Identifying Insufficient Data Coverage in Databases with Multiple Relations

Yin Lin
U. Michigan
irenelin@umich.edu

Yifan Guan
U. Michigan
yfguan@umich.edu

Abolfazl Asudeh
U. Illinois, Chicago
asudeh@uic.edu

H. V. Jagadish
U. Michigan
jag@umich.edu

ABSTRACT

In today's data-driven world, it is critical that we use appropriate datasets for analysis and decision-making. Datasets could be biased because they reflect existing inequalities in the world, due to the data scientists' biased world view, or due to the data scientists' limited control over the data collection process. For these reasons, it is important to ensure adequate data coverage across different groups over the intersection of multiple attributes. Often, the dataset to be analyzed is obtained through complex joins and predicate combinations over multiple relational tables in a database. Due to the sheer data volume we often have to deal with, determining adequate coverage can require an unacceptably long execution time.

In this paper, we provide an efficient approach for coverage analysis, given a set of attributes across multiple tables. To identify regions with insufficient coverage in the combinatorially large set of value combinations, we design an index scheme to avoid explicit table joins, achieve efficient memory usage, and support predicate combination at a high level of parallelism. We also propose *P-WALK*, a priority-based search algorithm, to traverse the lattice space. Since in practice, coverage assessment typically does not require precise COUNT aggregation results, we further present approximate methods to reduce computation time. Experimental evaluation using three real-world datasets shows the effectiveness, efficiency, and accuracy of proposed methods.

PVLDB Reference Format:
Y. Lin, Y. Guan, A. Asudeh, H. V. Jagadish. Identifying Insufficient Data Coverage in Databases with Multiple Relations. *PVLDB*, 13(11): 2229-2242, 2020.
DOI: https://doi.org/10.14778/3407790.3407821

1. INTRODUCTION

The era of big data is having an increasingly significant impact on science, engineering, and society. Computer algorithms are making more objective, accurate, and efficient decisions based on large datasets and data-driven techniques. Machine learning models, for example, make predictions on the basis of large datasets. Automated Decision Systems analyze data to inform human decisions where known cases include health issues [36], education [5], policing [44], criminal justice [25], etc. Although computer algorithms promise to be data-driven and "neutral", they are never really so; the data used to support the downstream machine learning and decision-making process is never truly objective [39]. This is because we embed human bias in the process of collecting, using, and analyzing the data. Additionally, the data itself also reflects existing inequality and prejudice in the world. Analyzing the representativeness, diversity, and suitability of decision support datasets is critical in identifying and removing inequalities and biases of downstream tasks.

It has been shown in [12, 6] that inadequate data collection can induce inaccuracy and discrimination for a specific category. This topic is critical when it comes to data-driven predictions in areas such as medical diagnosis and criminal justice, where a false signal can have catastrophic effects on people's lives. Many diseases are correlated with race, gender, and other demographic factors. For example, Ashkenazi Jewish women are known to have a higher risk of breast cancer [18]; it has also been shown [8] that the death rate of heart disease varies by race and ethnicity; the weight-to-height ratio varies across racial/ethnic groups, which makes a difference when assessing childhood obesity [52]. For sentinel health surveillance systems [45] used for monitoring diseases and detecting outbreaks, coverage analysis for the selected institutions and groups is of vital significance. When using machine learning to predict potential criminal acts, population bias [39], which refers to the demographic distortion between the studied and targeted population, is also exposing minority groups to a higher risk of model unfairness. For example, a widely-applied criminal justice tool systematically assesses defendants of marginalized communities to have a higher likelihood to re-offend [21]. The Boston government was until recently using a system to assign students to schools close to their homes [19]. However, this attempt ignored the lack of high-quality schools in predominantly minority neighborhoods.

All these examples indicate the importance of analyzing the data coverage of decision-support datasets. Given a set of attributes, each of which takes one of several discrete values[1], we would like to ensure that there are enough instances of the intersection of attributes to satisfy a given "coverage threshold". We would like to identify subgroups whose coverage is insufficient in that it does not meet the specified

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407821

[1]We can bucketize continuous-valued attributes.

threshold. These under-represented subgroups are at higher risk of experiencing intersectional unfairness [27, 17, 20] in downstream data-driven algorithms. We now develop one concrete example in further detail to better appreciate our coverage analysis task.

Example 1. An insurance company wants to train a model to quote the insurance premium for drivers based on their predicted probability of getting involved in severe and fatal accidents. They use the *UK Road Safety* dataset containing records of *Accidents* and *Vehicles*. The coverage is defined on a set of attributes, including driver information, road conditions, and vehicle conditions. Before the implementation of this model, the company would like to catch any imbalance in attribute value distribution, which might mislead the prediction results. For example, the database does not contain enough accident records for young people driving new cars. This could perhaps occur because most young people do not yet have the money to buy a new car. Whatever the reason, it would not be correct for the system to assume that the accident rates are similar for young people with old cars and with new cars; perhaps young people with new cars have a flashier lifestyle and drive more recklessly. To identify such cases, the company needs to know the attribute value combinations for which the count is low. This analysis requires considering a large number of potential groups, and using complex SQL queries to access the database.

Since the number of attribute value combinations is combinatorial, determining the count for each combination can be very time-consuming. Previous work has attempted to reduce this cost, for example, by exploiting a DAG of attribute value combinations starting from no attributes specified at the root and all attributes specified at the leaves. See related work (Section 7) for a brief survey. However, previous works assume that the dataset is a single table, and the coverage of data patterns can be determined with one pass over the dataset. In the real world, big data is more commonly stored and integrated into databases, or even data warehouses, with multiple tables. They might come from heterogeneous resources, and different groups of data users may have different downstream data applications. For a given set of attributes, coverage analysis typically involves exploring combinatorial value combinations of attributes from different tables and evaluating their coverage. In this process, joining the tables, constructing indexes at run time, and combining the predicate results across tables can be computationally expensive. Such OLAP operations typically access a large proportion of the data and require multiple passes over the data. In this case, making use of indexes and approximate query processing methods can greatly save time and effort.

To summarize, we make the following contributions:

- Database coverage analysis involves a large number of similar and computationally expensive queries. We present a compact and highly parallel index scheme to handle joins and cross-table predicate combinations.
- Identifying inadequate data collections over multiple categorical attributes does not have polynomial-time solutions [6]. We make use of the monotonicity property of data patterns and propose a priority-based algorithm to minimize the number of COUNT operations, which is the most time-consuming step of coverage analysis.
- Since approximate answers usually suffice for the coverage analysis problem, we consider sampling from the original database to respond quickly with smaller data sizes. We address challenges in sample-then-join estimation by inserting correlation-aware samplers and construct stratified samples in one pass over the data.
- We evaluate our approach on three real-world datasets having millions of records. We compare our method with baseline methods based on single-table algorithms and analyze the efficiency and accuracy of our method.

2. PRELIMINARIES

We follow the general setup of [6], making only the changes required by our problem. Consider a database $\mathcal{D}$ with multiple tables $\{T_1, \cdots, T_n\}$, where each table T_i has n_i tuples defined over m_i attributes $\{a_{i,1}, \cdots, a_{i,m_i}\}$. We use the notation $a_{i,j}$ to refer to the j-th attribute of T_i. We consider a user-specified set of "attributes of interest", $\mathcal{A} = \{A_1, \cdots, A_k\}$, where every $A_\ell \in \mathcal{A}$ is an attribute $a_{i,j}$ in a table T_i. Let T be the set of tables that have at least one attribute in the attributes of interest, i.e., $\mathsf{T} = \{T_i \mid a_{i,j} \in \mathcal{A}\}$. Without loss of generality, we assume $\mathsf{T} = \{T_{i_1}, \cdots, T_{i_{k'}}\}$ forms a join table, in form of $\times_{\mathsf{T}} = T_{i_1} \bowtie T_{i_2} \bowtie \cdots \bowtie T_{i_{k'}}$. In this paper, we consider the equi-join of tables, which includes a wide range of joins such as key-foreign key joins, star-joins and many-to-many joins. Our objective is to investigate the coverage over $\mathcal{A}$ in $\times_{\mathsf{T}}$, as defined in [6]. For convenience, we summarize the symbols in Table 1.

For attributes having continuous values or with high cardinalities, we first use the bucketization technique to reduce the computational complexity by grouping similar values into the same bucket. After bucketization, the total number of attribute value combinations is equal to $\prod_{i=1}^{k} c_i$, where c_i is the cardinality of attribute A_i. Definition 2.1 provides a formal statement of attribute value combinations.

Definition 2.1. (Pattern). A data pattern, $\mathcal{P}$, is a k-dimensional vector of value combinations for the attribute set $\mathcal{A} = \{A_1, A_2, A_3, ..., A_k\}$. The i-th position P_i represents the value of attribute A_i, which can either be X (meaning this element is unbound) or bound to a deterministic value. Formally,

$$\mathcal{P} = \langle P_1, P_2, P_3, ..., P_k \rangle$$

where P_i is either a deterministic value of A_i or $P_i = X$.

For example, consider a four-dimensional data pattern $\mathcal{P} = 0X1X$, which contains two bound elements and two unbound elements. This data pattern considers an attribute set with four attributes but with only two predicates, representing by bound element value assignments $A_1 = 0$ and $A_3 = 1$. In the join table $\times_{\mathsf{T}}$ of attribute set $\mathcal{A}$, we say a record *satisfies* pattern $\mathcal{P}$ if for all values $d_i = P_i$, this record satisfies $\mathcal{A}_i = d_i$.

We find it useful to allow "don't care" values for some attributes while specifying a pattern. It is permitted by unbound elements in the data pattern. We can thus have a root pattern with "don't care" for each attribute value, which matches every tuple in the database, leaf patterns that are fully instantiated attribute value combinations, and everything in between. The total number of distinct patterns is $\prod_{i=1}^{k}(c_i + 1)$, where we add one to the cardinality c_i in each term of the product to account for the "don't care" value.

