Selectivity Estimation for Queries Containing Predicates over Set-Valued Attributes

ZIZHONG MENG, Nanyang Technological University, Singapore
XIN CAO, University of New South Wales, Australia
GAO CONG, Nanyang Technological University, Singapore

Selectivity estimation aims to estimate the size of query results size accurately and efficiently. Despite being a well-researched area for decades, most existing estimators are designed to handle comparison predicates over numeric and categorical data. Nevertheless, Set-valued data are ubiquitous in many applications such as information retrieval and recommendation systems. However, these estimators may not be effective for handling predicates over set-valued data. In this work, we presents novel techniques for selectivity estimation on queries involving predicates over set-valued attributes. We first propose the set-valued column factorization problem, whereby each each set-valued column is converted to multiple numeric subcolumns, and set containment predicates are converted to numeric comparison predicates. This enables us to leverage any existing estimator to perform selectivity estimation. We then develop two methods for column factorization and query conversion, namely ST and ST-hist. We integrate ST and ST-hist into three estimators, Postgres, Neurocard, and DeepDB. We then conduct a comprehensive empirical analysis by comparing our approach against three baselines across three different datasets. The experimental results demonstrate that our methods exhibit superior estimation accuracy while maintaining high efficiency compared to the baseline techniques.

CCS Concepts: • **Information systems** → **Query optimization**.

Additional Key Words and Phrases: selectivity estimation, set-valued data, query optimization

ACM Reference Format:
Zizhong Meng, Xin Cao, and Gao Cong. 2023. Selectivity Estimation for Queries Containing Predicates over Set-Valued Attributes . *Proc. ACM Manag. Data* 1, 4 (SIGMOD), Article 261 (December 2023), 26 pages. https://doi.org/10.1145/3626755

1 INTRODUCTION

Selectivity estimation aims to estimate the size of a query result, which is a critical task in query optimization to generate cost models and query plans. Despite the prevalence of set-valued data in various applications, such as information retrieval, recommendation systems, and online social networks, most DBMSs and research studies neglect selectivity estimation on such data, focusing instead on queries over numeric and categorical data.

The queries containing predicates over set-valued attributes are prevalent in many applications. Table 1 presents an example dataset of user profiles, where *Age* is a numeric attribute, *User ID*, *User name* and *Location* are categorical attributes, and *Topic of interest* is a set-valued attribute. For example, to find the users aged 20 to 30 residing in Singapore, interested in art and travel, we can use the following SQL query: **SELECT** * **FROM** T **WHERE** $Age \leq$ **30 AND** $Age \geq$ **20 AND** $Location =$ **Singapore AND** ***Topic of interest*** $\supseteq$ **("Art", "Travel")**. This paper investigates the

Authors' addresses: Zizhong Meng, Nanyang Technological University, Singapore, zizhong001@e.ntu.edu.sg; Xin Cao, University of New South Wales, Australia, xin.cao@unsw.edu.au; Gao Cong, Nanyang Technological University, Singapore, gaocong@ntu.edu.sg.

2836-6573/2023/12-ART261
https://doi.org/10.1145/3626755

User ID	*User name*	*Age*	*Location*	*Topic of interest*
CD3487	John	25	US	Sport, Technique
AX9083	Tom	29	Singapore	Art, Sport, Travel
EG6754	Jerry	31	Singapore	Business, Technique, Travel
VC1364	Linda	26	Singapore	Art, Education, Travel
UY1638	Lily	23	US	Animation, Art

Table 1. Example table.

problem of selectivity estimation for queries containing predicates over set-valued attributes (*Topic of interest*) as well as numeric (*Age*) and categorical (*Location*) attributes. Another application of our study is **approximate query processing** for queries such as the popularity of "Art, Travel" in Singapore in a certain age group (i.e., a COUNT query). Our study can also be used on other data types with containment search operators. For example, containment operators in PostgreSQL for json [20] and hstore [19] types are similar to set containment search operator. We could also utilize our study on spatial containment functions (e.g., ST_Covers and ST_Contains in PostGIS [4, 5]) by taking each point or polygon as an element.

Limited support for set-valued attributes is provided by some DBMSs. For example, MySQL supports the set data type, but only up to 64 distinct types are allowed in one set column [31]; SQL server enables passing a set of data in a table-valued parameter [38]. However, they do not provide general selectivity estimation over set-valued data. Postgresql defines the array type for set-valued attributes and supports set containment operators. To the best of our knowledge, Postgresql is the only DBMS that offers a built-in selectivity estimator for set containment operators. There exists very little research on selectivity estimation for predicates over set-valued data, but they still have the following problems:

Favor highly frequent elements. Yang et al. propose two sampling-based methods for selectivity estimation on superset query [43]. Specifically, OT-sampling utilizes trie structures, while DC-sampling uses divide-and-conquer techniques. Both methods aim to capture the distribution of high-frequency elements. Thus, they may have limited performance on queries containing infrequent elements.

Cannot well capture the set element dependency. Postgresql treats each element as a seperate attribute (column) and introduces a basic model and an extended model [9] for selectivity estimation of set containment predicates [2, 23]. The basic model does not take into account any element dependencies. The extended model considers the distribution of set cardinalities, but this model still relies on random sampling of high-frequency elements, and thus it can only capture the dependency of high-frequency elements.

Cannot well capture the attribute correlations. One potential solution to handle queries with multiple attributes is to use the method proposed by [43] for set containment predicates, and to use existing estimators for other predicates. However, this approach neglects the correlations between the set-valued attribute and other attributes.

Challenges. State-of-the-art selectivity estimators aim to learn the joint probability distribution without attribute value independence (AVI) assumption. For example, Neurocard [44] adopts deep autoregressive models to learn the joint data distribution, while Deepdb [18] utilizes Sum-Product networks (SPNs) to capture the joint data distribution. A naive method for using existing selectivity estimators on set-valued attributes is to treat each element as an attribute (column), and the set-valued attribute is converted into multiple categorical columns. However, this approach is obviously inaccurate and inefficient due to the large number of columns and large search space, as well as the data sparsity in such a table. Another approach is to treat each set record as one attribute value, which can be considered as converting a set-valued column into a categorical column. However, this method can result in an extremely large domain size when the number of elements is very

large, making it difficult to learn the data distribution. Moreover, this method requires finding all subsets (resp. supersets) for a subset (resp. superset) query, which could be very slow.

Our solutions. Instead, we propose a novel approach for converting a set-valued attribute into a small number of numeric attributes by assigning orders to the values in each attribute (each attribute consists of a subset of elements). By doing so, we can convert a predicate over the set-valued attribute into predicates over the newly created numeric attributes. Ideally, the distribution after conversion should be easy to learn. A significant benefit of our idea is that it enables the utilization of any data-driven estimators for selectivity estimation after the conversion process. These estimators can capture the correlations between the converted attributes.

However, how to do the conversion is non-trivial. Selectivity estimation concerns both estimation accuracy and estimation efficiency. Estimators [36, 45] often prefer using fewer columns to decrease the estimation time and enhance the accuracy of their estimations. Hence, to facilitate the capture of attribute correlations by existing estimators, the conversion should not result in many columns. For each converted column, a large domain size would also lead to complex data distribution, resulting in poor estimation accuracy (e.g., histograms) or heavy estimation costs (e.g., learning-based models). Thus, it is preferable to have a small domain size in each converted column, and this will facilitate estimators to learn the data distribution to capture the element dependency. Another requirement is that the conversion will not dominate the query estimation time while providing high accuracy.

To address the challenges of conversion, we introduce the set-valued column factorization problem, which aims to convert set-valued data and queries. To address this problem, we propose two novel methods: ST and ST-hist. ST factorizes the set-valued column into a fixed number of numeric subcolumns and constructs a prefix tree (trie) structure on each subcolumn to search for subsets and supersets. While ST has a bounded number of subcolumns, it may not be efficient for dataset with a large number of elements and records as it can result in extremely large trie structures. Therefore, we propose ST-hist, which builds upon ST by replacing some subtrees of the trie with histograms to reduce the number of nodes in the trie. This technique enables ST-hist to achieve a linear conversion time in the number of elements in the subcolumn, making it more efficient and scalable than ST.

Contributions. Our work has the following contributions:

- To estimate the selectivity of queries containing predicates over set-valued attributes, we propose converting a set-valued column to numeric subcolumns, and then convert a set containment predicate to comparison predicates. This approach enables us to use existing data-driven estimators for estimation. To the best of our knowledge, this is the first work to focus on selectivity estimation for queries with predicates over set-valued attributes and other types of attributes without the AVI assumption.
- We define the set-valued column factorization problem for the conversion, and develop two new methods to solve it. ST conducts predicate conversion by constructing a trie structure. ST-hist is developed based on ST by constructing histograms on some nodes of the trie to replace and represent subtrees and performs approximate subset search and superset search.
- We integrate our methods into three existing estimators and compare them with three baselines on three real-life datasets. The results show that our methods outperform baselines by up to 120 times for accuracy while having similar estimation efficiency.

2 RELATED WORK

Our work focuses on selectivity estimation for queries that involve predicates over set-valued, numeric, and categorical attributes. While most existing estimators are designed on querying

numeric and categorical data, only a limited number of techniques have been developed for selectivity estimation over set-valued data.

Estimators for querying set-valued data. Yang et al. proposes two distinct sampling based selectivity estimation techniques for subset and superset queries. OT-sampling leverages a trie structure to capture high-frequency patterns [43] and achieves high accuracy when querying high frequent elements. However, this method may exhibit lower accuracy when querying low-frequency elements. DC-sampling employs a query-oriented divide-and-conquer strategy [43], and achieves performance that is highly dependent on the query distribution. Like other sampling-based methods, the performance of both techniques is sensitive to the sampling ratio. Hadjieleftheriou et al. propose a hash sampling algorithm for selectivity estimation in set similarity queries [13], which differ from set containment queries. PostgreSQL treats each element as a binary attribute and employs either independence assumptions or probabilistic models [9] to estimate the selectivity of subset queries and superset queries [23]. However, PostgreSQL relies on the AVI assumption for selectivity estimation. To the best of our knowledge, no existing estimator can accurately estimate selectivity for queries that involve predicates over both set-valued attributes and other attribute types without the AVI assumption.

Estimators for querying numeric and categorical attributes. Data-driven methods build models from datasets. Sampling based methods [12, 27] estimate selectivity from a subset of the data. Their performance is highly dependent on space consumption. Histogram methods [6, 11, 30, 33–35] build buckets to approximate the data distribution and make independence or partial independence assumptions. Probability models [9, 14] use Bayesian networks with conditional independence assumptions, but they are usually slow in estimation. Kernel density estimation (KDE) [17, 21] learns the distribution without any independence assumption, but its performance relies on bandwidth adjusting. Autoregressive models [16, 44, 45] decompose the joint distribution with product rule without any independence assumption and have high accuracy but low efficiency and require large space storage. Sum-product network (SPN) based methods [18, 46] decompose data into multiple SPNs and are efficient in estimation, but their accuracy is not very competitive. FACE [40] learns the joint distribution with normalizing flow. Iris [28] uses pre-trained summarization models without per-dataset training. Query-driven methods [1, 7, 22, 26, 32, 39] estimate the selectivity by leveraging the query workloads, and hybrid methods [17, 21, 42] construct models from both data and queries. Our proposed methods can be integrated into any data-driven estimator for selectivity estimation.

3 PROBLEM FORMULATION

Consider a relation T with n columns (attributes) denoted by $\{C_1, \cdots, C_n\}$, where each column could be set-valued, numeric, or categorical and has a finite element universe $\Sigma_i = \{e_1, \cdots, e_m\}$. If C_i is a set-valued column, each value t_i^j in C_i is a subset of Σ_i: $t_i^j \subseteq \Sigma_i$. For instance, *Topic of interest* in Table 1 is a set-valued column. If C_i is a numeric or categorical column, each value t_i^j of C_i is an element in Σ_i: $t_i^j \in \Sigma_i$. For instance, *Location* is a categorical column and *Age* is a numeric column in Table 1. The size of T is denoted by $|T|$ and the element universe size of C_i by $|\Sigma_i|$. The domain of C_i, denoted by $dom(C_i)$, consists of all distinct records in C_i and the size of the domain is denoted by $|C_i|$. Note that $dom(C_i) = \Sigma_i$ for numeric and categorical columns.

