a large sample size.

The third strategy, factorization based methods, is to decompose $\Pr(T)$ into multiple low-dimensional PDFs $\Pr(T')$ such that their suitable combination can approximate $\Pr(T)$. However, existing methods often fail to balance the three criteria. Some methods, including deep auto-regression [17, 62, 63] and Bayesian Network [12, 57], can losslessly decompose $\Pr(T)$ using *conditional factorization*. However, their probability computation speed is reduced drastically. Other methods, such as 1-D histogram [51] and sum-product network [20], assume global or local *independence* between attributes to decompose $\Pr(T)$. They attain high computation

efficiency but their estimation accuracy is low when the independence assumption does not hold. We present a detailed analysis of existing data-driven CardEst methods in Section 2.

Our Contributions. In this paper, we address the CardEst problem more comprehensively in order to satisfy all three criteria. We absorb the advantages of existing models and design a novel graphical model, called factorize-sum-split-product network (FSPN). Its key idea to *adaptively* decompose $\Pr(T)$ according to the dependence level of attributes. Specifically, the joint PDF of highly and weakly correlated attributes will be losslessly separated by conditional factorization and modeled accordingly. The joint PDF of highly correlated attributes can be easily modeled as a multivariate PDF. For the weakly correlated attributes, their joint PDF is split into multiple small regions where attributes are mutually independent in each. We prove that FSPN subsumes 1-D histogram, sum-product network and Bayesian network, and leverages their advantages.

Based on the FSPN model, we propose a CardEst method called FLAT, which is *fast*, *lightweight* and *accurate*. On a single table, FLAT applies an effective offline method for the structure construction of FSPN and an efficient online probability computation method using the FSPN. The probability computation complexity of FLAT is almost linear w.r.t. the number of nodes in FSPN. Moreover, FLAT enables fast incremental updates of the FSPN model.

For multi-table join queries, FLAT uses a new framework, which is more general and applicable than existing work [17, 20, 24, 63]. In the offline phase, FLAT clusters tables into several groups and builds an FSPN for each group. In the online phase, FLAT combines the probabilities of sub-queries in a fast way to get the final result.

In our evaluation, FLAT achieves state-of-the-art performance on both single table and multi-table cases in comparison with all existing methods [20, 23, 24, 29, 46, 57, 62, 64]. On single table, FLAT achieves up to 1–5 orders of magnitude better accuracy, 1–3 orders of magnitude faster probability computation speed (near $0.2ms$) and 1–2 orders of magnitude lower storage cost (only tens of KB). On the *JOB-light* benchmark [28, 30] and a more complex crafted multi-table workload, FLAT also attains the highest accuracy and an order of magnitude faster computation time (near $5ms$), while requiring only 3.3MB storage space. We also integrate FLAT into Postgres. It improves the average end-to-end query time by 12.9% on the benchmark workload, which is very close to the optimal result 14.2% using the true cardinality. This result confirms with a positive answer to the long-existing question whether and how much a more accurate CardEst can improve the query plan quality [44]. In addition, we have deployed FLAT in the production environment.

In summary, our main contributions are listed as follows:

1) We analyze in detail the status of existing data-driven CardEst methods in terms of the above three criteria (in Section 2).

2) We present FSPN, a novel unsupervised graphical model, which combines the advantages of existing methods in an adaptive manner (in Section 3).

3) We propose FLAT, a CardEst method with fast probability computation, high estimation accuracy and low storage cost, on both single table and multi-table join queries (in Section 4 and 5).

4) We conduct extensive experiments and end-to-end test on Postgres to demonstrate the superiority and practicality of our proposed methods (in Section 6).

2 PROBLEM DEFINITION AND BACKGROUND

In this section, we formally define the CardEst problem and analyze the status of data-driven CardEst methods. Based on the analysis, we summarize some key findings that inspire our work.

CardEst Problem. Let T be a table with a set of k attributes $A = \{A_1, A_2, \ldots, A_k\}$. T could either be a single or a joined table. Each attribute A_i in T is assumed to be either categorical, so that values can be mapped to integers, or continuous. Without loss of generality, we assume that the domain of A_i is $[LB_i, UB_i]$.

In this paper, we do not consider "LIKE" queries on strings. Any selection query Q on T may be represented in canonical form: $Q = (A_1 \in [L_1, U_1] \wedge A_2 \in [L_2, U_2] \wedge \cdots \wedge A_k \in [L_k, U_k])$, where $LB_i \le L_i \le U_i \le UB_i$ for all i. W.l.o.g., the endpoints of each interval can also be open. We call Q a *point query* if $L_i = U_i$ for all i and *range query* otherwise. If Q has no constraint on one side, we can set $L_i = LB_i$ or $U_i = UB_i$. For any query Q' where the constraint of an attribute A_i contains several intervals, we may split Q' into multiple queries satisfying the above form.

Let $\mathrm{Card}(T, Q)$ denote the number of records in T satisfying Q. The CardEst problem asks to accurately estimate $\mathrm{Card}(T, Q)$ without executing Q on T. CardEst is often modeled and solved from a statistical perspective. We can regard each attribute A_i in T as a random variable. The table T essentially represents a set of i.i.d. records sampled from the joint PDF $\Pr_T(A) = \Pr_T(A_1, A_2, \ldots, A_k)$. For any query Q, let $\Pr_T(Q)$ denote the probability of records in T satisfying Q. We have $\mathrm{Card}(T, Q) = \Pr_T(Q) \cdot |T|$. Therefore, estimating $\mathrm{Card}(T, Q)$ is equivalent to estimating the probability $\Pr_T(Q)$. Unsupervised CardEst solves this problem in a purely data-driven fashion, which can be formally stated as follows:

Offline Training: Given a table T with a set A of attributes as input, output a model $\widehat{\Pr}_T(A)$ for $\Pr_T(A)$ such that $\widehat{\Pr}_T(A) \approx \Pr_T(A)$.

Online Probability Computation: Given the model $\widehat{\Pr}_T(A)$ and a query Q as input, output $\widehat{\Pr}_T(Q) \cdot |T|$ as the estimated cardinality.

Data-Driven CardEst Methods Analysis. We use three criteria, namely *model accuracy*, *probability computation speed* and *storage overhead*, to analyze existing methods. The results are as follows:

1) Lossy FullStore [15] stores all entries in $\Pr_T(A)$ using compression techniques, whose storage grows exponentially in the number of attributes and becomes intractable [62, 63].

2) Sample and Kernel-based methods [18, 23, 29, 65] do not store $\Pr_T(A)$ but rather sample records from T on-the-fly, or use average kernels centered around sampled points to estimate $\Pr_T(Q)$. For high-dimensional data, they may be either inaccurate without enough samples, or inefficient due to a large sample size.

Alternatively, a more promising way is to *factorize* $\Pr_T(A)$ into multiple low-dimensional PDFs $\Pr_T(A')$ such that: 1) $|A'| << |A|$ so $\Pr_T(A')$ is easier to store and model; and 2) a suitable combination, e.g. multiplication, weighted sum and etc, of $\Pr_T(A')$ approximates $\Pr_T(A)$. Some representative methods are listed in the following:

3) 1-D Histogram [51] builds a (cumulative) histogram on each attribute and assumes all attributes are mutually independent, so $\widehat{\Pr}_T(Q)$ may be obtained in $O(|A|)$ time. However, the estimation errors may be high, since correlations between attributes are ignored.

4) M-D Histogram [7, 14, 46, 59] builds multi-dimensional histograms to model the dependency of attributes. They identify subsets of correlated attributes using models such as Markov network,

build histograms on each subset and assume the independence across different subsets. It improves the accuracy but the decomposition is still lossy. Meanwhile, it is space consuming.

5) Deep Auto-Regression (DAR) [17, 62, 63] decomposes the joint PDF according to the chain rule, i.e., $\Pr_T(A) = \Pr_T(A_1) \cdot \prod_{i=2}^{k} \Pr_T(A_i | A_1, A_2, \ldots, A_{i-1})$. Each conditional PDF can be parametrically modeled by a deep neural network (DNN). While the expressiveness of DNNs allows $\Pr_T(A)$ to be approximated well, probability computation time and space cost increase with the width and depth of the DNN. Moreover, for range query Q, computing $\Pr_T(Q)$ requires averaging the probabilities of lots of sample points in the range. Thus, the probability computation on DAR is relatively slow.

6) Bayesian Network (BN) [4, 12, 57] models the dependence structure between all attributes as a directed acyclic graph and assumes that each attribute is conditionally independent of the remaining attributes given its parents. The probability $\Pr_T(A)$ is factorized as $\Pr_T(A) = \prod_{i=1}^{k} \Pr_T(A_i | A_{\text{pa}(i)})$, where $\text{pa}(i)$ is the parent attributes of A_i in BN. Learning the BN structure from data and probability computation on BN are both NP-hard [5, 50].

7) Sum-Product Network (SPN) [20] approximates $\Pr_T(A)$ using several local and simple PDFs. An SPN is tree structure where each node stands for an estimated PDF $\widehat{\Pr}_{T'}(A')$ of the attribute subset A' on record subset $T' \subseteq T$ [45]. The root node represents $\widehat{\Pr}_T(A)$. Each inner node is: 1) a sum node which splits all records (rows) in T' into T'_i on each child such that $\widehat{\Pr}_{T'}(A') = \sum_i w_i \widehat{\Pr}_{T'_i}(A')$ with weights w_i; or 2) a product node which splits attributes (columns) in A' on each child as $\widehat{\Pr}_{T'}(A') = \prod_j \widehat{\Pr}_{T'}(A'_j)$ when all A'_j are mutually independent in T'. Each leaf node then maintains a (cumulative) PDF on a singleton attribute. The probability $\widehat{\Pr}_T(Q)$ can be computed in a bottom-up manner using the SPN node operations for both point and range queries. The storage overhead and probability computation cost are linear in the number of nodes of SPN.

The performance of SPN heavily relies on the local independence assumption. When it holds, the generated SPN is compact and exhibits superiority over other methods [20, 62]. However, real-world data often possesses substantial skew and strong correlations between attributes [57]. In this situation, SPN can not split these attributes using the product operation and might repeatedly apply the sum operation to split records into extremely small volumes [39], i.e., $|T'| = 1$. This would heavily increase the SPN size, degrade its efficiency and make the model inaccurate [6, 39].