Table 1: Table of Notations

Symbol	Description
$\times_{\mathsf{T}}$	Join table of tables in set T
A_i	The i-th attribute of the attribute set $\mathcal{A}$
P_i	The i-th dimension of data pattern $\mathcal{P}$
τ	Data coverage threshold
$cov(\mathcal{P})$	Coverage of data pattern $\mathcal{P}$
$\mathcal{G}(\mathcal{A})$	Data pattern graph for attribute set $\mathcal{A}$
MUP	Maximal Uncoverd Pattern

Given a set of "attributes of interest" $\mathcal{A}$ and a fixed coverage threshold τ, sufficient data coverage means that the join table $\times_{\mathsf{T}}$ of table set T contains at least τ records satisfying pattern $\mathcal{P}$. The value of coverage threshold τ varies according to the need of downstream applications. Identifying the insufficient coverage for the underlying data distribution enables us to get a better handle of potentially underrepresented groups in downstream algorithms.

Definition 2.2. (Covered/Uncovered Pattern). Given a database $\mathcal{D}$ and a data pattern $\mathcal{P}$, the coverage of a data pattern $cov(\mathcal{P})$ is defined as the number of records in the join table $\times_{\mathsf{T}}$ that satisfy pattern $\mathcal{P}$. With the fixed coverage threshold τ, pattern $\mathcal{P}$ is a covered pattern if $cov(\mathcal{P}) \geqslant \tau$. Otherwise, this pattern is said to be uncovered.

If we regard uncovered regions as data holes in the k-dimensional data space represented by the k attributes in set $\mathcal{A}$, larger data holes represent regions that cover more dimensions in the data space. In most situations, larger uncovered regions are more harmful than smaller ones. As a result, we aim to find the maximal data holes in the data space of the join table $\times_{\mathsf{T}}$. In Definition 2.3, we define the dominance of data patterns. Based on the dominance relation, we construct the pattern graph and define the Maximal Uncovered Pattern in Definition 2.4.

Definition 2.3. (Dominance of Data Patterns). A pattern $\mathcal{P}_1$ is said to be dominated by a pattern $\mathcal{P}_2$, if $\mathcal{P}_1$ can be obtained by assigning any of the unbound elements in $\mathcal{P}_2$ with a deterministic value. In this case, $\mathcal{P}_1$ is a descendant of $\mathcal{P}_2$, and $\mathcal{P}_2$ is the ancestor of $\mathcal{P}_1$. Furthermore, we say a pattern $\mathcal{P}$ is dominated by a pattern set $\mathcal{S}$ when there exists a pattern $\mathcal{P}' \in \mathcal{S}$ that dominates $\mathcal{P}$, and we say $\mathcal{P}$ dominates $\mathcal{S}$ when $\mathcal{P}$ dominates $\mathcal{P}' \in \mathcal{S}$ for some $\mathcal{P}'$.

For example, consider two pairs of data patterns ($\mathcal{P}_1 = 1XX$, $\mathcal{P}_2 = 1X0$) and ($\mathcal{P}_1 = 1XX$, $\mathcal{P}_3 = 110$). From the definition of dominance, we say that $\mathcal{P}_1$ dominates $\mathcal{P}_2$ and $\mathcal{P}_3$. On top of that, we define the parent-child relationship such that a parent can be obtained by replacing one of the bound elements from the child pattern with X. In this way, $\mathcal{P}_1$ is a parent of $\mathcal{P}_2$, and $\mathcal{P}_2$ is a parent of $\mathcal{P}_3$.

With the dominance relation of data patterns, we can construct the pattern graph $\mathcal{G}(\mathcal{A})$ for all possible value combinations. The number of layers in the pattern graph is $d+1$ where d is the number of dimensions of attribute set $\mathcal{A}$. Given a pattern $\mathcal{P}$ with k_b bound elements and k_u unbound elements, pattern $\mathcal{P}$ is in *Layer* k_b of the pattern graph. The number of parents of $\mathcal{P}$ is k_b, where each parent is obtained by changing one of $\mathcal{P}$'s bound element to unbound. The number of children for $\mathcal{P}$ is the sum of the cardinalities for all k_u unbound elements. This is because we pick a value from the data domain of each unbound attribute. Formally, the

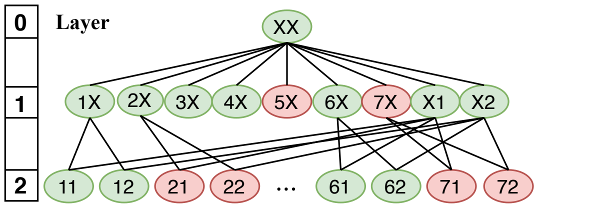


Figure 1: Pattern Graph for Attribute Set ($d = 2, c = 7, 2$)

number of parents of pattern $\mathcal{P}$ is $\#\ of\ parents(\mathcal{P}) = k_b$, while the number of children is $\#\ of\ children(\mathcal{P}) = \sum_{i=1}^{k_u} c_i$.

Figure 1 shows an example of the pattern graph for the running example in Section 4. The attribute set has a dimension of 2, and the cardinalities for each attribute are 7 and 2, respectively. In *Layer* 0, there is only one data pattern XX whose attribute values are all unbound. In *Layer* d, all $\prod_{k=1}^{d} c_k$ data patterns contain only bound elements, where c_k represents the cardinality for attribute A_k. The total number of data pattern nodes is $\prod_{k=1}^{d}(c_k + 1)$. In the pattern graph, there are links between each parent-child pair. The hierarchical structure of pattern graphs based on the dominance of patterns is very helpful when pruning the searching process among the combinatorial explosion of value combinations. Once we find an uncovered pattern, all its descendants are uncovered patterns. Similarly, for each covered pattern, all its ancestors are covered.

In Figure 1, we color all uncovered patterns in red and covered patterns in green. For example, pattern node $7X$ in *Layer* 1 is uncovered, and therefore all its children are uncovered. Similarly, pattern node 61 in *Layer* 2 is covered and all its parent nodes $6X$ and $X1$ are covered patterns.

Definition 2.4. (Maximal Uncovered Pattern, MUP). Pattern $\mathcal{P}$ is said to be a maximal uncovered pattern if its coverage $cov(\mathcal{P}) < \tau$, and for all its parent pattern $\mathcal{P}'$, $cov(\mathcal{P}') \geqslant \tau$.

In other words, an MUP refers to an uncovered pattern whose parents are all covered. In Figure 1, node $5X$, $7X$, 21, and 22 are MUPs since all their parents are covered. However, node 71 and 72 are not MUPs since they have an uncovered parent $7X$. When the data distribution is more uniform, MUPs are more likely to appear in lower layers. In contrast, when the data distribution is skewed, MUPs are more likely to appear in the upper layers.

All descendants of an MUP must also necessarily be uncovered. A user would typically prefer to be informed about a single MUP rather than the potentially large number of uncovered descendant patterns. Based on definitions above, our coverage analysis problem can be formulated as follows:

Problem 1 (Database Coverage Analysis). *In database $\mathcal{D}$, given a coverage threshold τ, analyze the coverage over attribute set $\mathcal{A}$ in the join table $\times_{\mathsf{T}}$, and find all maximal uncovered patterns $\mathcal{M} = \{\mathcal{P} \mid \mathcal{P} \in \mathcal{M}$ if P is an MUP$\}$.*

3. BASELINE METHODS

We provide a motivating example in *Internet Movie Data Base (IMDB)* to analyze the drawbacks of straightforward baseline methods. We construct a set of "attributes of interest" comprising 6 attributes from 5 relations. Figure 2 shows the query graph of the example. The circles are the attributes in set $\mathcal{A}$, while the squares are the relations in

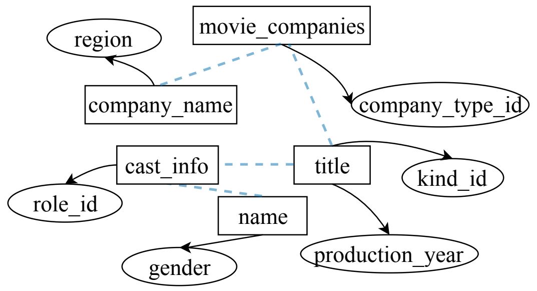


Figure 2: Query Graph for Motivating Example

set T, that contain attributes in $\mathcal{A}$. We use dashed edges to represent join predicates and solid arrows to indicate which relation the attribute is part of. Our coverage requirement is to ensure that we have enough instances for each categorical group. For example, we wish to be alerted if, in our database, all action movies are from big American studios, or all docudramas are from independent producers in recent years. According to the definitions in Section 2, this attribute set has a dimension of 6. The total number of data patterns is $\prod_{i=1}^{6}(c_i+1)$, where c_i denotes the cardinality of the i-th attribute.

An MUP identification algorithm has two major steps: (1) searching (and pruning) the pattern graph, and (2) computing the coverage of the related pattern when visiting a node in the graph. It has been proven that it is not possible to identify the MUPs of even a single table with a polynomial-time algorithm [6]. In view of the combinatorially large MUP search space (pattern graph), for the second step, we consider two types of baseline methods here. We can either check the coverage for each pattern individually using a conjunctive COUNT query or perform group-by to get the counts for all patterns in the leaf layer, and adding them up to determine the coverage for other nodes. We present below query execution plans for baseline methods.

