Discovering Approximate Inclusion Dependencies

Qingdong Su
School of Computer Science, Fudan University, China
qdsu22@m.fudan.edu.cn

Zhikang Wang
School of Computer Science, Fudan University, China
zkwang22@m.fudan.edu.cn

Zijing Tan
School of Computer Science, Fudan University, China
Shanghai Key Laboratory of Data Science
zjtan@fudan.edu.cn

Shuai Ma
SKLSDE Lab, Beihang University, China
mashuai@buaa.edu.cn

ABSTRACT

Inclusion dependencies (INDs) are widely used in data management tasks. The discovery techniques of INDs have thus received a lot of attention, for discovering INDs valid in data. However, real-world data quality issues may lead to partial violations of INDs. This paper makes the first effort to provide a comprehensive study on the discovery of approximate INDs (AINDs), aiming to identify INDs with error rates below a given threshold. This paper introduces a new definition of AIND based on deletion semantics, in addition to the existing definition based on insertion semantics. A discovery method is developed that can be configured to identify AINDs based on either of these semantics. The method combines partitioning techniques to handle tables that cannot all fit into memory simultaneously, with novel approaches to quantify AIND violations based on partitioned tables. To improve efficiency, the method employs a novel three-layer filtering structure and techniques that can potentially prune invalid candidate AINDs and identify valid AINDs without necessarily processing all tuples. We conduct an extensive experimental evaluation and verify the following: the proposed method significantly outperforms existing methods for AIND discovery based on insertion semantics, the AIND discoveries with insertion and deletion semantics can provide complementary results, and our discovery method can effectively deal with dirty dataset containing various types of errors.

PVLDB Reference Format:
Qingdong Su, Zhikang Wang, Zijing Tan, and Shuai Ma. Discovering Approximate Inclusion Dependencies. PVLDB, 18(4): 1210 - 1222, 2024.
doi:10.14778/3717755.3717777

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/A-IND/AINDD.

1 INTRODUCTION

Understanding the relationships between data is crucial for effectively utilizing the data, while these relationships are typically hidden in the data and are too costly to be manually defined. To this end, data profiling techniques [1, 2] have been extensively studied for automatically finding various data relationships from data. The goal of this work is to present an efficient solution to the discovery of approximate inclusion dependencies (AINDs). We first briefly review the definition of inclusion dependencies (INDs), and then highlight the advantage of AINDs compared to (exact) INDs.

Table 1: Student Information

	Name	Department	Origin
t_1	Alice Smith	Chemical Science	California
t_2	Bob Johnson	Electrical Engineering	California
t_3	Charlie Brown	Mechanical Engineering	California
t_4	David Williams	Environmental Engineering	Texas
t_5	Emily Davis	Computer Engineering	Taxas
t_6	Frank Miller	Computer Engineering	Texas
t_7	Grace Martinez	Artificial Intelligence	Texas
t_8	Henry Wilson	Artificial Intelligence	Texas
t_9	Ivy Clark	Aerospace Engineering	Texas

Table 2: Department Information

	Department	Number
t_1	Computer Science	200
t_2	Electrical Engineering	150
t_3	Mechanical Engineering	180
t_4	Chemical Engineering	160
t_5	Environmental Engineering	130
t_6	Aerospace Engineering	120
t_7	Architecture	110

IND is one of the most well known types of data dependency and has found applications in many data management tasks, such as schema design [3, 30], table joins [13, 24, 51], data integration [17, 18, 35] and query optimization [16, 26], among others. In brief, a unary IND is in the form of $R_1.A \subseteq R_2.B$, where R_1 and R_2 are two relational schemas, and A (resp. B) is an attribute of R_1 (resp. R_2). This IND is satisfied by an instance r_1 of R_1 and an instance r_2 of R_2, if for every tuple t of r_1, there exists a tuple s of r_2 such that $t[A] = s[B]$. This definition is based on the assumption of a clean and complete dataset, while data in practice often contain errors due to inaccuracies, inconsistencies or incompleteness [14, 19]. Dirty data hinders the application of exact INDs, as illustrated below.

Example 1: Consider Table 1 with the student information and Table 2 with the department information of a university. It is expected that all the values in the attribute Department of Table 1 are present in the attribute Department of Table 2, as every student should belong to a department. This relationship can be specified as an IND and suggest joining the two tables to find interesting facts.

Proceedings of the VLDB Endowment, Vol. 18, No. 4 ISSN 2150-8097.
doi:10.14778/3717755.3717777

Table 3: State Information

	State	Postal Code
t_1	California	CA
t_2	Texas	TX
t_3	West Virginia	WV
t_4	Wisconsin	WI
t_5	Wyoming	WY

However, the values in the attribute Department of tuples t_7 and t_8 in Table 1 are absent from Table 2, thereby violating the IND. As another example, all the values in the attribute Origin of Table 1 are expected to appear in the attribute State of Table 3, but this IND also fails due to a spelling error in tuple t_5 of Table 1. □

To recover INDs so as to facilitate table joins, we need relaxations in the satisfaction of INDs, which motivates the notion of AINDs that can hold with exceptions. Some criteria are needed to measure the error rates of AINDs, and an AIND is considered as valid if its error rate is below a given threshold. One notion of AINDs, *a.k.a.* partial INDs, is defined based on *insertion semantics* [5, 33]. Consider the IND involving the attributes Department of Table 1 and Table 2. The satisfaction of it can be recovered by adding one tuple to Table 2 with the value of "Artificial Intelligence" in attribute Department. As "Artificial Intelligence" is one of the seven distinct values in the attribute Department of Table 1, the *error rate* of this IND is 1/7 based on the AIND definition under insertion semantics. This rate is calculated as the ratio of the number of distinct values absent from Table 2 to the total number of distinct values in the attribute Department of Table 1. Now, consider the IND across Table 1 and Table 3. The error rate in this case is 1/3, as there are three distinct values in the left-hand-side (LHS) attribute Origin, and one of those values does not appear on the right-hand-side (RHS) attribute State.

As a complement to insertion semantics, we propose a new AIND definition based on *deletion semantics*. The core idea is that, to eliminate the impact of data errors, we can not only add values to the RHS attribute of an IND but also remove erroneous values from the LHS attribute. Consequently, the error rate of an IND is calculated as the proportion of tuples whose values in the LHS attribute of the IND are absent from all the values in the RHS attribute of the IND. In contrast to the insertion semantics, the deletion semantics takes into account the occurrence frequencies of values. Under the deletion semantics, the error rate of the second IND is calculated as 1/9. The insertion semantics measures violations from the perspective of the number of distinct values, while the deletion semantics approaches the problem from the perspective of the number of tuples. They offer complementary options for users in AIND discovery.

The advantages of AINDs make them desirable in many applications. In data lake environments, due to the fact that tables often come from different data sources, data quality issues are very common, very likely leading to the invalidity of exact INDs. The existence of AINDs between tables can imply the possibility of table join operations, which is crucial for data analysts [10, 12, 13, 24, 50, 51]. The vast scale of tables in data lake scenarios and the lack of standardized attribute naming also highlight the quest for automatic discovery techniques for AINDs. The problem of AIND discovery is necessarily challenging, given that exact IND discovery is already known to be a tough problem [7]. To our best knowledge, existing methods for AIND discovery [5, 33] are simple extensions of the methods for exact IND discovery, and only consider AINDs based on insertion semantics.

Contributions & Organizations. We make the first effort to give a comprehensive study on the discovery problem of AINDs.

(1) A new definition of AIND (Section 3). We give a new definition of AIND based on *deletion semantics*, as a complement to the existing definition based on *insertion semantics*.

(2) AIND Discovery method (Sections 4 and 5). We provide an algorithm that can be configured to discover AINDs based on either of the semantics, equipped with a set of novel techniques: (a) the combination of partitioning techniques, to quantify AIND violations based on partitioned tables when there are many tables that cannot all fit into memory simultaneously; (b) a three-layer filtering structure to first estimate and finally compute numbers of AIND violations; and (c) an enumeration method for finding all valid AINDs, which has an early termination feature, potentially determining the validity or invalidity of candidate AINDs before accessing all tuples.

(3) Experimental study (Section 6). We integrate our algorithm into the Metanome data profiling platform [37] and perform an in-depth experimental evaluation. Our method significantly outperforms existing methods for discovering AINDs based on insertion semantics. AIND discoveries utilizing insertion and deletion semantics can offer complementary results. Our discovery method can effectively handle dirty datasets containing various types of errors.

2 RELATED WORK

IND is known as one of the most important types of data dependency, and hence, discovery techniques for IND and its variants have drawn much attention. We investigate related work in this section.

Discovery of Exact INDs. Exact IND discovery methods have a long history. These algorithms differ not only in the indexing techniques for validating INDs (pruning non-INDs), but also in the scheduling methods for data between memory and disk. This is because IND discovery methods are usually faced with a huge number of tables that are too large to be loaded into memory simultaneously, a departure from discovery methods of other dependencies.

An early IND discovery algorithm, called as Demarchi, is proposed in [33]. It builds an inverted index of the entire data set at once and does not scale with large datasets. The algorithm SPIDER [6] first collects attribute values into files and then compares the values to prune non-INDs in a similar spirit to merge sort. SPIDER usually suffers from the limitation of opening too many files at the same time. The algorithm SINDD and its improvement SINDD++ are presented in [42] and [44]. They introduce the concept of attribute clustering to eliminate the number of redundant operations resulting from the direct computation of inverted indexes, and propose the idea of partitioning to address the scalability problem for large datasets. The algorithm BINDER [38] also adopts the idea of partitioning, and for each partition, calculates unary INDs by building an inverted index in a similar way to [6, 33]. Another algorithm, called MANY [47], is optimized for a large number of short input relations, *i.e.,* the number of data tables is very large while the number of tuples in each table is not large. It first generates unary candidate INDs by leveraging a Bloom filter, and then verifies each of them with intersection of sets based on memory hashing. Most

studies on IND discovery focus on unary INDs with only one LHS attribute and one RHS attribute, while there are also researches on the discovery of n-ary INDs that possibly have multiple attributes on the LHS and RHS [25, 33, 34, 36]. The techniques for discovering n-ary INDs usually assume that the set of unary INDs is known. Last but not least, an experimental evaluation of unary and n-ary IND discovery methods has been conducted in [11].