Definition 1 (Query Predicate). *A query can be defined as a conjunction of predicates on single columns, expressed as $q = q_1 \times q_2 \times \cdots \times q_l$. Each predicate consists of a column, an operator and a literal, i.e. $q_i = \langle col, op, val \rangle$. We consider predicates on set-valued columns, numeric columns and categorical columns in this work. For a predicate on a set-valued column, the operator could be subset operator or superset operator, and the literal is a set value. E.g., $q_i = \langle C_1 \subseteq (1, 2, 4) \rangle$ is a subset*

predicate on C_1 and $q_i = \langle C_1 \supseteq (1, 3)\rangle$ is a superset predicate on C_1. For a predicate on a numeric or a categorical column, the operator could be point operator or range operator, and the literal is a numeric value or categorical value. E.g., $q_i = \langle C_2 > 5\rangle$ is a range predicate and $q_i = \langle C_2 = 1\rangle$ is a point predicate.

Definition 2 (Selectivity Estimation). *The selectivity of a query predicate q is defined as the proportion of tuples in the table T that satisfy the predicate q, i.e. $sel(q) = \frac{1}{|T|}\sum_{t\in T}\theta(t)$, where $\theta(t) = 1$ if the tuple t satisfies q, and $\theta(t) = 0$ otherwise. The objective of selectivity estimation is to estimate the selectivity of a query predicate without executing the query. Specifically, $actsel(q)$ represents the actual selectivity of q, while $estsel(q)$ represents the estimated selectivity.*

Problem. Given a relation T with set-valued, numeric and categorical columns, the objective of this study is to perform selectivity estimation *accurately* and *efficiently* on queries with range and/or point predicates on numeric and/or categorical columns, as well as set containment search predicates on set-valued columns.

4 PROPOSED METHODS

We first discuss the challenges of the problem and define a set-valued column factorization problem to address these challenges in Section 4.1. Next, we develop two methods to solve the set-valued column factorization problem, i.e., ST (Section 4.3) and ST-hist (Section 4.4).

4.1 Conversion of Set-valued Columns and Set Containment Search Predicates

C
a, b
b, c, e
a, b, d
d, e

Table 2. A set column.

C'_a	C'_b	C'_c	C'_d	C'_e
1	1	0	0	0
0	1	1	0	1
1	1	0	1	0
0	0	0	1	1

Table 3. Convert each element to a binary column.

C'
$a, b \rightarrow 0$
$b, c, e \rightarrow 1$
$a, b, d \rightarrow 2$
$d, e \rightarrow 3$

Table 4. Convert the set column to a numeric column.

$C'_1(a, b)$	$C'_2(c, d)$	$C'_3(e)$
$a, b \rightarrow 0$	$null \rightarrow 0$	$null \rightarrow 0$
$b \rightarrow 1$	$c \rightarrow 1$	$e \rightarrow 1$
$a, b \rightarrow 0$	$d \rightarrow 2$	$null \rightarrow 0$
$null \rightarrow 2$	$d \rightarrow 2$	$e \rightarrow 1$

Table 5. Convert the set column into multiple numeric subcolumns.

To solve the problem of selectivity estimation for queries containing predicates over set-valued attributes, we need to address two challenges: how to learn the joint element distribution in the set-valued columns and how to capture the column correlations in the datasets. However, none of the existing methods can address both challenges simultaneously. For example, the method proposed in [43] could work with other estimators to handle set containment predicates and other types of predicates, but they work independently, neglecting the correlations between set-valued columns and other columns.

State-of-the-art data-driven estimators for categorical and numeric data can effectively capture column correlations by approximating the joint data distribution. This motivates us to convert set-valued columns to numeric columns first and then use these estimators to perform the estimation.

However, how to do the conversion to capture the element dependencies and column correlations is non-trivial. Two naive methods are converting each element to a binary column or converting the entire set-valued column into one numeric column, but both suffer from certain drawbacks as explained below.

Convert each element to a binary column. Table 3 illustrates an example conversion of Table 2. For example, the record (a, b) is converted to $(1, 1, 0, 0, 0)$. Given this approach, a set containment predicate can be efficiently converted to predicates with comparison operators on the converted table. For example, a subset predicate "$C \subseteq (a, b, c)$" on Table 2 can be converted to the predicate "$C'_d = 0$ AND $C'_e = 0$" on Table 3, while a superset predicate "$C \supseteq (a, b)$" on Table 2 can be converted to the predicate "$C'_a = 1$ AND $C'_b = 1$" on Table 3. However, set-valued columns typically follow a power law distribution, where most elements only appear in a few records. As a result, the data distribution after conversion is sparse and skew, making it difficult to learn. In addition, such a method generates a large number of columns in the converted table if the element universe (denoted as Σ) is large, leading to two problems. First, the search space becomes too large ($2^{|\Sigma|}$), making it challenging to learn the joint distribution of the columns in the converted table. Second, it affects the estimation efficiency since the complexity of most estimators depends on the number of columns.

Convert the set-valued column to one numeric column. This can be achieved by assigning a distinct value to each distinct set record in the domain. For example, the set-valued column C in Table 2 is converted to column C' in Table 4, where (a, b), (b, c, e), (a, b, d) and (d, e) are mapped to 0, 1, 2 and 3, respectively. The search space for this method is the domain size of the column, which is smaller than that of the first method, making it easier to learn the distribution. However, this method requires enumerating and searching all subset (resp. superset) records when estimating a subset (resp. superset) predicate, which can be extremely slow with a large domain.

Based on the above observations, we propose to convert each set-valued column into a few numeric subcolumns by assigning each element to one subcolumn and converting each set record to a new record with numeric values in the subcolumns. With this conversion, the number of converted columns is small and it is expected that the search space and the domain size of each subcolumn are acceptable. Table 5 is converted from Table 3 by assigning a, b to subcolumn C'_1, c, d to subcolumn C'_2 and e to subcolumn C'_3. Considering the subcolumn C'_1, the four set records in C are mapped to (a, b), (b), (a, b) and *null* in C'_1 respectively, so there are three distinct set values (a, b), (b) and *null* in C'_1, which are mapped to 0, 1 and 2, respectively. The mapping in the other two subcolumns is similar. The search space of this method is the product of the domain size of all subcolumns, which is much smaller than the first naive approach (12 in Table 5 v.s. 32 in Table 3), so it is easier to learn the joint distribution accurately. Compared with the second naive approach, the domain size of each subcolumn is smaller. For example, the maximum domain size in Table 5 is 3 while the domain size in Table 4 is 4. Hence, the data distribution of each column is easier to learn and the predicate conversion is faster.

Now the problem is how to partition elements into subcolumns. Based on the preliminary experiments on existing estimators, we observed most estimators have better accuracy and efficiency on datasets with fewer columns and smaller domain sizes. Hence, we aim to minimize the domain size of each subcolumn.

The next problem is how to perform predicate conversion. Given a set containment search predicate over the set-valued column, we need to convert it to comparison predicates over the converted numeric subcolumns so that existing estimators can handle the converted predicates. We expect an efficient conversion that is much faster than the estimation step. The predicate conversion is determined by the column conversion step, but it could be very slow on set-valued data with

a large element universe and a large domain size. Hence, we consider converting the set-valued column approximately to reduce the domain size of each numeric subcolumn. For example, we could use the same numeric value to represent *null* and c in C'_2 in Table 5, and then the domain size of C'_2 is reduced from 3 to 2. With the approximate set-valued data conversion, the selectivity of the converted predicate is not equal to the selectivity of the original predicate due to information loss. For example, considering a predicate $q : C \supseteq (c)$ on Table 2, the converted predicate q' on Table 5 should be $C'_2 = 1$. q and q' are equivalent and both of their selectivities are $\frac{1}{4}$ (the second record). However, if we consider the approximate set-valued data conversion that uses 1 to represent both *null* and c in C'_2, the selectivity of q' is $\frac{2}{4}$ (the first and second records), which is not equal to the selectivity of q. Hence, we also aim to reduce the error using the converted predicates when performing the approximate set-valued column conversion.

Based on all observations and discussions above, we give the definition of the set-valued column factorization problem below.

Definition 3 (Set-valued column factorization). *Given a set-valued column C with element universe Σ, find a method f to factorize C into k numeric subcolumns: $f(C) = C' = \{C'_1, \cdots, C'_k\}$, and find a function g to convert a set containment predicate q on C to the conjunction of multiple comparison predicates $q' = g(q) = q'_1 \times q'_2 \times \cdots \times q'_k$ on C'. The objectives of finding f and g are as follows:*

- *minimizing the maximum domain size of subcolumns* $\max_{i=1}^{k} |C'_i|$.
- *minimizing the error between the actual selectivity of q and $g(q)$, measured by q-error [29]:* $error(q, g(q)) = \max(\frac{actsel(q)}{actsel(g(q))}, \frac{actsel(g(q))}{actsel(q)})$.
- *minimizing the time complexity of g.*

The function f contains 3 steps. (1) Partition the elements into k clusters. (2) Factorize C into k subcolumns $C' = \{C'_1, \cdots, C'_k\}$ that for each set-valued record t in C, the new set value t'_i in C'_i is a subset of t that contains only those elements in the i-th cluster. (3) For each set value in each subcolumn, map it to a numeric value. Table 5 is an example of set-valued column factorization based on Table 2.

Note that after the conversion, all predicates will become predicates with comparison operators over numeric and categorical columns, including the converted subcolumns, and then existing estimators can be utilized to estimate selectivity. This is a significant benefit of our proposed solutions. Since we handle each set-valued column factorization separately, for ease of presentation, we consider the relation T with only one set-valued column C to describe our proposed conversion methods. We denote the element universe of C by Σ, the element universe of each subcolumn C'_i by Σ'_i. Given a subset predicate or superset predicate q, we denote the set value in q by t_q, and the number of elements in t_q by $|q|$. We use column C in Table 2 as the example in the rest of this paper.

4.2 Partitioning Elements

We proceed to present the first two steps of our proposed function f, i.e., (1) how to partition the elements into k clusters, and (2) factorize column C into k subcolumns.

One simple idea is to adopt the greedy approach to minimizing the maximum domain size of subcolumns. We regard each element as a column and randomly select k columns as seed columns. For each non-seed column, we merge it into the seed column which has the minimum domain size after merging. We update the seed column data after merging. Consider Table 2 and Table 3. Suppose we select C'_a and C'_c as two seed columns, and we merge other columns: For C'_b, the domain size of C'_a and C'_c will be 2 and 3 respectively if C'_b is merged into them. Hence, we merge C'_b into C'_a.

Similarly, C'_d and C'_e are merged into C'_c and C'_a, respectively. However, the cost of this algorithm is $O(k|\Sigma||T|)$, which is very expensive when $|\Sigma|$ and $|T|$ are large.

To improve the partitioning efficiency, we proceed to present a two-phase partitioning algorithm. We observe that the domain size of a cluster depends on the relationships of the elements in it. The domain size would be large if the elements in this cluster co-occur in many records with different combinations. This motivates us to first group elements into l clusters such that no elements in a cluster are contained in the same records. By doing so, the domain size of each cluster is the number of elements in this cluster, which is much smaller than the case that the elements co-occur in many same records. Then the second step is using the greedy algorithm to further partition the l clusters into k clusters. The cost of the second phase partition is $O(kl|T|)$. To reduce the cost, we aim to minimize the number of clusters l in the first step. Now we give the following problem definition.

