A Unified Model for Cardinality Estimation by Learning from Data and Queries via Sum-Product Networks

Abstract

Cardinality estimation is a fundamental component in database systems, crucial for generating efficient execution plans. Despite advancements in learning-based cardinality estimation, existing methods may struggle to simultaneously optimize the key criteria: estimation accuracy, inference time, and storage overhead, limiting their practical applicability in real-world database environments. This paper introduces QSPN, a *unified model* that integrates both data distribution and query workload. QSPN achieves high estimation accuracy by modeling data distribution using the simple yet effective Sum-Product Network (SPN) structure. To ensure low inference time and reduce storage overhead, QSPN further partitions columns based on query access patterns. We formalize QSPN as a tree-based structure that extends SPNs by introducing two new node types: QProduct and QSplit. This paper studies the research challenges of developing efficient algorithms for the offline construction and online computation of QSPN. We conduct extensive experiments to evaluate QSPN in both single-table and multi-table cardinality estimation settings. The experimental results have demonstrated that QSPN achieves superior and robust performance on the three key criteria, compared with state-of-the-art approaches.

ACM Reference Format:

. 2026. A Unified Model for Cardinality Estimation by Learning from Data and Queries via Sum-Product Networks. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 18 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Cardinality estimation (CardEst), which estimates the result size of an SQL query on a relational database, is a fundamental component of query optimization in database management systems (DBMSs). Traditional CardEst methods [1, 2, 4, 10, 11, 18, 24] rely on simplifying assumptions, such as column independence, often leading to substantial estimation errors. To overcome these limitations, learning-based CardEst models [6, 12, 16, 20, 40, 41, 43] have emerged as state-of-the-art solutions, significantly improving accuracy by capturing data distributions and query patterns.

Despite the advancements in learning-based CardEst models, deploying them in real-world DBMS systems still requires balancing three key criteria [9, 15, 33, 34, 43]: *estimation accuracy*, *inference time*, and *storage overhead*. Data-driven approaches [12, 40, 41, 43] leverage probabilistic models, such as Sum-Product Networks [12, 20, 43] and Deep Auto-Regressive [40, 41], to capture the joint distribution of all columns of the relational data. While these methods typically achieve high estimation accuracy, they suffer from high inference time and substantial storage overheads, particularly when handling complex data distributions. Query-driven methods [6, 16], on the other hand, train regression models that directly map SQL queries to their estimated cardinalities based on a set of training queries, bypassing the need to model data distributions. Although these methods are efficient and lightweight, they struggle with generalization, particularly when encountering queries that significantly deviate from those in the training set.

Given the strengths and limitations of existing methods, a promising direction is to develop a *unified model* that leverages both data and queries, aiming to achieve the three key criteria: *high estimation accuracy*, *low inference time*, and *lightweight storage overhead*. Such an approach can overcome the weaknesses of purely data-driven or query-driven CardEst models by leveraging complementary information from both sources.

Our Proposals. In this paper, we propose to learn from both data and queries via the Sum-Product Network (SPN) model. As shown in Figure 1(a), traditional SPN models [12, 43] recursively partition columns (*i.e.,* the Product node) and rows (*i.e.,* the Sum node) into *local subsets*, making it easier to compute the joint probability distribution from these local distributions. However, they often suffer from *high inference time* and *large model size* when columns in a database are highly correlated. This issue arises because many intermediate nodes (*e.g.,*Sum nodes) must be introduced to ensure that the columns in partitioned subsets are treated as independent.

To address this problem, we propose QSPN that extends traditional SPNs by incorporating query workload information. The high-level idea behinds QSPN stems from an observation in many real-world query workloads: queries often exhibit **specific access patterns** on the columns of relational tables, which can be effectively leveraged to enhance both the efficiency and accuracy of cardinality estimation. For instance, when retrieving movie comments by different types, production year is usually a search criteria meanwhile *i.e.,* these two columns are frequently queried together, whereas company type is seldom focused together with type.

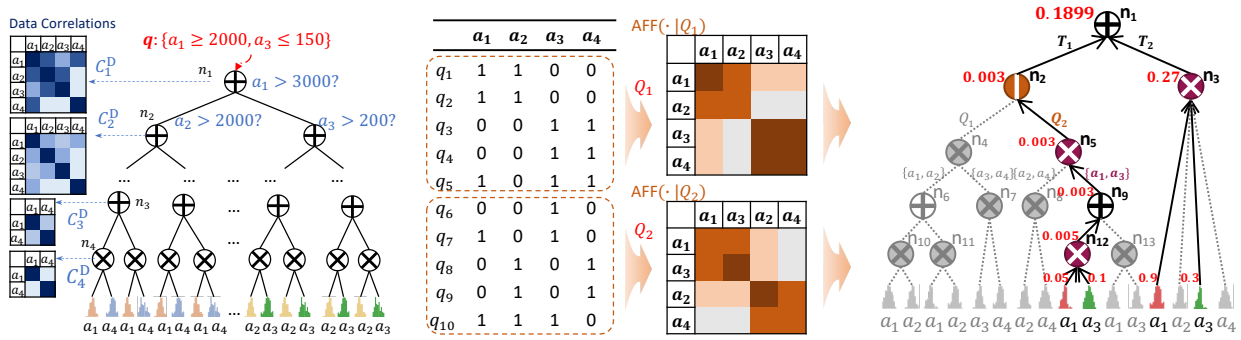
By integrating query workload information, QSPN can jointly partition columns based on both data correlations and query access patterns, thereby reducing model size and improving inference efficiency without sacrificing estimation accuracy. Table 1 presents a comparison between QSPN and existing approaches based on the three key criteria mentioned above.

Example 1. *We consider the CardEst task for an example table T with highly correlated columns a_1, a_2, a_3, a_4, as illustrated in Figure 1(a). SPN partitions T into different row subsets via* Sum *nodes (*e.g., *node n_1, which partitions rows based on whether $a_1 > 3000$) to reduce column correlations within each subset. However, as depicted in the figure , when columns exhibit high correlations, the SPN requires numerous* Sum *nodes to break down the joint distribution*

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2026/06

https://doi.org/XXXXXXX.XXXXXXX



(a) Traditional SPN Model. (b) Partitioning Columns based on Query Patterns. (c) QSPN Model and CardEst Computation.

Figure 1: High-level idea of QSPN.

Table 1: Cardinality Estimation Methods Comparison

Method	Category	Accurate Estimation	Fast Inference	Lightweight Storage
Postgress	Traditional	○	●	●
MHist	Traditional	○	○	◐
Sampling	Traditional	○	●	○
MSCN [16]	Query-Driven	○	●	◐
LW-XGB [6]	Query-Driven	○	●	●
Naru [41]	Data-Driven	●	○	◐
DeepDB [12]	Data-Driven	◐	◐	◐
FLAT [43]	Data-Driven	●	◐	○
UAE [35]	Hybrid	●	○	◐
Ours	**Hybrid**	●	●	●

into local distributions over individual columns. This leads to a substantial increase in model size. Moreover, when processing a query q, the inference procedure must traverse a large number of these nodes, significantly increasing inference time.

As illustrated in Figure 1(b), QSPN *leverages the* ***access patterns*** *of the query workload Q,* i.e., *how often certain columns are accessed together by the same queries. Within subset Q_1 of Q, we observe that columns a_1 and a_2 are frequently accessed together by queries q_1 and q_2, while a_3 and a_4 are jointly accessed by queries q_3 and q_4. Based on this pattern, we partition the columns into $\{a_1, a_2\}$ and $\{a_3, a_4\}$ for queries in Q_1, even if their data remain highly correlated. Similarly, for queries in subset Q_2, the columns can be partitioned into $\{a_1, a_3\}$ and $\{a_2, a_4\}$. These query-aware column partitions allow* QSPN *to construct more compact SPN models, reducing model size and inference time while maintaining accuracy.*

As shown in Figure 1(c), we formalize QSPN as a tree-based structure that extends traditional SPNs by introducing two new node types: QProduct and QSplit. Specifically, QProduct partitions columns based on query-specific access patterns (*e.g.,* grouping frequently accessed columns together) within a given workload, while QSplit refines the workload itself into more granular subsets to capture workload variations. Moreover, QSPN retains the SPN's ability to partition columns and rows based on data correlations. By partitioning columns based on both data correlations and query access patterns, QSPN effectively reduces the number of intermediate nodes in the SPN, which improves inference efficiency, reduces storage overhead while maintaining high estimation accuracy.

Key Challenges and Solutions. We study the technical challenges that naturally arise in our proposed QSPN approach.

Offline QSPN *Construction.* A key challenge in offline QSPN construction is integrating query workload information into the SPN framework while maintaining inference efficiency. Unlike conventional SPNs, QSPN must consider query co-access patterns, making the partitioning problem more complex. To address this, QSPN develops efficient algorithms for two core problems, *i.e.,* QProduct for query-aware column partitioning and QSplit for workload partitioning.

Online QSPN *Computation.* The online stage of QSPN presents two key challenges. First, accurately computing query cardinalities while minimizing inference overhead is non-trivial, especially for queries with unseen access patterns. Second, as data distributions and query workloads evolve, QSPN must maintain accuracy without requiring frequent full retraining. To tackle these challenges, we develop an online inference algorithm, and introduce an incremental update mechanism that enables QSPN to efficiently adapt to workload and data changes.

Multi-Table CardEst with QSPN. Extending QSPN to multi-table cardinality estimation is challenging due to the complex distributions of join keys and their impact on base table query predicates. To address this, we introduce a novel approach that allows QSPN to generalize to multi-table queries while maintaining high accuracy and efficient inference performance.

Contributions. Our contributions are summarized as follows.

(1) We propose QSPN, a query-aware Sum-Product Network that integrates data and queries for CardEst (Section 3).

(2) We develop effective algorithms for offline QSPN construction and online QSPN computation, balancing estimation accuracy, inference time, and model size (Sections 4 and 5). We extend QSPN to support multi-table cardinality estimation (Section 6).

(3) We conduct a comprehensive experimental study on widely used CardEst benchmarks. Extensive results demonstrate that QSPN achieves superior performance (Section 7).

2 Preliminaries

2.1 Problem Formalization

Data. This paper considers a relational table T with columns (or attributes) $A = \{a_1, a_2, \ldots, a_{|A|}\}$ and tuples $T = \{t_1, t_2, \ldots, t_{|T|}\}$,

where T may be either a single table or a joined table. We define the domain of each column a_i as $[LB_i, UB_i]$, where LB_i and UB_i represent the lower and upper bounds of a_i, respectively.

Queries. Similar to existing works [12, 35, 43], this paper focuses on queries that consist of a *conjunction* of predicates, where each predicate over column a_i can be represented as $L_i \le a_i \le U_i$, with $LB_i \le L_i \le U_i \le UB_i$. Without loss of generality, the endpoints of interval $[L_i, U_i]$ can also be open, *e.g.,* $(L_i, U_i]$ or $[L_i, U_i)$, which are omitted in this paper for simplicity. In particular, a point query over a_i can be represented as $L_i = U_i$. For ease of presentation, we assume each column has only one interval $[L_i, U_i]$, though this can be easily extended to the cases with multiple intervals per column.

This paper considers a *query workload* as a set of queries $Q = \{q_1, q_2, \ldots, q_{|Q|}\}$, typically extracted from real query logs. Given this workload Q, we introduce *query-column access matrix* (or *access matrix* for short) to represent the **access patterns** of queries in Q on columns in T. This matrix provides a structured way to capture which queries reference which columns, enabling more optimization opportunities in query-aware cardinality estimation.

Definition 2.1 (Query-Column Access Matrix). For each query q_i and each column a_j, we define $\texttt{acc}(q_i, a_j)$ as an indicator function that specifies whether column a_j is accessed by query q_i. Specifically, $\texttt{acc}(q_i, a_j) = 1$ if q_i accesses a_j, and $\texttt{acc}(q_i, a_j) = 0$ otherwise. Then, we define the *access matrix* for a workload Q and a column set A, denoted as $\textbf{ACC}(Q, A)$, as a binary matrix where each entry $\texttt{acc}(q_i, A_j)$ indicates whether query q_i accesses column a_j.

In particular, we use $\textbf{acc}(q_i)$ to represent the i-th row of the access matrix, corresponding to the access pattern of query q_i across all columns, and $\textbf{acc}(a_j)$ to denote the j-th column of the access matrix, capturing how column a_j is accessed by queries.

Cardinality Estimation. Given a new query q, we define $\texttt{Card}(q, T)$ as the *cardinality* of q over table T, *i.e.,* the number of tuples in T that satisfy the query conditions. The goal of *cardinality estimation* is to compute an estimate $\widehat{\texttt{Card}}(q, T)$ that approximates $\texttt{Card}(q, T)$ efficiently and accurately without executing q on T. Specifically, we study the following problems: (1) *Offline CardEst Training*, which trains a CardEst model by leveraging both data T and workload Q, capturing the underlying data distribution and query access patterns, and (2) *Online CardEst Inference*: which uses the trained CardEst model to estimate the cardinality $\widehat{\texttt{Card}}(q, T)$ for a given query q.

2.2 SUM-Product Network (SPN) Model

Sum-Product Network (SPN) [23] is a data-driven model with extensions such as DeepDB [12] and FLAT [43]. SPN-based approaches address the CardEst problem by modeling the *joint probability distribution* $P_T(A)$, where each attribute $a_i \in A$ is treated as a random variable. Given a query q with selection predicates $A_i \in [L_i, U_i]_{i=1}^m$, the estimated cardinality is computed as $\widehat{\texttt{Card}}(q, T) = |T| \cdot \sum_{v_1 \in [L_1, U_1]} \cdots \sum_{v_m \in [L_m, U_m]} P_T(v_1, \ldots, v_m)$, where the summation iterates over all possible values within the query's range constraints. SPN approximates $P_T(A)$ by *decomposing* the joint probability distribution into multiple *local probability distributions*. This decomposition is realized through a hierarchical, tree-based structure, where each node represents a *local* joint probability distribution $P_{T'}(A')$.