This work differs in the following. We discover unary AINDs that allow the existence of violations, and propose novel data structures and techniques. (a) We extend partitioning techniques to AIND discovery. It is highly non-trivial to quantify AIND violations in case of partitioned data, which significantly differs from checking the validity of exact INDs. (b) We design a three-layer filtering structure, which is first employed for a rough calculation to quickly discard invalid AINDs, and is then used to calculate the exact number of violations. (c) We introduce techniques to confirm AINDs as valid or invalid in early stages. An AIND can be verified to be valid before accessing all tuples, which is a unique characteristic that distinguishes AIND discovery significantly from exact IND discovery.

Discovery of approximate INDs. To our best knowledge, we are not aware of methods that are specifically designed for discovering AINDs. The methods of Demarchi [33] and SPIDER [5, 6] are extended to find AINDs based on insertion semantics. Our discovery method can be configured to find AINDs based on either insertion or deletion semantics, and significantly beats prior methods by employing a set of novel techniques introduced for AIND discovery.

Other studies on IND discovery. There are more studies on IND discovery, but their settings are significantly different from ours. [29] introduces an algorithm to find all unary INDs in a distributed setting, by partitioning tables and distributing computations across a map-reduce-style framework. Conditional INDs [9, 32] are proposed to state INDs that do not hold on the whole dataset but only on partial data specified by binding semantically related data values. A discovery method for conditional INDs is presented in [4]. Algorithm FAIDA [28] adopts a strategy that sacrifices correctness for efficiency in exact IND discovery; it guarantees completeness of the discovery result but may provide some false-positive INDs. Algorithms for maintaining the discovered INDs in response to data updates are studied in [43, 45]. The concept of temporal INDs and a discovery method are proposed in [8], aiming to find INDs that have consistently existed throughout the long-term history of Wikipedia tables. A recent study [22] introduces similarity inclusion dependencies (SINDs), which relax the requirement of equality to similarity in value comparison. SINDs can address minor errors at the character-level or token-level, but are not suitable for handling errors resulting from more complex causes. In Example 1, the notion of SIND can be used to tackle the error in "Taxas" by considering "Taxas" and "Texas" as the same, but cannot handle the data incompleteness problem that exists in Table 2. We contend that AINDs can treat different dirty data problems in a unified approach and are more general than SINDs. The discovery method of SINDs [22] mainly focuses on the integration of similarity indexes to speed up the calculation of value differences, while the key of AIND discovery is to quantify AIND violations. Consequently, novel techniques are required for efficiently discovering AINDs.

Discovery of other approximate dependencies. To deal with real-life dirty data, methods for discovering approximate dependencies have been studied for, *e.g.,* functional dependencies [27], denial constraints [31, 39, 49] and order dependencies [20, 21, 23, 46]. As opposed to INDs, these dependencies are violated by tuple pairs and only tuple deletions (but not insertions) can eliminate the violations. These distinctions give rise to entirely different discovery methods.

3 PRELIMINARIES

In this section, we present two definitions of AIND and formalize their discovery problems.

For two relational instances (tables) r_1 and r_2 of schema R_1 and R_2 respectively, we use t and s to denote tuples of r_1 and r_2, and A and B to represent attributes of R_1 and R_2, respectively. Specifically, tuples in r_1 are $t_1, t_2, \ldots, t_{|r_1|}$ and the set of attributes of R_1 is $\{A_1, A_2, \ldots, A_{|R_1|}\}$ respectively, where $|r_1|$ (resp. $|R_1|$) denotes the number of tuples (resp. attributes) of r_1 (resp. R_1). We denote by $t[A]$ the value of t in A. Similarly for r_2 and R_2.

Inclusion dependency. A unary inclusion dependency (IND) λ from R_1 to R_2 is defined as $R_1.A \subseteq R_2.B$. λ is satisfied by r_1 and r_2, iff $\forall t \in r_1$, $\exists s \in r_2$ such that $t[A] = s[B]$.

We now present the definitions of AIND, to tolerate real-life dirty data. In particular, we introduce a new definition of AIND based on deletion semantics, as a complement to existing definition of AIND based on insertion semantics [6, 33].

Approximate IND with the insertion semantics. We use $\mathsf{AIND_i}$ to represent approximate IND with the insertion semantics. An $\mathsf{AIND_i}$ is in the form of $R_1.A \subseteq_\epsilon^i R_2.B$, where ϵ is a given error threshold. It is satisfied by r_1 and r_2 if, in r_1, the proportion of distinct values $t[A]$ that are not present in attribute B of r_2 falls below the given threshold. Specifically, $R_1.A \subseteq_\epsilon^i R_2.B$ holds, iff

$$\frac{|\{t[A] \mid t \in r_1 \wedge \forall s \in r_2,\, s[B] \neq t[A]\}|}{|\pi_A(r_1)|} \leq \epsilon$$

In the formula, $|\pi_A(r_1)|$ represents the number of distinct values in attribute A, for the tuples of r_1.

Example 2: Recall the IND $R_1.Department \subseteq R_2.Department$ studied in Example 1. The existence of t_7 and t_8 in Table 1 makes the IND invalid. It can be seen that the $\mathsf{AIND_i}$ $R_1.Department \subseteq_\epsilon^i R_2.Department$ has an error rate of $\frac{1}{7}$, as "Artificial Intelligence" is the only one of the 7 distinct values for attribute Department in Table 1 that does not appear in Table 2. If a threshold ϵ no less than $\frac{1}{7}$ is used, then the $\mathsf{AIND_i}$ is considered as valid.

The satisfaction of $\mathsf{AIND_i}$ is measured based on the minimum number of tuple insertions to make the original IND hold true; the insertion of one tuple with the value of "Artificial Intelligence" in its attribute Department into Table 2 suffices. Note that the satisfaction of $\mathsf{AIND_i}$ considers the number of distinct values. □

Approximate IND with the deletion semantics. We use $\mathsf{AIND_d}$ to represent approximate IND with the deletion semantics. An $\mathsf{AIND_d}$ is in the form of $R_1.A \subseteq_\epsilon^d R_2.B$, where ϵ is a given error threshold. It is satisfied by r_1 and r_2 if, in r_1, the proportion of tuples whose value in A that is not present in attribute B of r_2 is below the given threshold. Specifically, $R_1.A \subseteq_\epsilon^d R_2.B$ holds, iff

$$\frac{|\{t \mid t \in r_1 \wedge \forall s \in r_2,\, s[B] \neq t[A]\}|}{|r_1|} \leq \epsilon$$

Example 3: Recall the IND $R_1.Origin \subseteq R_3.State$ investigated in Example 1. This IND is invalid due to an error in tuple t_5 of Table 1. It can be verified that the AIND_d $R_1.Origin \subseteq_\epsilon^d R_3.State$ has an error rate of $\frac{1}{9}$, as t_5 is the only one of the 9 tuples of Table 1 whose value in Origin is absent from the values of State of Table 3.

AIND_d adopts a measurement that considers the occurrence frequency of values, which is related to the number of required tuple deletions to recover the validity of the original IND; the deletion of t_5 from Table 1 suffices to make $R_1.Origin \subseteq R_3.State$ valid. Note that the satisfaction of AIND_d concerns the number of tuples. □

AIND discovery problem. With a set of relational instances (tables) $\{r_1, \ldots, r_m\}$ and an error threshold ϵ, the problem of AIND_i (resp. AIND_d) discovery is to find all AIND_is (resp. AIND_ds) that are satisfied by r_i and r_j, where $i, j \in [1, m]$.

Please note that ϵ is a user-defined parameter that indicates the level of violations that can be tolerated. However, using the same value of ϵ has different meanings in the two semantics. In AIND_i, it represents the maximum allowable proportion of distinct values that are absent from the RHS. For AIND_d, it represents the maximum allowable proportion of tuples containing attribute values that do not appear on the RHS. In the following sections, we present a discovery method that can be configured for AIND_i discovery or AIND_d discovery. Users can choose either of the two semantics and set the corresponding threshold based on their needs.

4 PARTITIONING DATA

In this section, we first review the data partitioning techniques employed by exact IND discovery methods [38, 42, 44]. We then highlight the challenges of adapting partitioning to AIND (AIND_i and AIND_d) discovery, and finally present our solutions.

4.1 Bucket, Partition and Bucket Fragment

IND discovery methods are commonly applied in scenarios involving a huge number of tables, *e.g.,* data lakes. To address the issue of too many tables that are impossible to accommodate in memory, data partitioning techniques have been introduced by previous exact IND discovery methods [38, 42, 44].

Bucket and partition. For each attribute of each table, the values in it are partitioned into a specified number of mutually exclusive parts, where each part is referred to as a *bucket* and identified by a bucket number. We denote the i-th bucket for attribute A of table r by $bucket(r.A, i)$. By using the same partitioning method on all attributes, the same values appearing in different attributes of different tables will be placed in buckets with the same number. A *partition* is a collection of buckets with the same bucket number from all attributes of all tables.