Inspirations. We find that independent factorization has low storage cost and supports fast inference but may incur huge estimation errors; conditional factorization can accurately decompose the PDF but the inference is costly. To this end, we propose a new unsupervised model, called factorize-split-sum-product network (FSPN), which applies both kinds of factorization methods in an adaptive manner. This design choice makes it possible to obtain a CardEst method that can simultaneously satisfy all three criteria.

3 THE FSPN MODEL

In this section, we present FSPN, a new tree-structured graphical model. We first explain the key ideas of FSPN with an example and then present its formal definition. Finally, we compare FSPN with aforementioned models.

Key Ideas of FSPN. FSPN can factorize attributes with different dependence levels accordingly. The conditional factorization approach is used to split highly and weakly correlated attributes. Then, highly correlated attributes are directly modeled together while weakly correlated attributes are recursively approximated using the independent factorization approach. Figure 1(a) gives an example of table T with a set A of four attributes *water turbidity* (A_1), *temperature* (A_2), *wave height* (A_3) and *wind force* (A_4). We elaborate the process to construct its FSPN in Figure 1(b) as follows:

At first, we examine the pairwise correlations and find that A_3 and A_4 are globally highly correlated, so they can not be easily decomposed as independent attributes. Instead, we can losslessly separate them from other attributes as early as possible and process each part respectively. Let $H = \{A_3, A_4\}$ and $W = \{A_1, A_2\}$. We apply the conditional factorization approach and factorize $\Pr_T(A) = \Pr_T(W) \cdot \Pr_T(H|W)$ (as node N_1 in step ①). $\Pr_T(W)$ and $\Pr_T(H|W)$ are then modeled in different ways.

The two attributes A_1 and A_2 are weakly correlated on T. If we split all records in T into T_1 and T_2 based on whether A_1 is less than 50 (as node N_2 in step ②), A_1 and A_2 are independent on both T_1 and T_2. This situation is called *contextually independent*, where T_1 and T_2 refer to the specific context. Since $\Pr_{T_1}(W) = \Pr_{T_1}(A_1) \cdot \Pr_{T_1}(A_2)$ (as node N_4 in step ③), we then simply use two univariate PDFs (such as histograms in leaf nodes L_1 and L_2 in step ③) to model $\Pr_{T_1}(A_1)$ and $\Pr_{T_1}(A_2)$ on T_1, respectively. Similarly, we also model $\Pr_{T_2}(W) = \Pr_{T_2}(A_1) \cdot \Pr_{T_2}(A_2)$ on T_2 (as node N_5).

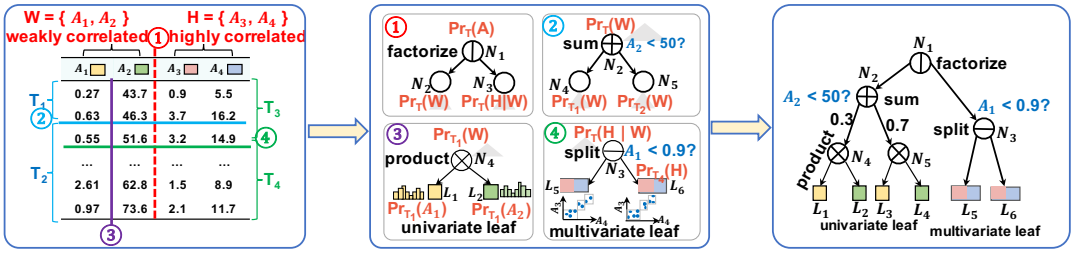
For the conditional PDF $\Pr_T(H|W)$, we do not need to specify $\Pr(H|w)$ for each value w of W. Instead, we can recursively split T into multiple regions T_j in terms of W such that H is independent of W in each context T_j, i.e., $\Pr_{T_j}(H) = \Pr_{T_j}(H|W)$. At this time, for any value w of W falling in the same region, $\Pr_{T_j}(H|w)$ stays the same, so we only need to maintain $\Pr_{T_j}(H)$ for each region. We refer to this situation as *contextual condition removal*. In our example, we split T into T_3 and T_4 (as nodes N_3 in step ④) by whether the condition attribute A_1 is less than 0.9. W is independent of H on each leaf node region, so we only need to model $\Pr_{T_3}(H)$ and $\Pr_{T_4}(H)$. Thus, we model them as two multivariate leaf nodes L_5 and L_6 in step ④. Note that, attribute values in H are interdependent and their joint PDFs $\Pr_{T_3}(H)$ and $\Pr_{T_4}(H)$ are sparse in the two-dimensional space, so they are easy modeled as a multivariate PDF.

Finally, we obtain an FSPN in Figure 1(c) containing 11 nodes, where 5 inner nodes represent different operations to split data and 6 leaf nodes keep PDFs for different parts of the original data.

Formulation of FSPN. Let $\mathcal{F}$ denote a FSPN modeling the joint PDF $\Pr_T(A)$ for records T with attributes A. $\mathcal{F}$ is a tree structure. Each node N in $\mathcal{F}$ is a 4-tuple (A_N, C_N, T_N, O_N) where:

- $T_N \subseteq T$ represents a set of records where the PDF is built on. It is called the *context* of node N.
- $A_N, C_N \subseteq A$ represent two sets of attributes. We call A_N and C_N the *scope* and *condition* of node N, respectively. If $C_N = \emptyset$, N represents the PDF $\Pr_{T_N}(A_N)$; otherwise, it represents the conditional PDF $\Pr_{T_N}(A_N|C_N)$. The root of $\mathcal{F}$, such as N_1 in Figure 1(c), represents the joint PDF $\Pr_T(A)$ with $A_N = A$, $C_N = \emptyset$ and $T_N = T$.
- O_N stands for the *operation* specifying how to split data to generate its children in different ways:

1) A Factorize (⫿) node, such as N_1 in step ①, splits highly



(a) Ocean Observation Data Table (b) Detailed Steps to Construct the FSPN Structure (c) The FSPN Structure

Figure 1: An ocean observation data table and its corresponding FSPN.

correlated attributes from the remaining ones by conditional factorization only when $C_N = \emptyset$. Let $H \subseteq A_N$ be a subset of highly correlated attributes. It generates the left child $N_L = (A_N - H, \emptyset, T_N, O_L)$ and the right child $N_R = (H, A_N - H, T_N, O_R)$. We have $\Pr_{T_N}(A_N) = \Pr_{T_N}(A_N - H) \cdot \Pr_{T_N}(H|A_N - H)$.

2) A Sum ($\oplus$) node, such as N_2 in step ②, splits the records in T_N in order to enforce contextual independence only when $C_N = \emptyset$. We partition T_N into subsets $T_1, T_2, \ldots, T_n$ and generate each child $N_i = (A_N, \emptyset, T_i, O_i)$ with weight $w_i = |T_i|/|T_N|$ for N. We can regard N as a mixture of models, i.e., $\Pr_{T_N}(A_N) = \sum_{i=1}^{n} w_i \Pr_{T_i}(A_N)$.

3) A Product ($\otimes$) node, such as N_4 in step ③, splits the scope A_N of N only when $C_N = \emptyset$ and contextual independence holds. Let $A_1, A_2, \ldots, A_m$ be the mutually independent partitions of A_N. N generates children $N_j = (A_j, \emptyset, T_N, O_j)$ for all $1 \le j \le m$ such that $\Pr_{T_N}(A_N) = \prod_{j=1}^{m} \Pr_{T_N}(A_j)$.

4) A Split ($\ominus$) node, such as N_3 in step ④, partitions the records T_N into disjoint subsets $T_1, T_2, \ldots, T_d$ only when $C_N \neq \emptyset$. For each $1 \le i \le d$, N generates the child $N_i = (A_N, C_N, T_i, O_i)$. Note that for any value c of C_N, there exists exactly one j such that c falls in the region of T_j. The semantic of split is different from sum. The split node divides a large model of $\Pr_{T_N}(A_N|C_N)$ into several parts by the values of C_N. Whereas, the sum node decomposes a large model of $\Pr_{T_N}(A_N)$ to small models on the space of A_N.

5) A Uni-leaf ($\square$) node, such as L_1 and L_2 in step ③, keeps the univariate PDF $\Pr_{T_N}(A_N)$, such as histogram or Gaussian mixture model, only when $|A_N| = 1$ and $C_N = \emptyset$.

6) A Multi-leaf ($\square\square$) node, such as L_5 and L_6 in step ④, maintains the multivariate PDF $\Pr_{T_N}(A_N)$ only when $C_N \neq \emptyset$ and A_N is independent of C_N on T_N.

The above operations are recursively used to construct $\mathcal{F}$ with three constraints: 1) for a factorize node, the right child must be a split node or multi-leaf; the left child can be any type in sum, product, factorize and uni-leaf; 2) the children of a sum or product node could be any type in sum, product, factorize and uni-leaf; and 3) the children of a split node can only be split or multi-leaf nodes.

Differences with SPN. As the name suggests, FSPN is inspired by SPN and its successful application in CardEst [20]. However, FSPN differs from SPN in two fundamental aspects. First, in terms of the underlying key ideas, FSPN tries to adaptively model attributes with different levels of dependency, which is not considered in SPN. Second, in terms of the fundamental design choices, FSPN can split weakly and highly correlated attributes, and models each class differently: 1) weakly correlated attributes are modeled by sum and product operations; and 2) for highly correlated attributes, FSPN uses split and multi-leaf nodes. SPN only uses the first technique on all attributes. As per our analysis in Section 2, this can generate a large structure since local independence can not easily hold.

Moreover, a simple extension of SPN with multi-leaf nodes also seems unlikely to mitigate its inherent limitations. This is because multi-leaf nodes can only efficiently model highly correlated attributes, as their joint PDF can be easily reduced to and modeled in a low dimensional space. Otherwise, their storage cost grows exponentially so the model size would be very large. FSPN guarantees that multi-leaf nodes are only applied on highly correlated attributes, whereas SPN and its extensions lack such mechanism. Our experimental results in Section 6.1 exhibit that the model size of SPN with multi-leaf nodes are much larger than FSPN and may exceed the memory limit on highly correlated table.