The key idea of SPN focuses on introducing intermediate nodes, which fall into one of the following two categories. (1) A `Sum` node partitions its tuple set T' into a collection of disjoint subsets $T' = \bigcup_i T'_i$. Each subset T'_i corresponds to a child node with a probability distribution $P_{T'_i}(A')$. The overall distribution at the `Sum` node is then computed as a weighted sum of its children's distributions $P_{T'}(A') = \sum_i w_i \cdot P_{T'_i}(A')$, where the weight w_i is determined by the proportion of tuples in each subset, given by $w_i = |T'_i|/|T'|$. (2) A `Product` node partitions its attribute set A' into disjoint subsets $A' = \bigcup_j A'_j$. Each subset A'_j corresponds to a child node that models the probability distribution $P_{T'}(A'_j)$. By assuming independence among these subsets, the overall distribution at the `Product` node is computed as $P_{T'}(A') = \prod_j P_{T'}(A'_j)$ This decomposition allows SPNs to efficiently approximate complex joint distributions by leveraging independence between attributes in a particular data subset. Given a query q for cardinality estimation, SPN estimates the cardinality $\widehat{\texttt{Card}}(q, T)$ in a bottom-up manner.

3 An Overview of QSPN

We propose `QSPN` that extends traditional SPNs by incorporating query workload information to partition columns based on their *access patterns*. We formalize `QSPN` as a tree-based structure that extends traditional SPNs by introducing two new node types, `QProduct` and `QSplit`, as shown in Figure 1(c). Formally, each node $\texttt{n}$ in `QSPN` is represented as a 4-tuple $(A_\texttt{n}, T_\texttt{n}, Q_\texttt{n}, O_\texttt{n})$, where $A_\texttt{n}$ denotes the column set, $T_\texttt{n}$ the corresponding table, and $Q_\texttt{n}$ the associated query workload. Each node captures the joint probability distribution $P_{T_\texttt{n}}(A_\texttt{n})$ conditioned on the queries in $Q_\texttt{n}$. Moreover, the node type $O_\texttt{n}$ represents how the joint probability of node $\texttt{n}$ is estimated from its child nodes, which is described as follows.

`QProduct`. A `QProduct` node, such as $\texttt{n}_5$ in Figure 1(c), partitions its column set $A_\texttt{n}$ into a set of disjoint subsets $\mathcal{A} = \{A_1, A_2, \ldots, A_m\}$, ensuring that columns in different subsets are infrequently co-accessed by queries in $Q_\texttt{n}$. For each subset A_i, a corresponding child node is created as $\texttt{n.child}_i = (A_i, T_\texttt{n}[A_i], Q_\texttt{n}[A_i], O_i)$. The joint probability distribution $P_{T_\texttt{n}}(A_\texttt{n})$ at node $\texttt{n}$ can then be computed as $P_{T_\texttt{n}}(A_\texttt{n}) = \prod_{i=1}^m P_{T_\texttt{n}}(A_i)$.

`QSplit`. A `QSplit`, such as $\texttt{n}_2$ in Figure 1(c), splits its query workload $Q_\texttt{n}$ into a set of disjoint query subsets $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_m\}$, ensuring that queries within the same subset share similar access patterns, while queries across different subsets exhibit distinct access behaviors. For each query subset Q_i, a corresponding child node is created as $\texttt{n.child}_i = (A_\texttt{n}, T_\texttt{n}, Q_i, O_i)$. Note that a `QSplit` does not directly compute a joint probability distribution $P_{T_\texttt{n}}(A_\texttt{n})$ but instead functions as a *query router*. Specifically, when estimating the cardinality of a query q, the `QSplit` identifies the query subset Q_{k^*} from $\mathcal{Q}$ that shares the most *similar access patterns* with q and routes q to the corresponding child node $\texttt{n.child}_{k^*}$ for cardinality estimation. We will discuss the method for determining whether a query subset exhibits the most *similar access patterns* with q later.

`Product`. A `Product` node, such as $\texttt{n}_3$ in Figure 1(c), partitions its column set $A_\texttt{n}$ into a set of disjoint subsets $\mathcal{A} = \{A_1, A_2, \ldots, A_m\}$,

ensuring statistical independence between columns in different subsets. For each subset A_i, a corresponding child node is created as $\mathsf{n.child}_i = (A_i, T_\mathsf{n}, Q_\mathsf{n}, O_i)$. The joint distribution $P_{T_\mathsf{n}}(A_\mathsf{n})$ is then computed using the equation in Section 2.2.

Sum. A Sum node, such as n_1 in Figure 1(c), partitions the table T_n of node n into disjoint subsets $\mathcal{T} = \{T_1, T_2, \ldots, T_m\}$. For each subset T_i, a corresponding child node is created as $\mathsf{n.child}_i = (A_\mathsf{n}, T_i, Q_\mathsf{n}, O_i)$. The joint distribution $P_{T_\mathsf{n}}(A_\mathsf{n})$ is then computed as a weighted sum of the distributions of its child nodes, as defined in Section 2.2.

Leaf. A Leaf node, such as any leaf in the tree shown in Figure 1(c), represents the 1-dimensional probability distribution $P_{T_\mathsf{n}}(A_\mathsf{n})$. Specifically, we use a histogram-based mechanism to capture this probability distribution. The construction cost of the histogram is $O(|T_\mathsf{n}|)$, and the query cost is approximately $O(1)$.

To support the above QSPN structure, in this paper, we introduce a framework that consists of both offline and online stages.

Offline QSPN Construction. In the offline stage, QSPN learns its structure from both data T and workload Q, capturing the underlying data distribution and query access patterns. Unlike conventional SPNs, QSPN introduces a new challenge of incorporating query co-access patterns into column partitioning. We propose efficient algorithms QProduct and QSplit, as presented in Section 4.

Online QSPN Computation. In the online stage, we utilize QSPN for cardinality estimation. Specifically, for queries with access patterns that differ from those in the training workload, we develop an online inference algorithm, ensuring both accuracy and efficiency. Second, we introduce an incremental update mechanism that selectively identifies and updates only the affected parts of the model. Details of online QSPN inference are provided in Section 5.

Multi-Table Cardinality Estimation. This paper also explores extending QSPN to support multi-table CardEst. The key difficulty lies in accurately modeling join key distributions while effectively handling multi-table query predicates. Traditional methods rely on heuristic bucket-based approaches, which suffer from poor accuracy. Learning-based CardEst models [40, 43], on the other hand, train on a materialized outer-join table, incurs excessive time and storage costs. To tackle this, we introduce an effective approach, which is described in Section 6.

4 Offline QSPN Construction

Given a relational table T with column set A and a query workload Q, QSPN construction generates a QSPN tree that models the joint probability distribution $P_T(A)$ conditioned on Q. To achieve this, the construction process recursively decomposes the joint probability distribution into local probability distributions in a top-down manner. Specifically, during the construction of each node, QSPN attempts different node types in the following order: Leaf, Product, QProduct, QSplit, and Sum.

Construction of Leaf Nodes. During the recursive process, if A contains only a single column, this indicates that the joint probability distribution has been fully decomposed into a local distribution over the specific column. In this case, the construction process creates a Leaf to model the 1-dimensional probability distribution $P_{T_n}(A)$ using a histogram. This choice is motivated by the histogram's accuracy, efficiency, and lightweight nature, making it well-suited for modeling such distributions.

Construction of Product Nodes. A Product node is constructed when the column set A of a node exhibits statistical dependencies suitable for partitioning. Following prior works [12, 23, 43], we use the *Randomized Dependence Coefficient* (RDC) to measure statistical dependencies between columns. Details on RDC can be found in the original paper [19]. Then, we employ a partition-based method. Specifically, using these RDC values, we construct a graph where vertices represent columns and edges represent dependencies weighted by the RDC values. Then, we remove the edges with RDC values below a threshold, and divide the graph into connected components, each representing an independent subset of columns.

Construction of Sum Nodes. During the recursive process, if all other types of nodes fail to meet the decomposition criteria, node n defaults to a Sum, which splits the data T into subsets T_i. To achieve this, following prior works [12, 43], we use the K-Means clustering algorithm, as it partitions the data into clusters, which helps to reduce data correlation within each subset.

4.1 Construction of QProduct

QProduct partitions columns according to their query access patterns, grouping frequently co-accessed columns together while separating those that are rarely co-accessed into distinct subsets.

Formalization of QProduct. We first formally define *access affinity* using the query-column access matrix as follows.

Definition 4.1 (Access Affinity). The *Access Affinity* between columns a_i and a_j with respect to query workload Q, denoted by $\mathtt{AFF}(a_i, a_j|Q)$, is defined as how frequently both columns are referenced together by queries in Q, *i.e.,*

$$\mathtt{AFF}(a_i, a_j|Q) = \mathbf{acc}(a_i) \cdot \mathbf{acc}(a_j). \tag{1}$$

Using access affinity, we formally define the QProduct operation.

*Definition 4.2 (*QProduct*).* Given a query workload Q and a column set $A = \{a_1, a_2, \ldots, a_{|A|}\}$, QProduct partitions A into a set of disjoint column subsets $\mathcal{A} = \{A_1, A_2, \ldots, A_m\}$, minimizing the *inter-partition affinity* (*IPA*) of partitioning $\mathtt{IPA}(\mathcal{A}|Q)$, where

$$\mathtt{IPA}(\mathcal{A}|Q) = \sum_{1 \le k < l \le m} \sum_{a_i \in A_k} \sum_{a_j \in A_l} \mathtt{AFF}(a_i, a_j|Q). \tag{2}$$

For example, consider the workload Q_1 in Figure 1(b). The inter-partition affinity for the column partitioning $\{\{a_1, a_2\}, \{a_3, a_4\}\}$ is 0, whereas for the partitioning $\{\{a_1, a_3\}, \{a_2, a_4\}\}$ it is 4. Based on these results, QProduct selects the partition $\{\{a_1, a_2\}, \{a_3, a_4\}\}$ for workload Q_1 to minimize inter-partition affinity.

Algorithm Design for QProduct. We first analyze the complexity of the QProduct construction problem, as shown below.

Lemma 1. *The problem of* QProduct *construction is equivalent to the minimum k-cut problem.*

We omit the proof due to the space constraint. The minimum k-cut problem, even for fixed k, is computationally expensive to solve. We abstract each column to a vertice. For example, the minimum 2-cut (*i.e.,* the classic min-cut problem) can be solved in $O(|A|^3)$ time using algorithms such as Stoer-Wagner [30]. For $k = 3$, the

time complexity is $O(|A|^3\tau(|A|))$ [22], where $\tau(|A|)$ represents the cost of computing the objective function, which is $O(|A|^2)$ in our QProduct construction problem, yielding an overall complexity of $O(|A|^5)$. Similarly, a recent algorithm for $k = 4$ achieves a complexity of $O(|A|^6\tau(|A|))$ (or $O(|A|^8)$). While the problem is polynomial-time solvable for fixed $k \geq 5$, the complexity increases dramatically (*e.g.,* $O(|A|^{16})$ for a minimum 5-cut algorithm as suggested by [13]), rendering such algorithms impractical for real-world use.

Given the computational expense of partitioning the column set to minimize IPA, we design an algorithm **PartitionByAFF** that achieves effective and efficient ($O(|A|^2)$) results. The algorithm first constructs a graph $G = (A, E)$, where vertices A represent columns, and an edge $e_{ij} \in E$ connects columns a_i and a_j if their affinity $\mathtt{AFF}(a_i, a_j|Q)$ exceeds a threshold τ. Next, the algorithm identifies the connected components of G, with the vertices in each connected component $G_i \subseteq G$ forming a column partition A_i.

4.2 Construction of QSplit

When considering the entire workload Q, QProduct may struggle to derive a meaningful column partitioning with a sufficiently low IPA score due to the presence of queries exhibiting diverse access patterns. The following example illustrates this challenge.

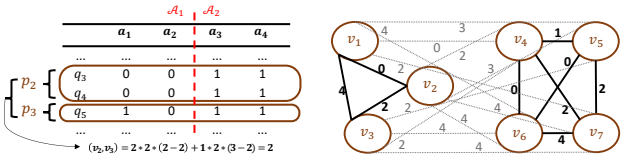
Example 2. *Given the workload $Q = \{q_1, q_2, \ldots, q_{10}\}$, as shown in Figure 1(b), an optimal column partitioning for* QProduct *with $k = 2$ results in $\{\{a_1, a_2, a_3\}, \{a_4\}\}$, yielding a minimum IPA score of 6, which remains relatively high. The primary reason for this is that different subsets of queries within Q exhibit different access patterns, leading to conflicting preferences for column partitioning. Specifically, queries in $Q_1 \subset Q$ favor the column partitioning $\{\{a_1, a_2\}, \{a_3, a_4\}\}$, while those in $Q_2 \subset Q$ prefer $\{\{a_1, a_3\}, \{a_2, a_4\}\}$. These conflicting preferences arise due to the distinct access patterns exhibited by queries in different subsets.*

Formalization of QSplit. To address the above challenge, we introduce the QSplit operation, which partitions the query workload into n subsets, *i.e.,* $\mathcal{Q} = \{Q_1, \ldots, Q_n\}$, ensuring that each subset exhibits more consistent access patterns, thus enabling QProduct to derive more meaningful column partitions.

*Definition 4.3 (*QSplit*).* Given a query workload Q and a column set $A = \{a_1, a_2, \ldots, a_{|A|}\}$, QSplit aims to partition Q into a set of disjoint query subsets $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_n\}$, minimizing the objective $\sum_{k=1}^{n} \mathtt{IPA}(\mathcal{A}_k^*|Q_k)$, where $\mathcal{A}_k^*$ is the *optimal* column partition given query subset Q_k.

For example, by splitting the workload Q in Figure 1(b) into two subsets ($n = 2$), an effective partitioning results in $Q_1 = \{q_1, \ldots, q_5\}$ and $Q_2 = \{q_6, \ldots, q_{10}\}$. Given these subsets, the optimal QProduct column partitioning for Q_1 is $\mathcal{A}_1^* = \{\{a_1, a_2\}, \{a_3, a_4\}\}$ with $\mathtt{IPA}(\mathcal{A}_1^*|Q_1) = 2$. Similarly, for Q_2, the optimal partitioning is $\mathcal{A}_2^* = \{\{a_1, a_3\}, \{a_2, a_4\}\}$ with $\mathtt{IPA}(\mathcal{A}_2^*|Q_2) = 2$.