Formally, for a table r of schema R, and a specified number N of buckets, the i-th bucket for r on an attribute $A \in R$ is defined as: $bucket(r.A, i) = \{t[A] \mid \text{PartiFunc}(t[A]) = i, t \in r\}$, where $i \in [0, N-1]$. Herein, PartiFunc is the partitioning function, and a commonly used function is to first calculate the hash value for $t[A]$, and then take the modulus of the hash value by N. Note that only distinct values are stored in buckets and buckets are disjoint. The number of values in a bucket is referred to as the size of the bucket. The i-th partition, denoted by P_i, is $\{ bucket(r.A, i) \}$ for each attribute A of each table r considered in the discovery.

Table 4: relation table r_1

	A	B	C
t_1	1	1	6
t_2	2	2	5
t_3	1	3	6
t_4	4	4	6
t_5	1	5	1

Table 5: relation table r_2

	D	E
t_1	1	9
t_2	1	9
t_3	3	1
t_4	3	12
t_5	1	2

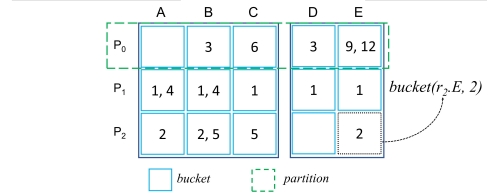


Figure 1: Partitioning Result

Example 4: For r_1 and r_2 shown in Table 4 and Table 5, assuming we use modulo division by three as the partitioning method, the resulting buckets and partitions are shown in Figure 1. We have P_0 = $\{ bucket(r_1.A, 0), bucket(r_1.B, 0), bucket(r_1.C, 0), bucket(r_2.D, 0), bucket(r_2.E, 0) \}$. Note that the same values in different attributes are placed in buckets with the same number. □

Exact IND discovery with partitions. Not all partitions need to be kept in memory at the same time; when memory is limited, some buckets can be stored on disk in files. Exact IND discovery can be easily performed by processing partitions one by one, because the validity of INDs can be checked within each partition. It is clear that an IND holds true if and only if it holds true inside every partition. By handling partitions sequentially, only those candidate INDs that have been verified to hold true on previous partitions will be considered on subsequent partitions.

Example 5: (Example 4 continued.) $R_1.A \subseteq R_2.D$ remains a candidate IND after processing partition P_0, as no violations of it are found in P_0. But this candidate is pruned after processing P_1 and no longer considered on P_2, as it is found to be violated in P_1. □

Fragment. When partitioning data, the process is to load a table into memory, then enumerate all the tuples of the table and place each attribute value into the corresponding bucket. Before all the tuples of a table are processed, some buckets of that table may be forced to be written to disk due to limited memory. Suppose a bucket is written to disk and the bucket in memory is cleared. When subsequent tuples of that table are processed, chances are that another value needs to be placed in that empty bucket. This situation results in *fragments* of the same bucket on disk and in memory; for each bucket, there may be several fragments on disk and at most one fragment in memory. It is important to note that in this case duplicate values may appear in these fragments, while they never exist in a bucket without fragmentation.

Some modifications to exact IND validation are needed to handle bucket fragmentation. To validate $R_1.A \subseteq R_2.B$, it now requires to check whether every value in each fragment of $bucket(r_1.A, i)$ is contained in a fragment of $bucket(r_2.B, i)$, for all $i \in [0, N-1]$.

4.2 Partitioning Data for AIND Discovery

Partitioning techniques are essential for handling large datasets, and limiting validations within partitions also helps improve efficiency. This paper aims to perform AIND discovery with partitions for the first time. We find that this leads to some new challenges.

(1) Validating AINDs with partitions differs significantly from validating exact INDs. As the satisfaction of an AIND is relaxed, violations found on a partition does not necessarily lead to the invalidity of the AIND. A more complex situation is that before accessing all partitions, an AIND may be determined as valid if the number of violations it causes is guaranteed to be no more than the maximum allowed number of violations.

(2) Fragmentation poses a new challenge for $\mathsf{AIND_i}$ discovery. Recall the number of distinct values in the LHS attribute is needed when checking the validity of an $\mathsf{AIND_i}$. When no bucket fragmentation exists, this number is equal to the total size of all buckets in that attribute. However, in the case of bucket fragmentation, duplicate values may exist in different fragments of the same bucket, making counting the distinct values for an attribute difficult.

Example 6: (Example 4 continued.) When processing table r_1, assume that memory limit has been reached after processing t_4, and $bucket(r_1.A, 1)$ is selected to be written to disk. After that, a new $bucket(r_1.A, 1)$ is needed in memory for $t_5[A]$. After processing r_1, $bucket(r_1.A, 1)$ has two fragments, which are $\{1, 4\}$ and $\{1\}$ respectively. Since there are duplicate values in them, the total size of fragments is larger than the number of distinct values in that bucket. Without loading all fragments into memory, we cannot determine the actual number of distinct values in the bucket, thus making it impossible to know the number of distinct values in attribute A. □

AIND discovery with partitions. We incorporate partitioning techniques into AIND discovery by storing new metadata during partitioning and presenting specially designed validation and pruning strategies. This section focuses on data partitioning methods and metadata design, while other techniques are discussed in Section 5.

In the partition-wise strategy for AIND validation, the number of violations is accumulated sequentially on partitions. Only when the number exceeds (or can never exceed) the *maximum allowed number*, a candidate AIND can be determined as invalid or valid. The calculation of the number and the maximum allowed number of violations involves measurement methods for $\mathsf{AIND_d}$ and $\mathsf{AIND_i}$.

(1) For an $\mathsf{AIND_d}$ $R_1.A \subseteq_\epsilon^d R_2.B$, (a) the number of violations is the number of values in A that are not present in B. Buckets containing only distinct values are not sufficient to compute this number, as it involves the occurrence frequency of values. We propose to save the frequency of each distinct value within bucket. The structure of elements within bucket is now in the form of <*value, frequency*>. In the sequel, we denote the frequency of *value* by *value.frequency*. (b) The maximum allowed number of violations is the product of the number of tuples and the error threshold ϵ. It can be easily calculated, as the number of tuples in a table can be obtained when partitioning the table.

(2) For an $\mathsf{AIND_i}$ $R_1.A \subseteq_\epsilon^i R_2.B$, (a) the number of violations is the number of distinct values in A that are not present in B. It can be computed based on buckets (or bucket fragments) containing only distinct values. (b) The maximum allowed number of violations is the product of the number of distinct values in A and the error threshold ϵ. Obtaining the number of distinct values in A becomes difficult in case of fragmentation, because the number can only be determined when all bucket fragments in A are read into memory. It is prohibitively expensive to do so, as this does not align with our memory management based on partitions and can lead to significant memory-to-disk swapping. To this end, we present the notion of *bounds* of buckets to effectively estimate the number.

Bounds of buckets. Suppose there are k fragments of a bucket b, denoted by $b_1, \ldots, b_k$, respectively. We denote by $b.size$ (resp. $b_i.size$) the size of b (resp. b_i). Due to the possible element overlap between the k fragments, we have the following result for $i \in [1, k]$:

$$max(b_i.size) \leq b.size \leq sum(b_i.size)$$

The left equal sign holds when the largest fragment contains all the elements of b, while the right equal sign holds when all the fragments have no overlapping elements. With the formula, the actual size of b can be estimated within a range of upper and lower bounds, which will be used for estimating the maximum allowed number of violations to help validate candidate $\mathsf{AIND_i}$s. The bounds are initially established in data partitioning when fragmentation occurs (Algorithm 1 in this Section), and will be updated as fragments are read into memory during AIND validation (Section 5).

Example 7: (Example 6 continued.) Before writing the first fragment $\{1, 4\}$ of $bucket(r_1.A, 1)$ to disk, we know the bucket size is at least 2. After bucket fragmentation occurs, the second fragment of the bucket in memory is $\{1\}$. Since the first fragment is now in the form of a disk file, we cannot determine the overlap between the two fragments. Instead, we know the bucket size is at least 2, which is the largest fragment size, and the bucket size is at most 3, which is the total size of all fragments. □

Algorithm. We provide an algorithm, called PAR (Algorithm 1), for partitioning all the tables in a given dataset U. It enumerates all tables, enumerates all tuples for each table, and for each attribute value of each tuple, employs the same partitioning function to determine the bucket containing the value. PAR generates all metadata required for AIND discovery while partitioning tables ($\mathsf{AIND_i}$ (or $\mathsf{AIND_d}$) discovery only utilizes part of them). This includes the number of tuples in a table r (*r.size*), and the metadata associated with each bucket, such as the size of the bucket in memory ($bucket(r.A, i).size$) and bounds ($bucket(r.A, i).lbound$ and *ubound*); note $bucket(r.A, i).size$ is the size of the fragment in memory in case of fragmentation. The occurrence frequency of each value (*t[A].frequency*) is also stored. Along the same lines as prior work, NULL values are discarded. But the number of NULLs in each attribute (*r.A.null*) is saved, which enables possible early termination of AIND validations (illustrated in Section 5).

When memory is running low (the ratio of used memory to total available memory is above a threshold α_1), Procedure WritetoDisk is called to continuously select the largest bucket for writing to disk until the ratio falls below another threshold α_2 (we set α_1 = 80% and α_2 = 60%). The largest bucket is identified using a max-heap each time. When a bucket is written to disk, a fragment of it is generated on the disk, the upper and lower bounds on the bucket size are updated based on the size of the fragment, and the size of the bucket in memory (the next fragment) is set to 0.