Generality of FSPN. We show that FSPN *generalizes* 1-D Histogram, SPN and BN models. First, when all attributes are mutually independent, FSPN becomes 1-D Histogram. Second, FSPN degenerates to SPN by disabling the factorize operation. Third, FSPN could equally represent a BN model on discrete attributes by iteratively factorizing each attribute having no parents from others. We put the transformation process in Appendix A.1. Based on it, we obtain Lemma 1 (proved in Appendix A.2) stating that the FSPN is no worse than SPN and BN in terms of expressive efficiency.

Lemma 1 *Given a table T with attributes A, if the joint PDF $\Pr_T(A)$ is represented by an SPN $\mathcal{S}$ or a BN $\mathcal{B}$ with space cost $O(M)$, then there exists an FSPN $\mathcal{F}$ that can equivalently model $\Pr_T(A)$ with no more than $O(M)$ space.*

4 SINGLE TABLE CardEst METHOD

In this section, we propose FLAT, a fast, lightweight and accurate CardEst algorithm built on FSPN. We first introduce how FLAT computes the probability on FSPN online in Section 4.1. Then, we show how FLAT constructs the FSPN from data offline in Section 4.2. Finally, we discuss how FLAT updates the model in Section 4.3.

4.1 Online Probability Computation

FLAT can obtain the probability (cardinality) of any query Q in a recursive manner on FSPN. We first show the basic strategy of probability computation with an example, and then present the detailed algorithm and analyze its complexity.

Basic Strategy. As stated in Section 2, the query Q can be represented in canonical form: $Q = (A_1 \in [L_1, U_1] \wedge A_2 \in [L_2, U_2] \wedge \cdots \wedge$

$A_k \in [L_k, U_k]$), where $L_i \le A_i \le U_i$ is the constraint on attribute A_i. Obviously, Q represents a *hyper-rectangle* range in the attribute space whose probability needs to be computed. In Figure 2, we give an example query Q on the FSPN in Figure 1(c).

First, considering the root node N_1, computing the probability of Q on this factorize node is a non-trivial task. For each point $r \in Q$, we can obtain its probability $\Pr_r(A_1, A_2)$ from node N_2 and the conditional probability $\Pr_r(A_3, A_4|A_1, A_2)$ from node N_3. However, for different r, $\Pr_r(A_3, A_4|A_1, A_2)$ is modeled by different PDFs on multi-leaf nodes L_5 or L_6 of N_3. Thus, we must split Q into two regions to compute the probability of Q (as step ① in Figure 2). To this end, we push Q onto N_3, whose splitting rule on the condition attributes ($A_1 < 0.9$) would divide Q into two hyper-rectangle ranges Q_1 and Q_2 on multi-leaf nodes L_5 or L_6, respectively. For Q_1 (or Q_2), the probability $\Pr(A_3, A_4|A_1, A_2) = \Pr(A_3, A_4)$ can be directly obtained from the multivariate PDF on L_5 (or L_6).

Then, we can compute the probability $\Pr(A_1, A_2)$ for each region Q_1 and Q_2 from N_2. Obviously, for the sum node (e.g. N_2) and product node (e.g. N_4), the probability of each region can be recursively obtained by summing (as step ③) or multiplying (as step ②) the probability values of its children, respectively. In the base case, the probability on the singleton attribute A_1 (or A_2) is obtained from the uni-leaf nodes L_1 and L_3 (or L_2 and L_4). Finally, since $\Pr(A_1, A_2)$ and $\Pr(A_3, A_4)$ are independent in Q_1 and Q_2, we can multiply and sum them together ((as step ④)) to obtain the probability of Q.

Algorithm Description. Next, we describe the online probability computation algorithm FLAT-Online. It takes as inputs a FSPN $\mathcal{F}$ modeling $\Pr_T(A)$ and the query Q, and outputs $\Pr_T(Q)$ on $\mathcal{F}$. Let N be the root node of $\mathcal{F}$. For any node N' in $\mathcal{F}$, let $\mathcal{F}_{N'}$ denote the FSPN rooted at N'. FLAT-Online recursively computes the probability of Q by the following rules:

Rule 1 (lines 2–3): Basically, if N is a uni-leaf node, we directly return the probability of Q on the univariate PDF of the attribute.

Rule 2 (lines 4–11): if N is a sum node or a product node, let $N_1, N_2, \ldots, N_t$ be all of its children. We can further call FLAT-Online on each child to obtain the probability on the PDF represented by each child. Then, node N computes a weighted sum (for sum node) or multiplication (for product node) of these probabilities.

Rule 3 (lines 12–18): if N is a factorize node, let LC and RC be its left and right child modeling $\Pr_T(W)$ and $\Pr_T(H|W)$, respectively. All descendants of RC are split or multi-leaf nodes. Let $L_1, L_2, \ldots, L_t$ be all multi-leaf descendants of RC. We assume that each split node divides the attribute domain space in a grid manner, which is ensured by the FSPN structure construction method in Section 4.2. Then, each L_i maintains a multivariate PDF on a hyper-rectangle range specified by all split nodes on the path from RC to L_i. Based on these ranges, we can divide the range of query Q into $Q_1, Q_2, \ldots, Q_t$. For each Q_i, the probability h_i on highly correlated attributes H could be directly obtained from L_i. The probability w_i on attributes W could be recursively obtained by calling FLAT-Online on $\mathcal{F}_{LC}$, the FSPN rooted at LC, and Q_i. After that, since H is independent of W on the range of each Q_i, we sum all products $h_i w_i$ together as the probability of Q.

Complexity Analysis. We assume that, on each leaf node, the probability of any range can be computed in $O(1)$ time, which can be easily implemented by a cumulative histogram or Gaussian mixture functions. Let n be the number of nodes in FSPN. Let f and m be the number of factorize and multi-leaf nodes in FSPN, respectively. The maximum number of ranges to be computed on each node is $O(m^f)$, so the time cost of FLAT-Online is $O(m^f n)$.

By our empirical testing, the actual time cost of FLAT-Online is almost linear w.r.t. the number of nodes in FSPN for two reasons. First, FSPN is compact on real-world data so both f and n are small. Second, the computation on many ranges in each node could be easily done in parallel. In our testing, the speed of FLAT-Online is even near the histogram method and 1–3 orders of magnitude faster than other methods (See Section 6.1).

Range	A_1	A_2	A_3	A_4
Bound	[0, 10]	[0, 100]	[0, 100]	[0, 100]
Leaf L_5	[0, 0.9)	[0, 100]	[0, 100]	[0, 100]
Leaf L_6	[0.9, 10]	[0, 100]	[0, 100]	[0, 100
Query Q	[0.6, 1.4]	[35, 65]	[2, 3]	[60, 70]
Query Q_1	[0.6, 0.9)	[35, 65]	[2, 3]	[60, 70]
Query Q_2	[0.9, 1.4]	[35, 65]	[2, 3]	[60, 70]

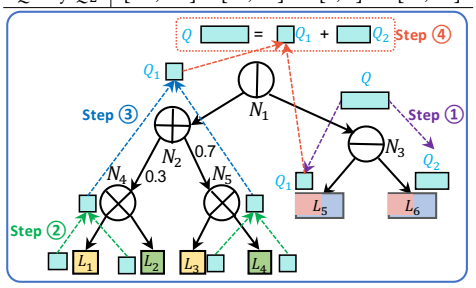


Figure 2: An example of the FLAT probability computation.

Algorithm FLAT-Online($\mathcal{F}$, Q)

1: let N be the root node of $\mathcal{F}$
2: **if** N is uni-leaf node **then**
3: **return** $\Pr_T(Q)$ by the univariate PDF on the attribute modeled by N
4: **else if** N is a sum node **then**
5: let $N_1, N_2, \ldots, N_t$ be the children of N with weights $w_1, w_2, \ldots, w_t$
6: $p_i \leftarrow$ FLAT-Online($\mathcal{F}_{N_i}$, Q) for each $1 \le i \le t$
7: **return** $\sum_{i=1}^{t} w_i p_i$
8: **else if** N is a product node **then**
9: let $N_1, N_2, \ldots, N_t$ be the children of N
10: $p_i \leftarrow$ FLAT-Online($\mathcal{F}_{N_i}$, Q) for each $1 \le i \le t$
11: **return** $\prod_{i=1}^{t} p_i$
12: **else**
13: let LC be the left child modeling $\Pr_T(W)$ and RC be the right child modeling $\Pr_T(H|W)$
14: let $L_1, L_2, \ldots, L_t$ be all the multi-leaf descendants of RC
15: split Q into $Q_1, Q_2, \ldots, Q_t$ by ranges of $L_1, L_2, \ldots, L_t$
16: get h_i of Q_i on variables H from the multivariate PDF on L_i for each $1 \le i \le t$
17: $w_i \leftarrow$ FLAT-Online($\mathcal{F}_{LC}$, Q_i) for each $1 \le i \le t$
18: **return** $\sum_{i=1}^{t} h_i w_i$

4.2 Offline Structure Construction

We show the general process to build an FSPN in Figure 3 and put the pseudocode of the algorithm FLAT-Offline in Appendix B.1 [66]. FLAT-Offline works in a top-down manner. Each node N takes the scope attributes A_N, the condition attributes C_N and the context of records T_N as inputs, and recursively decompose the joint PDF to build the FSPN rooted at N. To build the FSPN $\mathcal{F}$ modeling table T with attributes A, we can directly call FLAT-Offline($A, \emptyset, T$). We briefly scan its main procedures as follows:

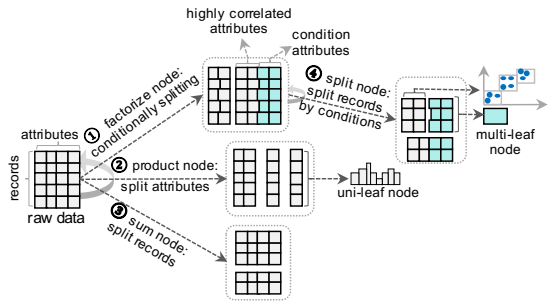


Figure 3: FLAT Structure Construction Process.