Algorithm Design for QSplit. The QSplit construction problem is highly challenging because it requires simultaneously minimizing $\mathtt{IPA}(\mathcal{A}_k^*|Q_k)$ for each partitioned subset Q_k. Thus, instead of directly optimizing $\mathtt{IPA}(\mathcal{A}_k^*|Q_k)$ for each Q_k, we focus on minimizing its upper bound, denoted as $\overline{\mathtt{IPA}}(\mathcal{A}_k|Q_k)$, which allows us to design a more tractable algorithm while still ensuring effective workload partitioning. Formally, the objective is to partition Q into $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_n\}$ while minimizing the upper bound of inter-partition affinity, *i.e.,* $\sum_{k=1}^{n} \overline{\mathtt{IPA}}(\mathcal{A}_k|Q_k)$.



(a) Upper bound design. (b) Our greedy algorithm.

Figure 2: An example of QSplit construction.

Next, we first design a mechanism to construct the upper bound $\overline{\mathtt{IPA}}(\mathcal{A}_k|Q_k)$ and prove that optimizing this upper bound is NP-hard. Finally, we propose an efficient heuristic algorithm that achieves effective partitioning results in practice.

Upper-bound design. Given a specific query set Q, we construct an upper bound $\overline{\mathtt{IPA}}(\mathcal{A}|Q)$ for the optimal column partitioning result $\mathtt{IPA}(\mathcal{A}^*|Q)$ by analyzing the access patterns of queries in Q. Since different queries (*e.g.,* q_1 and q_2 in Figure 1(b)) may share the same access pattern (e.g., $(1, 1, 0, 0)$), we introduce p_i to denote the i-th distinct access pattern in Q and let n_i represent the number of queries in Q that follow pattern p_I. For instance, in Figure 1(b), the first access pattern in Q is $p_1 = (1, 1, 0, 0)$, which corresponds to two queries, *i.e.,* $n_1 = 2$. Based on these notations, we construct the following upper bound for $\mathtt{IPA}(\mathcal{A}^*|Q)$.

Definition 4.4 (Upper Bound). Given a query set Q consisting of m distinct query patterns $\{p_1, p_2, \ldots, p_m\}$, let $\|p_i\|$ denote the L_2 norm of pattern p_i, n_i) represent the number of queries in Q corresponding to pattern p_i, and z_{ij} denote the dot product of two patterns p_i and p_j. We construct an upper bound for $\mathtt{IPA}(\mathcal{A}^*|Q)$ as

$$\overline{\mathtt{IPA}}(\mathcal{A}|Q) = \sum_{i<j} \left(n_i \cdot z_{ij} \cdot (\|p_i\| - z_{ij}) + n_j \cdot z_{ij} \cdot (\|p_j\| - z_{ij})\right).$$

Consider a query set Q with two query patterns, p_2 and p_3, as illustrated in Figure 2(a). To compute the upper bound $\overline{\mathtt{IPA}}(\mathcal{A}|Q)$, we first calculate the dot product $z_{23} = p_2 \cdot p_3 = 2$, as well as the norms $\|p_2\| = 2$ and $\|p_3\| = 3$. Given that $n_2 = 2$ and $n_3 = 1$, the contribution of patterns p_2 and p_3 to the overall upper bound is computed as: $n_2 \cdot z_{23} \cdot (\|p_2\| - z_{23}) + n_3 \cdot z_{23} \cdot (\|p_3\| - z_{23}) = 2$. As shown in Figure 2(a), the upper bound corresponds to a specific column partitioning strategy, *i.e.,* grouping columns that are co-accessed by multiple query patterns (*e.g.,* A_3 and A_2) while placing the remaining columns (*e.g.,* A_1 and A_4) in a separate group.

Lemma 2. *For any given query set Q, the upper bound $\overline{\mathtt{IPA}}(\mathcal{A}|Q)$ provides an upper estimate of the optimal result $\mathtt{IPA}(\mathcal{A}^*|Q)$.*

Due to the space limit, we omit the proof in this paper.

Hardness of upper-bound optimization. Next, we show that even optimizing the upper bound $\overline{\mathtt{IPA}}(\mathcal{A}|Q)$ is theoretically intractable.

Lemma 3. *The problem of partitioning Q into $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_n\}$ to minimize the upper bound of inter-partition affinity,* i.e.,, $\sum_{k=1}^{n} \overline{\mathtt{IPA}}(\mathcal{A}_k|Q_k)$, *is NP-hard.*

We prove this lemma via a reduction from the Max-Cut problem, which is NP-hard. Due to space constraints, we omit the proof.

A greedy algorithm for QSplit. Given the time complexity of solving our problem, we design a practical and efficient heuristic algorithm. The algorithm first constructs a graph $G = (V, E)$, where each vertex $v_i \in V$ represents a query pattern p_i, and an edge $e_{ij} \in E$ connects two patterns p_i and p_j with a weight of $n_i \cdot z_{ij} \cdot (\|p_i\| - z_{ij}) + n_j \cdot z_{ij} \cdot (\|p_j\| - z_{ij})$. Next, based on the constructed graph G, the algorithm aims to partition it into n sub-graphs, denoted as $\{G_1, G_2, \ldots, G_n\}$, such that the total weight of the edges *across* the sub-graphs is maximized. To achieve this, it first sorts all vertices in G based on their weighted degree in descending order and initializes n empty sub-graphs. Then, the algorithm iteratively processes each vertex and assigns it to an *appropriate* sub-graph. Specifically, in the i-th iteration, the algorithm considers vertex v_i and evaluates its potential assignment to each sub-graph G_j by computing the total edge weight of G'_j after incorporating v_i. The vertex v_i is then assigned to the sub-graph that results in the minimum weight summation. Figure 2(b) illustrates an example of our greedy algorithm partitioning the query set Q from Figure 1(b) into two subsets (*i.e.,* $m = 2$).

5 Online QSPN Computation

5.1 CardEst Inference with QSPN

When a new query q arrives, the Online CardEst Inference process using QSPN operates recursively by traversing the QSPN tree from the root to the leaf nodes, as shown in Figure 1(c). Specifically, during traversal, if a QSplit is visited, the process routes query q to the child node corresponding to the most relevant query subset. On the other hand, if a QProduct, Product, or Sum is visited, the process computes the joint probability accordingly as follows.

(1) For a Leaf node n, the algorithm computes the probability $\mathsf{n}.P_T(q)$ based on the histogram at leaf n.

(2) For a QProduct or Product node, n, the algorithm recursively invokes the CardEst inference process for each child of n where $q.A \cap \mathsf{n.child}_i.A \neq \emptyset$. It then multiplies the estimated probabilities of all the relevant child nodes to produce the estimation result. For example, at QProduct node n_5, the column set is partitioned into a_1, a_3 and a_2, a_4. Since q involves only a_1, a_3, the estimation continues with node n_9, while node n_8 is pruned.

(3) For a Sum node, n, the algorithm computes the weighted sum of the estimated probabilities of its child nodes.

Next, we explore two key implementation details of the CardEst Inference process: (1) query routing in QSplit nodes, and (2) subtree pruning in Sum and Product nodes.

Query Routing in QSplit Nodes. The objective of query routing in a QSplit node (say n_2 in Figure 1(c)) is to measure the degree to which a query set Q shares *similar access patterns* with a given query q, which guides the QSplit node in routing q to the appropriate child node (*e.g.,* node n_5). To this end, we formally introduce a matching score $\mathsf{S}(Q, q)$ between a query set Q and a query q:

$$\mathsf{S}(Q, q) = \frac{1}{|Q|} \sum_{(a_i, a_j), i<j} \mathsf{AFF}(a_i, a_j|Q) \cdot \mathsf{AFF}(a_i, a_j|\{q\}) \quad (3)$$

The intuition behind the matching score $\mathsf{S}(Q, q)$ is to measure how closely the access patterns of queries in Q align with the access pattern of q. For example, consider the query q executed in Figure 1(c). The access pattern for q is represented as:

$$\mathsf{AFF}(\cdot, \cdot|\{q\}) = [[1, 0, 1, 0], [0, 0, 0, 0], [1, 0, 1, 0], [0, 0, 0, 0]].$$

Considering the workload partitioning in Figure 1(b), we have $\mathsf{AFF}(\cdot, \cdot|Q_1) = [[3, 2, 1, 1], [2, 2, 0, 0], [1, 0, 3, 3], [1, 0, 3, 3]]$ and $\mathsf{AFF}(\cdot, \cdot|Q_2) = [[2, 1, 2, 0], [1, 3, 1, 2], [2, 1, 3, 0], [0, 2, 0, 2]]$ for Q_1 and Q_2, respectively. Based on these, we compute $\mathsf{S}(Q_1, q) = \frac{1}{5}$ and $\mathsf{S}(Q_2, q) = \frac{2}{5}$. Therefore, when the node n_1 is visited, the algorithm routes query q to its child node n_3, as Q_2, corresponding to n_3, shares more similar access patterns with q than Q_1.

Pruning Rules in Sum and Product Nodes. We employ pruning rules that leverage query q and pre-computed metadata to exclude irrelevant child nodes of a given node n. (1) For Product and QProduct nodes, let A_q denote the set of columns constrained by the query q. For a child node $\mathsf{n.child}_k$ of n, corresponding to the column subset A_k, pruning occurs if $A_k \cap A_q = \emptyset$. (2) For Sum nodes, it decides which child nodes contributes to $n.P_T(q)$ so that participate the computation *i.e.,* set of visited child nods C. Consider a child node $\mathsf{n.child}_k$ of n, corresponding to the table subset T_k. Before query processing, we pre-compute and store the range of values for each column A_i in T_k. During cardinality estimation, if the value ranges of T_k for any column do not overlap with the constraints specified by query q, the child node $\mathsf{n.child}_k$ can be safely pruned, as it does not contribute to the result.

5.2 QSPN Model Update

Data updates (ΔT, which include new tuples) and query workload shifts (ΔQ, which include new queries) impact the accuracy and inference efficiency of the original QSPN model. The high-level idea of the update method is to traverse QSPN in a top-down manner. Each time a node n is visited during the traversal, two steps are performed to update the subtree rooted at n. First, the method examines whether n, originally constructed using $\mathsf{n}.T$ and $\mathsf{n}.Q$, still fits $\mathsf{n}.T \cup \Delta T$ or $\mathsf{n}.Q \cup \Delta Q$. Second, if n no longer fits, the subtree rooted at n is reconstructed by calling the QSPN construction method (see Section 4), which generates a new subtree rooted at n'. Otherwise, each child node $\mathsf{n}.child_i$ is recursively updated. Note that the check and reconstruction steps are unnecessary if n is a Leaf, as histograms can be incrementally updated in $O(|\Delta T|)$.

The key challenge in the above update method is efficiently examining whether a node n still fits $\mathsf{n}.T \cup \Delta T$ or $\mathsf{n}.Q \cup \Delta Q$, as the corresponding data table $\mathsf{n}.T$ and query workload $\mathsf{n}.Q$ are not materialized at node n. To address this challenge, we maintain lightweight data structures in different types of nodes and design a mechanism to check whether node n is still up-to-date with respect to the data and query workload, as described below.

(1) If n is a Product node, we examine whether the column partitioning still holds, *i.e.,* whether columns in different partitions remain independent with respect to the updated data table $T \cup \Delta T$. To do this, we first compute $\mathsf{RDC}(a_i, a_j|\Delta T)$, where $a_i, a_j \in \mathsf{n}.A$, and then check whether there exist a_i, a_j from different child nodes, say a_i from $\mathsf{n}.child_k$ and a_j from $\mathsf{n}.child_l$ such that $\frac{|\mathsf{n}.T|}{|\mathsf{n}.T|+|\Delta T|}\mathsf{RDC}(a_i, a_j|\mathsf{n}.T) + \frac{|\Delta T|}{|\mathsf{n}.T|+|\Delta T|}\mathsf{RDC}(a_i, a_j|\Delta T)$ is larger

than a pre-defined threshold. If any such pair a_i, a_j is found, the subtree rooted at n needs to be reconstructed for more accurate CardEst. If the RDC between columns within a child node becomes less significant due to ΔT, we may reconstruct the subtree rooted at the child node to further partition the now-independent columns.

(2) If n is a QProduct node, the update examination strategy is similar to that of the Product case, except that we consider the access affinity $\texttt{AFF}(a_i, a_j|\texttt{n}.Q)$ for any column pair (a_i, a_j), instead of the correlation $\texttt{RDC}(a_i, a_j|\texttt{n}.T)$.

(3) If n is a QSplit, we examine whether the query routing strategy still holds for the updated workload $Q \cup \Delta Q$. To do this, for each workload partition Q_k corresponding to the child node $\texttt{n}.child_k$, we maintain the average of the matching scores of the queries in Q routed to Q_k, i.e., $\sum_{q \in Q_k} \texttt{S}(Q_k, q)/|Q_k|$. Then, for each query q' in the updated workload ΔQ, we assign q' to the child node with the maximum matching score (see Section 5.1) and update the average matching score accordingly. If the average matching score of any workload partition becomes less significant, *e.g.,* less than a predefined threshold, we reconstruct the subtree rooted at the QSplit node n, as the workload partition no longer reflects the access patterns of $Q \cup \Delta Q$.

(4) If n is a Sum node, we maintain the centroid for each tuple subset T_i from $\mathcal{T} = \{T_1, T_2, \ldots, T_m\}$ and the average distance between each tuple and the centroid of its assigned subset. Then, for each new tuple t in ΔT, we assign t to the tuple subset with the minimum distance to the centroid of that subset and update the average distance accordingly. If the average distance becomes significant, *e.g.,* exceeding a predefined threshold, we update the subtree rooted at n, as the Sum may no longer hold for $\texttt{n}.T \cup \Delta T$.

In this way, the QSPN update method minimizes unnecessary costs associated with model updates while ensuring accuracy in response to data updates and query workload shifts.