Algorithm 1: Partitioning Data Algorithm (PAR)

Input: a set U of relations $\{r_1, \ldots, r_m\}$ with schemas $\{R_1, \ldots, R_m\}$
Output: buckets for all tables

foreach $r \in U$ **do**
  $r.size \leftarrow 0$
  foreach $A \in R$ **do** $r.A.null \leftarrow 0$
  foreach $t \in r$ **do**
    $r.size \leftarrow r.size + 1$
    foreach $A \in R$ **do**
      if $t[A]$ *is NULL* **then**
        $r.A.null \leftarrow r.A.null + 1$
        continue
      $i \leftarrow \mathsf{PartiFunc}(t[A])$
      if $bucket(r.A, i)$ *is generated for the first time* **then**
        $bucket(r.A, i).\{size, lbound, ubound\} \leftarrow 0$
      if $t[A]$ *is contained in* $bucket(r.A, i)$ **then**
        $t[A].frequency \leftarrow t[A].frequency + 1$
      else
        add $t[A]$ into $bucket(r.A, i)$
        $t[A].frequency \leftarrow 1$
        $bucket(r.A, i).size \leftarrow bucket(r.A, i).size + 1$
      if $usedMemory/MaxMemory > \alpha_1$ **then**
        WritetoDisk()

Procedure WritetoDisk()
  while $usedMemory/MaxMemory > \alpha_2$ **do**
    $bucket(r.A, i) \leftarrow FindMaxBucket()$
    $bucket(r.A, i).lbound \leftarrow max(bucket(r.A, i).lbound, bucket(r.A, i).size)$
    $bucket(r.A, i).ubound \leftarrow bucket(r.A, i).ubound + bucket(r.A, i).size$
    write $bucket(r.A, i)$ to disk and clear it in memory
    $bucket(r.A, i).size \leftarrow 0$

5 AIND DISCOVERY METHOD

In this section, we first introduce a novel three-layer filtering structure for quantifying AIND violations (Section 5.1) and several rules that can potentially prune invalid candidate AINDs and identify valid AINDs before processing all partitions (Section 5.2). Combining them together, we finally give our discovery method (Section 5.3).

5.1 A Three-layer Filtering Structure

Validations of AIND candidates are carried out partition-wise. To quantify the violations of $R_1.A \subseteq_\epsilon^i R_2.B$ (or $R_1.A \subseteq_\epsilon^d R_2.B$), a basic building block is to calculate the number of violations of the AIND in the i-th partition, involving $bucket(r_1.A, i)$ and $bucket(r_2.B, i)$. We introduce a novel filtering structure to address the problem.

A filtering structure. We build a three-layer filter for each bucket. For a bucket with fragmentation, all the fragments are read into memory to build the filter. This is not a limitation because we only need multiple fragments of a bucket to be in memory, not all buckets for a particular attribute. Duplicate values in different fragments are removed once all fragments are merged, and the frequency of each distinct value is saved for the deletion semantics.

Every filter has M positions in each layer, and a hash function is utilized to map values in the bucket corresponding to the filter to the positions. The same hash function is used by all filters, and hence, same values in different buckets will have the same positions in their corresponding filters. We refer to M as the filter size; M should be an exponential power of 2, which enables efficient bitwise operations. (a) At the first layer, a bit-array with M bits (positions) is used, and "1" is set on a position if there exists at least one value that is mapped to the position. (b) At the second layer, a count array is used to save the number of distinct values that are mapped to each position. In case of the deletion semantics, in each position, the frequency of the value with the minimum frequency among all values mapped to the position is also stored. (c) At the third layer, the set of values mapped to a position is save in the position, together with the frequency of each value for deletion semantics.

Example 8: We showcase two sample buckets for insertion and deletion semantics in Figure 2a and Figure 2b, respectively. By setting the filter size M as 16, the filters for the two buckets are given in Figure 3a and Figure 3c respectively. Suppose values 4 and 20 are mapped to the fourth position within the two filters. In the filter for insertion semantics, at this position, (a) the value at the first layer is set to be 1; (b) the value at the second layer is set to be 2; and (c) the value at the third layer is the set {4, 20}. In contrast, in the filter for deletion semantics, (a) a pair <2, 3> is saved at the second layer, where 2 is the number of values mapped to the position, and 3 is the smaller of the frequencies of values 4 and 20 (as shown in Figure 2b); and (b) at the third layer, values are stored with their frequencies. □

Quantifying AIND violations. By leveraging our structure, the calculation for the number of violations (violation count) is divided into two parts: a rough calculation of the lower bound at first and then an exact calculation.

We consider $R_1.A \subseteq_\epsilon^i R_2.B$ (or $R_1.A \subseteq_\epsilon^d R_2.B$). To simplify the presentation, we denote the filter for $bucket(r_1.A, i)$ by f and the filter for $bucket(r_2.B, i)$ by f'. For f, we denote its k-th layer ($k \in [1, 3]$) by f_k, and the j-th element ($j \in [0, M-1]$) at the k-th layer by $f_k[j]$; similarly for f'. Recall that $f_1[j] = 1$ if some values are mapped to the j-th position, and $f_2[j]$ denotes the number of distinct values mapped to the j-th position. For the deletion semantics, we use $f_2[j].minFreq$ to denote the saved minimum frequency. At the third layer, $f_3[j]$ is the set of values mapped to position j. For each value v, we denote by $v.freq$ the frequency of it.

Rough calculation. In the rough calculation component, we only utilize the first two layers.

Insertion semantics. We find all the bit-array positions that are set to 1 in f_1. For each such position j, if $f_2[j]$ is larger than $f'_2[j]$, even in the best-case scenario where $f_3[j]$ includes all the values in $f'_3[j]$, there are still $f_2[j] - f'_2[j]$ violations. The following equation calculates the lower bound on the violation count:

$$\sum_{j=0}^{M-1} max(f_2[j] - f'_2[j], 0), \ where\ f_1[j] = 1 \tag{1}$$

Deletion semantics. The computation is more complex for deletion semantics, involving the occurrence frequencies of values. For a position j where $f_1[j] = 1$, there are at least $f_2[j] - f'_2[j]$ values that are absent, and each value occurs at least $f_2[j].minFreq$ times. The lower bound on the violation count is calculated as follows:

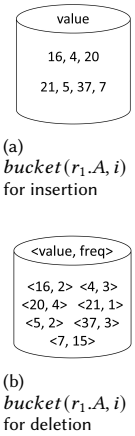


Figure 2: Sample Buckets

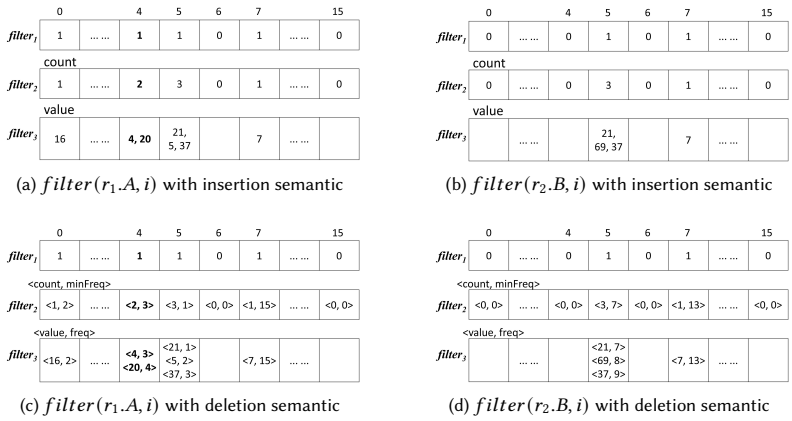


Figure 3: Sample Filters

$$\sum_{j=0}^{M-1} max((f_2[j] - f_2'[j]) \cdot f_2[j].minFreq, 0),\ where\ f_1[j] = 1 \quad (2)$$

Example 9: (Example 8 continued.) We give sample $filter(r_2.B, i)$ under insertion and deletion semantics in Figure 3b and Figure 3d respectively. Consider Figure 3a and Figure 3b for checking $R_1.A \subseteq_\epsilon^i R_2.B$. The lower bound on the violation count is (1 - 0) + (2 - 0) + (3 - 3) + (1 - 1) = 3. Consider Figure 3c and Figure 3d for checking $R_1.A \subseteq_\epsilon^d R_2.B$. The lower bound on the violation count is (1 - 0) × 2 + (2 - 0) × 3 + (3 - 3) × 1 + (1 - 1) × 15 = 8. □

Through rough calculation, we obtain a lower bound on the violation count for each candidate AIND in the current partition. By adding this to the total number of violations detected in prior partitions, we have a lower bound on the current violation count. If this lower bound is already enough to determine an AIND as invalid (details in Section 5.2), the exact calculation can be skipped. Otherwise, we proceed to the exact calculation stage.

Exact calculation. The third layer is also needed when calculating the exact violation count of a candidate AIND.

Insertion semantics. The violation count equals the total number of values that belong to $f_3[j]$ but do not belong to $f_3'[j]$, for all positions j where $f_1[j] = 1$. It is computed as follows:

$$\sum_{j=0}^{M-1} |f_3[j] \setminus f_3'[j]|,\ where\ f_1[j] = 1 \quad (3)$$

Deletion semantics. The violation count under the deletion semantics is the sum of the occurrence frequencies of all values that belong to $f_3[j]$ but do not belong to $f_3'[j]$, for all positions j where $f_1[j] = 1$. It is computed with the following equation:

$$\sum_{j=0}^{M-1} \sum_{v \in S} v.freq,\ where\ f_1[j] = 1\ and\ S = f_3[j] \setminus f_3'[j] \quad (4)$$

Example 10: Consider Figure 3a and Figure 3b when checking $R_1.A \subseteq_\epsilon^i R_2.B$. The exact violation count is 1 + 2 + 1 + 0 = 4 according to Equation 3. Consider Figure 3c and Figure 3d for checking $R_1.A \subseteq_\epsilon^d R_2.B$. According to Equation 4, the exact number of violations is 2 + (3 + 4) + 2 + 0 = 11. □

Complexity. (1) Building the filter of a bucket takes linear time in the bucket size, and the storage for the filter is also linear in the bucket size, as every position at the first or second layer requires limited storage. (2) Each rough calculation takes $O(M)$ where M is the filter size. This cost is irrelevant of the bucket size and usually very low. The exact calculation for $R_1.A \subseteq_\epsilon^i R_2.B$ (or $R_1.A \subseteq_\epsilon^d R_2.B$) in partition P_i takes linear time in the size of $bucket(r.A, i)$ at worst. (3) Rough calculation is much more cost-effective than exact calculation, and exact calculation is only necessary when rough calculation is insufficient to confirm a candidate as invalid.