Definition 4 (Minimum element independence partition). *Given a set-valued dataset T with element universe $\Sigma = \{e_1, \cdots, e_m\}$, find a minimum number of k partitions $P = (P_1, \cdots, P_k)$ that:*

- $\forall 1 \leq i < j \leq k, P_i \cap P_j = \phi$.
- $\bigcup_{i=1}^{k} P_i = \Sigma$.
- $\forall t \in T, \forall e_i, e_j \in t$ *that* $i \neq j, p(e_i) \neq p(e_j)$, *where* $p(e_i)$ *is the partition of* e_i.

This problem can be solved as a graph coloring problem. We construct a graph $G = (V, E)$ based on T, where $V = \{e_i | e_i \in \Sigma\}$ and $E = \{(e_i, e_j) | \exists t \in T, e_i \in t \wedge e_j \in t \wedge e_i \neq e_j\}$. The minimum element independence partition problem is equivalent to finding the chromatic number of G, i.e., the smallest number of colors to color all vertices in G such that no adjacent vertices share the same color. Graph coloring is known as an NP-hard problem [8] and many approximation algorithms have been developed. We use the largest first (LF) algorithm and the cost of the LF algorithm in our problem is $O(|V| + |E|)$ [41].

In summary, the two-phase partitioning algorithm works as follows. First, we use the LF algorithm to partition all elements into l element independence clusters, and then use the greedy algorithm proposed at the beginning of this section to further partition the l cluster into k clusters. The total cost is $O(|\Sigma| + |E| + kl|T|)$, where $|E|$ is the number of edges in graph G.

4.3 Settrie based Method ST

With the partitioned elements and subcolumns, it is necessary to map each set value in a subcolumn to a numeric value as the third step of factorization function f. A straightforward method for doing this is to use a hash function to map each distinct set value in the subcolumn to a unique numeric value. However, this method is highly inefficient for query conversion g, as it requires traversing all distinct set values in the subcolumn in order to find subsets (supersets) of a given subset (superset) search predicate.

To improve the query conversion efficiency, we propose a method called ST, developed from the settrie structure [37]. We construct a settrie structure for each subcolumn. Since the settrie for each subcolumn is constructed independently, for ease of discussion, we first assume that all elements are partitioned into one subcolumn C', and then we will discuss the benefits of multiple subcolumns in the cost analysis.

Settrie construction. Algorithm 1 provides the settrie construction procedure. Firstly, the elements are sorted in the descending order of frequency (line 1). The sorting order will be discussed in the cost analysis part. All tuples are inserted into the trie and are labeled in the trie. For each ordered tuple in the dataset, it is inserted from the root node. For each element e in the tuple, if the current node has a child with $element = e$, the current node is moved to this child, and the next element in the tuple is considered (lines 14–15). If the current node does not have a child

with $element = e$, a new node is constructed with $element = e$ and inserted into the children of the current node. The current node is then moved to the new node, and the next element in the tuple is considered (lines 16–19). When all elements in the tuple are traversed, the node with the last element is set as $last\_flag$, indicating that all elements in the path from root node to this node form a tuple in the dataset (line 20). We use the last node in the path to represent the tuple. After the settrie is constructed, a new label is given to each node to represent the corresponding tuple as a numeric value (line 7). We initialize $label = 0$ and use depth-first search (DFS) to label tuples. When the $last\_flag$ of a node is True in DFS, we set $node.label = label$ to represent the tuple and set $label = label + 1$ (line 24–26). Since all nodes in the subtree of a node are supersets of the node, we use the range $[node.start, node.end)$ to represent these supersets of the node. With the settrie, each set-valued record in the T is replaced with its corresponding numeric value in the settrie, and then the set-valued column C is converted to a numeric column C' (line 8). Figure 1 gives an example of settrie constructed based on column C in Table 2. All elements in C are ordered based on their frequency: (b, a, d, e, c). Nodes with heavy borders are nodes with $last\_flag$, and the red numbers are labels of the nodes representing tuples. The range in blue of a node matches supersets of the node.

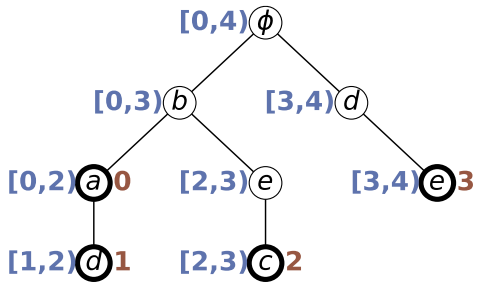


Fig. 1. Settrie on column C in Table 2.

For each set containment predicate, we need to convert it to comparison predicates on the numeric column without loss of information. Next we present the conversion algorithm for containment predicate.

Subset search. Given a subset search predicate q with an ordered set t_q, a tuple in the settrie is a subset of t_q if it can match some elements in t_q. Algorithm 2 returns all subsets of t_q by recursivly searching elements in t_q using the Function `subset_search`. We first input the root node of settrie and t_q into the Function `subset_search` (line 1–2). In each function call, if the $last\_flag$ of the current node is True, the current node is a subset of t_q, so we add the label of current node into the result array (line 5–6). If all elements in t_q have been searched, there is no need to search the children of the current node (line 7–8). Otherwise, we check all children of current node. If the element of a child node is current element in t_q, we search for the next element in t_q in the children of the child node; If the element of a child is larger than current element in t_q, we search for the next element in t_q in the children of the current node (line 9–13).

Superset search. Given a predicate q with an ordered set t_q, a tuple in the settrie is a superset of t_q if it contains all elements in t_q. Function `superset_search` in Algorithm 3 recursively finds all supersets of t_q in a settrie $node$. We initially pass the root node of settrie and the sorted predicate value t_q into the function. In each function call, if we have already searched all elements in t_q, the current node and all nodes in its subtree are supersets of t_q. Hence, we add them to the result array (line 5–7). Otherwise, we search the children of the current node. If the element of a child is in the range of the previous and current elements in t_q, we continue to search the current element in

Algorithm 1: Settrie construction

Input: set-valued column C, element universe Σ
Output: settrie $root$, numeric subcolumn C'

```
1  Sort the elements in the descending order of frequency;
2  Initialize root node root = ϕ;
3  for tuple in C do
4      Sort tuple with elements order;
5      set_insertion(root, tuple);
6  Initialize label = 0;
7  label_settrie(root, label);
8  Map C to a numeric subcolumn C' based on root;
9  return root, C';
10 Function set_insertion(root, tuple):
11     node = root;
12     for e in tuple do
13         children = node.children;
14         if ∃child ∈ children, child.element = e then
15             node = child;
16         else
17             Construct a new_node with element = e;
18             Insert new_node into children;
19             node = new_node;
20     node.last_flag=True;
21 end
22 Function label_settrie(node, label):
23     node.start = label;
24     if node.last_flag is true then
25         node.label = label;
26         label = label + 1;
27     for child in node.children do
28         label_settrie(child, label);
29     node.end = label;
30 end
```

the children of the child; If the element of a child is the same as the current element in t_q, we will search the next element in the children of the child (line 10–14).

Cost analysis. With an element universe Σ, there are at most $2^{|\Sigma|}$ different sets in a set-valued column. For a fixed probability $p \in (0, 1)$, we assume that for each $t \subseteq \Sigma$, the probability t in the column C is $p(t \in T) = p$. Hence, there are $p \cdot 2^{|\Sigma|}$ distinct sets in T. Given a predicate q, we use $\mathbb{E}_{sub}(q)$ and $\mathbb{E}_{sup}(q)$ to denote the expected number of nodes visited for subset search and superset search respectively. For subset search, Algorithm 2 only visits the nodes with element $e \in q$. Hence, the upper bound of $\mathbb{E}_{sub}(q)$ is $\sum_{i \in idx(q)} (\frac{2+(1-p)^{2^{|\Sigma|-i-1}}}{1+(1-p)^{2^{|\Sigma|-i-1}}})$ [37], where $idx(q)$ converts all elements in q to the indices in the element universe based on the descending order of frequency. For superset search, Algorithm 3 only visits elements whose indices are smaller than the index of q_m in the given order, where q_m is the last element in t_q when t_q in the descending order of frequency. Hence, the upper bound of $\mathbb{E}_{sup}(q)$ is $\sum_{i=0}^{idx(t_q[-1])-1} (\frac{2+(1-p)^{2^{|\Sigma|-i}}}{1+(1-p)^{2^{|\Sigma|-i}}})$ [37], where $idx(t_q[-1])$ is the index of the

Algorithm 2: Subset search

Input: settrie *root*, predicate q with set value t_q in element order
Output: subsets of t_q

```
Initialize result array r = φ;
subset_search(root, t_q, 0, r);
return r;
Function subset_search(node, t_q, idx, r):
    if node.last_flag is True then
        Add node.label to r;
    if idx = |q| then
        return;
    for child in node.children do
        if child.id = t_q[idx] then
            subset_search(child, t_q, idx + 1, r);
        else if child.id > t_q[idx] AND idx < q.size then
            subset_search(node, t_q, idx + 1, r);
```

Algorithm 3: Superset search

Input: settrie *root*, predicate q with value t_q in element order
Output: supersets of t_q

```
Initialize result array r = φ;
superset_search(root, t_q, 0, r);
return r;
Function superset_search(node, t_q, idx, r):
    if idx = |q| then
        Add the pair (node.start, node.end) into r;
        return;
    from = t_q[idx − 1] if idx > 0 else from = 0;
    upto = t_q[idx];
    for child in node.children do
        if child.id in (from, upto) then
            superset_search(child, t_q, idx, r);
        else if child.id = upto then
            superset_search(child, t_q, idx + 1, r);
```

last element of t_q in the element universe based on the descending order of frequency. To reduce the time of subset search and superset search, we could reduce the probability p, which is reducing the number of nodes in the settrie. For an element e as the i-th element when all elements are given an order, it will appear at most $\min(2^i, S(e))$ times, where $S(e)$ is the frequency of e in the column. Hence, we give the elements with higher frequency smaller labels to reduce the number of nodes. Therefore, we sort all elements in the descending order of frequency.

Settrie on multiple subcolumns. When $|\Sigma|$ is very large, $\mathbb{E}_{sub}(q)$ and $\mathbb{E}_{sup}(q)$ could be very large. To reduce the search time, we partition all elements into k subcolumns to reduce $|\Sigma|$ and split each record in the column into the k subcolumns. Then we construct a settrie for each subcolumn with Algorithm 1. Given a predicate q with set value t_q, we first split t_q into k parts based on the elements partition. Then for each subcolumn, we construct a predicate on the subcolumn with Algorithm 2 or Algorithm 3. The final predicate is the conjunction of all predicates with the AND

operator. Suppose the subcolumn C'_i has $|\Sigma'_i|$ elements and q_j is the elements in C'_i from q. The upper bound of $\mathbb{E}_{sub}(q)$ is $\sum_{j=0}^{k}\sum_{i\in idx(q_j)}(\frac{2+(1-p)^{2^{|\Sigma'_j|-i-1}}}{1+(1-p)^{2^{|\Sigma'_j|-i-1}}})$. For an element e in the subcolumn C_i, suppose $idx(e)$ is the index of e in Σ and $idx_i(e)$ is the index of e in Σ_i based on the descending order of frequency. we could get $|\Sigma| - idx(e) \geq |\Sigma_i| - idx_i(e)$ since the number of elements in Σ with lower frequency than e is larger than the number of elements in Σ_i with lower frequency than e. Hence, for any element e, $\frac{2+(1-p)^{2^{|\Sigma_i|-idx_i(e)-1}}}{1+(1-p)^{2^{|\Sigma_i|-idx_i(e)-1}}} \leq \frac{2+(1-p)^{2^{|\Sigma|-idx(e)-1}}}{1+(1-p)^{2^{|\Sigma|-idx(e)-1}}}$. Since all elements in q_j are from q, we could get $\sum_{j=0}^{k}\sum_{i\in idx(q_j)}(\frac{2+(1-p)^{2^{|\Sigma_j|-i-1}}}{1+(1-p)^{2^{|\Sigma_j|-i-1}}}) \leq \sum_{i\in idx(q)}(\frac{2+(1-p)^{2^{|\Sigma|-i-1}}}{1+(1-p)^{2^{|\Sigma|-i-1}}})$. Similarly, the upper bound of $\mathbb{E}_{sup}(q) = \sum_{j=0}^{k}\sum_{i=0}^{idx(q_j)-1}(\frac{2+(1-p)^{2^{|\Sigma_j|-i}}}{1+(1-p)^{2^{|\Sigma_j|-i}}}) \leq \sum_{i=0}^{idx(q)-1}(\frac{2+(1-p)^{2^{|\Sigma|-i}}}{1+(1-p)^{2^{|\Sigma|-i}}})$. Hence, subset and superset sesarch on multiple subcolumns are faster than them on single column.