6 Multi-Table CardEst with QSPN

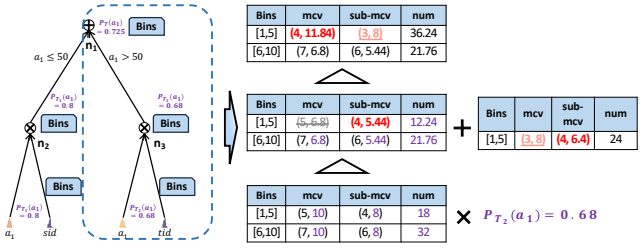
We introduce M-QSPN, a multi-table CardEst method built upon QSPN, as illustrated in Figure 3. For clarity, this section only considers a query q that performs an inner join between two tables S and T on the condition $S.sid = T.tid$, denoted as $q(S \bowtie T)$. The query also includes base table filter predicates $q(S)$ and $q(T)$. We assume that $S.sid$ and $T.tid$ share the same value domain D. Under this setting, the multi-table CardEst problem can be formalized as estimating the cardinality $|q(S \bowtie T)|$, which is computed as:

$$|q(S \bowtie T)| = |S||T| \sum_{v \in D} P(sid = v \wedge q(S)) \cdot P(tid = v \wedge q(T)), \quad (4)$$

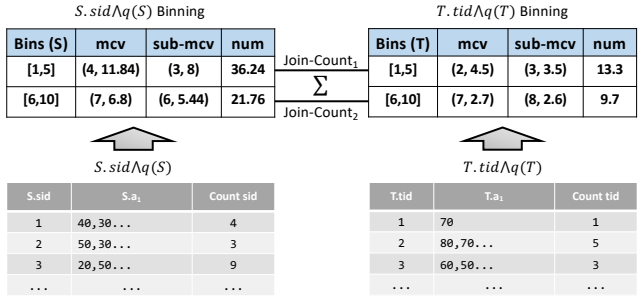
where $P(sid = v \wedge q(S))$ (or $P(tid = v \wedge q(T))$) denotes the probability that the join key $S.sid$ (or $T.tid$) equals v in the result table of the base filter predicates $q(S)$ (or $q(T)$).

As directly estimating the cardinality $|q(S \bowtie T)|$ using Equation (4) is computationally expensive, M-QSPN supports multi-table CardEst by *binning join keys*. Specifically, we divide the domain of the join keys (*sid* and *tid*) into a set of **bins** $\mathcal{B} = \{B_1, B_2, \ldots, B_n\}$. We then estimate $|q(S \bowtie T)|$ using the bins, *i.e.,*

$$|q(S \bowtie T)| = |S||T| \sum_{B \in \mathcal{B}} \sum_{v \in B} \{P(sid = v \wedge q(S)) \cdot P(tid = v \wedge q(T))\}.$$



(a) Binning Join-Key by QSPN Model on a Single Table.



(b) Multi-Table CardEst using Join-Key Binning

Figure 3: Illustration of our proposed M-QSPN method.

Then, the task is to estimate $\sum_{v \in B} P(sid = v \wedge q(S)) \cdot P(tid = v \wedge q(T))$ for each bin $B \in \mathcal{B}$. To achieve this, we propose maintaining basic statistics for each bin B of values. Formally, we define a bin for a join key, say $S.sid$, as $B = (\texttt{range}, \texttt{num}, \texttt{mcv}, \texttt{sub-mcv})$, where range is the identifier and domain of the bin B, num is the (estimated) number of tuples in B, mcv is the (estimated) frequency of most common value in B and sub-mcv is similar with mcv except for representing frequency of the second most common value.

Example 3. *Figure 3(b) shows an example of the aforementioned range-based binning: the bin B_1^S for sid, corresponding to range $[1, 5]$, has* num = 36.24, mcv = 11.84 *with value 4 and* sub-mcv = 8 *with value 3. Similarly, we can compute the corresponding bin B_1^T for the query result $q(T)$ over table T. Then, we can estimate $\sum_{v \in B_1^S} \{P(sid = v \wedge q(S)) \cdot P(tid = v \wedge q(T))\}$ based on bins B_1^S and B_1^T using the estimation technique detailed later.*

In this section, we address two challenges in the above estimation process. First, while it is straightforward to compute statistics for a given bin B over a join key, the task becomes more complex when considering the base table predicates, such as $q(S)$, because these predicates may have intricate correlations with the join keys. The second challenge lies in estimating $\sum_{v \in B} \{P(sid = v \wedge q(S)) \cdot P(tid = v \wedge q(T))\}$ based on the generated bins from $q(S)$ and $q(T)$ respectively. The following two subsections present our approach to addressing these challenges.

6.1 Binning Generation

Computations on Nodes. To tackle the first challenge, we introduce a *binning generation* method based on our single-table QSPN model. Algorithm 4.1 presents the pseudo-code for the Binning Generation algorithm.

Algorithm 4.1 `M-QSPN-BinningGen`

Input: n: A M-QSPN Node on Table T; q: A Query; J: Set of Join Keys
Output: B^T: Table T Binning, P_{n}: Probability Estimated at Node n

Compute $\mathsf{n}.P_T(q)$ in the same way of **`QSPN-Online`** (n, q)
if $J \cap \mathsf{n}.A \neq \emptyset$ **then**
  if $\mathsf{n}.O$ = `Leaf` **then**
    $B^T \leftarrow$ Generate q-filtered by bins on n
  else if $\mathsf{n}.O$ = `Product` $\vee$ $\mathsf{n}.O$ = `QProduct` **then**
    $B^T \leftarrow$ Tag $\sum_i P_i$ from n.`child` returning probabilities on n.`child` returning bins
  else if $\mathsf{n}.O$ = `Sum` **then**
    $B^T \leftarrow$ Create Linked-List of bins from n.`child`
  else if $\mathsf{n}.O$ = `QSplit` **then**
    $B^T \leftarrow$ bins from the q-route n.`child`$_i$
  end if
  if n is root **then**
    $B^T \leftarrow$ **`BinningFnlProc`**(B^T)
  else if $\mathsf{n}.O$ = `Product` $\wedge$ $|J \cap \mathsf{n}.A| > 1$ **then**
    $B^T \leftarrow$ **`BinningFnlProc`**(B^T)
    $B^T \leftarrow$ **`CrossCon`**(B^T)
  end if
else
  $B^T \leftarrow \emptyset$
end if
return $B^T, \mathsf{n}.P_T(q)$

Specifically, given base table predicates, such as $q(S)$ over table S, Binning Generation generates statistics for each bin $B \in \mathcal{B}$ corresponding to the data table that satisfies $q(S)$. To account for the intricate correlations between the predicate $q(S)$ and the join key, such as *sid*, this paper proposes a bottom-up traversal of the constructed QSPN of table S, as illustrated in Figure 3(a). Specifically, during the traversal, when a particular node n is visited, the key task is to generate a set of bins $\mathcal{B}_{\mathsf{n}}$ based on the bins of n's child nodes, and then return $\mathcal{B}_{\mathsf{n}}$ to n's parent node. To this end, we propose bin generation strategies tailored to different node types.

(1) If n is a `Leaf` corresponding to the join key, say *sid*, we directly return the pre-generated bins $\mathcal{B}_{\mathsf{n}}$ for the join key, as shown in Figure 3(a). If query $q(S)$ over table S includes predicates on *sid*, we filter the bins to retain only those that satisfy the predicates.

(2) If n is `Product` or `QProduct`, at most one child node of n, say n′, will return the bins $\mathcal{B}_{\mathsf{n}'}$ corresponding to its subtree, while the other child nodes return the estimated probabilities P_i based on the predicates in $q(S)$. Thus, we scale the `num` and `mcv` in each bin $B \in \mathcal{B}_{\mathsf{n}'}$ by a factor of $\prod_i P_i$. Take *e.g.,* n_3 in Figure 3(a) as an example: the bins from one of n_3.`child` are scaled by the probability $P_{T_2}(a_1) = 0.68$ from the other child so that the num, `mcv` and `sub-mcv` in all the bins are multiplied with 0.68. The scaling is reasonable because the child nodes of n are statistically independent or seldom co-accessed by queries.

(3) If n is a `Sum` and its column set A_{n} contains the join key (*e.g., sid*), then all child nodes of n return their respective bins, such as n_1 in Figure 3(a). In this case, we merge the bins from all child nodes. Specifically, if multiple bins cover the same value range (*e.g.,* $[1, 5]$), we aggregate their num values through summation. For the `mcv` (most common value) and `sub-mcv` (second most common value), we proceed as follows: if these `mcv` or `sub-mcv` share the same value in multiple bins, we sum them. If the values differ across bins, we compare all combinations of `mcv` and `sub-mcv` to identify the new top two most frequent values, which become the `mcv` and `sub-mcv` of the merged bin. For example, in n_1, bins from n_2 and n_3 both cover the range $[1, 5]$. Their original `sub-mcv` share the same value 4, *i.e.,* $(4, 5.44)$ and $(4, 6.4)$, which are combined to form the new `mcv` = 11.84 of 4. Meanwhile, the original `mcv` $(3, 8)$ from n_2 becomes the new `sub-mcv`.

(4) If n is a `QSplit` node, we can simply return the bins from its child node to which query $q(A)$ is routed.

Optimizations: Lazy Propagation. In practice, QSPN trees are significantly more complex than the example shown in Figure 3(a), which can lead to inefficiencies due to redundant operations. This issue primarily arises in `Sum` and `Product` nodes. When a bin passes through multiple `Sum` nodes, its `range` and associated statistics may be repeatedly compared and merged. Similarly, when traversing multiple `Product` nodes, the bin's num, `mcv`, and `sub-mcv` values may undergo repeated scaling. Since these operations are mathematically additive or multiplicative, we introduce a *Lazy Propagation* mechanism that defers and merges such operations.

(1) If n is a `Sum`, we use a linked list to maintain all bins from n.*child* and return it as the result of n. Note that this operation only stores pointers to the bins, and no bin is actually modified.

(2) If n is a `Product`, we only maintain a tag equal to the scaling factor $\prod_i P_i$, indicating that all returned bins (or linked lists of bins) should be scaled by this tag.

When the result is returned from the root of the QSPN, we obtain a recursively nested tree of linked lists containing bins and associated tags. We first traverse this tree to scale each bin using the accumulated tag values, where each bin is updated exactly once during this pass. This results in several sorted linked lists of scaled bins, as each list originates from a `Leaf`. We then apply a linear multi-way merge to produce the final bin for the join key. We refer to this procedure as the Final Process of Binning (**`BinningFnlProc`**) in the *Lazy Propagation* Optimization. Note that this method reduces the maximum number of modifications per bin from the depth of QSPN to two: one at the `Leaf` and one during **`BinningFnlProc`**.

Multiple Tables and Join Keys. The binning generation method described above can be extended to support multiple inner join conditions across multiple tables, such as $X \bowtie Y \bowtie Z$ with $(X.xid = Y.yid1) \wedge (Y.yid2 = Z.zid)$. The key difference lies in handling `Product` nodes, which must concatenate bins across different join keys. For instance, consider two join keys, $yid1$ and $yid2$, with bins B_1^{yid1}, B_2^{yid1} covering ranges $[0, 9]$, $[10, 19]$, and B_1^{yid2}, B_2^{yid2} covering $[20, 29]$, $[30, 39]$, respectively. At the `Product` n, we compute the Cartesian product of these bins, yielding a new bin set $\mathcal{B}$ with bins $B_{1,1}^{yid1,yid2}$, $B_{1,2}^{yid1,yid2}$, $B_{2,1}^{yid1,yid2}$, and $B_{2,2}^{yid1,yid2}$, corresponding to value ranges $([0, 9], [20, 29])$, $([0, 9], [30, 39])$, $([10, 19], [20, 29])$, and $([10, 19], [30, 39])$, respectively. This expansion assumes independence between join keys, consistent with the semantics of `Product`. For each value combination of join keys, we estimate the corresponding statistics under the assumption of independence. Specifically, suppose that we have b bins $B^1, B^2, \ldots, B^b$

on b different join keys on node n and we concatenate the b bins to a new bin B'. We compute $B'.\texttt{range} = (B^1.\texttt{range}, \ldots, B^b.\texttt{range})$ and $B'.\texttt{num} = |T_\texttt{n}| \prod_{1 \le i \le b} \frac{B^i.\texttt{num}}{|T_\texttt{n}|}$ according to the assumption of independence. The similar approach applies to $B'.\texttt{mcv}$ and $B'.\texttt{sub-mcv}$. We refer to this procedure as Crossing Concatenating (**CrossCon**), which is called on Product nodes with multiple child nodes returning bins on different join keys. Since **CrossCon** may produce a large number of bins, we apply scaling and invoke **BinningFnlProc** early at the Product to reduce computational overhead. Overall, combining **CrossCon** and **BinningFnlProc**, each bin is modified at most |join keys| + 1 times, which is less than the depth of QSPN.

6.2 Multi-Table CardEst based on Binning

To address the second challenge, we propose an effective method to estimate $\sum_{v \in B} \{P(sid = v \wedge q(S)) \cdot P(tid = v \wedge q(T))\}$ for each "matched" bin B, which is shared by both table S with predicates $q(S)$ and table T with predicates $q(T)$. For simplicity, we use B^S and B^T to denote the matched bins, *i.e.,* bins with the same range of join key values. Specifically, this method considers three cases.

(1) The first case is that both mcv and sub-mcv are dominant in their respective bins and share the same value. As shown in Figure 3(b), the bins $[6, 10]$ of B^S and B^T share the same mcv value, and the bins $[1, 5]$ share the same sub-mcv. In such cases, we directly compute their contribution to the total CardEst as 6.8×2.7 and 8×3.5, respectively. This approach applies to all matching combinations of mcv and sub-mcv (*e.g.,* $B^S.\texttt{mcv}$ matching $B^T.\texttt{sub-mcv}$).

(2) The second case is that mcv or sub-mcv match but are not dominant. In this case, other values in the bin may contribute significantly to the join result, so we adjust the count by a scaling factor: $\min\left(\frac{B^S.\texttt{num}}{B^S.\texttt{mcv}+B^S.\texttt{sub-mcv}}, \frac{B^T.\texttt{num}}{B^T.\texttt{mcv}+B^T.\texttt{sub-mcv}}\right)$. This ensures the join count does not exceed the total number of remaining tuples. For example, the join count for range $[1, 5]$ is computed as: $8 \times 3.5 \times \min\left(\frac{36.24}{11.84+8}, \frac{13.3}{4.5+3.5}\right)$ and for range $[6, 10]$: $6.8 \times 2.7 \times \min\left(\frac{21.76}{6.8+5.44}, \frac{9.7}{2.7+2.6}\right)$.