5.2 Rules to Determine Validity and Invalidity

We provide a set of rules to determine whether a candidate AIND is valid or invalid based on the number of violations already found and metadata information of the buckets, without necessarily processing all partitions. This enables early termination of some candidate AINDs, as only candidates that are not confirmed as valid or invalid will be processed on subsequent partitions. These rules illustrate the significant differences between AIND and exact IND validations. We consider $R_1.A \subseteq_\epsilon^i R_2.B$ (or $R_1.A \subseteq_\epsilon^d R_2.B$) in this subsection.

Rules for determining validity. A candidate AIND is confirmed as valid if its violation count can never exceed the maximum allowed number of violations. The rules applicable to insertion and deletion semantics are different, as presented below.

Insertion semantics. The number of distinct values is considered for insertion semantics. Assume for the buckets in A that have been processed, the number of distinct values is b, and a total of a violations have been detected. If the remaining buckets in A do not have fragmentation, then the total size of these buckets equals the number of remaining distinct values, denoted as c. Otherwise, c is defined as the sum of the upper bounds of these buckets. We have the following result for determining the validity.

Proposition 1: $R_1.A \subseteq_\epsilon^i R_2.B$ is valid, if $a + c \leq (b + c) \cdot \epsilon$.

Proof: Assume the actual number of distinct values in the remaining buckets is x, where $x \leq c$. Based on the fact that $\epsilon < 1$ and $x - c \leq 0$, we have the following starting from the given assumption:

$a + c \leq (b + c) \cdot \epsilon$

$a + c + x - c \leq (b + c) \cdot \epsilon + x - c$
$a + x \leq (b + c) \cdot \epsilon + x - c \leq (b + c) \cdot \epsilon + (x - c) \cdot \epsilon$
$a + x \leq (b + x) \cdot \epsilon$

That is, even if the remaining distinct values are all absent, the violation count is still less than the maximum allowed number. □

Deletion semantics. The number of values (not distinct values) is considered for deletion semantics. Assuming that in the buckets in A that have been processed, the number of values is b (this number is known as the frequency of each distinct value is saved), total violations detected is a, and the total number of NULL values in r_1 is n, we have the following result.

Proposition 2: $R_1.A \subseteq_\epsilon^d R_2.B$ is valid, if $a + |r_1| - n - b \leq |r_1| \cdot \epsilon$.

Rules for determining invalidity. A candidate AIND is confirmed as invalid if the violation count already exceeds the maximum allowed number of violations. Different from the previous case for establishing the validity of AINDs, we can first use the lower bound on the violation count obtained from rough calculation for confirming the invalidity of AINDs, and proceed with the exact violation count only when necessary. We present the rules for insertion and deletion semantics respectively.

Insertion semantics. Assume that the number of distinct values that have been processed in previous buckets is b, total violations detected is a, and the number (or the upper bound on the number in case of fragmentation) of distinct values in remaining buckets is c. We have the following result.

Proposition 3: $R_1.A \subseteq_\epsilon^i R_2.B$ is invalid, if $a > (b + c) \cdot \epsilon$.

Deletion semantics. Assuming in the buckets that have been processed, the total number of detected violations is a, we have the following result for confirming the invalidity of an $\mathsf{AIND_d}$. The rule is directly based on the definition.

Proposition 4: $R_1.A \subseteq_\epsilon^d R_2.B$ is invalid, if $a > |r_1| \cdot \epsilon$.

5.3 AIND Discovery Method

Putting our techniques together, we present AIND discovery method.

Algorithm. Our discovery method, referred to as AINDD (Algorithm 2), takes as inputs the buckets built with Algorithm PAR (Section 4.2) and the error threshold ϵ. It can be configured to identify all $\mathsf{AIND_i}$s (or $\mathsf{AIND_d}$s) satisfied by r_i and r_j from a given set of tables, by setting the semantics mode to insertion (or deletion).

AINDD utilizes a matrix to store the violation count detected for each AIND, and Σ to save the result set (lines 1-2). AINDD iterates through each partition in increasing order of the number of buckets with fragments in the partition. As AINDD enjoys the early termination property, not all partitions are necessarily needed for validating AINDs. Prioritizing partitions with a greater number of buckets entirely in memory may reduce disk operations and facilitate memory release, as all buckets in a partition can be released once the partition is processed. For each bucket, a three-layer filter is built as studied in Section 5.1 (line 8). All the fragments within the bucket are merged before constructing the filter (lines 6-7), which results in the adjustment of the bounds on the total bucket size in the attribute (not shown). Note that the exact total bucket size is still unknown if there are more buckets with fragments.

AINDD first tries to confirm the validity of candidates based on the known violation counts (lines 12-19). If unsuccessful, it then tries to prune invalid candidates based on rough and exact calculations respectively (lines 20-28). Candidate AINDs that are confirmed as valid or invalid have their positions in the matrix set to negative values (lines 18 and 25), enabling them to be quickly skipped in subsequent operations (line 11), and valid ones are collected in Σ (line 17). For a candidate whose validity and invalidity are not confirmed, the current violation count is saved in the matrix (line 29). AINDD utilizes additional flags to help improve efficiency. For each attribute, it keeps track of *LActive* and *RActive* flags, indicating whether there are still candidate AINDs with that attribute as the LHS or RHS attribute. The *Active* flag is set to true if either *LActive* or *RActive* is true. If the *Active* flag for an attribute is false, then it is safe to discard all the buckets in that attribute in all subsequent partitions. After processing a partition, AINDD updates these flags based on the newly determined valid and invalid AINDs (line 31). After processing all partitions, AINDD further collects valid results based on the violation counts saved in the matrix (lines 32-33). Note that if all flags are set to false after processing a partition, subsequent partitions will be promptly skipped.

Algorithm 2: AIND Discovery (AINDD)

Input: buckets for all tables, error threshold ϵ and semantics *mode*
Output: the set Σ of all valid AINDs

1 $Matrix[\cdot][\cdot] \leftarrow 0$
2 $\Sigma \leftarrow \emptyset$
3 Sort the partitions in ascending order based on the number of buckets containing fragments
4 **foreach** *partition* p_i **do**
5 **foreach** $bucket(r.A, i) \in p_i$ *where* $r.A.Active = true$ **do**
6 **foreach** *fragment frag of* $bucket(r.A, i)$ **do**
7 merge *frag* into $bucket(r.A, i)$
8 BuildFilter($bucket(r.A, i)$, *mode*)
9 **foreach** $f = filter(r.A, i)$ *where* $r.A.LActive = true$ **do**
10 **foreach** $f' = filter(r'.B, i)$ *where* $r'.B.RActive = true$ **do**
11 **if** $Matrix[r.A][r'.B] < 0$ **then continue**
12 $cnt \leftarrow Matrix[r.A][r'.B]$
13 **if** *mode= insertion* **then**
14 *flag* ← Check($r.A, r'.B, cnt$) with Proposition 1
15 **else** *flag* ← Check($r.A, r'.B, cnt$) with Proposition 2
16 **if** *flag = true* **then**
17 $\Sigma \leftarrow \Sigma \cup \{r.A \subseteq_\epsilon^{mode} r'.B\}$
18 $Matrix[r.A][r'.B] \leftarrow -2$
19 **continue**
20 $cnt \leftarrow$ RoughCalc(f, f', *mode*) + $Matrix[r.A][r'.B]$
21 **if** *mode = insertion* **then**
22 *flag* ← Prune($r.A, r'.B, cnt$) with Proposition 3
23 **else** *flag* ← Prune($r.A, r'.B, cnt$) with Proposition 4
24 **if** *flag = true* **then**
25 $Matrix[r.A][r'.B] \leftarrow -1$
26 **continue**
27 $cnt \leftarrow$ ExactCalc(f, f', *mode*) + $Matrix[r.A][r'.B]$
28 repeat lines 21-26 with *cnt* as the new input
29 $Matrix[r.A][r'.B] \leftarrow cnt$
30 free all the memory for buckets or filters in p_i
31 update *LActive*, *RActive* and *Active* flags for all attributes
32 **foreach** *valid* $r.A \subseteq_\epsilon^{mode} r'.B$ *according to* $Matrix[r.A][r'.B]$ **do**
33 $\Sigma \leftarrow \Sigma \cup \{r.A \subseteq_\epsilon^{mode} r'.B\}$

Table 6: Dataset Information

	# tables	size (MB)	max # tuples per table	total attribute number	max # distinct values per attribute	average # distinct values per attribute
CENSUS	4	112	199,524	42	99,800	2,462
WIKIPEDIA	2	615	14,024,428	11	5,475,188	649,330
TPC-H	8	1,257	6,001,215	61	4,580,667	236,137
BTC	22	2,988	523,808	375	523,808	396,491
GENOME	7	4,166	28,490,116	27	2,619,419	143,427
EEG	4	5,291	71,856	4,100	11,965	6,574
TPC-DS	25	8,748	85,610,088	419	1,920,800	129,310
IMDB	21	11,579	36,244,344	94	36,244,344	1,082,082

Example 11: For r_1 and r_2 in Table 4 and Table 5, AINDD considers all attribute pairs across the two tables as candidate AINDs. A candidate cannot be immediately discarded when some violations occur. Instead, the violation count is incremented. AINDD tries to confirm candidates as valid or invalid after processing each partition. Once all partitions have been processed, AINDD terminates and checks the violation counts to finalize the set of valid AINDs, in addition to those already identified during the processing. □

6 EXPERIMENTAL STUDY

Experimental setting. We first give the experimental setting.