4.4 Settrie with Histograms ST-hist

Challenges on ST. Section 4.3 discusses the cost of ST which is still exponential in the worst case with respect to the element universe size $|\Sigma|$. Another challenge is the large domain size on some datasets. Most estimators provide better estimation accuracy and efficiency on datasets with small column domain sizes. Therefore, we consider another method ST-hist, which has a fixed number of numeric subcolumns and a fixed domain size for each subcolumn, while minimizing the the error of actual selectivity of q and $g(q)$. ST-hist constructs histograms on the nodes of settrie with low frequencies. We also suppose all elements are in 1 subcolumn C' first for ease of discussion.

Given a user-defined number of nodes S, we keep the S nodes with the highest frequency in the settrie. Since the frequency of a node is the sum of frequencies of its children, all nodes in the subtree of a node must be removed if the node is removed. Hence, the removed nodes could construct multiple subtrees of the settrie, where the leaf nodes in subtrees are also leaf nodes in the settrie. For example, if we keep 2 nodes of the settrie in Figure 1, nodes b and a are kept. For each removed subtree, we construct histograms on its parent node.

Histogram construction. Algorithm 4 provides the details of histogram construction. Given a settrie, we first add frequency to each node in the settrie (line 1). The frequency of each node is initialized as 0. For each ordered tuple t in the set-valued column, we search t in the settrie and add 1 frequency to the matched nodes. We obtain the minimum frequency θ for the kept S nodes (line 2). All nodes with a frequency smaller than θ will be removed. We use the function `filter_nodes` to construct histograms, remove low frequency nodes and relabel high frequency nodes (line 3). In each function call, we first label the node if its $last\_flag$ is True and label the range of its supersets (line 14–17), which is similar to Algorithm 1. Then for each child of the current node with frequency lower than θ, we add the frequencies of the nodes in its subtree into a hashmap $freq\_map$ (line 18–20, line 8–11) and sum the frequencies of these nodes (line 21). Next, we construct a histogram (line 22–23) and keep a histogram label on the current node such that the labels of these low frequency nodes are replaced by this histogram label (line 24–26). For the children with a higher frequency than θ, we keep these nodes and continue to search their children (line 27–29). After all nodes have been filtered, we map each tuple $t \in C$ to a numeric value by searching for t in the settrie and obtain a numeric subcolumn C' (line 4). Then we remove all nodes with $keep$ = False (line 5).

Figure 2 gives an example of histogram construction. Suppose Figure 2a is a subtree of a full settrie. Numbers in red are labels of nodes and numbers in blue are frequencies of nodes. Suppose we want to keep 3 nodes on this subtree. Then the nodes in red are kept and the nodes in blue will

be merged into histograms. Figure 2b shows the subtree with histograms. The blue nodes are not actual nodes, but are histograms on node a, b and c. We first use a label to represent the histograms, and then we continue to label the unmerged nodes. E.g., we first label the histogram of node a as 0, and we recursively consider node b and node c respectively.

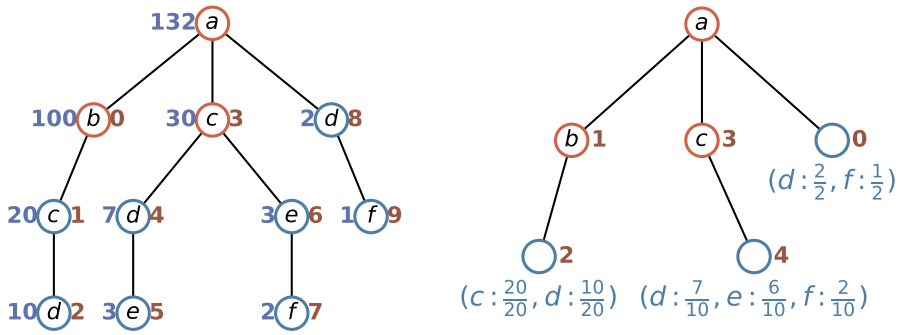
Algorithm 4: Histograms construction

Input: settrie $root$, set-valued column C, number of nodes S
Output: settrie with histograms $root$, numeric subcolumn C'

Add frequency to each node in $root$ as $node.freq$ based on C;
Get the minimum frequency θ of the S highest frequency nodes;
`filter_nodes`($root$, θ, 0);
Map C to a numeric subcolumn C' based on $root$;
Remove the nodes with $keep$ = False in $root$;
return $root$, C';
Function `get_frequency`($node$, $freq\_map$, $label$)**:**
    $node.keep$ = False;
    Add $node.freq$ to $freq\_map$;
    for *child in node.children* **do**
        `get_frequency`($child$, $freq\_map$, $label$);
end;
Function `filter_nodes`($node$, θ, $label$)**:**
    $node.start = label$;
    if *node.last_flag is True* **then**
        $node.label = label$;
        $label = label + 1$;
    Initialize a hashmap $freq\_map$;
    for *child in node.children with* $child.freq \leq \theta$ **do**
        `get_frequency`($child$, $freq\_map$, $label$);
    Sum the frequencies of not kept children as $total\_freq$;
    for *id in* $freq\_map$ **do**
        $node.hist[id] = \frac{freq\_map[id]}{total\_freq}$;
    if *node has histograms* **then**
        $node.hist\_label = label$;
        $label = label + 1$;
    for *child in node.children with* $child.freq > \theta$ **do**
        $child.keep$ = *True*;
        `filter_nodes`($child$, θ, $label$);
    $node.end = label$;
end

Given a set containment search predicate q, we need to convert it to comparison predicate based on the histogram based settrie. The search algorithm is different from the search algorithm in ST. Since one histogram label could represent different tuples in T not all these tuples are subsets or supersets of the given predicate value q_t, the conversion algorithm in Section 4.3 is biased for ST-hist. Hence, we give the conversion algorithm for ST-hist below which tries to reduce the bias.

Subset search. When we use Algorithm 2 on a predicate q with t_q and meet a node with histogram, we add the histogram label l into the result array. Suppose the current unmatched elements in t_q are t_q^r. We first initialize the probability $P = 1$. For each element e in the histogram, if e is not in



(a) Original settrie (b) Settrie with histograms

Fig. 2. Example of histograms based settrie construction.

t_q^r, we calculate $P = P \times (1 - p(e))$, where $p(e)$ is the probability of e in the histogram, i.e., $p(e) = node.hist[e]$. When we estimate the constructed predicate q' on the converted numeric column C' with an estimator, the probability that $C \subseteq t_q$ and $C' = l$ is $p(C \subseteq t_q \wedge C' = l) = p(C' = l) \times P$, where $p(C' = l)$ is the probability of $C' = l$ estimated by the estimator. E,g., suppose the predicate on the settrie in Figure 2b is $C \subseteq (a, b, d)$. The node label 1 is in the results based on Algorithm 2. Node a has a histogram with histogram label 0. Hence, we calculate the probability $P = 1 - p(f) = \frac{1}{2}$ and the estimated probability on label 0 is $\frac{1}{2}p(C = 0)$. Node b has a histogram with label 2. Hence, we calculate the probability $P = 1 - p(c) = 0$ and the estimated probability on label 2 is 0. Since c is not in the predicate value $t_q = (a, b, d)$, The label of node c and its histogram are not in the results.

Superset search. When we use Algorithm 3 on a predicate q with value t_q and meet a node with histogram, suppose the current unmatched elements in t_q is t_q^r. If all elements $e \in t_q^r$ are in the histogram, the proabbility $P = \prod_{e \in t_q^r} p(e)$; If there exists an element $e \in t_q^r$ not in the histogram, we return $P = 0$ since no nodes in the original subtree is the superset of t_q. When estimate the converted predicate q' on the converted numeric column C' with an estimator, the probability that $C \supseteq t_q$ and $C' = l$ is $p(C \supseteq t_q \wedge C' = l) = p(C' = l) \times P$, where $p(C' = l)$ is the probability of $C' = l$ estimated by the estimator. E.g., suppose the predicate on the settrie in Figure 2b is $C \supseteq (a, c, d, f)$. Node a and node b have histograms but element c and element f are not in the histograms respectively. Hence, the probability on these histogram are 0. Node c also has histogram and elements d and f are in the histogram. Hence, we calculate the probability $P = p(d)p(f) = \frac{14}{100}$ and the estimated probability on label 4 is $\frac{14}{100}p(C = 4)$.

Cost Analysis. In ST-hist, the number of tree nodes are reduced to the user-defined number S, so the subset and superset search need to explore at most S nodes. In the worst case, each node in the settrie has histogram and each histogram contains at most $|\Sigma|$ elements. For subset search, each element in the histogram should be checked whether in t_q, so the time complexity of subset search is $O(S|\Sigma|)$ in the worst case. For superset search, each element in q should be checked whether in the histogram, so the time complexity of superset search is $O(S|q|)$ in the worst case.

ST-hist on multiple subcolumns. If we keep a fixed number S nodes for each subcolumn, more subcolumns could keep more information. The procedure of ST-hist construction on multiple subcolumns is similar to ST-hist construction on single subcolumn, so we will not discuss again.

Remarks: OT-sampling [43] also adopts the trie structure for selectivity estimation on set containment search predicates. There are two main differences: (1) OT-sampling samples the data by only keeping the distribution of a few high-frequency elements. Instead, ST keeps the distribution of all elements and ST-hist keeps the distribution of most elements. Thus, OT-sampling performs poorly over less-frequent elements since their distributions are unknown. (2) OT-sampling can

only handle queries over a single set-valued column, and it keeps the set-valued data and the set containment search predicate. Thus, it cannot be implemented into existing estimators to capture attribute correlations like our methods, which convert the set-valued data to numeric data and convert the set containment search predicates to comparison predicates, so we could utilize any data-driven estimators to capture the attribute correlations.

5 MISCELLANEOUS ISSUES

We first discuss how to handle low selectivity elements in Section 5.1. Then we present the incremental settrie construction algorithm on data deletion, insertion and update for both ST and ST-hist in Section 5.2.

5.1 Handling Low Cardinality Elements

The elements in most datasets follow power law distribution, i.e, large cardinality for few elements and small cardinality for most elements. To reduce the tree size of ST, we remove all low cardinality elements in the dataset and add a column lc to indicate whether a record contains low cardinality elements, i.e., 1 indicates the record has low cardinality elements and 0 indicates the record has no low cardinality elements. We consider the following four cases for query conversion:

- **Subset query q without any low cardinality element.** Add a predicate $lc = 0$ to the converted predicate q' using AND operator for conjunction to indicate there is no low cardinality element in the query.
- **Subset query q with low cardinality elements.** Remove low cardinality elements from q and generate q' first. Then we also add the predicate $lc = 0$ to the q'. For each low cardinality element in q, we add a fixed small number ϵ to the estimated selectivity of q'.
- **Superset query q without any low cardinality element.** We use the superset query conversion algorithm in Section 4 since the selectivity of q is not influenced by whether a record contains low cardinality elements.
- **Superset query q with low cardinality elements.** Return the fixed small number ϵ directly.

In the experiment, we consider the element e with $|e| \leq 10$ as low cardinality elements and set $\epsilon = \frac{1}{|T|}$.