(3) This case occurs when the mcv and sub-mcv of B^S and B^T refer to different values. In this case, we focus on the dominant value, *i.e.,* the larger mcv, and treat the remaining tuples in the other bin as uniformly distributed over the non-mcv values. For instance, if $B^T.\texttt{mcv} > B^S.\texttt{mcv}$, we estimate the join count as: $\frac{B^S.\texttt{num}-B^S.\texttt{mcv}-B^S.\texttt{sub-mcv}}{|B^S.\texttt{range}|-1} \times B^T.\texttt{mcv}$. This captures the expected contribution from non-dominant values in B^S aligning with the dominant value in B^T, assuming uniformity within the residuals.

7 Experiments

7.1 Experimental Setting

Datasets. We evaluate QSPN on both single-table and multi-table datasets. Table 2 provides the statistics of the datasets.

Single-table Datasets. We use four single-table datasets. (1) *GAS* [28] is a real-world gas sensing dataset, and we extract the most informative 8 columns (Time, Humidity, Temperature, Flow_rate, Heater_voltage, R1, R5, and R7), following the existing works [37, 43]. (2) *Forest* [27] is a real-world forest-fire dataset from the US Forest Service (USFS) and US Geological Survey (USGS). (3) *Power* [29] is an electric power consumption dataset consisting of measurements gathered in a house located in Sceaux (7 km from Paris, France) between December 2006 and November 2010.

Multi-table Dataset. We evaluate multi-table cardinality estimation using the *IMDB* [14] dataset, a real-world dataset containing 50K movie reviews. This dataset is extensively used in existing works [10, 12, 16, 35, 40, 43] for multi-table cardinality estimation evaluation. The columns in *IMDB* typically have large domain sizes, which presents a greater challenge for CardEst.

Query Workloads. We describe below how the query workloads used in the experiments are prepared.

Synthetic Workloads for Single-Table CardEst. As there is no real query workload available for the above four single-table datasets, we use the following steps to synthesize workloads.

(1) Template generation: Following existing DBMS benchmarks [17, 31, 32], we first generate SQL templates containing different column combinations as query predicates, and then synthesize queries based on these templates.

(2) Template selection: Existing studies [12, 16, 34, 41, 43] have shown that data correlation significantly affects the performance of CardEst methods. To account for this, we cluster all generated templates into two groups: one containing templates that access highly correlated columns and the other containing templates that access weakly correlated columns. We then evenly sample templates from both groups to ensure a fair comparison.

(3) Query synthesis: Given a template, we generate queries by filling it with randomly selected predicates. Previous studies [34, 35, 41] have shown that query conditions significantly affect CardEst performance. To ensure a fair comparison, we follow the method from [34], generating query conditions in two steps: first, selecting a random tuple from the data table as the center of the range; then, determining the width of each range using either a uniform or exponential distribution, with a predefined ratio controlling the selection between them. This ensures the conditions cover a variety of query patterns for evaluation.

We generate a read-write hybrid workload to evaluate QSPN model updates, where each SQL statement is randomly selected to be either a query or a DML command (INSERT or DELETE). First, we create a training set with queries accessing weakly correlated columns and range constraints following a normal distribution. Then, we add at least 20% new data tuples with increased correlation by sampling from a table with sorted columns. These new tuples are included as DML commands. Finally, we generate a test set that includes (1) queries following the original patterns and distribution and (2) queries accessing highly correlated columns with range constraints following an exponential distribution.

Real-World Workload for Multi-Table CardEst. For our multi-table dataset *IMDB*, we use *JOB-light* [17], the most widely adopted multi-table workload for CardEst. *JOB-light* consists of 70 queries on the *IMDB* dataset, where each query includes joins on 2 to 5 tables, along with 1 to 5 range constraints. These queries present a significant challenge for cardinality estimation methods. In particular, to ensure a fair comparison with the hybrid-driven model *UAE*, we use *JOB-light* as the workload test set and adopt the extended workload provided by *UAE* as the workload training set. This extended workload contains 100,000 queries covering *JOB-light*.

Table 2: Statistics of Datasets.

Dataset	Tables	Tuples	Columns (Numeric)	Domain Size
GAS	1	4M	8 (8)	$10^1 \sim 10^2$
Forest	1	0.6M	54 (10)	$10^2 \sim 10^3$
Power	1	2M	9 (7)	$10^4 \sim 10^5$
IMDB	21	74M	108 (57)	$10^0 \sim 10^7$

Baselines. We compare QSPN against the following baselines.

MSCN [16] is a query-driven CardEst model based on regression models. We use the implementation of MSCN from [34] and set its hyper-parameters following the configurations in [34].

Naru [41] is a DAR-based model that fits the joint data distribution to compute CardEst. We use the implementation of Naru from [34] and set its hyper-parameters following the configurations in [34].

DeepDB [12] is a data-driven SPN-based model which fits joint data distribution to compute CardEst, following local independence assumption. We use the implemention of DeepDB from [34] and set its hyper-parameters following the configurations in [34].

FLAT [43] is a CardEst method based on DeepDB, which introduces factorization and multi-dimensional histograms. We use the implementation provided by the authors of FLAT [37] and tune its hyper-parameters as recommended in the original paper.

UAE [35] is a hybrid-driven CardEst method based on Naru, combining unsupervised losses from data with supervised losses from queries. We use the implementation provided by the authors of UAE [38] We also use it as multi-table CardEst method [39].

FactorJoin [36] is a multi-table CardEst method based on join-keys binning without modeling data distribution on generated outer-join tables. Following the paper [36], we implement it with uniform sampling (sample_rate=0.1).

Postgres [10] is the cardinality estimator of PostgreSQL, based on traditional statistics-based methods. We run and evaluate it using the connector from [34], which connects to PostgreSQL 12 with the default setting $stat\_target = 10000$.

Evaluation Metrics. We use the following metrics to comprehensively evaluate and compare CardEst methods.

Estimation Accuracy. For each query, we measure estimation accuracy using Q-error, defined as the ratio between the estimation est and the ground truth gt, *i.e.,* $Q\text{-error} = \frac{\min\{est,gt\}}{\max\{est,gt\}}$.

Inference Time. For each query in the test set, we measure the total time spent for CardEst as inference time. We then compute the mean inference time among the queries in the test set.

Storage Overhead. We measure the total size of the model or statistics file(s) for each method on each dataset and its corresponding workload training set, defining this as the storage overhead.

Experimental Settings. All evaluated methods are implemented in Python 3.7. Our experiments are conducted on an AMD-64 server with the following specifications: OS: Ubuntu 20.04.6 LTS; Dual CPU System: 2× Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz (20C40T); Main Memory: 1TB DDR4 ECC; Storage: 4×8 TB HDD (RAID5). We set the default hyper-parameters of QSPN (query-aware adaptive threshold of Product RDC, threshold of QProduct, threshold of QSplit, and threshold of Sum) on all datasets as: $\tau_p = (s = 5, l = 0.1, u = 0.3), \tau = 0.01, \tau_x = 0.7, \tau_s = 0.3$. Among these hyper-parameters, the τ_p is about the *RDC* threshold setting which we set following other SPNs. While τ_x and τ_s are hyper-parameters in QProduct algorithm and QSplit algorithm which we set after a grace search on workload samples.

Table 3: Evaluating Single-Table CardEst on the three Criteria: Estimation Errors, Inference/Construction Time and Model Size (Bold font indicates the best-performing method, while underlined font highlights the second-best).

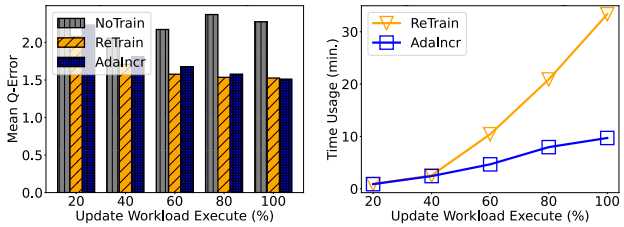
Dataset	Method	Estimation Mean Q-err	Inference Time (ms)	Construction Time (min.)	Model Size (MB)
GAS	Postgres	13.96	0.321	0.123	**0.01**
	MSCN	2.65	0.386	0.157	4.423
	LW-XGB	2.81	**0.066**	0.08	0.605
	Naru	1.12	13.658	149.205	1.923
	DeepDB	3.94	8.22	3.316	0.580
	FLAT	1.21	1.209	0.810	13.996
	UAE	1.10	2.013	301.390	0.747
	QSPN	1.22	1.089	1.919	0.209
Forest	Postgres	64.72	0.153	0.158	**0.07**
	MSCN	2.25	0.427	0.128	1.501
	LW-XGB	3.63	**0.074**	0.01	0.663
	Naru	1.50	38.651	80.026	2.216
	DeepDB	1.88	11.295	2.562	1.64
	FLAT	**1.30**	41.69	1.821	972.307
	UAE	1.6	76.721	188.661	29.147
	QSPN	1.81	3.092	2.167	1.261
Power	Postgres	551.04	**0.103**	0.206	**0.04**
	MSCN	4.32	0.421	0.144	4.402
	LW-XGB	4.204	0.466	0.02	0.062
	Naru	1.19	43.125	175.368	4.327
	DeepDB	13.05	12.731	3.722	3.711
	FLAT	1.27	2090.254	2.225	10100.46
	UAE	**1.16**	18.22	1316.239	12.599
	QSPN	1.51	0.988	0.782	0.623

7.2 Evaluation on Single-Table CardEst

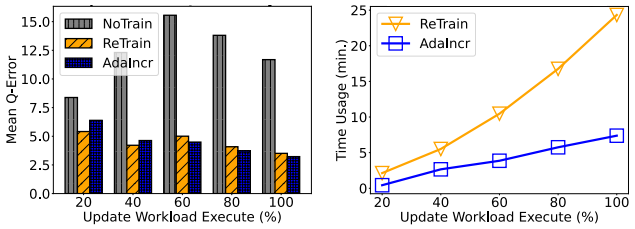
We first compare QSPN with the baseline methods on single-table CardEst. Table 3 reports the experimental results.

Estimation Accuracy. Our proposed QSPN achieves superior estimation accuracy, comparable to state-of-the-art (SOTA) data-driven methods (*FLAT* and *Naru*) and the hybrid method *UAE*. Specifically, the mean Q-errors of QSPN are minimal, ranging from 1.0 to 1.8. Moreover, QSPN overcomes the limitations of traditional SPN models (e.g., *DeepDB*), achieving up to an 88% reduction in Q-error. The superior performance of QSPN is primarily attributed to its ability to effectively handle strongly correlated data through column partitioning based on both data distribution and query co-access patterns. In contrast, query-driven methods such as *MSCN* and traditional CardEst methods (Postgres, Sampling, and MHist) struggle to achieve satisfactory estimation accuracy in most cases, as they fail to adequately learn the data distribution. Additionally, we observe that the Q-errors of *DeepDB* on the *GAS* and *Power* datasets rise sharply. This is due to the strong correlations present in these datasets, which pose a more significant challenge for *DeepDB*.

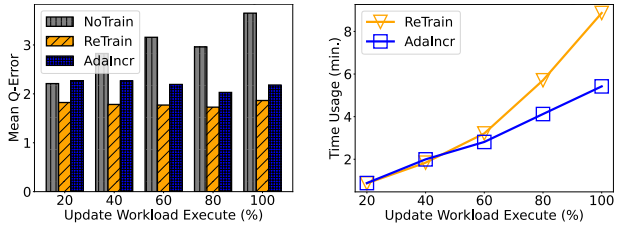
Inference Time. As shown in Table 3, QSPN demonstrates highly efficient inference performance. Specifically, the inference time of QSPN ranges from 0.422 ms to 3.092 ms across the four datasets, which is only slightly slower than the traditional CardEst method *Postgres* and the query-driven method *MSCN*. Moreover, compared to the traditional SPN-based method *DeepDB*, QSPN achieves up to a 92.2% reduction in inference time. This efficiency is due to QSPN's column partitioning strategy, which considers both data



(a) Dataset GAS.



(b) Dataset Forest.



(c) Dataset Power.

Figure 4: Accuracy and Time Usage on Hybrid-Workload.

correlations and query access patterns. By reducing the number of intermediate nodes in the SPN, QSPN improves inference efficiency and reduces storage overhead, all while maintaining high estimation accuracy. On the other hand, *Naru* and *UAE* are the slowest methods, as their underlying deep auto-regressive models suffer from high inference times due to the computationally expensive progressive sampling process. Additionally, the inference time of *FLAT* increases significantly on the *Forest* and *Power* datasets, due to the large number of factorize and multi-leaf nodes used to model the complex correlations in these datasets.

We also evaluate the construction (training) time of the methods and categorize them into three groups: (1) *Postgres*, *Sampling*, and *MSCN* require negligible time for training. (2) SPN-based methods, such as *DeepDB*, *FLAT*, and QSPN, take approximately 1-3 minutes for construction, which does not impose a significant burden on the DBMS. (3) DAR-based methods like *Naru* and *UAE* require 100 to 1000 times more training time than SPN-based models, making them impractical for real-world applications.

Storage Overhead. QSPN ranks among the top in storage efficiency, requiring only tens of KB to about 1 MB more than *Postgres*. *DeepDB* can also be considered a lightweight method, although its model size increases significantly on the *Forest* and *Power* datasets due to a larger number of nodes. On the other hand, *MSCN*, *Naru*, and *UAE*, which are based on CNN or DAR models, naturally have larger model sizes. *FLAT* and *Sampling* suffer from substantial storage overhead. The excessive model size of *FLAT* stems from the same factors that contribute to its slow performance.

Summary. The experimental results demonstrate that QSPN achieves superior and robust performance across the three key criteria, outperforming state-of-the-art approaches. This outcome aligns with the design objectives of QSPN, as presented in Table 1.