1. Separating highly correlated attributes with others: when $C_N = \emptyset$, FLAT-Offline firstly detects if there exists a set H of highly correlated attributes since the principle of FSPN is to separate them with others as early as possible (step ① in Figure 3). We find H by examining pairwise correlations, e.g. RDC [35], between attributes and iteratively group attributes whose correlation value is larger than a threshold τ_h. If $H \neq \emptyset$, we set N to be a factorize node. The left child and right child of N recursively call FLAT-Offline to model $\Pr_{T_N}(A_N - H)$ and $\Pr_{T_N}(H|A_N - H)$, respectively.

2. Modeling weakly correlated attributes: if $C_N = \emptyset$ and $H = \emptyset$, we try to split $\Pr_{T_N}(A_N)$ into small regions such that attributes in A_N are locally independent. Specifically, if $|A_N| = 1$, N is a uni-leaf node. We call the Leaf-PDF procedure to model univariate PDF $\Pr_{T_N}(A_N)$ using off-the-shelf tools. In our implementation, we choose histograms [46] and parametric Gaussian mixture functions [47] to model categorical and continuous attributes, respectively.

Otherwise, we partition A_N into mutually independent subsets based on their pairwise correlations (step ② in Figure 3). Two attributes are regarded as independent if their correlation value is no larger than a threshold τ_l. If A_N can be split into mutually independent subsets $A_1, A_2, \ldots, A_m$, we set N to be a product node and call FLAT-Offline to model each $\Pr_{T_N}(A_i)$. If not, the local independency does not exist, so we need to split the data (step ③ in Figure 3). Similar to [11], we apply a clustering method, such as k-means [26], to cluster T_N to $T_1, T_2, \ldots, T_n$ according to A_N . The records in the same cluster are similar, so the corresponding PDF becomes smoother and attributes are more likely to be independent. At this time, we set N to be a sum node and call FLAT-Offline to model $\Pr_{T_i}(A_N)$ with weight $w_i = |T_i|/|T_N|$ for each $1 \leq i \leq n$.

3. Modeling conditional PDF: when $C_N \neq \emptyset$, we model the conditional PDF $\Pr_{T_N}(A_N|C_N)$. First, we compute pairwise correlations across all attributes in A_N and C_N . If A_N is independent of C_N, N is a multi-leaf node. We model the multivariate PDF $\Pr_{T_N}(A_N)$ using the piecewise regression technique [40] and maintain its range in the attribute domain space.

Otherwise, we further split records in T_N (step ④ in Figure 3). Probability computation requires T_N to be divided into grids in terms of C_N. We apply a heuristic d-way partition method where d is a hyper-parameter. We choose the attribute $c \in C_N$ that maximizes the pairwise correlations between A_N and C_N. Intuitively, dividing the space by c would largely break their correlations. We set N to be a split node, evenly divide the range of c on T_N into d parts and get the clusters $T_1, T_2, \ldots, T_d$. After that, we call FLAT-Offline to model $\Pr_{T_i}(A_N|C_N)$ for each $1 \leq i \leq d$.

Complexity Analysis. Let n be the number of nodes in the resulting FSPN and s be the number of sum nodes. On each inner node, we can sample a set of r records from table T to compute the RDC scores between attributes. The time cost of calling RDC is $O(r \log r)$, so the total time cost is $O(n|A|^2 r \log r)$. On each sum node, we can also use the sampled records to compute the central points of the clusters and then assign each record to the nearest cluster. We denote the maximum iteration time in k-means as t. The total clustering time cost on all sum nodes is $O(stkr)$. Besides, on each node, we need to scan all records in T to assign them to the children (for inner nodes) or building the PDFs (for leaf nodes). The total scanning time cost is $O(n|T|)$. Therefore, the time complexity of FLAT-Offline is $O(n|A|^2 r \log r + n|T| + stkr)$. As n is often small, it is efficient. By our testing, learning the structure of an FSPN is faster than SPN and DAR to model the same joint PDF.

4.3 Incremental Updates

When the table T changes, we apply an incremental update method FLAT-Update to ensure the underlying FSPN model can fit the new data. To attain high estimation accuracy while saving update cost, we try to preserve the original FSPN structure to the maximum extent while fine-tuning its parameters for better fitting.

Let ΔT be the data inserted into (or deleted from) T. We could traverse the FSPN in a top-down manner to fit $T + \Delta T$ (or $T - \Delta T$). Specifically, for each factorize node N, since the conditional factorization is a lossless decomposition of the joint PDF, we propagate ΔT to its children. For each split node, we propagate each record in ΔT to the corresponding child according to its splitting condition.

On each original multi-leaf node L, we recheck whether the conditional independence still holds after adding (or deleting) some records. If so, we just update the parameters of its multivariate PDF by ΔT. Otherwise, we reset it as a split node and run of FLAT-Offline to further divide its domain space accordingly.

For each sum node, we store the centroids of all clusters in structure construction. We could assign each record in ΔT to the nearest cluster (or remove each record from its original cluster), propagate it to that child and update the weight of each child accordingly.

For each product node, we also recheck whether the independence between attributes subset still holds after adding (or deleting) some records. If not, we run FLAT-Offline to reconstruct the substructure of the FSPN accordingly. Otherwise, we directly pass ΔT to its children. On each uni-leaf node, we update its parameters of the univariate PDF by ΔT. Obviously, after updating, the generated FSPN can accurately fit the PDF of $T + \Delta T$ (or $T - \Delta T$).

We put the pseudocode of FLAT-Update in Appendix B.2 [66]. It can run in the background of the DBMS. In case of significant data change or schema changes, such as inserting or deleting attributes, the FSPN could be rebuilt by calling FLAT-Offline in Section 4.2.

5 MULTI-TABLE CardEst METHOD

In this section, we discuss how to extend FLAT algorithm to multi-table join queries. We first describe our approach on a high level, and then elaborate the key techniques in details.

Algorithm FLAT-Multi(D, Q)

1: organize all tables in D as a join tree J% offline
2: **for** each edge $(A, B) \in J$ **do**
3: **if** RDC$(a, b) \geq \tau_I$ for any attribute a of A and b of B **then**
4: $A \leftarrow \{A, B\}$
5: **for** each node T in J with attributes A_T of $\mathcal{T}$ **do**
6: add scattering coefficient columns S_T in T
7: $\mathcal{F}_T \leftarrow$ FLAT-Offline($A_T \cup S_T$, $\emptyset$, $\mathcal{T}$)
8: let $E = \{T_1, T_2, \ldots, T_d\}$ denote all nodes in touched by Q % online
9: **for** $i \leftarrow 1$ to d **do**
10: compute p_i in Eq. (1) by Technique II
11: **return** $|\mathcal{E}| \cdot \prod_{i=1}^{d} p_i$

Main Idea. To avoid ambiguity, in the following, we use printed letters, such as T, D, to represent a set of tables, and calligraphic letters, such as $\mathcal{T}, \mathcal{D}$, to represent the corresponding full outer join table. Given a database D, all information of D is contained in $\mathcal{D}$. DAR-based approach [62] builds a single large model on $\mathcal{D}$. It is easy to use and applicable to any type of joins between tables in D but suffer from significant limitations. First, no matter how many tables are involved in a query, the entire model has to be used for probability computation, which may be inefficient. Second, the size of $\mathcal{D}$ grow rapidly w.r.t. the number of tables in D, so its training cost is high even using samples from $\mathcal{D}$. Third, in case of data update of any table in D, the entire model needs to be retrained.

Another approach [20] builds a set of small models, where each captures the joint PDF of several tables $D' \subseteq D$. The joint PDF of attributes in $\mathcal{D}'$ (the full outer join table of D') is different from that in $\mathcal{D}$ since each record in $\mathcal{D}'$ can appear multiple times in $\mathcal{D}$. Therefore, the local model of $\mathcal{D}'$ needs to involve some additional columns to correct such PDF difference. When a query touches tables in multiple models, all local probabilities are corrected and merged together to estimate the final cardinality. This approach is more efficient and flexible, but it only supports the primary-foreign key join. This is not practical as many-to-many joins are very common in query optimization (see Section 6.3 for examples on the benchmark workload).

To overcome their drawbacks, our approach absorbs the key ideas of [20] and also builds a set of small local models. However, we extend this method to be more general and applicable. First, we develop a new PDF correction paradigm, inspired by [20], to support more types of joins, e.g., inner or outer and many-to-many (See the following Technique I). Second, we specifically optimize the probability computation and correction process based on our FSPN model (See Technique II). Third, we develop incremental model updates method for data changes (See Technique III).

Algorithm Description. We present a high-level description of our approach in the FLAT-Multi algorithm, which takes a database D and a query Q as inputs. The main procedures are as follows:

1. Offline Construction (lines 1–7): We first organize all tables in D as a tree J based on their joins. Initially, each node in J is a table in D, and each edge in J is a join between two tables. We do not consider self-join and circular joins in this paper. Based on J, we can partition all tables in D into multiple groups such that: tables are highly correlated in the same group but weakly correlated in different groups. Specifically, for each edge (A, B) in J, we sample some records from $A \bowtie B$, the outer join table, and examine the pairwise attribute correlation values between A and B. If some correlation values are higher than a threshold, we learn the model on $A \bowtie B$ together, so we merge $\{A, B\}$ to a single node. We repeat this process until no pair of nodes needs to be merged. After that, the probability across different nodes can roughly be assumed as independent on their full outer join table.

After the partition, each node T in J represents a set of one or more single tables. We add some scattering coefficient columns in its outer join table $\mathcal{T}$ for PDF correction. The details are explained in the following Technique I. Then, we construct a FSPN $\mathcal{F}_T$ on $\mathcal{T}$ using FLAT-Offline in Section 4.2. If $\mathcal{T}$ is large, we do not explicitly materialize it. Instead, we draw some samples from $\mathcal{T}$ using the method in [65] and train the FSPN model on them.

Figure 4 depicts a example database with three tables. The join between T_B and T_C is a many-to-many join. T_A and T_B are highly correlated so they are merged together into node T_1. Then, we build two FSPNs $\mathcal{F}_{T_1}$ and $\mathcal{F}_{T_2}$ on table $T_A \bowtie T_B$ and T_C, respectively.