Solution 1. Apply WHERE clause with JOIN predicates. For each data pattern we are visiting in the searching process, we use WHERE clause for bound elements value filtering and consider the join restriction to get the join result. For example, suppose we want to determine the pattern's coverage in the bottom (leaf) layer. The query would be:

```
SELECT COUNT(*)
  FROM title AS t, cast_info AS ci, company_name AS cn,
       name AS n, movie_companies AS mc
 WHERE t.kind = 'tv series'
   AND t.production_year = '2000 - 2010'
   AND n.gender = 'f'
   AND ci.role = 'director'
   AND cn.country_code = '[us]'
   AND mc.company_type = 'production companies'
   AND <join predicates>;
```

Issue 1. We profile the query execution plan using the PostgreSQL EXPLAIN function, to show how the involved tables are scanned (by index or sequential scan) and what kind of join algorithm will be applied. As indicated in the execution plan, the most time-consuming operation is the hash join to generate the temporary join table $\times_{\mathsf{T}}$. To consider attributes across multiple tables, join operations usually involve a large proportion of the data. The coverage analysis of a combinatorially large pattern set unavoidably requires multiple passes over the data. Performing join operations repeatedly is not desirable even if they can make use of indexes in original tables. Alternatively, materializing the join table could save the time to perform joins. However, creating the join table also has critical limitations: the long time latency of table scans and the lack of indexes of the join table. To create the join table and the corresponding indexes, prohibitively large storage space is also needed as many queries have a product/join of large input relations. Experimental results also show the enormous index creation time for each attribute in the join table, without which the cost of repeated full table scans (required to compute counts) becomes prohibitive.

Solution 2. Instead of checking the coverage of each data pattern individually via a conjunctive COUNT query, the group by operation computes the counts for all data patterns in the bottom (leaf) layer with a shared join operation. These numbers can be added up to obtain counts for nodes in the upper layers, and hence to find MUPs in the pattern graph. The corresponding query would be:

```
SELECT COUNT(*)
  FROM title AS t, cast_info AS ci, company_name AS cn,
       name AS n, movie_companies AS mc
 WHERE <join predicates>
 GROUP BY t.kind, t.production_year, n.gender, ci.role,
       cn.country_code, mc.company_type;
```

Issue 2. If the number of leaves is not too large, this is indeed an effective technique: a single scan of the table may suffice, and even more, efficient access methods may be possible. However, this bottom-up computation does not scale well, given the combinatorially large number of nodes in the pattern graph. The memory consumption of recording all bottom-layer patterns and their counts also increases quickly with the growth of the dimension and the cardinalities of attributes. In addition, note that the GROUP BY gives us counts for the leaf nodes: we still have to compute summations for other nodes in the pattern graph. If we want to identify MUPs in the pattern graph following top-down search, we need to add up the combinatorially large size of pattern counts each time we want to compute the coverage for a node in upper layers.

4. EXACT COVERAGE ANALYSIS

Index structures are traditionally used to speed up query evaluation in a database. Building upon the extensive work on this topic, in Section 4.1, we propose several schemes to avoid actual table joins and execute efficient predicate combinations across different tables, by using join indexes and bit-mapped index arrays. Then, in Section 4.2, we develop a priority-based search algorithm to minimize the number of COUNT operations required.

4.1 Index Structures for Pattern Coverage Assessment

Coverage computation of each data pattern is the key operation in the MUP identification process. However, in coverage exploration queries, creating the join table is a time and space consuming operation. Our method proposes to make use of the join index structure. We first illustrate the basic implementation of the join index and gradually explain further improvements in the following subsections.

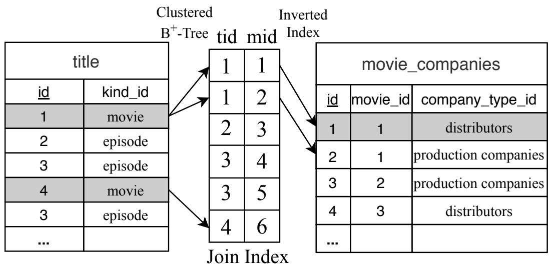


Figure 3: Using Join Index for Data Pattern Coverage Assessment, $cov(\mathcal{P} = 11)$

4.1.1 Join Index

Valduriez [49] proposed using a join index for efficient join operations. A join index is a data structure projecting, down to row identifiers, the join result of two relations. Assuming that each row in each relation is identified by a surrogate (row identifier RID) uniquely, the join index stores surrogates of record pairs from the two relations that appear in the join result. Formally, the join index on relation R and S with join predicate $f(R, S)$ is represented by:

$$JI = \{(r_i, s_j) \mid f(r_i, s_j) \text{ is true}\}$$

Even when index construction cost is taken into account, it has been shown in [49] that join index usually outperforms the hybrid-hash join and index nested loop join methods because join index is able to make efficient use of memory and has high adaptiveness to parallel execution.

We note that it is straightforward to generalize the idea of a join index from two tables to multiple tables. We do not discuss this generalization here in the interests of space.

A typical way to use a join index, also applicable in our scenario, is to identify records in the first table that satisfy the desired predicate, using single-table index structures, such as a B-tree. In our case, the desired predicate will be an equality predicate on the values of attributes in this table whose values are specified in the pattern of current interest. For each RID from one table, for a row satisfying such a predicate, we can use the join index to determine which rows satisfy the join restriction in other tables. Once again, these rows are identified by RID, and we may need to retrieve the actual row for each such RID to determine whether it satisfies the desired attribute pattern for attributes in the second (and third, fourth, ...) table. Consequently, building efficient indexes to enable fast access to join index records via RIDs in different tables is essential. In general, a simple method is to maintain multiple copies of the join index, each one clustered on RIDs of different tables, and then implement a clustered B^+-Tree to enable quick access.

In Figure 3, we show an example of using join index to determine the coverage for the "attributes of interest" set $\mathcal{A} = \{$`title.kind_id`, `movie_companies.company_type_id`$\}$. For simplicity, we encode the domain of each attribute as integers. The domain for `title.kind_id` is {1, 2, 3, 4, 5, 6, 7}, and the domain for `movie_companies.company_type_id` is $\{1, 2\}$. These two tables are related through a key-foreign key join. The join index table stores row identifiers in the join result. Say we want to compute the coverage of a data pattern $\mathcal{P} = 11$ (`<movie, distributors>`), which means that the number of instances in the join table satisfying the predicate conditions of `title.kind_id = 1` (`movie`) and `movie_companies.company_type_id = 1` (`distributors`).

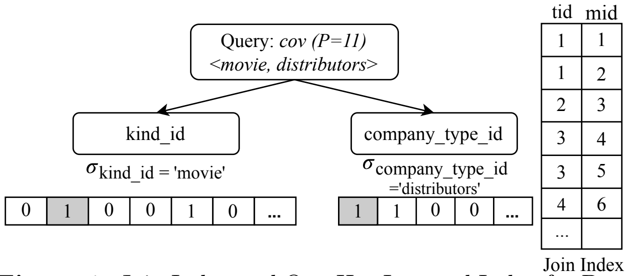


Figure 4: Join Index and One-Hot Inverted Index for Data Pattern Coverage Assessment, $cov(P = 11)$

The algorithm will firstly use the index on the predicate column of one table to get a set of entries (RIDs) to the join index table. Each entry to the join index will return a row identifier to retrieve attribute values in the other table.

4.1.2 One-Hot Inverted Indexes for Predicate Result Access

Making use of the join index overcomes the long time latency of the join operation. However, we still need to access the original table and check attribute values with RID entries from the join index. Generally, clustered B^+-Tree takes 2-3 disk I/Os for each index access on average.

A possible alternative is to use an "inverted index" and store RIDs of the rows in the second table with desired values for "attributes of interest". Of course, this only works for very selective predicates that produce small output results. Otherwise, these lists of RIDs can grow long, and we lose the central benefit of the join index, which is its small size compared to the actual join result. Similar challenges have been addressed in the context of data warehouses, through the use of bit-mapped index structures. In this subsection, we present the construction of (bit-mapped) One-Hot Inverted Indexes, which allows the process of checking attribute values in different tables to be completed in constant time complexity and in the memory without any disk I/Os.

One-Hot Inverted Indexes are a set of bit arrays, where each bit array is associated with one attribute value and stores its predicate result. Therefore, the total number of bit arrays we need to construct equals to the sum of cardinalities of all attributes $\sum_{i=1}^{d} c_i$, where d is the number of attributes in set $\mathcal{A}$ and c_i is the cardinality for the i-th attribute. The length of each bit array is associated with the length of the table this attribute belongs to.

The i-th bit in an inverted index represents the i-th row of the corresponding table; it is set to 1 when this row satisfies the predicate (the specified attribute value), and to 0 otherwise. To represent one predicate result of a table with 10 million rows, we only need about 1MB storage (10M bits). As a result, the One-Hot Inverted Indexes can be cached in memory. In addition, the construction of One-Hot Inverted Indexes can rely on the index structures on original tables and hence adds little time overhead.

Consider the example of $\mathcal{A} = \{$`title.kind_id`, `movie_companies.company_type_id`$\}$. We construct One-Hot Inverted Indexes for each attribute value. As shown in Figure 4, to get the coverage of a specific pattern, e.g., $\mathcal{P} = 11$, we can either scan the join index and check the bit arrays representing `kind_id = 1` and `company_type_id = 1`, or check the bit array on one table first, for all 1's in the bit array, access the join index and check the bit array on the other table. We can make this execution decision according to the join and the WHERE clause selectivity.

4.1.3 *Universal Inverted Indexes for Combining Predicates in Different Tables*

The construction of One-Hot Inverted Indexes allows us to avoid accessing the large original tables and efficiently combine predicates in the same table. However, since bit arrays representing attributes from different tables have different sizes and position mapping, we cannot simply perform AND operations on arrays from two or more tables. Scanning and accessing the join index is needed to "translate" bit arrays. In this section, we consider doing bit array extensions to support AND operations for combining predicates on different tables. If we can improve this process to pure bitwise operations, combining predicates can be very efficient, since we can execute 64 bits (or the word width on the computer system used) in parallel. We call the set of extended bit arrays as *universal inverted indexes*.