7.3 Evaluation on Dynamic Model Update

In this section, we evaluate the model update performance of QSPN on our read-write hybrid workload and compare the following alternatives: *NoTrain* (no model updates), *ReTrain* (periodic model reconstruction), and *AdaIncr* (our updating method in QSPN). As shown in Figure 4, *NoTrain* suffers from poor accuracy across all three datasets under hybrid workloads, whereas both *ReTrain* and *AdaIncr* maintain the accuracy of the QSPN model. Notably, *AdaIncr* reduces update time by 30% to 60% while achieving the same accuracy as *ReTrain*. Furthermore, when analyzing the trends in workload execution time, the total time usage of *AdaIncr* increases gradually, whereas that of *ReTrain* rises sharply. That is because the data and workload size is expanding along with the hybrid-workload updating both data and queries. In this situation, the cost of every time *ReTrain* rises sharply. While our *AdaIncr* continuously update the model so that we only process a small amount of data and change a small number of nodes each time. Due to space constraints, we only report the results for the hybrid-update setting, which involves both data updates and query workload shifts, *i.e.*, difficult case for model update. While the results for the data-update and query-update settings are provided in our technical report which reflects similar rules like Figure 4.

Notably, *AdaIncr* reduces update time by 30% to 60% while achieving the same accuracy as *ReTrain*. Furthermore, when analyzing the trends in workload execution time, the total time usage of *AdaIncr* increases gradually, whereas that of *ReTrain* rises sharply.

7.4 Evaluation of Multi-Table CardEst

In this section, we compare our proposed M-QSPN with the SOTA methods for multi-table CardEst and report the results in Table 4. We observe that *Postgres* exhibits poor estimation accuracy, and loses its advantage in inference time, as it requires multiple iterations to compute multi-table join cardinality. *UAE* achieves better estimation accuracy; however, both its inference and training times are unsatisfactory due to the inherent limitations of DAR-based models. Compared to *UAE*, *FLAT* provides a better trade-off between estimation accuracy and inference time. Among all evaluated approaches, our proposed M-QSPN model achieves the best performance. Specifically, compared to the best baseline, *FLAT*, M-QSPN improves estimation accuracy and inference efficiency by approximately three times, covering the shortage of *FactorJoin* on base table filtering. The only drawback of M-QSPN is its model size, primarily due to storing binnings on Leaf nodes with extra storage for convenience. However, this additional storage overhead can be eliminated by integrating binning into the histogram on each Leaf node during implementation.

Table 4: Evaluating Multi-Table CardEst on the Job-Light Workload of IMDB Dataset.

Method	Q-error						Inference Time (ms)	Construction Time (min.)	Model Size (MB)
	50th	90th	95th	99th	Max	Mean			
Postgres	7.99	161.79	818.29	2042.92	2093.13	128.61	1894.532	3.323	**9**
FLAT	2.73	16.71	44.52	120.91	203.11	10.33	32.081	46	91
FactorJoin	4.33	24.7	34.62	92.57	106.36	10.63	8626.417	**0.426**	13.213
UAE	**1.45**	17.74	29	184.42	184.95	11.72	123.6	27224	-
M-QSPN	2.54	**6.5**	**9.01**	**26.79**	**33**	**3.87**	**13.368**	27	450

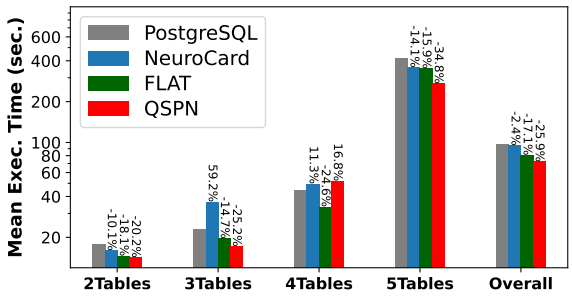


Figure 5: End-to-End Mean Query Execution Time.

7.5 Evaluation on End-to-End Query Execution

We evaluate the effect of CardEst models on end-to-end query execution in the PostgreSQL DBMS [10], following the experimental settings described in the End-to-End CardEst Benchmark [21]. Specifically, we conduct the End-to-End benchmark on the *IMDB* dataset [14] with *JOB-Light* workload [17], and compare our proposed QSPN with the PostgreSQL internal estimator [10], *NeuroCard* [40], and *FLAT* [43] in the End-to-End evaluation. The experimental results are reported in Figure 5. Our QSPN achieves the best performance in end-to-end query execution. Specifically, QSPN reduces the mean query execution time by 25.9%, compared to 2.4% for *Neurocard* and 17.1% for *FLAT*. This outcome shows that the accurate CardEst by QSPN and M-QSPN reach the superior performance optimization of our QSPN in multi-table CardEst. In particular, QSPN performs best on queries involving 2, 3, and 5 joined tables. For instance, for the most complex 5-table queries, our QSPN reduces execution time by more than 15% compared to *FLAT*. We also report the mean optimization time for CardEst: 60 ms for *NeuroCard*, 11 ms for *FLAT*, and 6 ms for our M-QSPN, demonstrating that M-QSPN achieves not only the best execution performance but also the best optimization efficiency.

8 Related Work

Traditional CardEst Methods. Traditional CardEst methods [1, 2, 4, 10, 11, 18, 24] rely on simplifying assumptions, such as column independence. Postgres [10] assumes that all columns are independent and estimates the data distribution of each column using histograms. Sampling-based methods [1, 2, 11, 18] sample tuples from the data and store them. In the online phase, these methods execute queries on the stored sample to estimate cardinalities. MHist [24] is a multi-histogram approach that accounts for data correlation by constructing multi-dimensional histograms. Bayes [4] performs cardinality estimation using probabilistic graphical models [5, 8, 33]. While effective, it can be slow, especially when dealing with datasets that have high correlation. The key limitation of these traditional methods is their reliance on simplifying assumptions, which often lead to significant estimation errors.

Learning-based CardEst Methods. Query-driven methods transform the CardEst problem into a regression task, mapping query workloads to ground truth cardinalities. MSCN [16] encodes a query into a standardized vector, which is then fed into a Multi-Layer Perceptron (MLP), where they undergo average pooling. The pooled representations are then concatenated and passed into a final MLP to output the selectivity. LW-XGB/NN [6] encodes a query into a simpler vector by concatenating the lower bounds and upper bounds of all range predicates in order. Then, this query vector is then input into a small neural network or XGBoost [3] model to predict the estimated cardinalities. Data-driven methods transform the CardEst problem into a joint probability problem, where each column is treated as a random variable. Naru [41] factorizes the joint distribution into conditional distributions using Deep AutoRegressive (DAR) models such as MADE [7]. Naru then employs progressive sampling [25] to compute cardinality estimates for range queries based on point probabilities. UAE [35] is an optimized version of Naru, tailored for query workloads. As a hybrid CardEst model, UAE improves the fitting of the DAR model on data with long-tail distributions by adjusting the sampling regions according to the queries. FactorJoin [36] is a multi-table CardEst method that trains single-table cardinality estimation models for each of the joined tables and uses a factor graph to capture the relationships and correlations between different join keys. Despite advancements in learning-based cardinality estimation, existing methods may struggle to simultaneously optimize the key criteria: *estimation accuracy*, *inference time*, and *storage overhead*, limiting their practical applicability in real-world database environments.

9 Conclusion

In this paper, we have introduced QSPN, a *unified model* that integrates both data distribution and query workload. QSPN extends the simple yet effective Sum-Product Network (SPN) model by jointly partitioning columns based on both data correlations and query access patterns, thereby reducing model size and improving inference efficiency without sacrificing estimation accuracy. We have formalized QSPN as a tree-based structure that extends SPNs by introducing two new node types: `QProduct` and `QSplit`. We have conducted extensive experiments to evaluate QSPN in both single-table and multi-table cardinality estimation settings. The experimental results have demonstrated that QSPN achieves superior and robust performance on the three key criteria, compared with state-of-the-art approaches.

References

[1] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. 2000. Towards estimation error guarantees for distinct values. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 268–279.

[2] Surajit Chaudhuri, Gautam Das, and Utkarsh Srivastava. 2004. Effective use of block-level sampling in statistics estimation. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 287–298.

[3] 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.

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

[5] 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.

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

[7] Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. 2015. MADE: masked autoencoder for distribution estimation. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning-Volume 37*. 881–889.

[8] Lise Getoor, Benjamin Taskar, and Daphne Koller. 2001. Selectivity estimation using probabilistic models. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. 461–472.

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

[10] Copyright © 1996-2024 The PostgreSQL Global Development Group. [n.d.]. PostgreSQL 12.20 Documentation: Chapter 59. Genetic Query Optimizer. ([n. d.]). https://www.postgresql.org/docs/12/geqo-intro2.html

[11] Peter J Haas, Jeffrey F Naughton, S Seshadri, and Lynne Stokes. 1995. Sampling-based estimation of the number of distinct values of an attribute. In *VLDB*, Vol. 95. 311–322.

[12] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: learn from data, not from queries! *Proceedings of the VLDB Endowment* 13, 7 (2020), 992–1005.

[13] Tsuyoshi Hirayama, Yuhao Liu, Kazuhisa Makino, Ke Shi, and Chao Xu. 2023. A Polynomial Time Algorithm for Finding a Minimum 4-Partition of a Submodular Function. (2023), 1680–1691. https://doi.org/10.1137/1.9781611977554.CH64

[14] Research institute for mathematics & computer science in the Netherlands. [n.d.]. imdb.tgz. ([n. d.]). https://homepages.cwi.nl/~boncz/job/imdb.tgz

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

[16] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf

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

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

[19] David López-Paz, Philipp Hennig, and Bernhard Schölkopf. 2013. The Randomized Dependence Coefficient. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger (Eds.). 1–9. https://proceedings.neurips.cc/paper/2013/hash/aab3238922bcc25a6f606eb525ffdc56-Abstract.html

[20] Alejandro Molina, Antonio Vergari, Nicola Di Mauro, Sriraam Natarajan, Floriana Esposito, and Kristian Kersting. 2018. Mixed Sum-Product Networks: A Deep Architecture for Hybrid Domains. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 3828–3835. https://doi.org/10.1609/AAAI.V32I1.11731

[21] Nathaniel-Han. [n.d.]. End-to-End-CardEst-Benchmark. ([n. d.]). https://github.com/Nathaniel-Han/End-to-End-CardEst-Benchmark

[22] Kazumasa Okumoto, Takuro Fukunaga, and Hiroshi Nagamochi. 2009. Divide-and-Conquer Algorithms for Partitioning Hypergraphs and Submodular Systems. 5878 (2009), 55–64. https://doi.org/10.1007/978-3-642-10631-6_8

[23] Hoifung Poon and Pedro Domingos. 2011. Sum-product networks: A new deep architecture. In *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*. IEEE, 689–690.

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

[25] Foster Provost, David Jensen, and Tim Oates. 1999. Efficient progressive sampling. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*. 23–32.

[26] UCI ML Repository. [n.d.]. Census Income Data Set. ([n. d.]). https://archive.ics.uci.edu/dataset/20/census+income

[27] UCI ML Repository. [n.d.]. Classification of pixels into 7 forest cover types. ([n. d.]). https://archive.ics.uci.edu/dataset/31/covertype

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

[29] UCI ML Repository. [n.d.]. Individual Household Electric Power Consumption. ([n. d.]). https://archive.ics.uci.edu/dataset/235/individual+household+electric+power+consumption

[30] Mechthild Stoer and Frank Wagner. 1997. A simple min-cut algorithm. *J. ACM* 44, 4 (1997), 585–591. https://doi.org/10.1145/263867.263872

[31] TPC. [n.d.]. TPC-DS. ([n. d.]). https://www.tpc.org/tpcds/default5.asp

[32] TPC. [n.d.]. TPC-H. ([n. d.]). https://www.tpc.org/tpch/default5.asp

[33] Kostas Tzoumas, Amol Deshpande, and Christian S Jensen. 2011. Lightweight graphical models for selectivity estimation without independence assumptions. *Proceedings of the VLDB Endowment* 4, 11 (2011), 852–863.

[34] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are we ready for learned cardinality estimation? *Proceedings of the VLDB Endowment* 14, 9 (2021), 1640–1654.

[35] 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 International Conference on Management of Data*. 2009–2022.

[36] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. 2023. FactorJoin: a new cardinality estimation framework for join queries. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[37] wuziniu. 2022. Github repository: FSPN. (2022). https://github.com/wuziniu/FSPN

[38] Jingyi Yang. [n.d.]. Github repository: pagegitss/UAE. ([n. d.]). https://github.com/pagegitss/UAE

[39] Jingyi Yang. [n.d.]. Github repository: pagegitss/UAE. ([n. d.]). https://github.com/pagegitss/UAE/tree/master/UAE_joins

[40] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: one cardinality estimator for all tables. *Proceedings of the VLDB Endowment* 14, 1 (2020), 61–73.

[41] 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. *Proc. VLDB Endow.* 13, 3 (2019), 279–292. https://doi.org/10.14778/3368289.3368294

[43] 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. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1489–1502.

[43] 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. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1489–1502.

[44] 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, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *Proceedings of the VLDB Endowment* 15, 4 (2021), 752–765.

A Additional Update Experiments

A.1 Single-Table

Table 5 reports the detailed single-table experimental results. The dataset census [26] is small *i.e.,* easy for CardEst models so we omit it in the main text.

A.2 Data-Update Only

We first consider the setting where only data tuples are updated (*i.e.,*, ΔT). As shown in Figure 6, data updates have a minimal impact on QSPN. This is because, in large datasets with high-cardinality

Table 5: Evaluating Single-Table CardEst on Key Criteria: Estimation Errors, Inference/Construction Time and Model Size.