5.2 Incremental Settrie Construction

Given a settrie $root_0$ constructed on the data T_0, we hope to construct a new settrie $root_1$ on the data after deletion, insertion, and update based on $root_0$ without reconstruction. Since one update is equivalent to one deletion and one insertion, we focus on deletion and insertion in this section. We propose incremental settrie construction algorithms for both ST and ST-hist. We denote the data to be deleted by T_1 ($T_1 \subseteq T_0$) and the data to be inserted by T_2.

Incremental ST construction on deletion. When we first construct the settrie $root_0$ on T_0, we store the frequency of each node on that node like the ST-hist construction discussed in Section 4.4 in the paper. When a record in T_0 is deleted, we search the record in $root_0$ with the same searching method in the settrie construction discussed in Section 4.3 in the paper and reduce the frequency by 1 for each matched node in the search path. After all records in T_1 are deleted, we remove all nodes with 0 frequency in $root_0$.

Incremental ST construction on insertion. If there are no new elements in the inserted data T_2, we could insert all records in T_2 into $root_0$ and relabel all records in the settrie according to Algorithm 1 in Section 4.3. If T_2 contains new elements (denote the new element universe by Σ'), we first sort the elements in Σ' in descending order of frequency and index them following the

indices of Σ_0, i.e., for $e \in \Sigma'$, $idx(e)$ is in range $[|\Sigma_0|, |\Sigma_0| + |\Sigma'| - 1]$. Then, we insert all records in T_2 into the settrie.

Incremental ST-hist construction on deletion. Deletion on ST-hist is similar to that on ST. The difference is that the histograms should be considered. When searching an element e in a deleted record t on a node in the settrie, we first check whether the node has a child with $element = e$. If so, we reduce the frequency of that child by 1 and continue to search for the next element at the child node. If not, we search all remaining elements in t in the histogram of the node and reduce the frequency by 1 for each matching histogram. We remove all nodes and histograms with 0 frequency.

Incremental ST-hist construction on insertion. The difference between incremental ST-hist construction and incremental ST construction on insertion is that we should keep the same number of nodes on ST-hist. Hence, after indexing all new elements and inserting all new records, we need to construct histograms with Algorithm 4 in the paper. Note that some nodes in $root_0$ already contain histograms, so the new histograms should be merged into them if they are attached to the same node.

When all deletion and insertion are finished, we relabel all nodes in the settrie.

Cost Analysis. The time cost of settrie construction is $O(|T|L)$, where L is the average length of each record in the set-valued column. Hence, the time cost of reconstruction is $O((|T_0|-|T_1|+|T_2|)L)$, which is linear to the size of total data. In contrast, the time cost of incremental construction is $O((|T_1| + |T_2|)L)$, which is linear to the size of incremental data. In general, we consider $|T_1| \ll |T_0|$ and $|T_2| \ll |T_0|$, so the time cost of incremental construction is much smaller than that of reconstruction.

6 EXPERIMENTS

The experiments focus on the following aspects:

- **Estimation Accuracy.** We evaluate the estimation error of the proposed techniques using three estimators.
- **Estimation Efficiency.** We evaluate the query conversion and estimation time of the proposed techniques on three estimators.
- **Total Runtime Studies.** We evaluate the total runtime on query execution with proposed techniques.
- **Hyper-parameter Studies.** We study the impact of different hyper-parameters to the performance.
- **Performance of Incremental Settrie Construction.** We compare the incremental settrie construction time and settrie reconstruction time of ST and ST-hist.
- **Impact of Element Distribution.** We evaluate the performance on datasets with different element distribution.
- **Scalability Study.** We evaluate the performance by varying the number of tuples.
- **Low Cardinality Threshold Selection.** We discuss how to select the low cardinality threshold.

6.1 Experimental Setup

6.1.1 Datasets. We use three real-world datasets with different size ($|T|$), number of columns (#Col), size of element universe ($|\Sigma|$) and average length of sets (AvgL) described in Table 6.

`GN` [10]. This dataset contains State, territories and associated areas of the United States. It contains 1 set-valued column, 3 categorical columns and 3 numeric columns.

`Twitter`. This dataset contains tweets posted from 2016-01-04 to 2016-02-04. It contains 1 set-valued column, 1 categorical column and 3 numeric columns.

Table 6. Datasets for experiments.

Dataset	$\|T\|$	#Cols	$\|\Sigma\|$	AvgL	#Subcols	#N	#N-hist
GN	2.2M	7	25K	3	3	47K	5K
Twitter	7.6M	5	118K	9	5	432K	20K
Wiki	5.3M	1	191K	13	10	938K	40K

Wiki. This dataset contains the first sentence of each English Wikipedia article. The dataset was extracted in September 2017 and contains only one set-valued column.

We remove all elements with cardinality less than 10 and add a column *lc* to indicate which record contains these elements as discussed in Section 5.1. The element frequency of all datasets follow power law distribution.

6.1.2 Estimators. We incorporate our two methods, ST, ST-hist, into three existing estimators, including an open source DBMS, Postgres and two learning based estimators, Neurocard and DeepDB.

- ST. Settrie based method (Section 4.3).
- ST-hist. Settrie with histograms based method (Section 4.4).

Postgres (version 15.1). Postgres constructs 1D-histograms for range predicates, MCV for equal predicates, independence model for subset predicates and graphical statistical model for superset predicates with AVI assumption for selectivity estimation.
Neurocard [44]. Neurocard adopts deep autoregressive models to learn the joint distribution. We allow column factorization and tune parameters in Neurocard to get good estimation accuracy and efficiency. Neurocard does not support set-valued data and set operators.
DeepDB [18]. DeepDB learns the joint distribution with SPNs. DeepDB does not support set-valued data and set operators.

6.1.3 Baselines. We compare with three baselines: Postgres, Sampling and OT-Sampling [43], which are the best baselines to the best of our knowledge. Sampling uses a fix number of tuples uniformly sampled from the whole dataset. OT-Sampling is a dedicated method for set containment estimation and its sampling keeps the distribution of high frequency elements. We consider the 12 most frequent elements for OT-Sampling and keep the sample size as 10^4 for both Sampling and OT-Sampling by following previous work [15, 43]. The two methods discussed in Section 4.1 suffer from either very inaccurate or very inefficient estimation, so we will not report their performance.

Table 6 shows the hyper-parameters we used in the experiments and the statistics on our methods. #Subcols is the number of subcolumns we set on ST and ST-hist. #N is the average number of nodes on each trie in ST. #N-hist is the number of nodes kept on each trie in ST-hist. Our methods are implemented in C++. We run our methods and all estimators on a Tesla V100 Xeon E5-2698.

6.1.4 Query Workloads. We consider three query types: regular query considering all elements, low frequency query considering only 80% elements with the lowest frequency and high frequency query considering only 5% elements with the highest frequency. We generate 1K queries for each query type on each dataset, where 500 of them contain subset predicates and 500 of them contain superset predicates. For each query, we first sample some categorical and numeric columns from the dataset; Then for each sampled categorical column, we uniformly sample a value from its domain and use equal or range operator. For each numeric column, we uniformly draw a range from its domain and use range operator. For each subset predicate, we uniformly draw 5-10 records from the set-valued column and take the union of the sampled records as the predicate value. For each superset predicate, we uniformly draw a record from the set-valued column, and then uniformly draw 2–4 elements from the record as the predicate value. The true selectivity of each query is obtained by executing it in Postgres.

Table 7. Estimation error on GN.
(a) Subset queries.

Estimator	Regular queries								High frequency queries								Low frequency queries							
	Multiple columns				Set-only column				Multiple columns				Set-only column				Multiple columns				Set-only column			
	Mean	Median	95th	Max	Mean	Median	95th	Max	Mean	Median	95th	Max	Mean	Median	95th	Max	Mean	Median	95th	Max	Mean	Median	95th	Max
Sampling	38.7	2.83	175	891	**1.16**	1.79	**1.54**	**2.72**	37.3	11.0	198	599	**1.06**	**1.05**	1.17	1.35	31.5	23.0	82.0	148	71.0	86.0	166	221
OT-Sampling	34.6	2.29	156	814	1.21	**1.13**	1.59	6.23	39.3	11.0	206	585	**1.06**	**1.05**	**1.15**	**1.28**	29.4	22.0	79.0	148	74.3	89.5	172	221
Postgres	10.4	7.12	26.9	189	8.53	7.54	15.9	33.2	7.48	3.34	27.4	**110**	4.12	3.88	6.02	10.4	3.03	1.40	11.6	**38.0**	1.25	1.21	1.56	2.75
Postgres_ST	7.60	3.57	23.1	201	2.91	2.56	5.72	9.16	9.02	3.74	34.5	118	2.27	2.16	3.21	5.30	3.79	**1.39**	15.3	44.0	**1.15**	**1.11**	**1.41**	**2.27**
Postgres_ST-hist	14.5	9.54	40.6	270	12.2	10.3	24.6	50.7	10.57	4.92	37.0	156	6.27	5.94	9.02	14.3	3.24	1.69	10.0	44.0	2.81	1.70	2.50	136
Neurocard_ST	**2.49**	**1.52**	**5.35**	**121**	1.70	1.54	2.84	7.41	**2.46**	**1.38**	**4.78**	183	1.71	1.68	2.20	2.88	**2.53**	1.88	6.00	**38.0**	2.70	2.22	5.90	11.4
Neurocard_ST-hist	4.38	2.39	11.9	235	2.19	1.91	4.09	7.11	3.44	1.68	7.20	316	1.27	1.22	1.71	2.43	2.56	1.63	**5.93**	**38.0**	2.56	1.80	7.23	13.3
DeepDB_ST	25.9	19.4	72.3	163	8.64	6.93	20.7	45.0	9.98	3.34	32.2	254	1.81	1.60	3.24	6.78	345	320	590	1082	404	384	597	1210
DeepDB_ST-hist	22.0	16.6	63.0	156	2.57	1.67	8.08	26.0	26.2	3.73	37.5	6394	2.36	2.11	4.16	15.5	3.65	2.72	7.86	40.5	2.06	2.01	3.00	3.58

(b) Superset queries.

Estimator	Regular queries								High frequency queries								Low frequency queries							
	Multiple columns				Set-only column				Multiple columns				Set-only column				Multiple columns				Set-only column			
	Mean	Median	95th	Max	Mean	Median	95th	Max	Mean	Median	95th	Max	Mean	Median	95th	Max	Mean	Median	95th	Max	Mean	Median	95th	Max
Sampling	34.3	12.0	157	627	27.8	1.26	171	1037	37.1	12.0	169	985	15.8	**1.12**	106	**312**	18.2	15.0	37.0	56.0	37.5	36.0	83.0	86.0
OT-Sampling	35.4	12.0	188	439	19.9	**1.18**	110	518	36.9	11.0	144	1497	13.6	1.14	76.0	431	18.3	16.0	38.0	56.0	36.8	34.5	80.0	86.0
Postgres	150	15.0	428	13040	85.1	3.43	198	5773	222	26.0	706	23791	169	2.57	506	13042	11.4	11.0	29.0	72.0	9.51	7.88	20.2	28.5
Postgres_ST	60.3	6.69	170	9703	11.9	2.07	31.1	692	108	14.7	352	8989	39.5	1.89	72.7	2305	11.2	11.0	28.0	72.0	5.66	5.13	11.8	22.7
Postgres_ST-hist	10.2	6.25	30.6	210	80.5	5.79	187	7917	143	15.0	578	8989	67.3	3.88	106	11612	12.2	12.0	30.0	72.0	6.94	1.78	37.0	50.0
Neurocard_ST	3.25	**1.50**	**11.0**	**130**	**3.70**	1.38	15.1	**236**	**4.08**	**1.58**	**15.3**	83.5	**7.29**	1.32	22.2	720	**5.20**	**2.68**	**20.0**	**54.0**	**1.85**	1.46	**4.62**	**12.0**
Neurocard_ST-hist	**3.19**	1.54	**11.0**	**130**	4.30	1.35	**11.3**	635	**4.08**	1.63	17.2	**74.0**	8.70	1.32	**22.1**	720	6.92	3.18	**20.0**	56.0	1.95	**1.36**	**4.62**	**12.0**
DeepDB_ST	11.5	2.25	54.0	654	375	3.49	990	51636	151	22.9	630	7480	70.1	3.05	353	2308	47.0	11.0	54.1	1959	146	6.23	1182	3041
DeepDB_ST-hist	44.9	4.19	121	7240	37.9	3.59	107	2306	120	18.0	635	4065	47.4	2.53	86.8	2308	70.7	16.3	401	979	203	64.2	649	1784

Table 8. Estimation error on Twitter.
(a) Subset queries.