Our focus would be on many-to-one equijoins, such as key-foreign key joins and star joins, which constitute more than 90 percent of real-world cases [40]. For key-foreign key joins, each row in the *child* table would find exactly one match in the *parent* table. Similarly, in the star join schema, there will be a large *fact* table with foreign keys referencing any number of *dimension* tables. For this reason, the join size for many-to-one join is the same as the size of the child or the fact table. This property makes it possible to extend bit arrays for small parent/dimension tables to the same size as the child/fact table and support efficient AND operations [40]. For less common many-to-many equijoins for large tables, extending bit arrays might not be efficient since we need to extend bit arrays on both sides, and the final join result can be as large as the Cartesian product of the tables. In this situation, we use the method we proposed in Section 4.1.2.

Essentially, the construction of Universal Inverted Indexes is the extension of One-Hot Inverted Indexes on small tables to the same size of the large table. This extension process relies on the materialized join results in the join index: for each original One-Hot Inverted Indexes, we read RIDs of all 1's in parent/dimension tables, and set the corresponding bits representing rows in the child/fact table according to join index records.

In our running example, there is a key-foreign key join between the `title` and the `movie_companies` table. We need to extend the bit arrays representing the predicate results for the smaller parent table, `title`, to the same size as the `movie_companies` table. Figure 5 illustrates the extension process. The original bit array for `kind_id = 1` stores filter results as RIDs of the `title` table. The 1 in the first position means that the first row in the `title` table satisfies the predicate condition `kind_id = 1`. In the extension process, we find that the first row in `title` joins the first two rows in

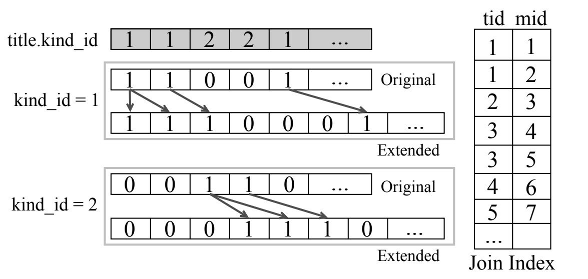


Figure 5: Bit Array Extensions on the Referenced Table

`movie_companies`. Therefore, in the extended bit array, we set the first two bits as 1. Similarly, since the 2nd and 5th row of `title` join the 3rd and 7th row of `movie_companies`, we set the corresponding bits as 1. Because we can view the many-to-one join as predicates over a wider column set, we can perform AND operations for extended bit arrays to combine predicates in different tables.

For now, we have discussed several optimizations to quickly determine the coverage for a given data pattern. Algorithm 1 illustrates a detailed process of counting pattern coverage. Coverage computation adopts different strategies for many-to-one and many-to-many joins. For many-to-one joins (line 2-6), the computation only relies on the extended Universal Inverted Indexes since they have incorporated the join result in the bit-mapped form. For many-to-many joins (line 7-14), the coverage computation scans the join index in parallel and checks corresponding bits in One-Hot Inverted Indexes for attribute values.

Algorithm 1 Count Pattern Coverage

```
Input: Data pattern $\mathcal{P} = a_1, a_2, ..., a_n$, join index $JI$, bit vectors $\mathcal{V}$ storing predicate and join results
Output: Coverage count $cnt$ of pattern $\mathcal{P} = a_1, a_2, ..., a_n$
 1: Initialize $cnt = 0$
 2: if only consists many-to-one join relation then
 3:     $bv = [1]$ * length of detail table
 4:     for each Universal Inverted Index $b_i$ that represents
          a predicate of bound elements in pattern $\mathcal{P}$ do
 5:         $bv = bv \wedge b_i$
 6:         $cnt$= number of 1's in $bv$
 7: else
 8:     for each records of row identifiers $r_{i,k}$ in $JI$ do
 9:         $Flag = True$
10:         for each bound element $a_i$ in pattern $P$ do
11:             Find One-Hot Inverted Index $b_i$ represents $a_i$
12:             Check position $r_{i,k}$ of $b_i$
13:             if $b_i[r_{i,k}] = 0$ then $Flag = False$
14:         if $Flag = True$ then $cnt{+}{+}$
15: return cnt
```

4.2 MUP Identification Algorithms

In the coverage analysis task, we aim to determine the coverage for every possible value combinations and find all maximal uncovered patterns (MUPs). We not only need to optimize the execution of COUNT queries, but also need a well-designed traversal algorithm to minimize the number of COUNT operations performed. In this section, we discuss various algorithms to efficiently prune the search space.

4.2.1 *Pattern Graph Traversal*

One obvious choice is to traverse the pattern graph (recall Fig. 1) top-down from the root in a breadth-first manner. If a node has sufficient coverage (the count for the associated pattern is greater than or equal to a specified threshold), then all its immediate children are added to the queue. If a node is found not to have sufficient coverage, it is reported as a MUP, and all its descendants can be removed from further consideration since each must be uncovered but none can be maximally uncovered. (There is some additional bookkeeping required to avoid visiting a node multiple times, since the graph is a lattice and not a tree. This is straightforward, and not conceptually interesting; hence we do not describe fully). Such a top-down traversal works well to find "large" MUPs, close to the root, very quickly. However, if the MUPs are low in the pattern graph, we may end up computing counts for almost every pattern.

An alternative could be to try a bottom-up traversal, starting from the leaf nodes in the pattern graph. If a node is uncovered when we compute its count, then its parents are added to the queue. If a node is covered, then we do not need to explore any of its ancestors, since they must all also be covered. Furthermore, its children could possibly be MUPs, but we have to check other parents of each child to determine whether it is a MUP. This algorithm also requires bookkeeping to avoid re-visiting nodes. It also needs a final MUP determination phase. These small costs could be worth it if the MUPs are close to the leaves. However, we note that the number of leaves is large, and may even constitute the vast majority of the pattern graph. Exploring all the leaves may leave out only a small fraction of non-leaf nodes in the pattern graph.

To address the difficulties with the top-down and bottom-up algorithms, [6] proposed the deep diver (DD) algorithm and showed that it performed much better. DD uses a depth-first search in the pattern graph, diving deeper to identify MUPs. However, like the first two methods, it treats all categorical attribute values equally, so that the search process cannot benefit from the different pruning efficiency of nodes. We propose a new algorithm, *P-WALK*, which makes use of priority-based search to visit the node with the highest pruning efficiency first. In addition, the pruning of *P-WALK* not only relies on the discovered MUPs but also on the coverage status of all previously visited nodes.

4.2.2 *P-WALK Algorithm*

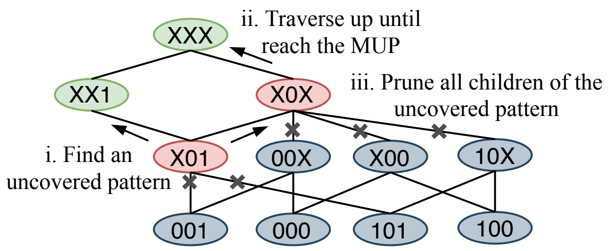
Like previous algorithms, *P-WALK* also makes use of the data patterns' monotonicity property: for each covered pattern, all its parents are covered, and for each uncovered pattern, all its children are uncovered. The dominance of the data patterns enables us to prune all parents/children branches once we find a covered/uncovered pattern. In view of this property, the main idea of *P-WALK* is to prioritize visiting nodes that have the greatest pruning power. To heuristically approach the "greatest pruning power", we define a priority scoring function as follows in Definition 4.1.

Definition 4.1. (Priority Scoring Function) The priority scoring function is a heuristic function to decide the order in which to check the coverage of each data pattern. Formally, the priority for each data pattern is:

$$priority = \omega_p \times n_p + \omega_c \times n_c$$

where n_p and n_c are the number of parent nodes and child nodes for each data pattern, ω_p and ω_c are the weights for parents and children.

If we assign a higher weight for child nodes, the priority search algorithm will be close to top-down BFS, because nodes in higher layers have more child nodes. On the other hand, when parent nodes are assigned higher weight, the search algorithm is more likely to traverse deep to the lower layer. For nodes in the same layer, the cardinality of its unbound attributes will also affect the priority score, preferring nodes with more neighbors to get higher pruning efficiency. Once we find a covered pattern, we visit the next node with the highest priority; once we find an uncovered pattern, we check its ancestors iteratively until we find MUPs.


ii. Traverse up until reach the MUP
iii. Prune all children of the uncovered pattern
i. Find an uncovered pattern
XXX
XX1
X0X
X01
00X
X00
10X
001
000
101
100
Figure 6: Example of *P-WALK* Pruning Process

We illustrate the iterative checking-ancestor operation and the pruning of *P-WALK* algorithm in Figure 6. It shows part of the pattern graph with 3 binary attributes. The red and green nodes stand for uncovered and covered nodes respectively, while grey nodes represent pruned nodes. Suppose we first visit the node $P = X01$ and find it is uncovered. *P-WALK* would first go up until it reaches a MUP that dominates it. The MUP in this example is $P = X0X$. In this process, we prune its ancestors whenever we find a covered pattern, e.g., $P = XX1$, and prune its descendants, whenever we find an uncovered pattern, e.g., $P = X01, X0X$.

We combine this *P-WALK* search algorithm with the index-based data pattern coverage counting algorithm we proposed in Sec. 4. The pseudo-code of *P-WALK* is shown in Alg. 2.

In Alg. 2, searching the nodes with higher pruning efficiency is crucial to reduce unnecessary count operations. When a node is dominated by MUPs or dominates a covered pattern, we can prune this branch based on the coverage monotonicity property (line 8-9). For each covered node, in line 11-13, we push all its children to the priority queue and add the node to the covered set. For each uncovered pattern, in line 15-23, *P-WALK* starts a local search that goes up to its ancestors to find MUPs that dominates this pattern.