Dataset	Method	Estimation Errors						Inference Time (ms)	Construction Time (min.)	Model Size (MB)
		50th	90th	95th	99th	Max	Mean			
GAS	Postgres	3.42	33.37	48.46	143.34	376.21	13.96	0.321	0.123	**0.01**
	Sampling	**1.02**	**1.14**	**1.19**	**1.54**	**1.97**	**1.06**	1.347	**0.002**	250.118
	MHist	1.84	4.63	12.97	290.38	5564	28.21	187.55	17.600	3.09
	MSCN	1.46	3.61	6.45	22.21	43.96	2.65	0.386	0.157	4.423
	LW-XGB	1.46	3.23	4.27	12.38	154.22	2.81	**0.066**	0.08	0.605
	Naru	1.07	1.26	1.39	1.87	2.47	1.12	13.658	149.205	1.923
	DeepDB	1.06	9.32	14.22	48.23	83.05	3.94	8.22	3.316	0.580
	FLAT	**1.02**	1.7	2.12	3.12	3.94	1.21	1.209	0.810	13.996
	UAE	1.06	1.21	1.30	1.71	2.14	1.10	2.013	301.390	0.747
	QSPN	1.06	1.69	2.12	3.12	3.94	1.22	1.089	1.919	0.209
Census	Postgres	6.75	118.1	393.67	2362	2362	91.25	0.133	0.045	**0.01**
	Sampling	1.18	6.31	24.6	72.58	97	4.71	0.893	**0.001**	1.994
	MHist	**1**	**1**	**1**	**1**	**1**	**1**	68.69	4.830	0.98
	MSCN	2.37	6.25	13.07	78	79	4.85	0.391	0.108	0.085
	LW-XGB	1.05	2.62	4.33	11.07	13.23	1.65	**0.053**	0.01	0.138
	Naru	1.03	1.23	1.40	1.69	3.00	1.09	5.985	1.522	0.128
	DeepDB	1.12	1.54	1.78	3	6.86	1.25	0.438	0.090	0.026
	FLAT	1.11	1.4	1.72	3	6.67	1.23	0.938	0.107	0.104
	UAE	1.06	1.15	1.25	1.76	2.67	1.09	2.28	13.044	0.829
	QSPN	1.12	1.42	1.7	3	6.67	1.23	0.422	0.141	0.059
Forest	Postgres	3.43	40.67	109.74	1002.98	9215.5	64.72	0.153	0.158	**0.07**
	Sampling	1.07	**1.44**	1.89	25.01	169	2.64	0.994	**0.001**	47.378
	MHist	1.59	5.46	10.12	40.87	1427	8.86	420.915	20.000	3.83
	MSCN	1.49	3.53	5.36	11.73	61.5	2.25	0.427	0.128	1.501
	LW-XGB	1.44	4.63	11.03	41.92	139.43	3.63	**0.074**	0.01	0.663
	Naru	1.17	1.79	2.31	7.39	**22.00**	1.50	38.651	80.026	2.216
	DeepDB	1.07	1.57	2.02	4.81	148	1.88	11.295	2.562	1.64
	FLAT	**1.02**	1.48	**1.80**	3.67	38.71	**1.30**	41.69	1.821	972.307
	UAE	1.16	1.82	2.75	12.96	25.20	1.6	76.721	188.661	29.147
	QSPN	1.05	1.48	1.84	**3.08**	186.5	1.81	3.092	2.167	1.261
Power	Postgres	3.71	162.04	925.33	13374.34	48341	551.04	**0.103**	0.206	**0.04**
	Sampling	1.04	1.42	4	60.96	394	3.96	1.147	**0.001**	118.273
	MHist	15	104.73	349.63	5350.84	15829	211.63	182.491	13.350	2.72
	MSCN	1.88	8.31	15.58	32.45	155	4.32	0.421	0.144	4.402
	LW-XGB	1.34	5.59	14.14	46.49	242.08	4.204	0.466	0.02	0.062
	Naru	1.06	1.35	1.61	3.48	7.95	1.19	43.125	175.368	4.327
	DeepDB	1.04	3.42	5.45	198.2	1876	13.05	12.731	3.722	3.711
	FLAT	**1.01**	**1.22**	**1.39**	**2.27**	50.59	1.27	2090.254	2.225	10100.46
	UAE	1.04	1.27	1.58	4.15	**5.00**	**1.16**	18.22	1316.239	12.599
	QSPN	1.05	1.74	2.57	7.98	51.33	1.51	0.988	0.782	0.623

attributes, incremental updates typically do not significantly alter the overall data distribution. *NoTrain* results in a significant Q-error, whereas both *ReTrain* and *AdaIncr* maintain the accuracy of the QSPN model. Compared with *ReTrain*, our *AdaIncr* achieves over a 10× reduction in update time while keeping the QSPN model even more accurate than *ReTrain*, as its efficiency allows for more frequent updates.

A.3 Query-Update Only

As shown in Figure 7, query updates do impact QSPN because they alter the query-column access patterns. *NoTrain* is not a viable option, as it leads to a sharp increase in mean Q-error. As expected, both *ReTrain* and *AdaIncr* maintain the accuracy of the QSPN model, with *AdaIncr* significantly reducing the time required for model updates. However, compared to the data-update-only scenario, *AdaIncr* requires more time for model updates in this case, as more subtrees need to be reconstructed to adapt to workload shifts. In contrast to the data-update-only scenario, the QSPN model remains unaffected by workload shifts on the *Census* dataset, as it constructs only a few `QProduct` and `QSplit` nodes due to the weak correlations in the data.

B Additional Proof

B.1 PROOF of LEMMA 1

Lemma 1. *The problem of* `QProduct` *construction is equivalent to the minimum k-cut problem.*

Proof. Recall that the minimum k-cut problem is defined as follows: Given an undirected graph $G = (V, E)$ with edge weights $w : E \to \mathbb{R}$ and an integer k, the goal is to partition the vertex set V into k disjoint subsets $C_1, C_2, \ldots, C_k$ such that the following objective is minimized: $\sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \sum_{v_1 \in C_i, v_2 \in C_i} w(\{v_1, v_2\})$, which corresponds to the sum of weights of edges crossing different partitions.

In the context of `QProduct` construction, the column set A and the access affinity $\texttt{AFF}(a_i, a_j)$ between two columns can be naturally mapped to the vertex set V and edge weights $w(v_1, v_2)$ in the minimum k-cut problem, respectively. Moreover, the objective of

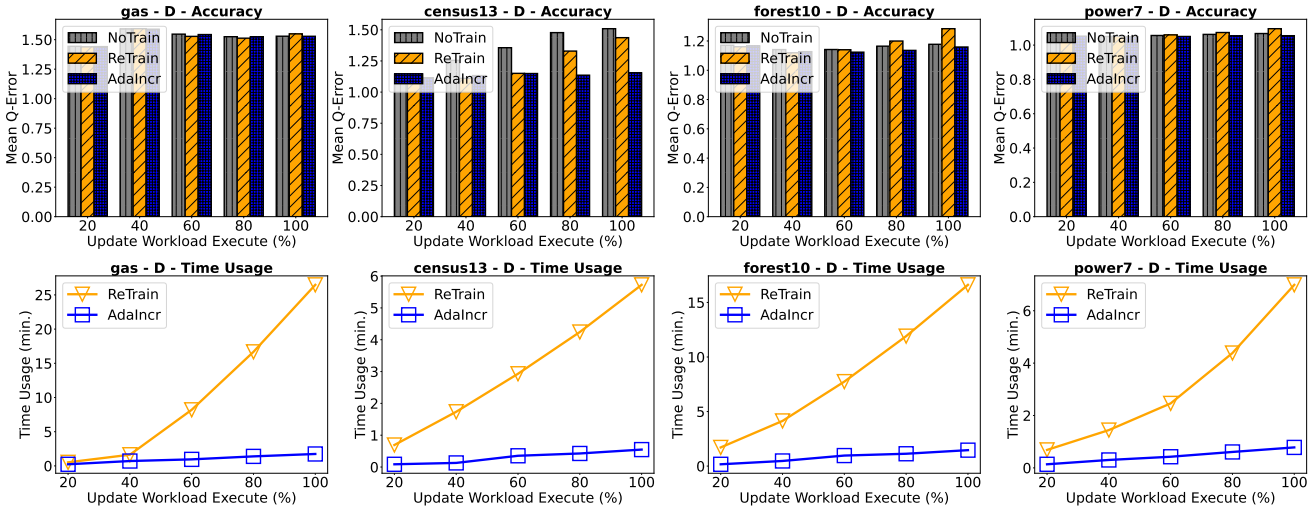


Figure 6: Evaluation on Dynamic Model Update (Data-Update Setting).

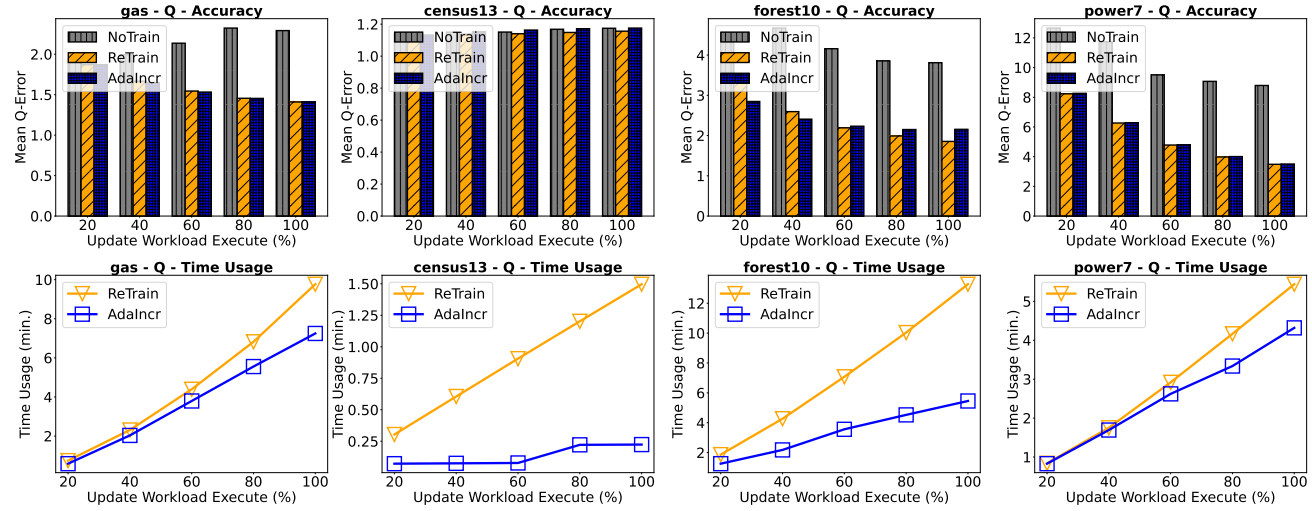


Figure 7: Evaluation on Dynamic Model Update (Query-Update Setting).

minimizing the inter-partition affinity (IPA) in QProduct construction is equivalent to the objective of the minimum k-cut problem. Thus, we prove the lemma.

We prove that the QProduct construction problem can be reduced from K-CUT problem. Given an arbitrary undirected graph $G = (V, E)$ and parameter K, we make a column set $A = \{a_1, \ldots, a_{|V|}\}$ corresponding to V and we set $\texttt{AFF}(a_i, a_j|Q) = \texttt{AFF}(a_j, a_i|Q) = w_l / \max\{w | e = (u, v, w) \in E\}$ for each edge $e_l = (v_i, v_j, w_l)$. Note that the AFF matrix we set must be legal since the assumed query workload Q which contains $\texttt{AFF}(a_i, a_j|Q) * \max\{w | e = (u, v, w) \in E\}$ queries with predicates only on column a_i and a_j for each $\texttt{AFF}(a_i, a_j|Q)$ value is easy to generate. Then we solve QProduct construction problem to partition A into K subsets: $A = \{A_1, \ldots, A_K\}$ and we get the answer of the K-CUT problem: minimized $IPA(A|Q)$ multiplying $\max\{w | e = (u, v, w) \in E\}$.

So far, there is no efficient deterministic algorithm for K-CUT problem. When $K = 2$, it can be solved by an $O(|A|^3)$ but actually slow MIN-CUT algorithm [30]. When $K = 3$, it can be solved by an $O(|A|^3\tau(|A|))$ 3-CUT algorithm [22] where $\tau(|A|)$ is the cost of computing goal function which is $O(|A|^2)$ under the background of our QProduct construction problem *i.e.,* the algorithm costs $O(|A|^5)$. When $K = 4$, it can be solved by an $O(|A|^6\tau(|A|))$ *i.e.,* $O(|A|^8)$ 4-CUT algorithm [13]. When $K \geq 5$, there is no polynomial time algorithm found so far while the research of 4-CUT algorithm [13] made a corollary that the 5-CUT algorithm should cost $O(|A|^{14}\tau(|A|))$ *i.e.,* $O(|A|^{16})$. □

B.2 PROOF of LEMMA 2

Lemma 2. *For any given query set Q, the upper bound* $\overline{\texttt{IPA}}(\mathcal{A}|Q)$ *provides an upper estimate of the optimal result* $\texttt{IPA}(\mathcal{A}^*|Q)$.

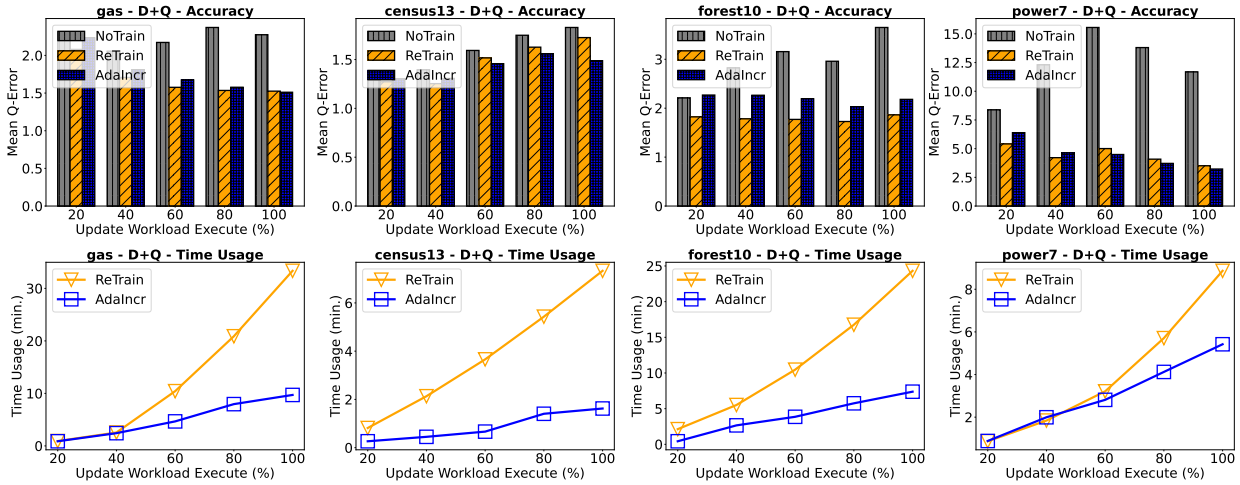


Figure 8: Evaluation on Dynamic Model Update (Hybrid-Update Setting).