Datasets. We use 8 datasets in our experiments. CENSUS, WIKIPEDIA, BTC, GENOME, EEG and IMDB are real datasets, while TPC-H and TPC-DS are commonly-used synthetic benchmark datasets. The properties of them are given in Table 6. These datasets exhibit significant differences in terms of the maximum number of tuples in a single table, total number of attributes, and the maximum as well as average number of distinct values per attribute.

Algorithms. All the following algorithms are implemented in Java and integrated into the Metanome system [37] for comparison.

(1) We develop methods AINDD$_i$ and AINDD$_d$ for discovering AIND$_i$s and AIND$_d$s respectively, based on Algorithm 2 by setting different semantics modes. Recall that method PAR (Algorithm 1) is employed to partition data to facilitate the discovery process. We totally need to set three parameters: the number N of partitions for method PAR, the filter size M for our filtering structure, and the error threshold ϵ. We use the following formula to determine N:

$$N = max(\left\lfloor \frac{\sum_{r\in U}\sum_{A\in R}|\pi_A(r)|}{\gamma \cdot \sum_{r\in U}|R|} \right\rfloor, 10) \quad (5)$$

This formula first calculates the average of the number of distinct values across all attributes in a dataset U. It then divides this average by γ to determine N, and also ensures N is at least 10. We estimate $|\pi_A(r)|$ using the HyperLogLog method [15], and set γ to 10,000 as large tables are anticipated. M is set to 1,024 through experiments. Unless otherwise stated, we set ϵ to 0.01.

(2) We develop two baseline methods for discovering AIND$_i$s based on Demarchi [33] and SPIDER [5, 6], called as A-DeMarchi and A-SPIDER. Although such extensions have been briefly mentioned in the literature, we are not aware of previous implementations.

(3) Our method is used to discover exact INDs by setting ϵ to 0. This version, referred to as AINDD*, is compared with one of the state-of-the-art exact IND discovery methods, known as BINDER [38][1].

[1]http://hpi.de/en/naumann/projects/repeatability/data-profiling/ind (last accessed 2025/1/26).

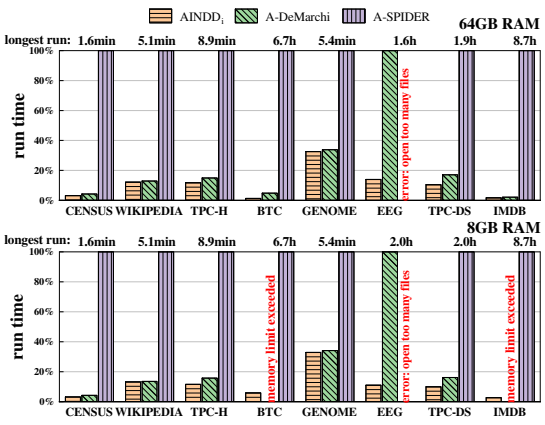


Figure 4: Comparison Of AIND$_i$ Discovery Methods

(4) Our method is compared to Algorithm SAWFISH [22][2] in terms of the ability to identify valid INDs from dirty data. As noted in Section 2, SAWFISH discovers SINDs, which relax the requirement of equality to similarity in value comparison.

Measurement & Running environment. For each algorithm, the reported runtime comprises disk I/O resulting from data partitioning and memory limitation (if applicable), as well as dependency discovery. The average of three runs is presented here. All the experiments are run on a machine powered by an Intel Xeon Bronze 3204 1.90G CPU, with 128GB of memory and CentOS Linux.

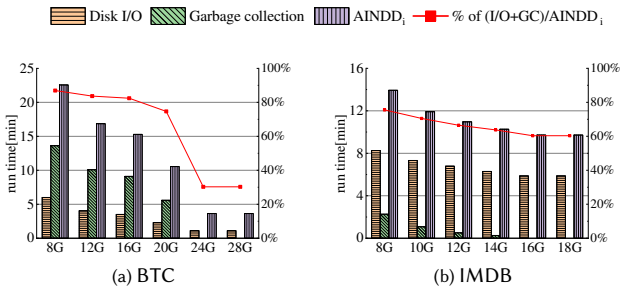
Experimental results. We next report our findings.

Exp-1: AIND$_i$ discovery methods. We compare AINDD$_i$ against A-DeMarchi and A-SPIDER in Figure 4. We use the longest algorithm time on each dataset as a baseline and present the proportion of other algorithm time relative to this baseline time. AINDD$_i$ and A-SPIDER can deal with large datasets that cannot be fully kept in memory, while A-DeMarchi is a completely memory-based algorithm. To also test different disk and memory scheduling strategies, two sets of experiments are conducted by limiting the available memory of Metanome [37] to 8GB and 64GB respectively, where the 8GB setting can be used to simulate discovery situations on large-scale real-world datasets. Similar settings are adopted in [11, 22, 38].

(1) When the available memory is 64GB, the only failure case occurs in A-SPIDER for the dataset EEG. This is because A-SPIDER creates a file for each attribute, and as EEG has 4,100 attributes, this ultimately leads to the file opening failure. AINDD$_i$ performs the best. It is on average 28X and up to 76X faster than A-SPIDER, and on average 2.3X and up to 7.1X faster than A-DeMarchi. Note that the complexity of AIND$_i$ validation depends on the number of distinct values, and the number of candidate AIND$_i$s is related to the number of attributes. The combined influence of these factors implies that larger datasets do not necessarily lead to longer time.

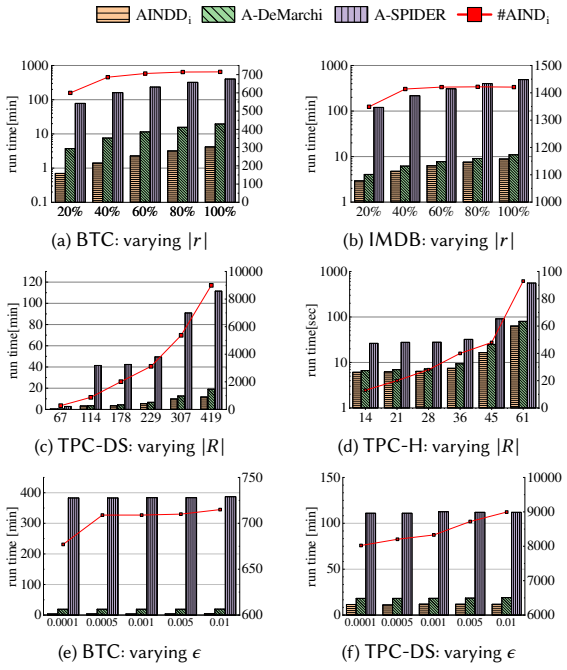
(2) When the available memory is limited to 8GB, it is impossible to fully keep some large datasets in memory. Since A-SPIDER always creates a disk file for each attribute regardless of the available memory, its performance is largely not affected by the memory limitation. A-DeMarchi cannot handle situations of insufficient memory,

[2]https://github.com/HPI-Information-Systems/Sawfish (last accessed 2025/1/26).



(a) BTC (b) IMDB

Figure 5: AINDD$_i$ with Varying Memory Settings



(a) BTC: varying $|r|$ (b) IMDB: varying $|r|$

(c) TPC-DS: varying $|R|$ (d) TPC-H: varying $|R|$

(e) BTC: varying ϵ (f) TPC-DS: varying ϵ

Figure 6: Scalability of AIND$_i$ Discovery Methods

making it fail on BTC and IMDB. Although memory limitation may incur additional disk operations and negatively affect the performance of AINDD$_i$ (for instance, the runtime of AINDD$_i$ on IMDB increases from 8.8 minutes to 14.1 minutes), AINDD$_i$ can well schedule memory and disk usage and efficiently handle all datasets.

(3) Insufficient memory can lead to more disk I/O operations (where some buckets are written back to disk and subsequently read back) and memory cleanup (garbage collection). In Figure 5, we present the time spent on them, along with their proportions in the total time of AINDD$_i$, using datasets BTC and IMDB. The results indicate that when memory is severely limited, these two components can account for the vast majority of the total time. Their proportions decrease with the increase of the available memory, as expected. When memory is sufficient, forced memory cleanup no longer occurs, and the disk I/O overhead is limited to the reading of the dataset. Disk I/O consistently comprises a considerable portion of the processing in the dataset IMDB due to its large size.

Exp-2: Scalability. Using 64GB of memory, we study the scalability. Besides the runtime, the number of AIND$_i$s discovered is reported. We present results on some datasets due to space limitation. For the experiments involving $|r|$ and ϵ, the trends are generally similar across all datasets. For the experiments involving $|R|$, we select datasets where the increase in $|R|$ significantly impacts the discovery results.

Varying $|r|$. We first test the impact of $|r|$, where $|r|$ is the number of all tuples in the tables of a dataset.

(1) By varying the proportion of tuples in BTC, the runtime of different algorithms is depicted in Figure 6a. As the proportion varies from 20% to 100%, AINDD$_i$ takes 42s to 4min9s, as opposed to 3min44s to 19min28s taken by A-DeMarchi, and 1h18min to 6h42min taken by A-SPIDER. All algorithms demonstrate good scalability with respect to $|r|$.

(2) We vary $|r|$ on dataset IMDB and report the results in Figure 6b. As the proportion varies from 20% to 100%, the number of AIND$_i$s discovered only slightly increases from 1,349 to 1,421. All algorithms scale well with $|r|$, which confirms our observations on BTC.

Varying $|R|$. We then study the impact of $|R|$, which is the number of all attributes in the tables of a dataset.