2. Online Processing (lines 8–12): Let $E = \{T_1, T_2, \ldots, T_d\}$ denote all nodes in J touched by the query Q and Q_i be the sub-query on T_i. By our assumption, the probability of each Q_i is independent on the table $\mathcal{E} = \mathcal{T}_1 \bowtie \mathcal{T}_2 \bowtie \cdots \bowtie \mathcal{T}_d$. We can efficiently correct the probability from the local model $\mathcal{F}_{T_i}$ on $\mathcal{T}_i$ to $\mathcal{E}$ by a new paradigm. Finally, we multiply all probabilities to get the final result.

Technique I: Probability Correction Method. We need to correct the probability to account for the effects of joining from two aspects. We elaborate the details with the example query Q in Figure 4(e). Q is divided into two sub-queries: Q_1 ($T_{B.B_2} > 0.5$ on node T_1) and Q_2 ($T_{C.C_2} < 0.3$ on node T_2). First, on node T_1, the FSPN $\mathcal{F}_{T_1}$ is built on table $T_A \bowtie T_B$ instead of table T_B individually. As each record in T_B can occur multiple times in $T_A \bowtie T_B$, the probability obtained by $\mathcal{F}_{T_1}$ needs to be *down-scaled* to remove the effects of T_A. Second, the probability obtained on node T_2 is defined on table T_C individually but not on $T_B \bowtie T_C$. Therefore, the probability of Q_2 (and also Q_1) needs to be *up-scaled* to add the effects of joining.

The above corrections are achieved by adding extra columns in table $\mathcal{T}_i$ of each node T_i. These columns track the number of times that a record in a single table A appears in $\mathcal{T}_i$, i.e., the scattering effect. Previous works [20, 62] add columns to process the scattering effects of each join in only one side. However, our solution considers the scattering effects on two sides of each join. It is more practical by supporting more join types in one framework, and more general by processing down-scale and up-scale effects at the same time.

For each pair of joined tables (A, B) in a node T_i, we add two additional attributes $S_{A,B}$ and $S_{B,A}$ in $\mathcal{T}_i$. $S_{A,B}$ indicates how many records in B can join with this record in A and vice versa. We call such $S_{A,B}$ *scattering coefficient*. In Figure 4(d), we add two columns $S_{A,B}$ and $S_{B,A}$ in the table $T_A \bowtie T_B$ of T_1. These columns are be used to down-scale the effects of untouched tables inside each node.

Similarly, for up-scale correction, we can regard node T_i as the root of the join tree J. For each distinct sub-tree of J rooted at T_i containing nodes $E' = \{T'_1, T'_2, \ldots, T'_d\}$, we add a column $S_{T_i,E'}$ in table $\mathcal{T}_i$ indicating the scattering coefficient of each record in $\mathcal{T}_i$ to the outer join table $\mathcal{E}' = \mathcal{T}'_1 \bowtie \mathcal{T}'_2 \bowtie \cdots \bowtie \mathcal{T}'_d$. For the node T_2 in Figure 4(c), we add the column $S_{T_2,\{T_1,T_2\}}$ indicating the scattering coefficient of each record in T_C when joining with $T_A \bowtie T_B$. The method to compute the values of these scattering coefficient columns has been proposed in [65]. Briefly speaking, we can obtain the values of $S_{T_i,E'}$ by recursively aggregating over all

(a) Table T_A

A_1	A_2
b	0
c	2
b	3
c	4

(b) Table T_B

B_1	B_2	B_3
1	0.3	M
0	0.6	D
3	0.4	D
2	0.7	M
4	0.5	K
3	0.2	K

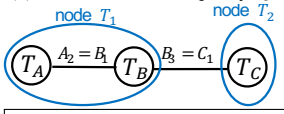
(c) Table T_C of node T_2

C_1	C_2	$S_{T_2,\{T_1,T_2\}}$
D	0.2	2
M	0.7	2
D	0.8	2
K	0.9	2

(d) Table $T_A \rtimes T_B$ of node T_1

A_1	A_2	B_1	B_2	B_3	$S_{A,B}$	$S_{B,A}$	$S_{T_1,\{T_1,T_2\}}$
null	null	1	0.3	M	0	0	1
b	0	0	0.6	D	1	1	2
b	3	3	0.4	D	2	1	2
c	2	2	0.7	M	1	1	1
c	4	4	0.5	K	1	1	1
b	3	3	0.2	K	2	1	1

(e) Join Tree J and Query Q



Q: select count(*) from T_B full outer join T_C on $T_{B.B_3} = T_{C.C_1}$ where $T_{B.B_2}$>0.5 and $T_{C.C_2}$<0.3

Figure 4: Example databases and join query.

sub-trees rooted at T_i's children. Using dynamic programming, the time cost of computing scatter coefficient values over all nodes is linear w.r.t. table size.

As all tables form a join tree, the number of added scattering columns in each node is linear w.r.t. its number of tables. In each node T_i, all scattering coefficient columns are learned together with other attributes when constructing the FSPN $\mathcal{F}_{T_i}$.

We can estimate the cardinality by the following lemma. We put the detailed correctness proof in Appendix C of the technical report [66]. In a high order, for each record with down-scale value s and up-scale value e, we correct its probability satisfying Q_i by a factor of e/s. We set e or s to 1 if it is 0 since records with zero scattering coefficient also occur once in the full outer join table.

Lemma 2 *Given a query Q, let $E = \{T_1, T_2, \ldots, T_d\}$ denote all nodes in J touched by Q. On each node T_i, let $S = \{S_{A_1,B_1}, S_{A_2,B_2}, \ldots, S_{A_n,B_n}\}$, where each (A_j, B_j) is a distinct join such that B_j is not in Q. Let $s = (s_1, s_2, \ldots, s_n)$ where $S_{A_j,B_j} = s_j \in \mathbb{N}$ for all $1 \le i \le n$ denote an assignment to S and $dlm(s) = \prod_{j=1}^{n} \max\{s_j, 1\}$. Let*

$$p_i = \frac{|\mathcal{T}_i|}{|\mathcal{E}|} \cdot \sum_{s,e} \left(\Pr_{\mathcal{T}_i}(Q_i \wedge S = s \wedge S_{T_i,E} = e) \cdot \frac{\max\{e,1\}}{\mathrm{dlm}(s)} \right). \quad (1)$$

Then, the cardinality of Q is $|\mathcal{E}| \cdot \prod_{i=1}^{d} p_i$.

Consider again query Q in Figure 4(e). For the sub-query Q_1 on node T_1, we need to down-scale by $S_{B,A}$ and up-scale by $S_{T_1,\{T_1,T_2\}}$. By Eq. (1), we have $p_1 = (1 * 2 + 1 + 1)/8 = 1/2$. Similarly, we have $p_2 = 1/4$ for sub-query Q_2, so the final cardinality of Q is $8 * (1/8) = 1$.

As a remark, if two tables A and B are inner joined in Q, we can add the constraint $S_{A,B} > 0$ and $S_{B,A} > 0$ (or $S_{A,E} > 0$ and $S_{B,E} > 0$ if A and B in different nodes) in Eq. (1) to remove all records in A or B that have no matches. Similarly, we only add $S_{B,A} > 0$ or $S_{A,B} > 0$ to Q for left and right join, respectively.

Technique II: Fast Probability Computation: Notice that, the value p_i in Eq. (1) involves summing over the probabilities of each assignment to the down-scale value s and up-scale value e. If we directly obtain all these probabilities, the time cost is very high. Instead, we present an optimized method to compute p_i, which only requires a *single* traversal on the underlying FSPN model.

Specifically, on any node T in the join tree, let S_T and A_T denote the scattering coefficient and attribute columns in $\mathcal{T}$, respectively. When constructing the FSPN $\mathcal{F}_T$, we first use a factorize root node to split the joint PDF $\Pr_{\mathcal{T}}(S_T, A_T)$ into $\Pr_{\mathcal{T}}(A_T)$ on the left child LC and $\Pr_{\mathcal{T}}(S_T|A_T)$ on the right child RC. Each leaf node L of RC models a PDF of S_T. By FSPN's semantic, the probabilities of any query Q on A_T and S_T are independent on each L. Then, we have

$$\begin{aligned} \Pr'_{\mathcal{T}}(Q) &= \sum_{L} \left(\Pr_L(A_T) \cdot \sum_{s,e} \left(\Pr_L(S = s \wedge S_{T,E} = e) \cdot \frac{\max\{e,1\}}{\mathrm{dlm}(s)} \right) \right) \\ &= \sum_{L} \left(\Pr_L(A_T) \cdot \mathbb{E}\left[\frac{\max\{e,1\}}{\mathrm{dlm}(s)} \right] \right). \end{aligned} \quad (2)$$

For the left part, the probability $\Pr_L(A_T)$ could be computed with the FSPN rooted at node LC using the method in Section 4.1. For the right part, it is a *fixed* expected value of $\max\{e,1\}/\mathrm{dlm}(s)$ of S_T. Therefore, we can pre-compute the expected value for each possible $S, S_{T,E} \subseteq S_T$ on each leaf L. After that, each p_i in Eq. (1) could be obtained by traversing the FSPN $\mathcal{F}_{T_i}$ *only once*. By our empirical analysis in Section 4.1, the CardEst time cost for multi-table queries is also near linear w.r.t. the number of nodes in FSPNs.

Technique III: Incremental Updates. Next, we introduce how to update the underlying FSPN models in multi-table cases. We put the pseudocode of our algorithm FLAT-Update-Multi in Appendix B.2 [66] and describe the procedures as follows.

First, we consider the case of inserting some records ΔC in a table C of the node T. It affects $\mathcal{T}$ in three aspects: 1) each record in ΔC can join with other tables in T. We use $\Delta\mathcal{T}_+$ to denote all new records inserted into $\mathcal{T}$; 2) each record in $\mathcal{T}$, which does not find a match in table C (null) but can join with the new records in ΔC, needs to be removed. We denote them as $\Delta\mathcal{T}_-$; and 3) the scattering coefficient of each record in $\mathcal{T}$, which can join with new records in ΔC, needs to be enlarged. We denote these records as $\Delta\mathcal{T}_*$. We can directly join ΔC with $\mathcal{T}$ to identify $\Delta\mathcal{T}_+$, $\Delta\mathcal{T}_-$ and $\Delta\mathcal{T}_*$ accordingly.