Proof. Due to the space constraints, we provide only a proof sketch. First, we show that $\texttt{IPA}(\mathcal{A}^*|Q)$ can be decomposed into the IPA scores of distinct query patterns, *i.e.,* $\texttt{IPA}(\mathcal{A}^*|Q) = \sum_{p_i \in Q} \texttt{IPA}(\mathcal{A}^*|\{p_i\})$. Next, we prove that for any query pattern $p_i \in Q$, its optimal IPA score $\texttt{IPA}(\mathcal{A}^*|\{p_i\})$ is bounded by the upper term $n_i \cdot z_{ij} \cdot (\|p_i\| - z_{ij})$, where p_j is any query pattern satisfying $z_{ij} > 0$. To see why this holds, note that z_{ij} represents the number of shared accessed columns between patterns p_i and p_j. The term $(\|p_i\| - z_{ij})$ captures the remaining accessed columns in p_i that are not shared with p_j. Thus, $n_i \cdot z_{ij} \cdot (\|p_i\| - z_{ij})$ is the IPA score of the specific partitioning scheme as illustrated in Figure 2(a), which is larger than the optimal result $\texttt{IPA}(\mathcal{A}^*|\{p_i\})$. Moreover, if $z_{ij} = 0$ for all $j \neq i$, meaning that p_i shares no common accessed columns with any other query patterns, then the optimal column partition for p_i incurs no IPA cost, implying $\texttt{IPA}(\mathcal{A}^*|\{p_i\}) = 0$. Thus, we conclude that $\texttt{IPA}(\mathcal{A}^*|Q) \leq \overline{\texttt{IPA}}(\mathcal{A}|Q)$, proving the lemma. □

B.3 PROOF of LEMMA 3

Lemma 3. *The problem of partitioning Q into $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_n\}$ to minimize the upper bound of inter-partition affinity,* i.e.,, $\sum_{k=1}^{n} \overline{\texttt{IPA}}(\mathcal{A}_k|Q_k)$, *is NP-hard.*

Proof. To convert an arbitrary graph $G(V, E)$ to QSplit problem, we generate $\mathbf{q_i}$ corresponding to each $v_i \in V$ and we generate $w = |\mathbf{q_i}|z_{i,j}(x_i - z_{i,j}) + |\mathbf{q_j}|z_{i,j}(x_j - z_{i,j})$ for each $e_{ij} \in E$. To maximize the sum of edge weights that are cut off in G, the sum of edge weights remained in all components is minimized, which is equivalent to minimizing $\texttt{SIQ}(\mathcal{Q}|A)$ after converting G to QProduct problem. Thus, minimizing $\texttt{SIQ}(\mathcal{Q}|A)$ can be reduced to MAX-CUT problem. □

B.4 Complexity Analysis of Offline QSPN Construction

Given data T, workload Q and column set $|\mathcal{A}| = m$, supposing a balanced c-way QSPN tree with n nodes (s Sum nodes, x QSplit nodes, u Product | QProduct nodes, less than and around n Leaf nodes) and z sample size for RDC. For each Leaf node, the number of tuples is around $|T|/s$ so its complexity is $O(n|T|/s)$. For each of the data tuples, it will go through $log_c s$ Sum nodes and cost $O(cm)$ in each node so the complexity of Sum nodes is $O(cm|T|log_c s)$. Similarly, the complexity of QSplit nodes is $O(cm|Q|log_c t)$. The cases are different with Product and QProduct, since their cost contains RDC and AFF matrixes computation. For RDC matrix computation, the cost is constant $O(zlogz)$. So the complexity of Product nodes is $O(nm^2 zlogz)$. For AFF matrix calculation, the cost is $O(m^2|Q|)$. So the complexity of QProduct nodes is $O((s + x + u)m^2|Q|)$. In total, the complexity of QSPN construction is $O(m^2[nzlogz + (s+x+u)|Q|] + cm(|T|log_c s + |Q|log_c x) + n|T|/s)$.

B.5 Complexity Analysis of Online CardEst Inference with QSPN

Given a query q with selectivity (probability) p and constraints on m columns, suppose the QSPN model is a balanced c-way tree with N nodes (s Sum nodes, x QSplit nodes, and u Product/QProduct nodes). It is known that the cost of a Leaf node with a histogram is $O(1)$, the cost of a QSplit node is $O(m^2 c)$, and the cost of other middle nodes is $O(mc)$. Thus, the worst-case complexity of the bottom-up inference method is $O(xm^2 c + (N-x)mc)$. Since there are usually a small number of QSplit nodes in a QSPN model for query routing, the time cost of our fast inference method is approximately $\max(p, \frac{1}{s}) \cdot \frac{m}{u} \cdot Nmc$.

C Additional Detailed Algorithm

C.1 Offline QSPN Construction Algorithm

We show our top-down **ConstructQSPN** (Algorithm 1) corresponding to Section 4 including its sub function **PartitionByAFF** (Algorithm 1.1) which is explained in Section 4.1 and sub function **SplitWorkload** (Algorithm 1.2) which is explained in Section 4.2. In **ConstructQSPN** algorithm, each node n to construct are tried different type n.O in the heuristic order: Leaf, Product, QProduct, QSplit, Sum for a accuracy-efficiency balanced QSPN model. And the specific node construction are executed in sub functions.

Algorithm 1.2 `SplitWorkload` (Q, A)

Input: Q: A Query Workload; A: A Column Set
Output: $\mathcal{Q} = \{Q_1, \dots Q_m\}$: A Collection of Sub-workloads

$\mathbf{q_1}, \dots, \mathbf{q_w} \leftarrow$ Merge Queries q sharing the same $\mathbf{acc}_q$
$\mathtt{sim} \leftarrow$ **ComputeSim** ($\mathbf{ACC}(Q, A)$) for each (q_i, q_j) from Q
Construct a graph $G = (V, E)$ where vertice V represents $\mathbf{ACC}(Q, A)_1, \dots, \mathbf{ACC}(Q, A)_{|\mathbf{ACC}(Q,A)|}$ and $e_{kl} \in E$ represents $\mathtt{sim}(\mathbf{ACC}(Q, A)_k, \mathbf{ACC}(Q, A)_l)$
Cut off some biggests edges of G
Sort $v \in V$ by weighted degrees in desc order
Create m empty set: $R_1, \dots, R_m$
for $1 \le i \le |\mathbf{ACC}(Q, A)|$ **do**
    Put $\mathbf{ACC}(Q, A)_i$ into the subset R_j with $\min_j(\sum_{\mathbf{ACC}(Q,A)_i \in R_j} \mathtt{sim}(i, \mathbf{ACC}(Q, A)_k))$
end for
Create m empty query subsets: $Q_1, \dots, Q_m$
for $q \in Q$ **do**
    Put q into Q_k if $\exists \mathbf{ACC}(Q, A)_l \in R_k$ and q obeys $\mathbf{ACC}(Q, A)_l$
end for
$\mathcal{Q} \leftarrow \{Q_1, \dots, Q_m\}$
return $\mathcal{Q}$

Algorithm 2 `QSPN-Online` $(\mathtt{n}, q)$

Input: n: A QSPN Node; q: A Query
Output: $P_{\mathtt{n}}$: Probability Estimated at Node n

if $\mathtt{n}.O = \mathtt{Leaf}$ **then**
    return $\mathtt{n}.P_T(q)$ based on the histogram at leaf n
else if $\mathtt{n}.O = \mathtt{Product} \mid \mathtt{QProduct}$ **then**
    $P_i \leftarrow$ **QSPN-Online** $(\mathtt{n.child}_i, q)$ for each child of n with the columns satisfying $q.A \cap \mathtt{n.child}_i.A \neq \emptyset$
    return $\prod_i P_i$ from all child nodes of n
else if $\mathtt{n}.O = \mathtt{QSplit}$ **then**
    $k \leftarrow$ **RouteQuery** $(q, \{\mathtt{n.child}_i\})$
    return **QSPN-Online** $(\mathtt{n.child}_k, q)$
else
    $C \leftarrow$ **SelectChildren** $(\{\mathtt{n.child}_i\}, q)$
    $P_i \leftarrow$ **QSPN-Online** $(\mathtt{child}, q)$ for each $\mathtt{child} \in C$
    return weighted summation $\sum_i w_i \cdot P_i$
end if

Algorithm 3 `QSPN-Update` $(\mathtt{n}, \Delta T, \Delta Q)$

Input: n: A QSPN Node, ΔT: Data Update subset, ΔQ: New Queries subset
Output: Updated QSPN node $\mathtt{n}'$

$U \leftarrow False$
if $\mathtt{n}.O = \mathtt{Leaf}$ **then**
    $\mathtt{n}' \leftarrow \mathtt{n}$
    Modify Histogram at $\mathtt{n}'$ by ΔT
    return $\mathtt{n}'$
else if $\mathtt{n}.O = \mathtt{Product} \mid \mathtt{QProduct}$ **then**
    $U, \mathtt{n}' \leftarrow$ **ColumnPartitionCheck** $(\mathtt{n}, \Delta T, \Delta Q)$
else if $\mathtt{n}.O = \mathtt{QSplit}$ **then**
    $U, \mathtt{n}' \leftarrow$ **WorkloadSplitCheck** $(\mathtt{n}, \Delta Q)$
else if $\mathtt{n}.O = \mathtt{Sum}$ **then**
    $U, \mathtt{n}' \leftarrow$ **DataPartitionCheck** $(\mathtt{n}, \Delta T)$
end if
if U **then**
    $\mathtt{n}' \leftarrow$ **QSPN-Offline** $(\mathtt{n}'.A, \mathtt{n}'.T \cup \Delta T, \mathtt{n}'.Q \cup \Delta Q)$
else
    $\mathtt{n}'.\mathtt{child}_i \leftarrow$ **QSPN-Update** $(\mathtt{n}'.\mathtt{child}_i, \Delta T_i, \Delta Q_i)$ for each child of $\mathtt{n}'$
end if
return $\mathtt{n}'$

Algorithm 1 `Construct`QSPN (A, T, Q)

Input: A: A Column Set; T: A Table; Q: A Query Workload
Output: n: A QSPN Node

Initialize a QSPN node $\mathtt{n} = (A, T, Q, O)$
if $|A| = 1$ **then**
    $\mathtt{n}.O \leftarrow \mathtt{Leaf}$
    $\mathtt{n}.Pr_T(A) \leftarrow$ **BuildHist** (A, T)
else if $\mathcal{A} \leftarrow$ **PartitionByRDC** (A, T) is *non-singleton* **then**
    $\mathtt{n}.O \leftarrow \mathtt{Product}$
    $\mathtt{n.child}_i \leftarrow$ **Construct**QSPN (A_i, T, Q) for $\forall A_i \in \mathcal{A}$
else if $\mathcal{A} \leftarrow$ **PartitionByAFF** (A, Q) is *non-singleton* **then**
    $\mathtt{n}.O \leftarrow \mathtt{Product}$
    $\mathtt{n.child}_i \leftarrow$ **Construct**QSPN (A_i, T, Q) for $\forall A_i \in \mathcal{A}$
else if $\mathcal{Q} \leftarrow$ **SplitWorkload** (Q, A) is *non-singleton* **then**
    $\mathtt{n}.O \leftarrow \mathtt{QSplit}$
    $\mathtt{n.child}_i \leftarrow$ **Construct**QSPN (A, T, Q_i) for each $Q_i \in \mathcal{Q}$
else
    $\mathtt{n}.O \leftarrow \mathtt{Sum}$
    $\mathcal{T} \leftarrow$ **ClusterTable** (T, A)
    Compute weight $w_i \leftarrow |T_i|/|T|$ for each sub-table $T_i \in \mathcal{T}$
    $\mathtt{n.child}_i \leftarrow$ **Construct**QSPN (A, T_i, Q) for each $T_i \in \mathcal{T}$
end if
return node n

Algorithm 1.1 `PartitionByAFF` (Q, A)

Input: Q: A Query Workload; A: A Column Set
Output: $\mathcal{A} = \{A_1, \dots, A_m\}$: A Collection of Column Sets

for any pair (a_i, a_j) from A **do**
    Compute $\mathtt{AFF}(a_i, a_j|Q)$ for a_i and a_j
end for
Construct a graph $G = (A, E)$ where vertices A represent columns, and an edge $e_{ij} \in E$ exists between columns a_i and a_j if their affinity $\mathtt{AFF}(a_i, a_j|Q)$ is larger than a threshold τ.
$\mathcal{G} \leftarrow$ **ConnectedComponents** (G)
return $\mathcal{A} = \{A_i \mid$ vertex set in each component $G_i \in \mathcal{G}\}$

C.2 Online QSPN Inference Algorithm

We show our top-down **QSPN-Online** (Algorithm 2) based on query routing and pruning rules whose sub function **RouteQuery** is explained in Section 5.1. Benefiting from sufficient optimization utilizing the features of QSPN, the **QSPN-Online** algorithm provides accurate and fast CardEst in online phase using lightweight QSPN model which are constructed both on data and workload before.

C.3 Online QSPN Update Algorithm

We show our efficient top-down **QSPN-Update** (Algorithm 3) based on node examining whose sub function **ColumnPartitionCheck**, **WorkloadSplitCheck** and **DataPartitionCheck** are explained in Section 5.2. The **QSPN-Update** algorithm achieves adaptive incremental QSPN update to keep the model accurate against data update and query workload shift in online phase of dynamic scenes, which is manual-triggering-free and brings little extra cost.