(1) The runtime of all methods on TPC-DS is shown in Figure 6c, where $|R|$ is varied by dropping tables. As the number of candidate AIND$_i$s is the square of the number of attributes, all methods are very sensitive to $|R|$. Specifically, as $|R|$ varies from 67 to 419, the number of AIND$_i$s discovered grows from 272 to 8,995, and the time of AINDD$_i$ increases by 31.8X, as opposed to that of A-DeMarchi increases by 52X and that of A-SPIDER increases by 41X. Different from the other two methods, AINDD$_i$ enjoys the early termination property and shows better scalability *w.r.t.* $|R|$.

(2) We vary $|R|$ on dataset TPC-H in Figure 6d. As $|R|$ increases from 14 to 61, the number of AIND$_i$s discovered grows from 13 to 93, and the runtime of AINDD$_i$ increases by 10.4X, while A-DeMarchi and A-SPIDER increase by 12X and 21.1X, respectively.

Varying ϵ. We finally study the impact of the error threshold ϵ.

(1) By varying ϵ on BTC, the results are shown in Figure 6e. The runtime of A-DeMarchi and A-SPIDER is nearly unaffected by changes of ϵ. Due to the absence of pruning rules, they simply calculate the number of violations for each candidate AIND$_i$ across the entire dataset and finally identify valid ones that meet the requirement specified by ϵ. In contrast, the impact of ϵ on AINDD$_i$ is more intricate. Increasing ϵ allows for more tolerance of violations, helping the early identification of valid AIND$_i$s but hindering the early identification of invalid ones. With the increase of ϵ, the number of valid AIND$_i$s increases from 677 to 715, thereby causing the runtime of AINDD$_i$ to increase from 3.4 minutes to 4.2 minutes.

(2) By varying ϵ, the results on TPC-DS are given in Figure 6f, which are similar to those in Figure 6e. As ϵ varies from 0.0001 to 0.01, the number of valid AIND$_i$s increases from 7,912 to 8,995, while the runtime of AINDD$_i$ slightly increases from 11min32s to 11min43s.

Exp-3: Insertion and deletion semantics. With 64GB of memory, we compare the efficiency of AINDD$_i$ and AINDD$_d$, as well as the result sizes (including the intersection of their results), as shown in Figure 7.

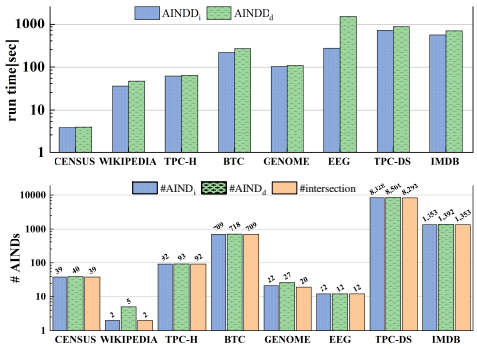


Figure 7: Comparison of AINDD$_i$ and AINDD$_d$

(1) AINDD$_d$ consistently takes more time than AINDD$_i$, which is reasonable as AINDD$_d$ needs to consider the occurrence frequency associated with each distinct value. AINDD$_i$ is on average 21.5% faster than AINDD$_d$. The performance gap becomes notably evident on EEG, as EEG contains a much larger number of attributes (4,100) compared to other tested datasets.

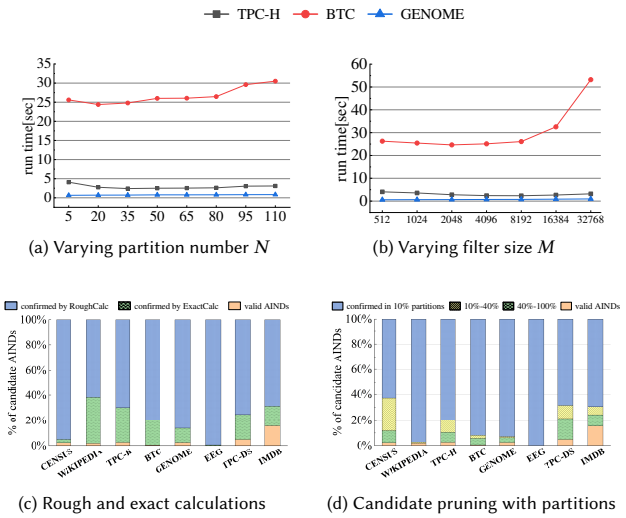
(2) On the tested datasets, the result set of AINDD$_d$ is always not smaller than that of AINDD$_i$. The two result sets exhibit a containment relationship on most datasets, although the difference is usually not significant. On certain datasets, such as TPC-DS and GENOME, the results of AINDD$_i$ and AINDD$_d$ are complementary.

(3) We manually check the results on GENOME [48], which contains attributes with clear semantic information. With ϵ = 0.01, 22 AINDD$_i$s and 27 AINDD$_d$s are discovered, and among them, 17 AINDD$_i$s and 21 AINDD$_d$s are identified as meaningful INDs. As an example, *ratings.item_id* $\subseteq_\epsilon^d$ *metadata.item_id* is valid only under the deletion semantics. We find that the values present in the LHS attribute but absent in the RHS attribute occur with low frequency. As a result, the error rate calculated based on insertion semantics is notably higher than that based on deletion semantics, and surpasses ϵ.

Exp-4: Parameter settings. We study the setting of partition number N and filter size M, using 64GB of memory. To better analyze the results, we only consider the discovery part (Algorithm 2).

(1) In Figure 8a, we report the time of AINDD$_i$ on three datasets by fixing M = 1,024 and varying N. As N increases from 5 to 20, the runtime on TPC-H reduces by about 33%, while the gains on BTC and GENOME are less significant. The size of each bucket (filter) decreases with the increase of N, which usually favors the effectiveness of the filtering. With further growth in N, the improvement becomes less significant. A too large N can cause performance to decrease, as further reducing the filter size no longer brings notable benefits, but instead increases the overhead of initialization. AINDD$_i$ can achieve good performance when N is in a large range, making the setting of N relatively easy. It can also be verified that Equation 5 can be used to obtain a reasonable setting for N.

(2) In Figure 8b, we use the parameter N calculated according to Equation 5 on each dataset and report the time of AINDD$_i$ by varying M. The time remains stable when M is in a relatively wide range. Specifically, when M ranges from 512 to 4,096, the standard deviation of time values in a single dataset is at most 0.75. The performance degrades when M is too large, which results in decreased



(a) Varying partition number N

(b) Varying filter size M

(c) Rough and exact calculations

(d) Candidate pruning with partitions

Figure 8: Parameter Settings and Optimizations

efficiency of rough calculation, without being able to further enhance the pruning power. The results show that after setting N, a wide range of M settings can achieve satisfactory performance.

Exp-5: Optimizations. We verify our optimization techniques.

(1) Among all the candidate AINDD$_i$s, we report the proportions confirmed invalid through rough calculation and exact calculation respectively, as well as the proportion of valid AINDD$_i$s. The results in Figure 8c indicate that rough calculation can effectively prune the vast majority of candidates; as mentioned earlier, rough calculation has a very low computational complexity.

(2) After the invalidity of a candidate is confirmed in a partition, it does not need to be reconsidered in subsequent partitions. We illustrate in Figure 8d the proportion of candidate AINDD$_i$s that are confirmed across different proportions of partitions. The results show that after processing only 10% of the partitions, the proportion of candidates found invalid exceeds 60% across all datasets, and it even surpasses 90% in several datasets. This clearly demonstrates the effectiveness of using data partitioning techniques in discovery.

Exp-6: Exact IND discovery. We compare AINDD* against exact IND discovery method BINDER in Figure 9.

(1) AINDD* outperforms BINDER on all tested datasets when the available memory is 64GB, and is on average 2X and up to 2.3X faster. We find AINDD* has advantages over BINDER mainly in two aspects. (a) In data partitioning, AINDD* employs an *adaptive* strategy, writing buckets to disk only when memory is about to run out. In contrast, BINDER uses an *eager* strategy to write more buckets to disk because it expects large datasets. When memory is actually sufficient, the eager strategy results in more disk operations than the adaptive strategy. (b) The three-layer filtering structure used in AINDD* also proves effective for exact IND validation. In particular, the rough calculation component yields significant pruning power with low computational complexity on certain datasets.

(2) With a memory capacity of 8GB, the advantage of the adaptive strategy over the eager strategy degrades on datasets requiring additional disk I/O. Hence, the superiority of AINDD$_i$ decreases

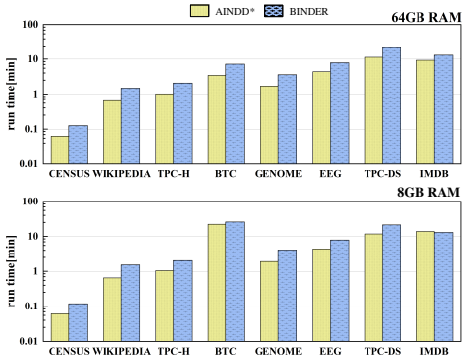


Figure 9: Comparison Of Exact Discovery Algorithms

on BTC, and BINDER performs better in IMDB. We find that this is because the filtering structure of AINDD_i takes into account the need for quantifying violations, resulting in a larger memory footprint compared to the indexing structure used by BINDER, which hinders AINDD_i in case of limited memory. Additionally, when the exact calculation component of AINDD_i is used to verify an IND, its performance is slightly weaker than the structure of BINDER, which is designed specifically for exact IND validation.

Exp-7: Effectiveness. We verify the effectiveness of our approach.