Next, we describe how to incrementally update the FSPN $\mathcal{F}_T$ built by Technique II. Recall that the root node N of $\mathcal{F}_T$ is a factorize node separating attributes and scattering coefficient columns, which enables fast incremental update. The left child LC of N models $\Pr_{\mathcal{T}}(A_T)$ on all attribute columns. We could update it to fit the data $\mathcal{T} + \Delta\mathcal{T}_+ - \Delta\mathcal{T}_-$ by directly calling the FLAT-Update method in Section 4.3. The right child RC of N models $\Pr_{\mathcal{T}}(S_T|A_T)$ on all scattering coefficients columns. Each multi-leaf L of RC only stores some expected values of S_T defined by Eq. (2). We can pre-build a hash table on the probability of each assignment s of S_T. Then, based on the changes of scattering columns in $\Delta\mathcal{T}_+$, $\Delta\mathcal{T}_-$ and $\Delta\mathcal{T}_*$, we can incrementally update all expected values.

Finally, as $\mathcal{T}$ changes, we need to propagate the effects to other nodes T' to update all scattering columns $S_{T',E}$. For efficiency, it can run in the background asynchronously. Specifically, after each time interval such as one day, we scan all tables and recompute the scattering coefficients using the method in [65]. Then we incrementally update the expected values stored in FSPN $\mathcal{F}_{T'}$.

For the case of deleting some records ΔC in a table C of the node T, the updating could be done in a very similar way. At this time, we obtain $\Delta\mathcal{T}_-$ containing all removed tuples joining with ΔC previously, $\Delta\mathcal{T}_+$ containing all added tuples having no matches in table C and $\Delta\mathcal{T}_*$ containing all original records whose scattering coefficients are reduced. Then we update the FSPN $\mathcal{F}_T$ and $\mathcal{F}_{T'}$ of other nodes T' in the same way as the insertion case. Notice that, the data insertion and deletion can also be done simultaneously as long as we maintain the proper set of records $\Delta\mathcal{T}_+$, $\Delta\mathcal{T}_-$ and $\Delta\mathcal{T}_*$. In the complex case of creating new tables or deleting existing tables in the database, the model could be retrained offline.

6 EVALUATION RESULTS

We have conducted extensive experiments to demonstrate the superiority of our proposed FLAT algorithm. We first introduce the experimental settings, and then report the evaluation results of CardEst algorithms on the single table and multi-table cases in Section 6.1 and 6.2, respectively. Section 6.3 reports the effects of updates. Finally, in Section 6.4, we integrate FLAT into the query optimizer of Postgres [8] and evaluate the end-to-end query optimization performance.

Baselines. We compare FLAT with a variety of representative CardEst algorithms, including:

1) Histogram: the simplest 1-D histogram based CardEst method widely used in DBMS such as SQL Server [34] and Postgres [8].

2) Naru: a DAR based algorithm proposed in [63]. We adopt the authors' source code from [64] with the var-skip speeding up technique [31]. It utilizes a DNN with 5 hidden layers (512, 256, 512, 128, 1024 neuron units) to approximate the PDFs. The sampling size is set to 2,000 as the authors' default. We do not compare with the similar method in [17], since their performance is close.

3) NeuroCard [62]: an extension of Naru onto the multi-table case. We also adopt the authors' source code from [36] and set the sampling size to 8,000 as the authors' default.

4) BN: a Bayesian network based algorithm. We use the Chow-Liu Tree [4, 16] based implementation to build the BN structure, since its performance is better than others [12, 57].

5) DeepDB: a SPN based algorithm proposed in [20]. We adopt the authors' source code from [19] and apply the same hyper-parameters, which set the RDC independence threshold to 0.3 and split each node with at least 1% of the input data.

6) SPN-Multi: a simple extension of SPN with multivariate leaf nodes. It maintains a multi-leaf node if the data volume is below 1% and attributes are still not independent.

7) MaxDiff: a representative M-D histogram based method [46]. We use the implementation provided in the source code repository of [64]. We do not compare with the improved methods DBHist [7], GenHist [14] and VIHist [59] are they are not open-sourced.

8) Sample: the method uniformly samples a number of records to estimate the cardinality. We set the sampling size to 1% of the dataset. It is used in DBMS such as MySQL [48] and MariaDB [52]. We do not compare with other method such as IBJS [29] since their performance has been verified to be less competitive [20, 62, 64].

9) KDE: kernel density estimator based method for CardEst. We have implemented it using the scikit-learn module [33].

10) MSCN: a state-of-the-art query-driven CardEst algorithm described in [24]. For each dataset, we train it with 10^5 queries generated in the same way as the workload.

Regarding FLAT hyper-parameters as described in Section 4.2, we set the RDC threshold $\tau_l = 0.3$ and $\tau_h = 0.7$ for filtering independent and highly correlated attributes, respectively, and set $d = 2$ for d-way partition of records. Similar to DeepDB, we also do not split a node when it contains less than 1% of the input data. The sensitivity analysis of hyper-parameters are put in Appendix D [66].

Evaluation Metrics. Based on our discussion in Section 1, we concentrate on examining three key metrics: estimation accuracy, time efficiency and storage overhead. For estimation accuracy, we adopt the widely used q-error metric [17, 20, 24, 28, 30, 63] defined as the larger value of$\mathrm{Card}(T,Q)/\widehat{\mathrm{Card}}(T,Q)$ and $\widehat{\mathrm{Card}}(T,Q)/\mathrm{Card}(T,Q)$, so its optimal value 1. We report the whole q-error distribution (50%, 90%, 95%, 99% and 100% quantile) of each workload. For time efficiency, we report the estimation latency and model training time. For storage overhead, we report the model size.

Environment. All above algorithms have been implemented in Python. All experiments are performed on a CentOS Server with an Intel Xeon Platinum 8163 2.50GHz CPU having 64 cores, 128GB DDR4 main memory and 1TB SSD.

6.1 Single Table Evaluation Results

We use two single table datasets: 1) GAS is real-world gas sensing data obtained from the UCI dataset [49] and contains 3,843,159 records. We extract the most informative 8 columns (*Time*, *Humidity*, *Temperature*, *Flow_rate*, *Heater_voltage*, *R1*, *R5* and *R7*); and 2) DMV [42] is a real-world vehicle registration information dataset and contains 11,591,877 tuples. We use the same 11 columns as [64].

For each dataset, we generate a workload containing 10^5 randomly generated queries. For each query, we use a probability of 0.5 to decide whether an attribute should be contained. As stated in Section 2, the domain of each attribute A is mapped into an interval, so we uniformly sample two values l and h from the interval such that $l \leq h$ and set $A \in [l, h]$.

Estimation Accuracy. Table 1 reports the q-error distribution for different CardEst algorithms. As main take-away, their accuracy can be ranked as FLAT $\approx$ Naru$\approx$ SPN-Multi > BN > DeepDB >> Sample/MSCN >> KDE >> MaxDiff/Histogram. The details are as follows:

1) Overall, FLAT 's estimation accuracy is very high. On both datasets, the median q-error (1.001 and 1.002) is very close to 1, the optimal value. On GAS, FLAT attains the highest accuracy. The accuracy of Naru and SPN-Multi is comparable to FLAT, which is marginally better than FLAT on DMV. The high accuracy of Naru and stems from its AR based decomposition and the large DNN representing the PDFs. SPN-Multi achieves high accuracy as it models the PDFs of attributes without independence assumption.

2) The accuracy BN and DeepDB is worse than FLAT. At the 95% quantile, FLAT outperforms BN by 3.6× and DeepDB by 71× on GAS. The error of BN mainly arises from its approximate structure construction. DeepDB appears to fail at splitting highly correlated attributes. Thus, it causes relatively large estimation errors for queries involving these attributes.

Table 1: Performance of CardEst algorithms on single table.

Dataset	Algorithm	50%	90%	95%	99%	Max	Size (KB)	Training Time (Min)
GAS	Histogram	2.732	53.60	163.0	$2 \cdot 10^6$	$3 \cdot 10^7$	**34**	**1.3**
	Naru	1.007	1.145	1.340	2.960	16.50	6, 365	216
	BN	1.011	1.208	1.550	4.780	36.80	108	8.2
	DeepDB	1.039	1.765	2.230	95.12	619.2	218	54
	SPN-Multi	1.005	1.169	1.289	1.461	3.702	31,253	62
	MaxDiff	2.211	86.7	196.0	$3 \cdot 10^4$	$8 \cdot 10^5$	$3 \cdot 10^5$	310
	Sample	1.046	1.625	2.064	6.017	3, 410	-	-
	KDE	3.307	5.469	6.742	471.0	$2 \cdot 10^4$	-	27
	MSCN	2.610	68.47	129.0	$1 \cdot 10^5$	$7 \cdot 10^5$	2, 663	662
	FLAT (Ours)	**1.001**	**1.127**	**1.183**	**1.325**	**3.178**	198	19
DMV	Histogram	1.184	2.541	41.72	710.0	$2 \cdot 10^5$	**24**	**1.6**
	Naru	1.006	1.184	1.368	**6.907**	**49.03**	7, 564	146
	BN	1.003	1.264	1.818	9.800	176.0	59	5.4
	DeepDB	1.005	1.574	2.604	27.90	534.0	247	48
	SPN-Multi	1.004	**1.163**	**1.347**	7.225	58.37	53,267	53
	MaxDiff	1.802	6.304	28.81	4, 320	$3 \cdot 10^4$	$7 \cdot 10^5$	249
	Sample	1.122	1.619	9.010	551.0	7, 077	-	-
	KDE	3.493	15.07	104.0	589.0	$5 \cdot 10^4$	-	48
	MSCN	1.215	2.612	4.420	17.90	1, 192	2, 566	744
	FLAT (Ours)	**1.002**	1.255	1.795	9.805	76.50	53	2.4

3) The accuracy of MSCN and Sample appears unstable. FLAT outperforms MSCN by 109× and 1.8× on GAS and DMV, respectively. As MSCN is query-driven, its accuracy relies on if the workload is "similar" to the training samples. Whereas, FLAT outperforms Sample by 4.5× and 56× on GAS and DMV, respectively as the sampling space of DMV is much larger than GAS.

4) FLAT largely outperforms Histogram, MaxDiff and KDE since Histogram and MaxDiff makes coarse-grained independence assumption and KDE may not well characterize high-dimensional data by tuning a good bandwidth for kernel functions [23].