Algorithm 2 *P-WALK* Coverage Analysis Algorithm

```
    Input: Pattern graph $\mathcal{G}(\mathcal{A})$, coverage threshold $\tau$
    Output: Maximal uncovered patterns $\mathcal{M}$
 1: procedure INITIALIZATION
 2:     $neighbors \leftarrow$ empty priority queue sorted nodes by
    the priority scoring function
 3:     $\mathcal{C}$ an empty set to store covered nodes
 4: procedure COVERAGEASSESSMENT
 5:     push the root node $P = XX...X$ to $neighbors$
 6:     while $neighbors$ is not empty do
 7:         $p$ = pop a node from $neighbors$
 8:         if $p$ is dominated by $\mathcal{M}$ or $p$ dominates $\mathcal{C}$ then
 9:             continue
10:         $cov(p)$ = COUNT_PATTERN_COVERAGE($p$)
11:         if $cov(p) \geqslant \tau$ then
12:             push all children of $p$ into $neighbors$
13:             push $p$ into $\mathcal{C}$
14:         else
15:             Initialize stack $S' \leftarrow$ empty stack
16:             push all parents of $p$ into $S'$
17:             while stack $S'$ is not empty do
18:                 pop a node $p'$ from $S'$
19:                 if $cov(p') < \tau$ then
20:                     if all parents of $p'$ are covered then
21:                         push $p'$ into $\mathcal{M}$
22:                     else push all uncovered parents into $S'$
23:                 else push $p'$ into $\mathcal{C}$
24: return maximal uncovered patterns $\mathcal{M}$
```

5. APPROXIMATE COVERAGE ANALYSIS

Coverage assessment queries return numerical results to determine whether pattern coverage is greater than or equal to the given threshold. Providing a precise COUNT result is not required. Sampling-based techniques are common methods to estimate query results by taking samples from the data. In this section, we consider the use of sampling-based techniques to further improve upon the performance we obtained with index-based techniques in the preceding section.

For coverage analysis task, using the Bernoulli sampler to pass rows uniformly-at-random is insufficient given the following drawbacks. Firstly, the estimated aggregation results have a higher variance for small groups and highly selective predicates. To identify the insufficient data coverage, we are actually interested in small groups generated via predicate conjunctions. In this case, using stratified sampling to improve the accuracy of small groups would be necessary. In Section 5.1, we discuss how we adapt **distinct sampler** [26] to construct stratified samples in one pass over the data.

Additionally, to avoid the costly join-then-sample operation [11, 14], we would like to push samplers past join operations so that sampling (and the associated downsizing) occurs first. Pushing Bernoulli/Uniform samplers past joins is not effective, as joining independently obtained random samples is likely to severely under-estimate the join size. We adopt the **Correlated sampling method** [50] to improve the efficiency and accuracy of sample-then-join.

After sampling, we run *P-WALK* MUP identification algorithm and check the pattern coverage more efficiently with much smaller data size. This remains the same as discussed above in Section 4, except for the use of estimated counts. We provide below respective application scenarios, detailed implementation, and variance analysis for the two samplers.

5.1 Distinct Sampler

Frequently, a data warehouse is organized in a star schema, where large fact tables contain foreign keys of other relations. As mentioned in Section 4.1.3, such joins produce results equal in size to the original fact table. In fact, these joins can be regarded as stratifying predicates over a table with a wider column set, including columns from the small dimension tables. The same logic applies to sampling. As such, we apply the distinct sampler [26] only to the fact tables and maintain the original small dimension tables.

Consider the join $T_i \bowtie T_j$ where T_i is the referenced table and T_j is the referencing table. We want to construct the sample table $T_{s_{i,j}}$ from T_j, with columns A_s that include the join column on T_i as well as the columns $a_{j,\ell} \in \mathcal{A}$ from T_j. To ensure $T_{s_{i,j}}$ includes enough records for each group, we consider stratified sampling for every possible value combination in A_s. Specifically, for every value combination $\mathcal{C}$ from the cardinality of A_s, we would like to maintain a sample of portion p of it, while considering $p \in (0, 0.1]$.[2] For example, $p = 0.1$ means that we would like to include 10% of instances matching $\mathcal{C}$ in $T_{s_{i,j}}$. For value combinations that include less than k instances in T_j, we include all of them in $T_{s_{i,j}}$. Following the rule-of-the-thumb in central limit theorem, we set $k = 30$.

The distinct sampler uses the reservoir sampling technique [2] while considering reservoirs of sizes larger than S for different value combinations. Given parameters A_s, f, p, the sampler $\Gamma^{\mathcal{D}}_{A_s,f,p}$ returns at least f rows (if available), for each value combination C of attributes in the column set A_s and pass the other rows with the probability of p in database $\mathcal{D}$. It does so by performing a three-stage sampling process in one pass through the table. First, for each distinct value combination, the distinct sampler passes the first f rows with weight $= 1$. If the value count of this data pattern is less than f, the number of records contained in the samples would be the exact frequency count for this value. Subsequently, distinct sampler maintains a reservoir with size S; for value frequencies (counts) within $(f, f + S/p]$, distinct sampler passes the first f tuples with weight 1 and passes the others in the reservoir with the weight of $(\text{freq} - f)/S$. During this process, the reservoir keeps flushing rows with probability $\frac{S}{\text{freq}-f}$. If the value frequency is larger than $f + S/p$, the sampler will pass the rest of rows with probability p. Formally, Equation (1) shows how distinct sampler works when observing different value frequency.

$$\Gamma^{\mathcal{D}}_{A_s,f,p} = \begin{cases} \text{1. if freq} \leqslant f\text{, pass all rows with w} = 1 \\ \text{2. if } f < \text{freq} \leqslant f + S/p\text{, pass rows in the} \\ \text{reservoir with w} = (\text{freq} - f)/S \\ \text{3. if freq} > f + S/p\text{, pass rows with w} = 1/p \end{cases} \tag{1}$$

We select S and f such that $k = S + f$ to ensure to include all instances of value combinations with frequencies k in the sample table $T_{s_{i,j}}$. Having the sample table $T_{s_{i,j}}$ constructed, we use the *Horvitz-Thompson (HT) estimator* [26] for aggregate estimation. For each group G, the estimated COUNT aggregation result $C(G)$ can be written as:

$$C_{T_{s_{i,j}}}(G) = \sum_{t \in T_{s_{i,j}}(G)} \frac{C(t)}{\Pr[t \in T_{s_{i,j}}(G)]} \tag{2}$$

Because the HT estimator is unbiased, we get $E[C_{T_{s_{i,j}}}(G)] = C(G)$. Also, since $Var[X] = E[X^2] - E^2[X]$:

$$\text{Var}[C_{T_{s_{i,j}}}(G)] = \sum_{i,j \in G} \left(\frac{P[i,j \in T_{s_{i,j}}(G)]}{\Pr[i \in T_{s_{i,j}}(G)] \cdot \Pr[j \in T_{s_{i,j}}(G)]} - 1 \right) \cdot C(i)C(j) \tag{3}$$

For the distinct sampler, the weight of samples is the reciprocal of the sampling probability. Therefore, we have:

$$P[i \in T_{s_{i,j}}(G)] = \begin{cases} 1 & \text{freq} \leqslant f \\ \max(f/\text{freq}, p) & \text{freq} > f \end{cases} \tag{4}$$

If $i \neq j$, $\Pr[i, j \in T_{s_{i,j}}(G)] = \Pr[i \in T_{s_{i,j}}(G)]\Pr[j \in T_{s_{i,j}}(G)]$.

5.2 Correlated Sampling Method

For many-to-many joins (like joining two fact tables) of large relations, we apply correlated sampling method to both of the relations and pick the rows having join values in the same data space. Although it will introduce higher variance, it is practical in reducing the enormous join size of large tables, in which situation the join size could as large as the Cartesian product of large input relations.

Although distinct sampler reduces the estimation variance for small groups, when the size of dimension tables becomes very large, the distinct sampler would keep almost every record when stratifying the join columns. Therefore, even for a few many-to-one join situations, we prefer to use the correlated sampling algorithm.

[2]As suggested by [26], we do not consider sampling rate p above 0.1 for higher performance gain.

Correlated sampling method [50] first picks a hash function $h(\cdot)$, which maps the join values to a uniform distribution within range [0,1]. For instance, $h(v) = ((kv + b) \bmod \mathsf{p})/\mathsf{p}$, where $k, b \in [1, \mathsf{p})$, and p is a large prime number [29]. Suppose the sampling probability is p in the correlated sampling method. Similarly, we consider p range from 0.1 to 10^{-5}. For each table, if the join value v_i for row i is smaller than p after the hash mapping, i.e., $h(v_i) < p$, this row would be included in our samples. The unbiased estimated join size for the samples would be $\widehat{J} = J'/p$, where J' is the join size for the sampled set. To consider the coverage of data patterns over attribute of interest set $\mathcal{A}$, the estimated COUNT aggregation is essentially the estimated join size subject to a set of predicates of $\mathcal{A}$. When we apply WHERE clause restrictions $a_j \in \mathcal{A}$ to table T_j, the estimated aggregation result would be $\widehat{J}^{(\mathcal{A})} = J'^{(\mathcal{A})}/p$. The variance for correlated sampling method is given by [50]:

$$Var(\widehat{J}) = \left(\frac{1}{p} - 1\right) \sum_v F_1(v)^2 F_2(v)^2 \qquad (5)$$

Where $F_1(v)$ and $F_2(v)$ are the frequency of a join value appears in the two tables. The variance after applying filters in A becomes smaller as we replace $F_j(v)$ with $F_j^{(a_j)}(v)$.

6. EXPERIMENTAL EVALUATION

To validate the efficiency of the proposed methods, we conducted experiments on a Linux machine with a 3.8 GHz Intel Xeon processor and 64GB memory. The bit array construction and intersection operation are implemented in C++17, using CRoaring [30] Library, one of the most performant bitmap compression methods [51]. The index construction process is done via parallel client-end queries to any general-purpose database systems. In this paper, the datasets for experiments are stored in PostgreSQL 12.2. The *P-WALK* algorithm is implemented in Java. *P-WALK* calls the C++ functions to check the pattern coverage for each visiting node individually.

In the coverage checking process, we rank all bit arrays according to their cardinality, which reflects predicate selectivity and perform pairwise AND operation of all bit arrays starting from the most-filtering predicate. We classify a pattern as uncovered once the cardinality of the intermediate result is less than the coverage threshold.