Estimator	Regular queries								High frequency queries								Low frequency queries							
	Multiple columns				Set-only column				Multiple columns				Set-only column				Multiple columns				Set-only column			
	Mean	Median	95th	Max	Mean	Median	95th	Max	Mean	Median	95th	Max	Mean	Median	95th	Max	Mean	Median	95th	Max	Mean	Median	95th	Max
Sampling	75.4	16.0	360	1512	5.60	2.00	7.86	1362	78.7	11.0	415.3	1799	1.57	1.24	3.34	9.51	53.6	35.5	146	286	38.8	25.0	107	315
OT-Sampling	127	28.0	589	1833	56.5	**1.28**	464	2437	127	15.0	675	2512	13.1	**1.16**	**2.12**	1677	52.7	34.5	144	286	4.85	**2.52**	26.0	89
Postgres	10.5	5.58	36.1	110	9.49	5.88	28.7	77.3	6.40	2.61	23.0	166	3.08	2.19	8.68	19.0	9.06	7.26	22.35	68.0	**3.29**	3.18	**4.60**	**6.58**
Postgres_ST	6.27	3.60	20.2	58.3	5.33	3.51	15.6	40.6	6.31	2.59	24.5	166	2.79	2.22	6.41	16.4	10.9	6.59	39.0	140	10.4	6.88	42.2	141
Postgres_ST-hist	6.34	3.74	21.2	54.6	5.46	3.62	15.4	36.3	10.1	6.73	41.3	138	5.29	2.58	13.6	133	5.46	3.62	15.4	36.3	11.2	6.78	40.0	143
Neurocard_ST	**3.28**	**1.83**	**8.59**	137	1.63	1.44	2.77	**3.86**	3.55	**1.63**	**9.77**	318	1.54	1.43	2.44	4.19	**2.74**	**1.84**	**6.67**	**23.4**	4.43	5.49	6.31	6.78
Neurocard_ST-hist	4.84	2.56	12.8	180	**1.58**	1.41	**2.57**	5.94	4.20	1.84	11.4	91.1	**1.36**	1.23	2.17	**3.82**	4.94	2.46	16.2	62.0	4.05	4.31	4.89	7.63
DeepDB_ST	4.05	2.70	11.3	**39.2**	3.90	2.78	10.9	37.3	4.05	2.81	10.7	**31.9**	4.08	3.11	10.0	16.2	31.6	22.7	114	366	27.8	18.3	112	377
DeepDB_ST-hist	8.10	4.75	25.1	80.3	7.90	4.58	25.5	73.7	**3.46**	2.37	11.4	27.9	4.50	2.80	12.7	67.2	65.3	39.4	197	633	48.1	31.8	195	654

(b) Superset queries.

Estimator	Regular queries								High frequency queries								Low frequency queries							
	Multiple columns				Set-only column				Multiple columns				Set-only column				Multiple columns				Set-only column			
	Mean	Median	95th	Max	Mean	Median	95th	Max	Mean	Median	95th	Max	Mean	Median	95th	Max	Mean	Median	95th	Max	Mean	Median	95th	Max
Sampling	100	24.0	483	2987	78.0	1.28	578	1985	92.9	16.0	457	2387	68.5	1.21	384	2050	38.3	25.0	107	315	171	180	306	356
OT-Sampling	78.2	20.0	325	1594	60.7	**1.26**	405	1522	84.4	14.0	356	4672	81.6	**1.18**	602	2824	36.7	24.0	103	230	87.4	47.0	284	354
Postgres	447	24.0	1265	51536	2043	30.5	13255	68877	452	24.2	1216	22625	1560	25.3	8939	69366	37.0	22.4	113	361	106	37.4	281	333
Postgres_ST	174	9.13	476	20502	516	5.87	2568	49557	184	7.80	397	17445	692	5.18	2693	49557	16.4	3.79	90.1	230	81.0	4.12	281	281
Postgres_ST-hist	134	11.1	500	2433	348	9.79	1158	49557	113	8.67	353	15585	387	7.27	925	49557	16.0	2.85	90.1	137	80.4	3.02	280	281
Neurocard_ST	5.35	1.69	18.0	217	**5.06**	1.43	**16.2**	**169**	4.47	1.72	16.1	70.0	**7.01**	1.46	15.1	**912**	9.90	1.97	42.1	154	31.4	1.95	281	281
Neurocard_ST-hist	**4.65**	**1.38**	**17.0**	**211**	6.82	1.33	32.0	372	**3.17**	**1.36**	**11.0**	**106**	10.6	1.32	**11.4**	1579	**5.19**	**1.33**	**25.0**	**124**	**5.55**	**1.35**	**40.1**	**105**
DeepDB_ST	19.3	2.32	89.0	2083	94.7	1.99	317	11569	13.2	1.88	46.5	1061	51.7	1.85	164	2754	18.2	6.40	90.1	230	82.7	6.53	280	320
DeepDB_ST-hist	25.4	3.88	118	1416	31.7	4.04	127	1180	18.9	2.95	75.4	890	25.0	2.89	84.6	2522	457	436	1192	2527	548	480	1202	2957

Table 9. Estimation error on Wiki.

Estimator	Subset queries												Superset queries											
	Regular queries				High frequency queries				Low frequency queries				Regular queries				High frequency queries				Low frequency queries			
	Mean	Median	95th	Max	Mean	Median	95th	Max	Mean	Median	95th	Max	Mean	Median	95th	Max	Mean	Median	95th	Max	Mean	Median	95th	Max
Sampling	172	129	460	843	221	2.16	760	1072	26.9	23.0	78.0	76.0	74.5	18.5	336	1368	75.7	7.03	367	2491	35.4	27.0	80.0	89.0
OT-Sampling	85.2	5.75	373	843	65.4	1.65	438	1159	26.5	24.0	49.0	76.0	59.8	14.0	270	1710	50.1	4.29	208	1544	35.3	29.0	79.0	89.0
Postgres	5.85	4.39	13.8	23.0	6.01	5.50	13.4	25.3	**1.66**	**1.46**	**2.83**	**4.81**	188	8.82	544	19115	243	7.88	1551	9306	183	148	413	450
Postgres_ST	**2.16**	**1.78**	5.13	**16.8**	3.65	2.50	9.80	16.5	4.26	3.95	7.91	8.63	94.9	4.52	353	5269	119	3.26	640	8626	5.06	4.75	7.80	27.0
Postgres_ST-hist	**2.16**	**1.78**	**4.99**	17.7	3.84	2.62	10.1	17.7	4.36	4.06	8.08	8.90	5.58	2.06	19.5	40.0	144	5.26	775	8626	5.58	2.06	19.5	40.0
Neurocard_ST	7.25	6.53	15.2	36.5	4.15	3.81	8.35	13.5	27.9	25.9	51.9	56.7	16.0	3.17	61.3	698	14.4	2.55	52.0	**527**	4.29	2.39	9.36	228
Neurocard_ST-hist	6.10	5.22	13.0	31.7	3.45	3.23	6.35	12.7	24.2	22.4	44.9	49.0	**13.1**	**2.20**	**56.1**	**477**	**11.3**	**1.90**	**45.1**	**527**	**1.86**	**1.23**	**3.23**	**27.0**
DeepDB_ST	5.61	2.35	9.25	280	2.19	**1.64**	4.90	**7.56**	25.1	22.0	49.0	76.1	92.3	4.25	317	9500	117	3.08	553	8626	5.92	4.31	16.0	40.2
DeepDB_ST-hist	6.93	3.85	15.1	280	**2.15**	1.84	**4.35**	9.31	27.8	25.0	51.0	70.0	72.2	8.06	199	5336	64.6	8.06	223	1952	327	284	811	1488

6.1.5 Evaluation Metrics. We use the q-error [29] to evaluate the estimation accuracy: $error(q) = \max(\frac{actsel(q)}{estsel(q)}, \frac{estsel(q)}{actsel(q)})$. We use query conversion time and query estimation time to evaluate the estimation efficiency.

6.2 Experimental Results

6.2.1 Accuracy Evaluation. Tables 7–9 give the estimation error on GN, Twitter and Wiki, respectively. We report results on queries with predicates only on set-valued column (labeled as set-only column in the tables) and queries with predicates on both set-valued column and other columns

(labeled as multiple columns). We observe that ST and ST-hist based estimators usually have the best accuracy. In terms of the median error, Postgres_ST and Postgres_ST-hist outperform the baselines by up to 2.46 times on subset queries and 13.1 times on superset queries. Neurocard_ST and Neurocard_ST-hist outperform the baselines by up to 19.2 times on subset queries and 18.7 times on superset queries. DeepDB_ST and DeepDB_ST-hist outperforms the baselines by up to 5.92 times on subset queries and 10.3 times on superset queries. Next we give the main observations.

Observations on Regular queries and high frequency queries. Our methods and baselines have similar performance on regular queries and high frequency queries, so we discuss them together.

(1) Sampling and OT-Sampling usually have the best median error performance on `GN` and `Twitter` for queries on set-only column, but poor performance for queries on multiple columns. This is because search space of multiple columns are much larger than that of single column. Sampling and OT-Sampling have large error on `Wiki` since the element universe and average record length on `Wiki` is larger, so it is more difficult for samples to represent the data distribution well. OT-Sampling has better performance than Sampling on most high frequency queries since OT-Sampling keeps the distribution of high frequency elements.

(2) ST based estimators outperform Postgres by up to 19.9 times for queries on set-only column since Postgres is based on element independence assumption while ST based estimators can capture element correlations inside a subcolumn. For example, Postgres_ST generates histograms to represent the distribution of a subcolumn, so ST are able to greatly improve Postgres in selectivity estimation for queries with predicates on set-valued attributes.

(3) Learning based estimators with ST or ST-hist outperform the three baselines in most cases for queries on multiple columns since ST and ST-hist utilize estimators to learn the correlations of elements in different subcolumns when learning the joint data distribution. However, all baselines do not model joint data distribution. Hence, ST and ST-hist preform much better for queries with predicates on multiple columns.

(4) ST-hist has worse performance than ST on Postgres and DeepDB. This is because some information are lost in ST-hist construction. However, ST-hist has comparable performance with ST on Neurocard, which is because ST-hist reduces subcolumn domain size to simplify the distribution, so Neurocard_ST-hist could learn the distribution easier than Neurocard_ST. However, this simplification does not improve the performance of Postgres and DeepDB since they have built-in histograms or CDFs to simplify the distribution.

Observations on low frequency queries.

(1) Sampling and OT-Sampling perform poor on low frequency queries. Sampling is random and OT-Sampling generates samples based on the distribution of high frequency elements. Hence, it is highly likely that low frequency elements are not sampled.

(2) Postgres and Neurocard with ST and ST-hist outperform Postgres by up to 71.8 times on superset queries since they can well capture element correlations when elements are in the same subcolumn. However, they are on a par with Postgres on subset queries. This is because elements in subset queries usually have low correlations, and Postgres could well capture the correlations with element independence assumptions.

(3) Postgres and Neurocard with ST and ST-hist outperform Sampling and OT-Sampling by up to 53.3 times since ST and ST-hist consider low frequency elements while Sampling and OT-Sampling focus on high frequency elements.

(4) The performance of DeepDB_ST and DeepDB_ST-hist are very unstable since CDF used by DeepDB has poor performance on point and IN predicates with low frequency values.