Estimation Latency. Figure 5 reports the average latency of all CardEst methods. Since only MSCN and Naru provide the implementation optimized for GPUs, we compare all CardEst methods on CPUs for fairness. We provide the comparison results on GPUs in Appendix E.1 [66]. In summary, their speed on CPUs can be ranked as Histogram ≈ FLAT > MSCN > SPN-Multi/DeepDB > KDE/Sample >> Others. The details are as follows:

1) Histogram runs the fastest, it requires around 0.1*ms* for each query. FLAT is close with a latency around 0.2*ms* and 0.5*ms* on DMV and GAS, respectively. Both are much faster than all other methods. This can be credited to the FSPN model used in FLAT being both compact and easy to traverse for probability computation. MSCN is also fast since it only requires a forward pass over DNNs.

2) DeepDB, SPN-Multi, KDE and Sample need up to 10*ms* for each query. FLAT is 1–2 orders of magnitude faster than them because the FSPN model used in our FLAT is more compact than the SPN model in DeepDB and SPN-Multi. In addition, KDE and Sample need to examine large amount of samples, thus less efficient.

3) MaxDiff, BN and Naru need 10–100*ms* for each query. FLAT is 2–3 orders of magnitude faster than them, e.g., 213× and 599× faster than Naru on GAS and DMV, respectively. The time cost of MaxDiff is spent on decompressing the joint PDF. The inference on BN is NP-hard and hence inefficient. Naru requires repeated sampling for range querie so it is computationally demanding.

Model Training Time. As shown in the last column in Table 1, FLAT is very efficient in training. Specifically, on DMV, FLAT is 61× and 20× faster than Naru and DeepDB in training. This is due to the structure of FSPN is much smaller than SPN, and our training process does not require iterative gradient updates as required for SGD-based training of DNNs [2].

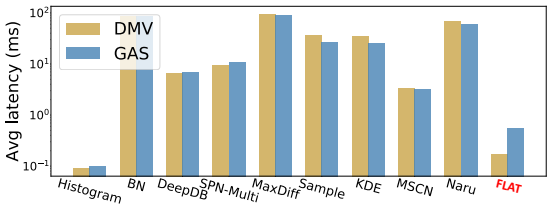


Figure 5: Estimation latency of CardEst algorithms.

Storage Overhead. Storage costs are given in Table 1. The storage cost of Histogram and BN is proportional to the attribute number so they require the smallest storage. FLAT is also very small requiring about 2× of Histogram. DeepDB requires more storage space than FLAT since the learned SPN has more nodes. They consume 10–100KB of storage. MSCN and Naru consume several MB since they store large DNN models. SPN-Multi requires tens of MB as it needs to maintain the multi-leaf nodes on not highly correlated attributes, as we discussed in Section 3. The storage cost of MaxDiff is the highest since it stores the compressed joint PDF.

Model Node Number. To give more details, we also compare the number of nodes (or neurons) in DeepDB, SPN-Multi and Naru. The 5-layer DNN in Naru is fully connected and contains 2, 432 neurons. The SPN used in DeepDB contains 873 and 823 nodes on GAS and DMV, respectively. SPN-Multi contains 825 and 787 nodes on GAS and DMV, respectively. Whereas, the FSPN in FLAT only uses 210 and 20 nodes on GAS and DMV, respectively. FSPN uses 21×, 7.4× and 7× less nodes than DNN, SPN and SPN-Multi to model the same joint PDF.

Stability. We also examine FLAT on synthetic datasets. The results in Appendix E.2 show that FLAT is stable to varied correlations and distributions and relatively robust to varied domain size.

6.2 Multi-Table Evaluation Results

We evaluate the CardEst algorithms for the multi-table case on the IMDB benchmark dataset. It has been extensively used in prior work [20, 28, 30, 62] for cardinality estimation. We use the provided *JOB-light* query workload with 70 queries and create another more complex and comprehensive workload *JOB-ours* with 1, 500 queries.

JOB-light's schema contains six tables (*title*, *cast_info*, *movie_info*, *movie_companies*, *movie_keyword*, *movie_info_idx*) where all other tables can only join with *title*. Each *JOB-light* query involves 3–6 tables with 1–4 filtering predicates on all attributes. *JOB-ours* uses the same schema as *JOB-light* but each query is a range query using 4–6 tables and 2–7 filtering predicates. The predicate of each attribute is set in the same way as on single table. Figure 6 illustrates the true cardinality distribution of the two workloads. The scope of cardinality for *JOB-ours* is wider than *JOB-light*. Note that, the model of each CardEst method is the same for the two workloads. As the attributes are highly correlated on IMDB, the model size of SPN-Multi exceeds our memory limit, so we can not evaluate it.

Results on *JOB-light*. Table 2 reports the q-error and storage cost of CardEst methods on the *JOB-light* workload. We observe that:

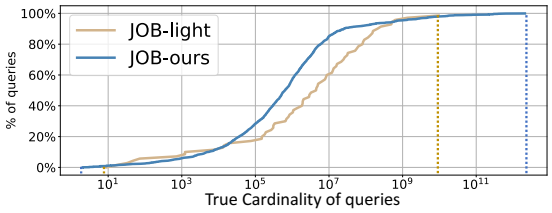


Figure 6: Cardinality distribution of workload on IMDB.

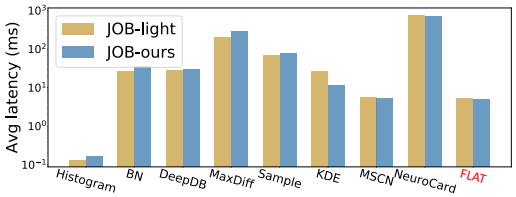


Figure 7: Estimation latency on IMDB.

Table 2: Performance of CardEst algorithms on *JOB-light*.

Algorithm	50%	90%	95%	99%	Max	Size (KB)
Histogram	8.310	1, 386	6, 955	$8 \cdot 10^5$	$2 \cdot 10^7$	**131**
NeuroCard	1.580	4.545	5.910	8.480	**8.510**	7, 076
BN	2.162	28.00	74.60	241.0	306.0	237
DeepDB	1.250	2.891	3.769	25.10	31.50	$3.7 \cdot 10^4$
MaxDiff	32.31	5, 682	$5 \cdot 10^4$	$4 \cdot 10^6$	$4 \cdot 10^7$	$4 \cdot 10^5$
Sample	2.206	65.80	1, 224	$5 \cdot 10^4$	$1 \cdot 10^6$	-
KDE	10.56	563.0	4, 326	$4 \cdot 10^5$	$8 \cdot 10^6$	-
MSCN	2.750	19.70	97.60	622.0	661.0	3, 421
FLAT (Ours)	**1.150**	**1.819**	**2.247**	**7.230**	10.86	3, 430

Table 3: Performance of CardEst algorithms on *JOB-ours*.

Algorithm	50%	90	95%	99%	Max	Training Time (Min)
Histogram	15.71	7480	$4 \cdot 10^4$	$1 \cdot 10^6$	$4 \cdot 10^8$	**2.7**
NeuroCard	1.538	9.506	81.23	8012	$1 \cdot 10^5$	173
BN	2.213	25.60	2456	$2 \cdot 10^5$	$7 \cdot 10^6$	7.3
DeepDB	1.930	28.30	248.0	$1 \cdot 10^4$	$1 \cdot 10^5$	68
MaxDiff	45.50	8007	$2 \cdot 10^5$	$9 \cdot 10^6$	$1 \cdot 10^9$	79
Sample	2.862	116.0	3635	$3 \cdot 10^5$	$4 \cdot 10^7$	-
KDE	8.561	1230	$1 \cdot 10^4$	$9 \cdot 10^5$	$2 \cdot 10^8$	15
MSCN	4.961	45.7	447.0	8576	$1 \cdot 10^5$	1, 744
FLAT (Ours)	**1.202**	**6.495**	**57.23**	**1120**	$\mathbf{1 \cdot 10^4}$	53

1) The accuracy of FLAT is the highest among all algorithms. NeuroCard is only a bit better w.r.t the maximum q-error, which reflects only one query in the workload. At the 95% quantile, FLAT outperforms NeuroCard by 2.6×, BN by 33×, DeepDB by 1.7× and MSCN by 43×. The reasons have been explained in Section 6.1.

2) In terms of storage size, Histogram and BN are still the smallest and MaxDiff is still the largest. FLAT 's space cost is 3.3MB, which is 10.8× and 2.1× less than DeepDB and NeuroCard, respectively. In comparison with the single table case, FLAT 's space cost is relatively large. This is because for the multi-table case, FSPN needs to process more attributes—the scattering coefficients columns and materialize some values for fast probability computation. However, it is still reasonable and affordable for modern DBMS.

Results on *JOB-ours*. On this workload, FLAT is also the most accurate CardEst method. As reported in Table 3, we observe that:

1) The performance of FLAT is better than NeuroCard and still much better than others. At the 95% quantile, FLAT outperforms NeuroCard, DeepDB and MSCN by 1.4×, 4.3× and 7.8×, respectively. The performance of other algorithms drops significantly on this workload. A similar observation is also reported in [62]. This once again demonstrates the shortcomings of these approaches, especially for complex data and difficult queries.

2) The q-error of FLAT on *JOB-ours* is relatively larger than that on *JOB-light* because *JOB-ours* is a harder workload. As shown in Figure 6, the true cardinality of the tail 5% queries in *JOB-ours* is often less than 100. However, the performance of FLAT is still reasonable since the median value is only 1.2.

We also examine the detailed q-errors of FLAT and other CardEst methods with different number of tables and predicates in queries. Due to space limits, we put the results in Appendix E.3 of the technical report [66]. The results show that the accuracy of our FLAT is more stable with number of joins and predicates.

Time Efficiency. Figure 7 exhibits the average estimation latency on the two workload. Obviously, Histogram is still the fastest while MaxDiff is still the slowest. FLAT requires around 5*ms* for each query, which is still much faster than others. It outperforms BN by 5×, Sample by 12.4×, KDE by 4.8× and DeepDB by 5.2×. The training time on the IMDB dataset is given in the last column of Table 3. FLAT is faster than NeuroCard and close to DeepDB.