Our experiments are conducted as follows. (1) We analyze the index construction time and memory consumption of our method. (2) For pattern coverage assessment to compute the COUNT result for a given data pattern, we compare the runtime of index-based methods (Sec. 4.1) with read-optimized DBMS solutions, under varying data size and data pattern level. (3) For MUP identification algorithms, we compare *P-WALK* (Sec. 4.2) with three baseline single table algorithms: *Pattern Breaker, Pattern Combiner*, and *Deep Diver* proposed in [6]. We vary the coverage threshold and compare the number of COUNT operations needed by different algorithms. (4) For the exact MUP identification runtime, we compare our method (Sec. 4) with the three baseline traversal algorithms using group-by and conjunctive COUNT queries to determine the pattern coverage. (5) For the approximate coverage analysis method (Sec. 5), we evaluate the MUP identification accuracy of the distinct sampler and the correlated sampling method and compare the performance with the Bernoulli (naïve) sampler. In addition, we analyze the time efficiency of sampling-based methods.

6.1 Dataset Description

We use three real-world datasets for evaluations:

- *IMDB*[3]: IMDB is a real-world database containing facts related to movies. It is widely used in many data analysis and machine learning tasks. The data is publicly available on the IMDB website. We use the *imdbpy*[4] tool to transform the text files into a relational database containing 21 tables and 14.1GB data. The two largest tables `cast_info` and `movie_info` contain 58,892,067 and 24,131,800 records, respectively. We ignored text attributes like names, notes, and attributes used as checksums or join keys. We include all other attributes distributed in 6 different tables for coverage analysis. Continuous attributes like `production_year` are grouped in buckets, each ten years wide.
- *UK Road Safety*[5]: UK Road Safety is a detailed dataset released by the UK government of more than 2 million road accidents and involved vehicles in the United Kingdom from 2005 to 2017. It contains more than 2 million records about the road and weather conditions of accidents, demographic information of drivers and vehicle information. We used 8 attributes about traffic accidents with cardinalities ranging from 3 to 10, namely `light_condition`, `weather_condition`, `road_type`, `day_of_week`, `speed_limit`, `urban_rural`, `severity`, `surface_condition`, and 4 attributes about drivers, namely `driver_age`, `driver_sex`, `driver_home`, `vehicle_age`, with cardinalities ranging from 4 to 12 for studying the coverage in the database.
- *SF Bay Area Bike Share*[6]: SF Bay Area Bike Share is the dataset of Bay Wheels, a bicycle sharing system in San Francisco. It records information like weather, the number of available docks and bikes in every minute, the location and construction time of each station, and bike trips among these stations. The dataset can be used to investigate trends such as how weather influences bike trips; how bike trips patterns differ by the time of day and the time of week. We chose attributes including `location`, `temperature`, `humidity`, `visibility`, `wind_speed`, `cloud_cover`, and `availability_of_docks_and_bikes` in each station for coverage analysis.

6.2 Performance Evaluation

6.2.1 Index Construction Overhead

Our method not only works when the indexes already exist. The join table and inverted indexes can also be constructed and maintained on the fly. In Table 2, for each database, we report the number of tables $|\mathsf{T}|$ and the number of records for join tables $|\Join_{\mathsf{T}}|$. We compare the memory consumption of all index structures with the size of databases and the join tables containing only the "attributes of interest". We note that the sizes of indexes are small because in the bit-mapped representation, each row is only related to several bits indicating whether it satisfies the predicate restrictions. In addition, we report the join index creation time and the total time to construct all bit-mapped arrays in parallel for the set of "attributes of interest".

[3]ftp://ftp.funet.fi/pub/mirrors/ftp.imdb.com/pub/frozendata/
[4]https://imdbpy.readthedocs.io/en/latest/usage/ptdf.html
[5]https://www.kaggle.com/tsiaras/uk-road-safety-accidents-and-vehicles
[6]https://www.kaggle.com/benhamner/sf-bay-area-bike-share

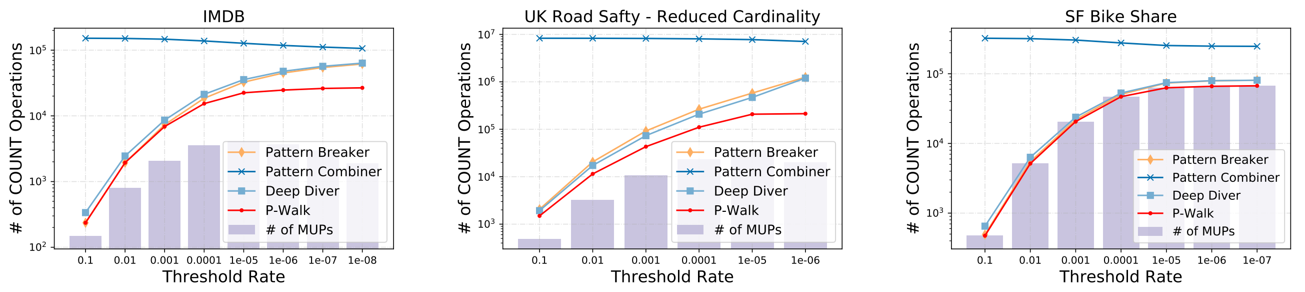


Figure 7: Number of COUNT Operations for Searching Algorithms

Table 2: Index Construction Overhead for Datasets: *IMDB* (I), *UK Road Safety*(U), *SF Bay Area Bike Share* (S)

Dataset	I	U	S
# of tables, $\|T\|$	6	2	3
# of records, $\|\bowtie_T\|$	135.3M	2.1M	62.6M
Database Size(GB)	14.1	1.26	4.45
Join Table Size(GB)	9.57	0.12	0.85
Join Index Size	1.6GB	16.5MB	0.15GB
Inverted Indexes Memory Consumption	0.44GB	7.36MB	0.14GB
Join Index Creation Time(s)	18.38	8.47	10.80
Total Inverted Indexes Creation Time(s)	17.19	6.05	21.53

6.2.2 *Exact Coverage Analysis*

(1) Exact - Data Pattern Coverage Assessment. Determining the coverage of a data pattern is in fact to compute the COUNT result in the join table $\bowtie_T$ for the corresponding group. We compare the conjunctive COUNT query execution time of our index-based solution with the following baseline methods: (i) database query in PostgreSQL (ii) optimized query based on PostgreSQL materialized views (iii) database query in column-based systems MonetDB [10] and HyPer [37]. We use the largest *IMDB* dataset for this experiment. The query execution time is highly related to the data size, the number of bound attributes in the data pattern and the selectivity of WHERE clause predicates. To reduce the variance from WHERE clause selectivity, we randomly generate predicate values for data patterns and record the average execution time. In Fig. 8, the y-axis shows the runtime of data pattern coverage assessment in milliseconds. We first fix the layer of data patterns to be 6 (same as $|\mathcal{A}|$, the dimension of the attribute set) and compare the execution time of different methods under varying data size (Fig. 8, left). We observe that the query execution time of materialized views swells because of the lack of index structures. Compared with the baseline methods, our bit-mapped solution scales well as the data size grows. We note that the runtime of bit-mapped solution is close to zero because of the highly parallel in-memory execution.

Next, we fix the data size as the original database and vary the number of bound attributes in the data pattern. The experimental result also exhibits the efficiency and robustness of our method (Fig. 8, right). In this experiment, we observe a surge of running time for some baseline methods when the data pattern is in the upper layer. This is because the indexes on original tables fail to increase the computation efficiency with fewer predicate restrictions.

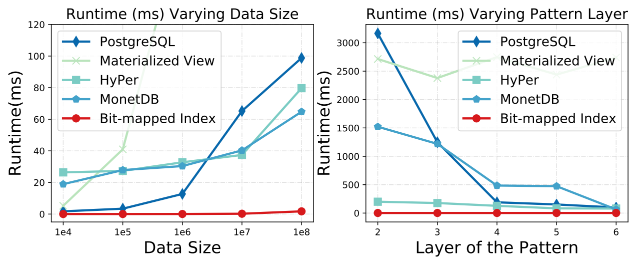


Figure 8: Runtime (ms) for Data Pattern Coverage Assessment (IMDB)

(2) Exact - MUP Identification. We profiled the MUP identification process and demonstrated that the most time-consuming operation is the COUNT operation to determine the coverage of a data pattern. To validate the pruning and searching efficiency of *P-WALK*, we compare the number of COUNT needed for all searching algorithms. The number of COUNT operations is affected by the distribution of MUPs at each level, and MUP distribution is, in turn, affected by the value of the coverage threshold τ. In our evaluation, we choose the value of τ to be dependent on $|\bowtie_T|$, the size of the join table. We define t as the threshold rate, where $t = \tau / |\bowtie_T|$. We vary τ from $0.1 \times |\bowtie_T|$, where most of the patterns are uncovered, to 1, where most of the data patterns are covered. In Fig. 7, the bars show the number of MUPs as the coverage threshold rate varies. As the threshold rate decreases, for the bottom-up *Pattern Combiner* algorithm, the number of COUNT operations decreases. On the contrary, the number of COUNT increases for the other algorithms. This phenomenon can be explained by the "moving down" tendency of MUPs (will be discussed in Sec. 6.2.4) when we define smaller coverage thresholds. We find that *P-WALK* algorithm outperforms the others in all scenarios. In the *UKRoad* dataset with low threshold rate t, the number of COUNT operations of *P-WALK* is more than 60 times less than the others. In addition, the growing tendency of COUNT operations of *P-WALK* also indicates the same trend as the MUP number variation tendency, which means the priority search heuristically approaches the path with higher pruning power.