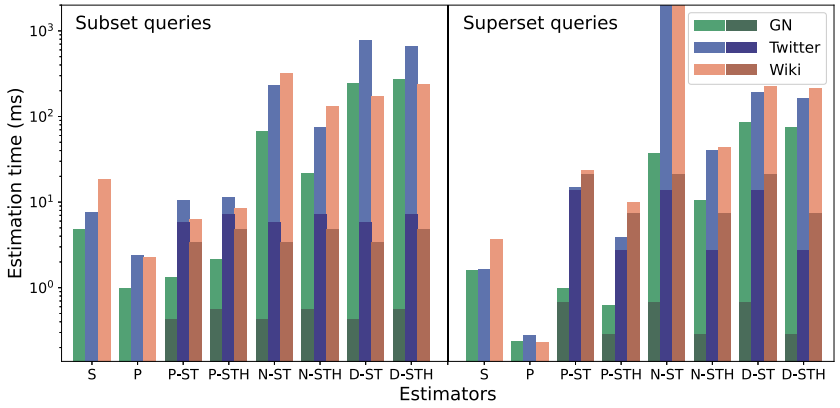


Fig. 3. Estimation time. S, P, N and D refer to Sampling, Postgres, Neurocard and DeepDB respectively. Dark color is conversion time and light color is estimation time.

6.2.2 Estimation Efficiency. The estimation time of our methods comprises query conversion time and query estimation time. Figure 3 shows both of them. There is no query conversion time for Sampling and Postgres. Sampling and OT-Sampling have the same estimation time, so we use Sampling for them. Overall, we make the following observations:

(1) ST can convert most subset queries within 5*ms* although the cost of query conversion is exponential to the size of element universe. That is because most queried elements have high frequency and we use high-frequency-first-order in settrie construction. We only need to search a small part of the tree. However, The conversion time of superset queries can be more than 15*ms* on datasets with a large element universe and large domain size (`Twitter` and `Wiki`). This is because the cost of superset search is much larger than the cost of subset search as discussed in Section 4.3. The estimation time of Neurocard_ST is not competitive on dataset with a large domain size (`Twitter` and `Wiki`) since the model size on these datasets is very large. DeepDB estimates each value in the IN predicates sequentially, so estimation of DeepDB_ST on subset queries is less efficient than that on superset queries.

(2) ST-hist takes similar conversion time as ST on subset queries, but takes less time than ST on superset query conversion. This is because only few nodes in a settrie are traversed for subset query search. The estimation time of Postgres_ST-hist is similar with Postgres_ST since Postgres constructs a fixed number of buckets. Neurocard_ST-hist outperforms Neurocard_ST by 2.5–170 times in estimation efficiency since the model size of Neurocard_ST-hist is much smaller than that of Neurocard_ST. DeepDB_ST-hist has similar estimation time with DeepDB_ST since the estimation time of DeepDB is dominated by the number of values in the IN predicate but not the domain size.

6.2.3 Total Runtime Studies. We evaluate the impact of our study on query optimization on `IMDB` dataset [24]. Previous work [24, 25] report that `IMDB` has strong correlations and it has been widely used in selectivity estimation evaluation [18, 44]. Since `IMDB` does not contain set-valued columns, we concatenate the column feature_name from `GN` to the table *title* and the column tags from `Twitter` to the table *movie_info*. The queries are generated based on the benchmark JOB-light [18, 22, 44]. Each query contains 4–8 operators on 3–6 tables with set containment search predicates. We generate 200 queries with subset containment predicates and 200 queries with superset containment predicates. We follow previous work [3] to modify Postgres so that it could import external estimation results. Figure 4 demonstrates end-to-end running time, which is the sum

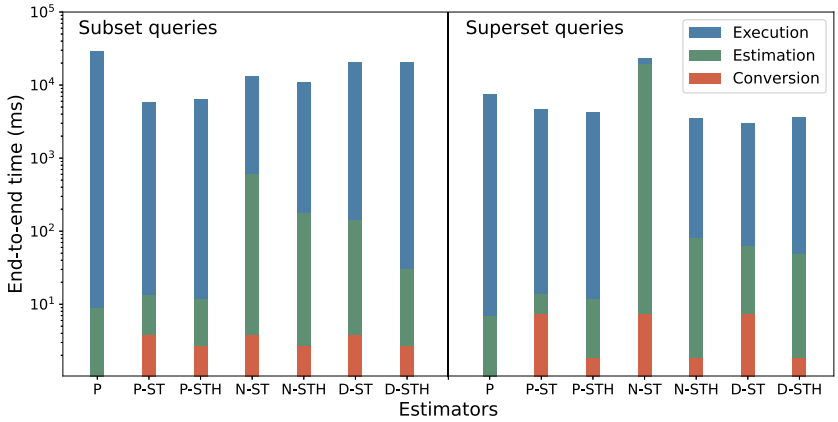


Fig. 4. End-to-end query execution time. P, N and D refers to Postgres, Neurocard and DeepDB respectively.

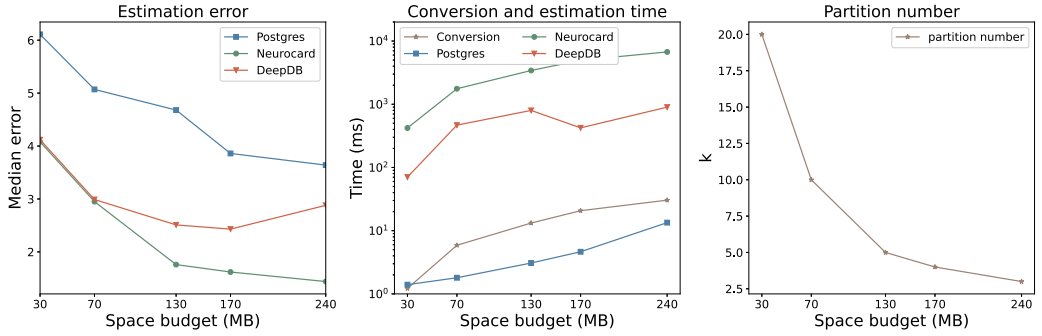
of conversion, estimation and execution time. Overall, we get the following observations. (1) Our methods outperform Postgres on the execution time. This shows that more accurate estimation can translate into better query plan. Specifically, our methods only take 20%–70% of the time on subset query execution and 30%–60% of the time on superset query execution in average. (2) Most ST and ST-hist based estimators demonstrate superior performance over Postgres concerning end-to-end time, with the exception of Neurocard_ST on superset queries due to its inefficient estimation. However, Neurocard_ST-hist has short end-to-end time since ST-hist significantly reduces the estimation time while keeping high accuracy.

6.2.4 Hyper-parameter studies. We study the effect of the storage space budget on the performance of ST and ST-hist. We could adjust the number of partitions k in ST and ST-hist or the number of nodes in S in ST-hist to satisfy the space budget. Subset queries and superset queries have similar performance when varying hyper-parameters, and we use Figure 5 to show the performance of both on `Twitter`. The results on other datasets are qualitatively similar. Overall, we have the following observations.

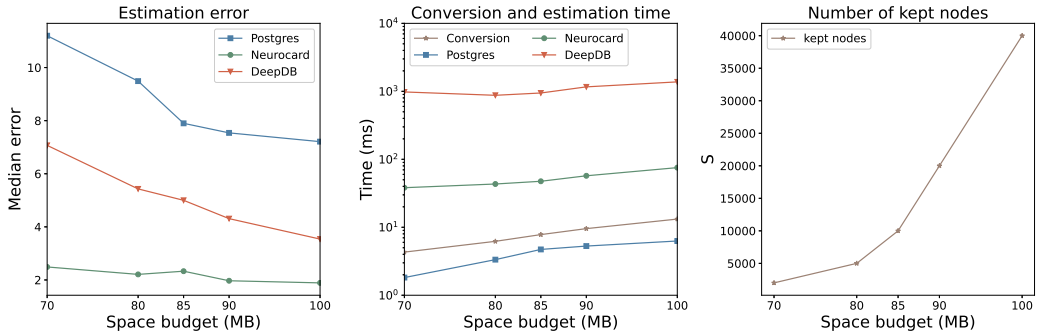
Varying the number of partitions k on ST and ST-hist. We observe similar results on ST and ST-hist. Due to space limitation, we only report the results on ST in Figure 5a. It can be seen that space consumption increases when reducing the number of partitions, and space consumption is roughly inversely proportional to the number of partitions k. By increasing the space budget, (1) the accuracy is improved since a smaller number of partitions simplifies the distribution, and (2) the conversion and estimation time increase since the size of each settrie increases dramatically.

Varying the number of kept nodes S on ST-hist. The space consumption is slightly increased by using more kept nodes. By increasing the space budget, (1) the accuracy is improved since fewer nodes are replaced by the histograms, and (2) the conversion and estimation time increases.

Parameter selection discussion. Based on the above observation, we find that there is a trade-off between space budget, conversion and estimation time, and accuracy. Hence, we have the following rules for selecting the parameters k and S. (1) In ST, we can first randomly select a k (e.g., $k = 10$) and then generate a settrie. First, if the size of the settrie does not exceed the space budget, we can further consider tuning the value of k based on the requirement of accuracy and efficiency. If the application focuses more on efficiency, we can increase k such that the space budget is still satisfied and the conversion and estimation are faster. If the application requires better accuracy, we can decrease k, but this may exceed the space budget. Second, if we run out of space during construction, we have to increase the value k to reduce space consumption. Assume that the



(a) Varying k on ST



(b) Varying S on ST-hist

Fig. 5. Varying hyper-parameters based on space budget.

construction progress is $x\%$ when we run out of space with value k. We roughly estimate that the size of the settrie for the dataset is $\frac{B}{x\%}$. Next, we can increase the value k to $k' = k\frac{B}{x\%B} = \frac{k}{x\%}$. We repeatedly perform this operation until the space budget can be satisfied. (2) For ST-hist, we can first set k similarly to the procedure in ST. Next, by utilizing the histogram, the space cost compared to ST is always smaller. From Figure 5b, we can observe that a larger S can reduce the errors while affecting the efficiency slightly. Thus, if the application cares more about accuracy, we can always use a large S value. Otherwise, we set S to a smaller value to obtain better efficiency sacrificing the accuracy.

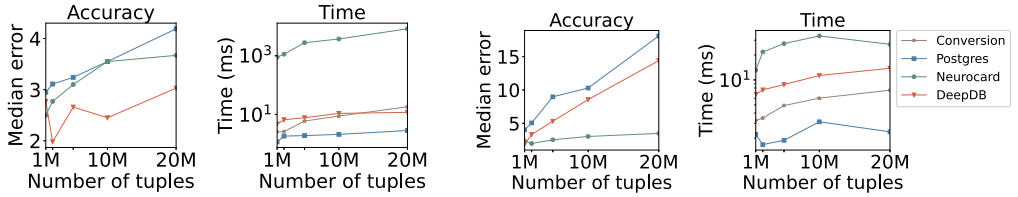
6.2.5 Performance of Incremental Settrie Construction. We compare the performance of incremental construction and reconstruction on `Twitter`. We use 80% tweets in `Twitter` as the base dataset and the remaining as two sets of insertion data (each contains 10% `Twitter`). Next, we first delete 5% of `Twitter` from the base dataset and insert the first set, and then we delete 5% again and insert the second set. We first construct a settrie on the base data and then incrementally construct the settrie as described above for the update.

Table 10 gives the performance of both ST and ST-hist, where RC refers to reconstruction and IC refers to incremental construction. We observe that (1) the incremental construction is much faster than the reconstruction, (2) the reconstruction time is proportional to the total data size while the incremental construction time is proportional to the size of update data and (3) the query performance on incremental construction and reconstruction are almost same.

6.2.6 Impact of element distribution. Since most real-world datasets follow power-law distribution, we use `Twitter` as template dataset to generate datasets with different element distribution.