6.3 Effects of Updates

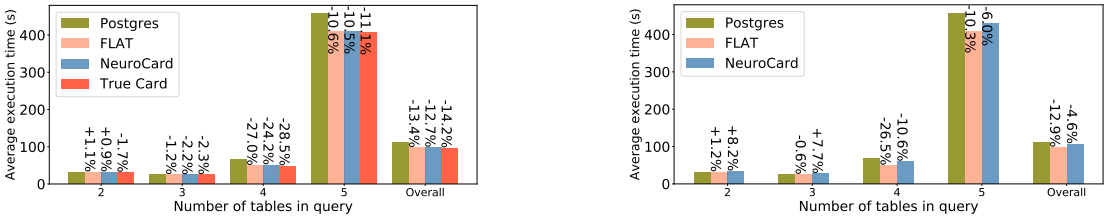
We examine the performance of our incremental update method. Specifically, for data insertion evaluation, we train the base model on a subset of IMDB data before 2004 (80% of data) and insert the rest data for updating. For data deletion, we train the base model on all data and delete the data after 1991. We compare the accuracy on the *JOB-light* workload and the update time cost of our update method with two baselines: he original stale model and the new model retained on the whole data. From Table 4, we observe that:

1) The accuracy of the retrained model is the highest but it requires the highest updating time. The accuracy of the non-updated model is the lowest since the data distribution changes.

2) Our update method makes a good trade-off: its accuracy is close to the retrained model but its time cost is much lower. This shows that our FSPN model can be incrementally updated on its structure and parameters to fit the new data in terms of both insertion and deletion. This is a clear advantage since the entire model does not need to be frequently retrained in presence of new data.

6.4 End-to-End Evaluation on Postgres

To examine the performance of ML-based CardEst algorithms in real-world DBMS, we integrate our FLAT and NeuroCard into the query optimizer of Postgres 9.6.6 to perform an end-to-end test. We do not compare with DeepDB since it can not support many-to-many join. However, for many star-join queries between a primary key and multiple foreign keys in the workload, the sub-queries on joining foreign keys are many-to-many joins. Meanwhile, we add the method which uses the true cardinality of each sub-query during query optimization as the baseline. We report the results



(a) **Query Time Excluding CardEst Latency** (b) **End-to-End Query Time**

Figure 8: Comparison of CardEst algorithms integrated into Postgres.

Table 4: Effects of updates on IMDB.

Update	Method	50%	90%	95%	99%	Max	Time (Min)
Insertion	Non-Updated	1.201	2.297	3.862	18.93	47.14	0
	Retrained	1.150	1.819	2.247	7.230	10.86	53
	Our Method	1.153	1.821	2.480	8.914	13.72	1.2
Deletion	Non-Updated	1.218	2.263	3.905	15.47	56.21	0
	Retrained	1.129	1.763	2.253	6.815	15.3	49
	Our Method	1.134	1.791	2.432	8.285	19.78	1.0

of the *JOB-light* workload on the IMDB benchmark dataset. The results on *JOB-ours* are similar and put in the Appendix E.4 [66].

We disable parallel computing in Postgres and only allow primary key indexing to minimize the impact of other factors [28, 58].We report the total query time excluding the CardEst time cost in Figure 8(a) and the end-to-end query time (including plan compiling and execution) in Figure 8(b). We observe that:

1) Accurate CardEst results can help the query optimizer generate better query plans. Without considering the CardEst latency, both NeuroCard and FLAT improve over Postgres by near 13%. Their improvement is very close to the optimal result using true cardinality in query compiling (14.2%). This verifies that the accuracy of FLAT is sufficient to generate high-quality query plans.

2) For the end-to-end query time, the improvement of FLAT is more significant than NeuroCard. Overall, FLAT improves the query time by 12.9% while NeuroCard only improves 4.6%. This is due to the CardEst needs to do multiple times in query optimization. The latency of NeuroCard is much longer than FLAT and degrades its end-to-end performance.

3) The improvement of FLAT becomes more significant on queries with more joins. On queries joining 4 tables, FLAT improves the end-to-end query time by 26.5% because the search space of the query plans grows exponentially w.r.t. the join number. If a query only joins 2 or 3 tables, its query plan is almost fixed. When it joins more tables, the inaccurate Postgres results may lead to a sub-optimal query plan while our FLAT providing more accurate CardEst results can find a better plan. This phenomenon has also observed and explained in [44].

7 RELATED WORK

We briefly review prior work on query-driven CardEst methods and machine learning (ML) applied to problems in databases. The data-driven CardEst methods have already been discussed in Section 2.

Query-Driven CardEst Methods. Initially, prior research has approached query-driven CardEst by utilizing feedback of past queries to correct generated models. Representative work includes correcting and self-tuning histograms with query feedbacks [3, 10, 22, 54], updating statistical summaries in DBMS [55, 61], and query-driven kernel-based methods [18, 23]. Later on, with the advance of deep learning, focus shifted to learning complex mappings from "featurized" queries to their cardinalities. Different types of models, such as deep networks [32], tree-based regression models [9] and multi-set convolutional networks [24], were applied. In general, clear drawbacks of query-driven CardEst methods are as follows: 1) their performance heavily relies on the particular choice of how input queries are transformed into features; 2) they require large amounts of previously executed queries for training; and 3) they only behave well, when future input queries follow the same distribution as the training query samples. Therefore, query-driven CardEst methods are not flexible and generalizable enough.

ML Applied in Databases. Recently, there has been a surge of interest in using ML-based methods in order to enhance the performance of database components, e.g. indexing [41], data layout [25], query execution [43] and scheduling [37]. Among them, learned query optimizers are a noteworthy hot-spot. [38] proposed a query plan generation model by learning embeddings for all queries. [27] applied reinforcement learning to optimize the join order. We are currently trying to integrate FLAT with these two approaches to design an end-to-end solution for query optimization in databases.

Moreover, it is worth mentioning that the proposed FSPN model is a very *general* unsupervised model, whose scope of application is not limited to CardEst. We are in the process of trying to apply to other scenarios in databases that also require modeling the joint PDF of high-dimensional data, such as approximate group-by query processing [56], hashing [25] and multi-dimensional indexing [41].

8 CONCLUSIONS

In this paper, we propose FLAT, an unsupervised CardEst method that is simultaneously fast in probability computation, lightweight in storage cost and accurate in estimation quality. It supports queries on both single table and multi-tables. FLAT is built on FSPN, a new graphical model which adaptively models the joint PDF of attributes and combines the advantages of existing CardEst models. Extensive experimental results on benchmarks and the end-to-end evaluation on Postgres have demonstrated the superiority of our proposed methods. In the future work, we believe in that FLAT could serve as a key component in an end-to-end learned query optimizer for DBMS and the general FSPN model can play larger roles in more database-related tasks.

REFERENCES

[1] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *SIGMOD*. 1383–1394.

[2] Léon Bottou. 2012. Stochastic Gradient Descent Tricks. *Neural networks: Tricks of the trade* (2012), 421–436.

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

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

[5] Paul Dagum and Michael Luby. 1993. Approximating probabilistic inference in Bayesian belief networks is NP-hard. *Artificial intelligence* 60, 1 (1993), 141–153.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] Robert Gens and Domingos Pedro. 2013. Learning the structure of sum-product networks. In *ICML*. PMLR, 873–880.

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

[13] G. Graefe and W. J. Mckenna. 1993. The Volcano optimizer generator: extensibility and efficient search. In *ICDE*. 209–218.

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

[15] 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.

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

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

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

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

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

[21] Yannis E Ioannidis and Stavros Christodoulakis. 1991. On the propagation of errors in the size of join results. In *SIGMOD*. 268–277.

[22] 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.

[23] 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.

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

[25] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *SIGMOD*. 489–504.

[26] K Krishna and M Narasimha Murty. 1999. Genetic K-means algorithm. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 29, 3 (1999), 433–439.

[27] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196* (2018).

[28] 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.

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

[30] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *The VLDB Journal* 27, 5 (2018), 643–668.

[31] 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.

[32] Henry Liu, Mingbin Xu, Ziting Yu, Vincent Corvinelli, and Calisto Zuzarte. 2015. Cardinality estimation using neural networks. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*. 53–59.

[33] Luch Liu. 2020. Github repository: scikit-learn. *https://github.com/scikit-learn/scikit-learn* (2020).

[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] David Lopez-Paz, Philipp Hennig, and Bernhard Schölkopf. 2013. The randomized dependence coefficient. In *NIPS*. 1–9.

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

[37] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *SIGCOMM*. 270–288.

[38] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711* (2019).

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

[40] Victor E McZgee and Willard T Carleton. 1970. Piecewise regression. *J. Amer. Statist. Assoc.* 65, 331 (1970), 1109–1124.

[41] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-dimensional Indexes. In *SIGMOD*. 985–1000.

[42] State of New York. 2020. Vehicle, snowmobile, and boat registrations. *https://catalog.data.gov/dataset/vehicle-snowmobile-and-boat-registrations* (2020).

[43] Yongjoo Park, Ahmad Shahab Tajik, Michael Cafarella, and Barzan Mozafari. 2017. Database learning: Toward a database that becomes smarter every time. In *SIGMOD*. 587–602.

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

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

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

[47] Carl Edward Rasmussen. 2000. The infinite Gaussian mixture model. In *NIPS*. 554–560.

[48] 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).

[49] UCI ML Repository. 2020. Gas sensor array temperature modulation Data Set. *https://archive.ics.uci.edu/ml/datasets/Gas+sensor+array+temperature+modulation* (2020).

[50] Mauro Scanagatta, Antonio Salmerón, and Fabio Stella. 2019. A survey on Bayesian network structure learning from data. *Progress in Artificial Intelligence* (2019), 1–15.

[51] 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.

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

[53] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. 2019. Presto: Sql on everything. In *ICDE*. 1802–1813.

[54] 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.

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

[56] Saravanan Thirumuruganathan, Shohedul Hasan, Nick Koudas, and Gautam Das. 2020. Approximate query processing for data exploration using deep generative models. In *ICDE*. 1309–1320.

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

[58] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*. 1009–1024.

[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. 2020. Are we ready for learned cardinality estimation? arXiv:2012.06743 [cs.DB]

[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] 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.

[63] 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).

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

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

[66] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2020. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation [Technical Report]. *arXiv preprint arXiv:2011.09022* (2020).