In Fig. 9, we report the running time for MUP identification algorithms under varying thresholds. The time overhead of MUP identification is affected by the number of COUNT operations and the efficiency of COUNT execution. For baseline experiments, we choose the three baseline searching algorithms discussed above to prune the lattice space. To determine the coverage of data patterns, we found that relying on conjunctive COUNT query to determine the pattern coverage is prohibitive even with read-optimized techniques (e.g., materialized views, column-store, indexes

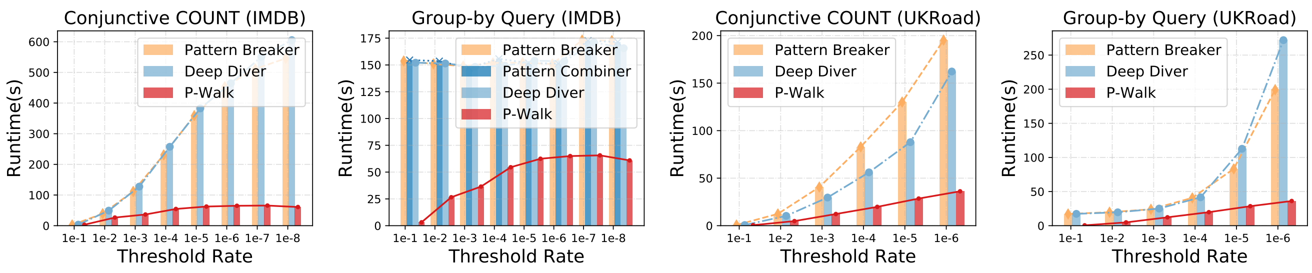


Figure 9: Runtime (ms) for MUP Identification

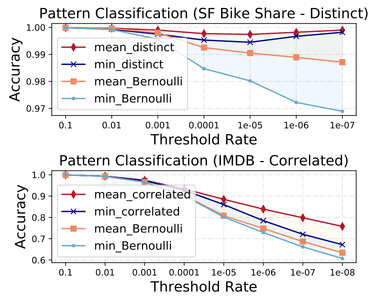


Figure 10: Pattern Classification

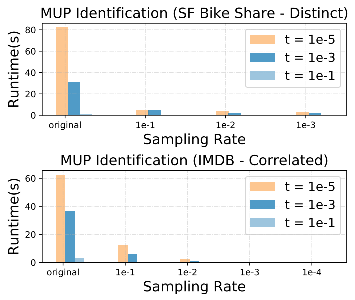


Figure 11: Approximate Efficiency

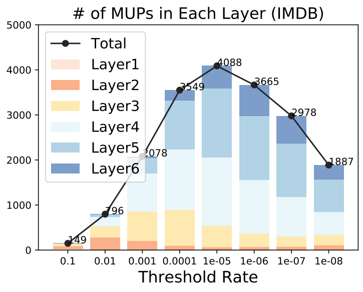


Figure 12: MUP Distribution

on original tables). Therefore, we constructed bitmaps for the join table in memory to optimize the conjunctive COUNT execution. For the baseline relying on group-by queries, we observed that the time to add up leaf pattern counts is prohibitive and used the inverted index in [6] to further optimize it. As is shown in Fig. 9, for the largest *IMDB* dataset, the time to execute the conjunctive COUNT query bursts as the number of COUNT operation needed for MUP identification increases. *Pattern Combiner* algorithm fails in *IMDB* dataset because of the long computation time for all leaf nodes. The performance of the group-by solution for the *IMDB* dataset is acceptable, but for the *UKRoad* dataset where the number of nodes in the pattern graph is large, the execution time for group-by solution bursts. The bottom-up *Pattern Combiner* fails for the *UKRoad* dataset because the large number of leaf nodes causes an OutOfMemoryError.

6.2.3 Approximate Method

(3) Approximate - MUP Identification and Data Pattern Classification Accuracy. We follow the insertion criteria in Sec. 5 to insert distinct sampler for many-to-one joins and apply correlated sampling method for many-to-many joins for the two largest datasets: *SF Bike Share* and *IMDB*, with more than 62 million and 135 million records, respectively. For the correlated sampling method, we consider the sampling rate p_c ranging from 0.1 to 10^{-5}. For the distinct sampler, we consider the sampling rate p_d from 0.1 to 10^{-3}, because we observe that due to the restriction of stratification, the sample size does not change much when $p < 10^{-3}$.

Table 3 shows the performance of the distinct sampler and the correlated sampling method. We consider the MUP identification accuracy under varying sampling rates and coverage threshold rates ($t = \tau/|\times_{\mathrm{T}}|$). As expected, we observe that the accuracy of MUP identification decreases as the sampling rate decreases. For better illustration, we show the minimum and average accuracy of different sample sizes for each given threshold rate. As is shown in the result, even with small samples, the accuracy of identifying MUPs is always higher than 96%. There is a slight decrease in accuracy when the threshold causes an increase of MUPs in

Table 3: Accuracy of MUP Identification

	Distinct sampling *SF Bike Share*		Correlated Sampling *IMDB*	
Threshold	min(acc)	avg(acc)	min(acc)	avg(acc)
$t = 10^{-1}$	99.99%	100%	99.99%	99.99%
$t = 10^{-2}$	99.93%	99.97%	99.44%	99.51%
$t = 10^{-3}$	99.74%	99.99%	98.45%	98.60%
$t = 10^{-4}$	99.52%	99.89%	97.25%	97.53%
$t = 10^{-5}$	99.45%	99.73%	96.62%	96.97%
$t = 10^{-6}$	99.67%	99.81%	96.78%	97.11%
$t = 10^{-7}$	99.82%	99.89%	97.02%	97.11%
$t = 10^{-8}$	-	-	97.63%	98.13%

middle layers. Although the accuracy of the correlated sampling method is slightly lower, it successfully reduces data size to 10^{-5} of the original data.

To further evaluate the effectiveness of the samplers, we regard the pattern coverage analysis task as a binary classification problem. For each pattern, based on the estimated COUNT result of samples, we classify it as a covered or an uncovered pattern. Figure 10 shows the minimum and average classification accuracy over different sampling rates of the distinct sampler, the correlated sampling method, and the baseline Bernoulli sampling. We observe that both the distinct sampler and the correlated sampling method outperform Bernoulli sampling. The distinct sampler maintains high accuracy as the coverage threshold decreases, which indicates the distinct sampler improves the estimation accuracy for small groups. The discrepancy in accuracy between Bernoulli sampling and correlated sampling can be easily explained by the performance of sample-then-join of these two methods. We find that the correlated sampling method has a good approximation to join-then-sample, while Bernoulli sampling outputs a much smaller join size.

(4) Approximate - MUP Identification Efficiency. To validate the time efficiency of sampling method, Fig. 11 shows the running time of approximate MUP identification of distinct sampler and correlated sampling method under varying sampling rates (x-axis) and coverage threshold rates (labels) on the two largest datasets: *IMDB* and *SF Bike Share*. We

observe a drastic decrease in running time when the data size shrinks to the tenth of the original size, under which a high MUP identification accuracy is maintained. Based on this result, we suggest sampling rates below $p = 0.1$ and considering the trade-off between accuracy and efficiency.

6.2.4 MUP Analysis

The distribution of MUPs differs across datasets under different coverage thresholds. This variation affects the performance of search algorithms and the accuracy of the sampling methods. Figure 12 shows the MUP distribution in different layers with varying threshold rates. In all three datasets, we observe that for a given threshold, most MUPs appear in the middle layers. As the coverage threshold decreases, the percentage of MUPs that are closer to the bottom layer increases, while the total number of MUPs has a "bell-curve" shape as the threshold rate varies. The "moving-down" tendency of MUPs with the decrease of threshold explains the reason why top-down MUP identification algorithms have an increasing number of COUNT operations while the bottom-up algorithms have the opposite tendency as the threshold goes down.

6.3 Setting Threshold Value

The algorithms we have developed work for any specified threshold value. The user can set the threshold as a certain percentage of the total number of records, or the user can refer to the number of predicate restrictions to tailor different thresholds for groups in different granularity. For a user with no special knowledge, we recommend they use the rule-of-thumb statistic in the central limit theorem. For example, Sudman [48] suggests that each "minority subgroup" requiring a minimum of 20-50 samples.

For the motivating example in Sec. 1 to predict the risks of drivers, using this rule-of-thumb, we set the coverage threshold as $\tau = 30$. Below, we interpret some representative MUPs founded by our proposed method. For value combinations in the first layer, which only consider one attribute, we observe that with $\tau = 30$, there are no MUPs in this layer. In the second layer $\ell = 2$, we found that there is not enough data for drivers with `vehicle_age` > 30 having accidents in `weekdays` or drivers `age` < 65 having `vehicle_age` > 30. When $\ell = 3$, we found that there are not enough records for `female` from the `rural area` involved in accidents in `sunny` days. Besides, the dataset contains not enough instances for `male` drivers `aged over 65` who are involved in `fatal` accidents. Our coverage analysis provides the dataset user all maximal empty spaces not covered adequately by the data. Without the traversal algorithm and the index structures, the execution time would be prohibitive as the size of the database and the set of value combinations grows.

7. RELATED WORK

The diversity and representativeness of data collection are essential topics for algorithmic fairness in data science, and have been widely discussed in fields such as sociology [9, 15], biology [24], information retrieval [3], and data ethics [7, 16]. Many efforts focus on identifying inadequate data coverage in datasets. [12] analyzes the training set coverage of the protected attribute contributing to machine learning discrimination. There are also data mining works [33, 32, 34] that aim to identify the largest empty regions in datasets. [6] defines the coverage over multiple categorical attributes and developed techniques for identifying empty spots in datasets. However, all these methods are limited to single table scenarios and never consider the increasing time complexity as the data volume grows.

Relational databases typically distribute the information across multiple tables. Joins are frequently required, and expensive. Optimization engines like [47] proposes methods to compute large batches of aggregate queries efficiently. In our proposed method, we use a variant of join indexes [49, 40, 41] proposed to materialized these joins of tables.