Table 10. Performance of reconstruction

		estimator	construction (ms) RC	construction (ms) IC	conversion (ms) RC	conversion (ms) IC	estimation (ms) RC	estimation (ms) IC	median error RC	median error IC
ST	1st	Postgres	45832	14475	2.42	2.45	1.90	2.04	6.96	7.03
		Neurocard					824	832	2.78	2.68
		DeepDB					312	393	2.54	3.09
	2nd	Postgres	48624	10043	2.52	2.63	1.92	1.81	6.98	7.20
		Neurocard					742	827	2.82	2.69
		DeepDB					283	287	2.65	2.62
ST-hist	1st	Postgres	56909	12485	2.36	2.19	2.31	2.23	7.70	7.68
		Neurocard					41.7	31.2	2.60	2.03
		DeepDB					262	402	3.32	2.97
	2nd	Postgres	60579	11503	2.34	2.26	2.22	2.18	7.66	7.70
		Neurocard					35.5	30.5	2.38	2.13
		DeepDB					271	378	3.08	3.16



(a) Performance on ST (b) Performance on ST-hist

Fig. 6. Varying the number of tuples.

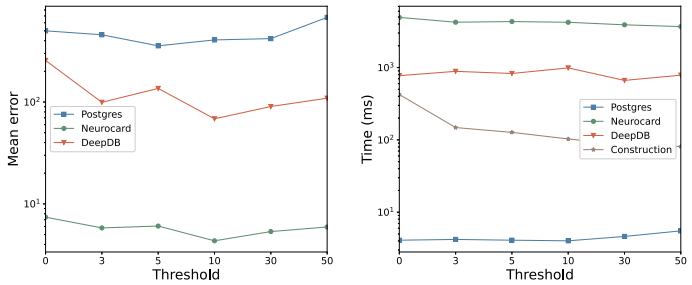
`Uniform` follows uniform distribution and the probability of each element in `Linear` increases linearly. Table 11 shows the evaluation performance. Our methods outperform baselines significantly since Postgres and OT-Sampling only consider a few high frequency elements, but elements in `Uniform` and `Linear` have more similar frequencies than a dataset with power-law distribution.

6.2.7 Scalability study. We evaluate the scalability of our methods on `Twitter` by keeping $k = 10$ and $S = 20$K while varying the number of tuples. Figure 6 demonstrates the performance. We observe that (1) increasing the number of tuples will reduce the estimation accuracy and slightly increase the conversion and estimation time since the number of distinct values increases, and (2) the estimation error of ST-hist is proportional to the number of tuples since we keep the number of nodes unchanged.

Table 11. Performance on different distribution

Estimator	`Uniform` Subset queries Mean	`Uniform` Subset queries Median	`Uniform` Superset queries Mean	`Uniform` Superset queries Median	`Linear` Subset queries Mean	`Linear` Subset queries Median	`Linear` Superset queries Mean	`Linear` Superset queries Median
OT-Sampling	27.4	27.0	69.0	1.00	24.9	24.0	95.8	1.00
Postgres	1486	1366	83.0	50.0	1673	1595	82.5	67.5
Postgres_ST	1.38	1.36	1.22	1.10	1.32	1.30	1.22	1.00
Postgres_ST-hist	3.20	3.17	4.11	1.00	3.49	2.42	3.02	1.00
Neurocard_ST	1.64	1.64	1.25	1.00	1.63	1.60	1.26	1.00
Neurocard_ST-hist	6.34	5.85	1.08	1.01	14.9	13.8	1.05	1.00
DeepDB_ST	11.0	10.1	1.17	1.1	9.15	8.73	1.41	1.00
DeepDB_ST-hist	7.85	7.23	11.8	2.13	8.75	8.36	2.54	1.15

6.2.8 Low cardinality threshold selection. We varying the low cardinality threshold from 0 to 50 on `Twitter` with ST. Figure 7 demonstrates the performance. The results on the other datasets are



(a) Estimation error (b) Construction and estimation time

Fig. 7. Varying low cardinality threshold on `Twitter`

qualitatively similar. Overall, we have three observations when increasing the number of tuples. (1) The estimation accuracy is slightly improved first and then slightly reduced. This is because removing low cardinality elements has small impact on accuracy, but could reduce the domain size. (2) The settrie construction time is significantly reduced initially and then slightly reduced, since most elements have low cardinality. (3) the estimation time keeps almost same. Overall, 5 to 10 is the best threshold to balance accuracy and construction time. Hence, we select 10 as the threshold.

6.2.9 Summary. We summarize our main findings as follows.
ST is more suitable for Postgres and DeepDB. ST-hist and ST have similar estimation time on Postgres and DeepDB since they generate similar number of histograms or CDFs for each subcolumn. However, ST-hist is less accurate than ST due to the information loss in data conversion.
ST-hist is the best for Neurocard. Neurocard is inefficient in estimation. ST-hist can greatly improve its efficiency due to the small domain size and small number of subcolumns. Additionally, ST-hist also simplifies the data distribution by reducing domain size of each subcolumn. Hence, Neurocard can learn the distribution better, which improves the estimation accuracy in many cases.
ST-hist is the most balanced option. ST-hist has the best overall efficiency in most cases with good accuracy. Thus, ST-hist is a good option if we do not know much about the estimator and dataset.

7 CONCLUSION

This work focuses on selectivity estimation for queries on dataset involving set-valued attributes. We propose to factorize set-valued columns and set containment predicates into numeric subcolumns and comparison predicates so that existing estimators can be utilized for the estimation task. We develop two factorization methods ST and ST-hist. Our experiments conducted on three real-world datasets demonstrate that existing estimators enhanced with our methods can achieve better accuracy than baselines while having high estimation efficiency.

Selectivity estimation for queries with predicates involving set-valued columns and other types of columns is an open problem. Our work is among the first to explore the topic. We expect that our work could inspire future studies in this area. In the future, we aim to set hyper-parameters automatically in our solutions.

ACKNOWLEDGMENTS

This research is supported in part by MOE Tier-2 grant MOE-T2EP20221-0015.

REFERENCES

[1] Ashraf Aboulnaga and Surajit Chaudhuri. 1999. Self-tuning histograms: Building histograms without looking at data. *ACM SIGMOD Record* 28, 2 (1999), 181–192.

[2] Array operators 2023. *Array operators in Postgresql.* https://www.postgresql.org/docs/12/functions-array.html

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

[4] Containment function 2023. *Containment function in PostGIS.* https://postgis.net/docs/ST_Contains.html

[5] Cover function 2023. *Cover function in PostGIS.* https://postgis.net/docs/ST_Covers.html

[6] 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.

[7] Anshuman Dutt, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2020. Efficiently approximating selectivity functions using low overhead regression models. *VLDB* 13, 12 (2020), 2215–2228.

[8] Michael R Garey, David S Johnson, and Larry Stockmeyer. 1974. Some simplified NP-complete problems. In *STOC*. 47–63.

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

[10] GN 2021. *Geographic names of U.S.* https://www.usgs.gov/u.s.-board-on-geographic-names/download-gnis-data

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

[12] Peter J Haas, Jeffrey F Naughton, and Arun N Swami. 1994. On the relative cost of sampling for join selectivity estimation. In *PODS*. 14–24.

[13] Marios Hadjieleftheriou, Xiaohui Yu, Nick Koudas, and Divesh Srivastava. 2008. Hashed samples: selectivity estimators for set similarity selection queries. *Proceedings of the VLDB Endowment* 1, 1 (2008), 201–212.

[14] Max Halford, Philippe Saint-Pierre, and Franck Morvan. 2019. An approach based on bayesian networks for query selectivity estimation. In *International Conference on Database Systems for Advanced Applications*. Springer, 3–19.

[15] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, et al. 2021. Cardinality estimation in DBMS: a comprehensive benchmark evaluation. *Proceedings of the VLDB Endowment* 15, 4 (2021), 752–765.

[16] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep learning models for selectivity estimation of multi-attribute queries. In *SIGMOD*. 1035–1050.

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

[18] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *VLDB* 13, 7 (2020), 992–1005.

[19] Hstore 2023. *Hstore type and operators in PostgreSQL.* https://www.postgresql.org/docs/current/hstore.html

[20] Json 2023. *Json type and operators in PostgreSQL.* https://www.postgresql.org/docs/current/functions-json.html

[21] Martin Kiefer, Max Heimel, Sebastian Breß, and Volker Markl. 2017. Estimating Join Selectivities using Bandwidth-Optimized Kernel Density Models. *VLDB* 10, 13 (2017), 2085–2096.

[22] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*.

[23] Alexander Korotkov and Konstantin Kudryavtsev. 2016. Selectivity Estimation for Search Predicates over Set Valued Attributes. *International Journal of Database Theory and Application* 9, 10 (2016), 285–294.

[24] 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.

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

[26] Lipyeow Lim, Min Wang, and Jeffrey Scott Vitter. 2003. SASH: A self-adaptive histogram set for dynamically changing workloads. In *VLDB*. 369–380.

[27] Richard J Lipton, Jeffrey F Naughton, and Donovan A Schneider. 1990. Practical selectivity estimation through adaptive sampling. In *SIGMOD*. 1–11.

[28] Yao Lu, Srikanth Kandula, Arnd Christian König, and Surajit Chaudhuri. 2021. Pre-training summarization models of structured datasets for cardinality estimation. *VLDB* 15, 3 (2021), 414–426.

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

[30] M Muralikrishna and David J DeWitt. 1988. Equi-depth multidimensional histograms. In *SIGMOD*. 28–36.

[31] MySQL 2017. *SET data type in MySQL.* http://download.nust.na/pub6/mysql/tech-resources/articles/mysql-set-datatype.html
[32] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. 2020. QuickSel: Quick Selectivity Learning with Mixture Models. In *SIGMOD*. 1017–1033.
[33] Viswanath Poosala, Peter J Haas, Yannis E Ioannidis, and Eugene J Shekita. 1996. Improved histograms for selectivity estimation of range predicates. *ACM Sigmod Record* 25, 2 (1996), 294–305.
[34] Viswanath Poosala and Yannis E Ioannidis. 1997. Selectivity estimation without the attribute value independence assumption. In *VLDB*, Vol. 97. 486–495.
[35] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. 1996. Improved Histograms for Selectivity Estimation of Range Predicates. In *SIGMOD*. 294–305.
[36] Postgresql 2023. *Postgresql 15.1.* https://www.postgresql.org/docs/15/index.html
[37] Iztok Savnik, Mikita Akulich, Matjaž Krnc, and Riste Škrekovski. 2021. Data structure set-trie for storing and querying sets: Theoretical and empirical analysis. 16, 2 (2021).
[38] SQL server 2023. *Table-valued parameters in SQL server.* https://learn.microsoft.com/en-us/sql/relational-databases/tables/use-table-valued-parameters-database-engine?view=sql-server-ver16
[39] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO-DB2's learning optimizer. In *VLDB*, Vol. 1. 19–28.
[40] Jiayi Wang, Chengliang Chai, Jiabin Liu, and Guoliang Li. 2021. FACE: A normalizing flow based cardinality estimator. *Proceedings of the VLDB Endowment* 15, 1 (2021), 72–84.
[41] Dominic JA Welsh and Martin B Powell. 1967. An upper bound for the chromatic number of a graph and its application to timetabling problems. 10, 1 (1967), 85–86.
[42] Peizhi Wu and Gao Cong. 2021. A Unified Deep Model of Learning from both Data and Queries for Cardinality Estimation. In *SIGMOD*. 2009–2022.
[43] Yang Yang, Wenjie Zhang, Ying Zhang, Xuemin Lin, and Liping Wang. 2019. Selectivity estimation on set containment search. *Data Science and Engineering* 4, 3 (2019), 254–268.
[44] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: one cardinality estimator for all tables. *VLDB* 14, 1 (2020), 61–73.
[45] 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. *VLDB* 13, 3, 279–292.
[46] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2021. FLAT: fast, lightweight and accurate method for cardinality estimation. *VLDB* 14, 9 (2021), 1489–1502.

Received April 2023; revised July 2023; accepted August 2023