The traversal of the powerset lattice has been studied in different contexts, such as frequent item-set mining [4], combinatorial set enumeration [46]. Functional dependency (FD) discovery [42] algorithms apply various traversal strategies to enumerate FD candidates. [23, 38, 53] are level-wise bottom-up algorithms using the apriori generation method. DFD [1] applies a depth-first random walk for FD candidate generation. However, FD discovery algorithms only enumerate attribute combinations instead of value combinations. In the pattern graph for coverage analysis, the number of neighbors and the pruning efficiency is different for nodes even in the same layer. The priority search we proposed heuristically prefers the nodes with higher pruning efficiency. In addition, recent work on data slicing [13] defines a partial ordering to enumerate the lattice of attribute values for discovering the top-k data slices where the trained model performs worse. However, it cannot be applied to MUP identification since it only cares about the top-k slices instead of pruning and exploring the entire lattice space.

For approximate coverage analysis, techniques include histogram [43], wavelet [14], online aggregation [22], etc. The first two methods fail to consider the coefficient of different attributes, while online aggregation lacks an efficient way to reduce the variance for small groups [31]. Therefore, we adopt query samplers [26, 2, 28], and correlation-aware sampling techniques [50] to reduce the data size. We would like to distinguish our work from the aggregate query optimization [35], because the coverage analysis explores the combinatorial set of value combinations for data resides in systems which might not be optimized for this purpose.

8. FINAL REMARKS

In this paper, we address the problem of database coverage analysis with multiple relations – computing information that data scientists ought to consider prior to engaging in any data analysis and data users ought to consider before accepting analysis results. Given a fixed coverage threshold and a set of categorical attributes across tables, we propose index techniques for efficient data pattern coverage analysis; we also develop a lattice-based priority search to identify all maximal empty regions. Since it is usually sufficient to obtain an approximate assessment of coverage, we develop approximate methods for further improvement. We hope that the efficient algorithms we develop in this paper will remove a barrier to adoption of coverage analysis as a necessary data pre-processing step before data analysis, just as data cleaning is a necessary pre-processing step. In addition, the maintenance of the index structures in face of database updates is a potential topic for future work.

Acknowledgements. We would like to thank the anonymous reviewers for their insightful comments and suggestions. We would like to thank Shengpu Tang for his helpful comments. This research is supported in part by NSF under grants 1741022 and 1934565.

9. REFERENCES

[1] Z. Abedjan, P. Schulze, and F. Naumann. Dfd: Efficient functional dependency discovery. In *CIKM*, pages 949–958. ACM, 2014.

[2] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *SIGMOD*, pages 487–498. ACM, 2000.

[3] R. Agrawal, S. Gollapudi, A. Halverson, and S. Ieong. Diversifying search results. In *WSDM*, pages 5–14. ACM, 2009.

[4] R. Agrawal, R. Srikant, et al. Fast algorithms for mining association rules. In *VLDB*, volume 1215, pages 487–499, 1994.

[5] M. J. Alves and C. V. Willie. Controlled choice assignments: A new and more effective approach to school desegregation. *The Urban Review*, 19(2):67–88, 1987.

[6] A. Asudeh, Z. Jin, and H. Jagadish. Assessing and remedying coverage for a given dataset. In *ICDE*, pages 554–565. IEEE, 2019.

[7] S. Barocas and A. D. Selbst. Big data's disparate impact. *Calif. L. Rev.*, 104:671, 2016.

[8] E. J. Benjamin, P. Muntner, and M. S. Bittencourt. Heart disease and stroke statistics-2019 update: a report from the american heart association. *Circulation*, 139(10):e56–e528, 2019.

[9] E. Berrey. *The enigma of diversity: The language of race and the limits of racial justice*. University of Chicago Press, 2015.

[10] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *Cidr*, volume 5, pages 225–237, 2005.

[11] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *SIGMOD*, volume 28, pages 263–274. ACM, 1999.

[12] I. Chen, F. D. Johansson, and D. Sontag. Why is my classifier discriminatory? In *NeurIPS*, pages 3539–3550, 2018.

[13] Y. Chung, T. Kraska, N. Polyzotis, K. H. Tae, and S. E. Whang. Slice finder: Automated data slicing for model validation. In *ICDE*, pages 1550–1553. IEEE, 2019.

[14] G. Cormode, M. Garofalakis, P. J. Haas, C. Jermaine, et al. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trend in Databases*, 4(1–3):1–294, 2011.

[15] F. Dobbin and A. Kalev. Why diversity programs fail and what works better. *Harvard Business Review*, 94(7-8):52–60, 2016.

[16] M. Drosou, H. Jagadish, E. Pitoura, and J. Stoyanovich. Diversity in big data: A review. *Big data*, 5(2):73–84, 2017.

[17] C. Dwork and C. Ilvento. Group fairness under composition. In *FAT**. ACM, 2018.

[18] K. M. Egan, D. Trichopoulos, M. Stampfer, W. Willett, P. Newcomb, A. Trentham-Dietz, M. Longnecker, and J. Baron. Jewish religion and risk of breast cancer. *The Lancet*, 347(9016):1645–1646, 1996.

[19] S. Feijo. Here's what happened when boston tried to assign students good schools clase to home. 2018.

[20] J. Foulds and S. Pan. An intersectional definition of fairness. *arXiv preprint arXiv:1807.08362*, 2018.

[21] B. Garrett and J. Monahan. Assessing risk: The use of risk assessment in sentencing. *Judicature*, 103:42, 2019.

[22] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, pages 171–182. ACM, 1997.

[23] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal*, 42(2):100–111, 1999.

[24] L. Jost et al. Mismeasuring biological diversity: response to hoffmann and hoffmann (2008). *Ecological Economics*, 68(4):925–928, 2009.

[25] S. M. Julia Angwin, Jeff Larson and L. Kirchner. Machine bias. 2016.

[26] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *SIGMOD*, pages 631–646. ACM, 2016.

[27] M. Kearns, S. Neel, A. Roth, and Z. S. Wu. An empirical study of rich subgroup fairness for machine learning. In *FAT**, pages 100–109. ACM, 2019.

[28] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann. Cardinality estimation done right: Index-based join sampling. In *CIDR*, 2017.

[29] D. Lemire and O. Kaser. Strongly universal string hashing is fast. *The Computer Journal*, 57(11):1624–1638, 2014.

[30] D. Lemire, O. Kaser, N. Kurz, L. Deri, C. O'Hara, F. Saint-Jacques, and G. Ssi-Yan-Kai. Roaring bitmaps: Implementation of an optimized software library. *Software: Practice and Experience*, 48(4):867–895, 2018.

[31] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join: Online aggregation via random walks. In *SIGMOD*, pages 615–629. ACM, 2016.

[32] B. Liu, W. Hsu, and S. Chen. Using general impressions to analyze discovered classification rules. In *KDD*, pages 31–36, 1997.

[33] B. Liu, L.-P. Ku, and W. Hsu. Discovering interesting holes in data. In *IJCAI*, pages 930–935. Springer, 1997.

[34] E. Loekito and J. Bailey. Fast mining of high dimensional expressive contrast patterns using zero-suppressed binary decision diagrams. In *SIGKDD*, pages 307–316. ACM, 2006.

[35] G. Moerkotte and T. Neumann. Accelerating queries with group-by and join by groupjoin. *PVLDB*, 4(11), 2011.

[36] M. S. Moore, A. Bocour, L. Jordan, E. McGibbon, J. K. Varma, A. Winters, and F. Laraque. Development and validation of surveillance-based algorithms to estimate hepatitis c treatment and cure in new york city. *Journal of Public Health Management and Practice*, 24(6):526–532, 2018.

[37] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *SIGMOD*, pages 677–689. ACM, 2015.

[38] N. Novelli and R. Cicchetti. Fun: An efficient algorithm for mining functional and embedded dependencies. In *ICDT*, pages 189–203. Springer, 2001.

[39] A. Olteanu, C. Castillo, F. Diaz, and E. Kiciman. Social data: Biases, methodological pitfalls, and ethical boundaries. *Frontiers in Big Data*, 2:13, 2019.

[40] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *ACM SIGMOD Record*, 24(3):8–11, 1995.

[41] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD*, pages 38–49. ACM, 1997.

[42] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J.-P. Rudolph, M. Schönberg, J. Zwiener, and F. Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *PVLDB*, 8(10):1082–1093, 2015.

[43] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB*, volume 97, pages 486–495, 1997.

[44] R. Richardson, J. Schultz, and K. Crawford. Dirty data, bad predictions: How civil rights violations impact police data, predictive policing systems, and justice. *New York University Law Review Online, Forthcoming*, 2019.

[45] D. D. Rutstein, R. J. Mullan, T. M. Frazier, W. E. Halperin, J. M. Melius, and J. P. Sestito. Sentinel health events (occupational): a basis for physician recognition and public health surveillance. *American journal of public health*, 73(9):1054–1062, 1983.

[46] R. Rymon. Search through systematic set enumeration. *Technical report, University of Pennsylvania*, 1992.

[47] M. Schleich, D. Olteanu, M. Abo Khamis, H. Q. Ngo, and X. Nguyen. A layered aggregate engine for analytics workloads. In *SIGMOD*, pages 1642–1659. ACM, 2019.

[48] S. Sudman. Applied sampling. Technical report, Academic Press New York, 1976.

[49] P. Valduriez. Join indices. *TODS*, 12(2):218–246, 1987.

[50] D. Vengerov, A. C. Menck, M. Zait, and S. P. Chakkappen. Join size estimation subject to filter conditions. *PVLDB*, 8(12):1530–1541, 2015.

[51] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson. An experimental study of bitmap compression vs. inverted list compression. In *SIGMOD*, pages 993–1008. ACM, 2017.

[52] R. C. Whitaker, J. A. Wright, M. S. Pepe, K. D. Seidel, and W. H. Dietz. Predicting obesity in young adulthood from childhood and parental obesity. *New England journal of medicine*, 337(13):869–873, 1997.

[53] H. Yao, H. Hamilton, and C. Butz. Fd_mine: Discovering functional dependencies in a database using equivalences, canada. In *ICDM*, pages 1–15. IEEE, 2002.