(1) We show AIND discovery can identify hidden INDs from real-life dirty data. In GENOME [48], the table *metadata* contains information on 84,661 movies, where attribute *item_id* serves as a unique identifier for each movie. It is expected that values in *item_id* in other tables all come from the *metadata* table, but data errors hinder exact INDs. We manually identify 38 meaningful INDs, and only 10 out of them are actually valid in GENOME. We perform AIND discovery and measure the discovery result by *precision*, *recall* and *F1-score*, where *precision* is defined as the proportion of the discovered AINDs that are among the 38 INDs, and *recall* is the proportion of the 38 INDs that are discovered. We compare our approach with SAWFISH [22]. SAWFISH uses edit distance (ED) or Jaccard similarity (JAC) to measure the similarity of strings, and treat strings with a distance smaller than the given threshold as equal. We set 0.4 (the default parameter used in [22]) or 0.8 as the JAC threshold, and 1 (the minimum distance) or 3 as the ED threshold.

We see the following from the results reported in Table 7. The *precision* of AINDD_i and AINDD_d first decreases but then almost stabilizes as ϵ increases, while the *recall* continuously increases. This leads to an overall improvement in the *F1-score*. Except for the setting of JAC = 0.8, SAWFISH has much lower *precision*, higher *recall*, and much lower *F1-score* compared to AINDD_i and AINDD_d; this is because the result set of SAWFISH is always much larger than those of AINDD_i and AINDD_d (not shown). Even using the smallest ED threshold, SAWFISH may still generate a large number of false positives. The setting of JAC = 0.8 implies a strict similarity measure, resulting in a very low *recall*. AINDD_i and AINDD_d are not only more effective but also more efficient. Under the tested settings, SAWFISH takes at least 2 minutes and up to 227 minutes, while AINDD_i and AINDD_d take at most 1.8 minutes.

(2) We verify that AIND discovery can effectively handle various types of errors. We consider 3 valid exact INDs and their related attributes in CENSUS. After introducing noise, we study the precision and recall of AIND and SIND discovery. Three types of noise injection methods are used: (a) insertion, placing a randomly generated new value of length one to six on an attribute related to exact INDs while setting other attributes to NULL; (b) deletion, setting the value of an attribute related to INDs to NULL; and (c) modification, adding a randomly generated suffix of length one to three to the original value. Insertion and deletion are applied to the LHS and RHS attributes of exact INDs respectively, and modification can be applied to any attribute of INDs. All noise injections will not introduce new valid exact INDs. We set the noise rate to $\frac{1}{1000}$ of the number of tuples. Besides a single noise type, we also use a mixed injection method of all types.

The experimental results are presented in Table 8 (P for precision and R for recall). We report the results of AINDD_d, since most attributes in CENSUS have a small number of distinct values. AINDD_d can effectively handle all cases by finding and only finding the expected INDs, and its discovery results show good stability with respect to changes of the threshold. For SAWFISH, by gradually increasing the ED threshold or decreasing the JAC threshold, constraints on string similarity are continuously relaxed until all original INDs are discovered (R = 1); further adjustment of the thresholds will only result in lower precision. In order to discover all INDs, SAWFISH often results in very low precision because a large number of meaningless SINDs are discovered as string similarity constraints are relaxed.

Table 7: Finding hidden INDs from GENOME

	AINDD_i				AINDD_d				SAWFISH			
ϵ	0.0001	0.001	0.01	0.1	0.0001	0.001	0.01	0.1	JAC=0.4	JAC=0.8	ED=1	ED=3
precision	0.92	0.82	0.77	0.76	0.92	0.77	0.78	0.74	0.21	1.00	0.32	0.18
recall	0.29	0.37	0.45	0.58	0.29	0.34	0.55	0.66	0.90	0.21	0.55	0.92
F1-score	0.44	0.51	0.57	0.66	0.44	0.47	0.65	0.69	0.34	0.35	0.41	0.30

Table 8: Handling Errors on CENSUS

	threshold	all-errors		left-insert		left-modify		right-delete		right-modify	
		P	R	P	R	P	R	P	R	P	R
SAWFISH	ED=1	0.00	0.00	0.00	0.00	0.00	0.00	0.33	1.00	0.67	0.67
	ED=3	0.00	0.00	0.00	0.00	0.21	1.00	-	-	0.21	1.00
	ED=6	0.04	1.00	0.04	1.00	-	-	-	-	-	-
	JAC=0.8	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	0.67
	JAC=0.4	0.12	0.67	0.04	0.33	0.19	1.00	-	-	0.18	1.00
	JAC=0.2	0.01	1.00	0.06	1.00	-	-	-	-	-	-
AINDD_d	$\epsilon = 0.001$	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
	$\epsilon = 0.01$	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**

7 CONCLUSION

We have provided the first comprehensive study on AIND discovery, by introducing a new definition of AINDs based on deletion semantics and developing a method that can be configured to identify AINDs based on insertion or deletion semantics. Our method employs novel data structures and techniques to improve efficiency. An extensive experimental study has been conducted to verify the efficiency and effectiveness of our approach.

We aim to adapt our method to diverse settings, including distributed [41], conditional [32] and incremental discovery [40, 45].

ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China 62172102, 61925203 and U22B2021. For any correspondence, please refer to Zijing Tan and Shuai Ma.

REFERENCES

[1] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2017. Data Profiling: A Tutorial. In *SIGMOD 2017*. 1747–1751.

[2] Ziawasch Abedjan, Lukasz Golab, Felix Naumann, and Thorsten Papenbrock. 2018. *Data Profiling*. Morgan & Claypool Publishers.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

[4] Jana Bauckmann, Ziawasch Abedjan, Ulf Leser, Heiko Müller, and Felix Naumann. 2012. Discovering conditional inclusion dependencies. In *CIKM*. 2094–2098.

[5] Jana Bauckmann, Ulf Leser, and Felix Naumann. 2010. Efficient and exact computation of inclusion dependencies for data integration. *Technical Report* (2010). https://hpi.de/fileadmin/user_upload/fachgebiete/naumann/publications/PDFs/2010_bauckmann_efficient.pdf

[6] Jana Bauckmann, Ulf Leser, Felix Naumann, and Veronique Tietz. 2007. Efficiently Detecting Inclusion Dependencies. In *ICDE*. 1448–1450.

[7] Thomas Bläsius, Tobias Friedrich, and Martin Schirneck. 2022. The complexity of dependency detection and discovery in relational databases. *Theor. Comput. Sci.* 900 (2022), 79–96.

[8] Leon Bornemann, Tobias Bleifuß, Dmitri V. Kalashnikov, Fatemeh Nargesian, Felix Naumann, and Divesh Srivastava. 2024. Efficient Discovery of Temporal Inclusion Dependencies in Wikipedia Tables. In *EDBT*. 399–411.

[9] Loreto Bravo, Wenfei Fan, and Shuai Ma. 2007. Extending Dependencies with Conditions. In *VLDB*. 243–254.

[10] Yuyang Dong, Kunihiro Takeoka, Chuan Xiao, and Masafumi Oyamada. 2021. Efficient Joinable Table Discovery in Data Lakes: A High-Dimensional Similarity-Based Approach. In *ICDE*. 456–467.

[11] Falco Dürsch, Axel Stebner, Fabian Windheuser, Maxi Fischer, Tim Friedrich, Nils Strelow, Tobias Bleifuß, Hazar Harmouch, Lan Jiang, Thorsten Papenbrock, and Felix Naumann. 2019. Inclusion Dependency Discovery: An Experimental Evaluation of Thirteen Algorithms. In *CIKM*. 219–228.

[12] Mahdi Esmailoghli, Jorge-Arnulfo Quiané-Ruiz, and Ziawasch Abedjan. 2022. MATE: Multi-Attribute Table Extraction. *Proc. VLDB Endow.* 15, 8 (2022), 1684–1696.

[13] Grace Fan, Jin Wang, Yuliang Li, and Renée J. Miller. 2023. Table Discovery in Data Lakes: State-of-the-art and Future Directions. In *SIGMOD*. 69–75.

[14] Wenfei Fan and Floris Geerts. 2012. *Foundations of Data Quality Management*. Morgan & Claypool Publishers.

[15] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete mathematics & theoretical computer science* (2007).

[16] Jarek Gryz. 1998. Query Folding with Inclusion Dependencies. In *ICDE*. 126–133.

[17] Laura M. Haas, Mauricio A. Hernández, Howard Ho, Lucian Popa, and Mary Roth. 2005. Clio grows up: from research prototype to industrial tool. In *SIGMOD*. 805–810.

[18] Oktie Hassanzadeh, Ken Q. Pu, Soheil Hassas Yeganeh, Renée J. Miller, Lucian Popa, Mauricio A. Hernández, and Howard Ho. 2013. Discovering Linkage Points over Web Data. *Proc. VLDB Endow.* 6, 6 (2013), 444–456.

[19] Ihab F. Ilyas and Xu Chu. 2019. *Data Cleaning*. ACM.

[20] Yifeng Jin, Zijing Tan, Jixuan Chen, and Shuai Ma. 2023. Discovery of Approximate Lexicographical Order Dependencies. *IEEE Trans. Knowl. Data Eng.* 35, 4 (2023), 3684–3698.

[21] Yifeng Jin, Zijing Tan, Weijun Zeng, and Shuai Ma. 2021. Approximate Order Dependency Discovery. In *ICDE*. 25–36.

[22] Youri Kaminsky, Eduardo H. M. Pena, and Felix Naumann. 2023. Discovering Similarity Inclusion Dependencies. *Proc. ACM Manag. Data* 1, 1 (2023), 75:1–75:24.

[23] Reza Karegar, Parke Godfrey, Lukasz Golab, Mehdi Kargar, Divesh Srivastava, and Jaroslaw Szlichta. 2021. Efficient Discovery of Approximate Order Dependencies. In *EDBT*. 427–432.

[24] Aamod Khatiwada, Roee Shraga, Wolfgang Gatterbauer, and Renée J. Miller. 2022. Integrating Data Lake Tables. *Proc. VLDB Endow.* 16, 4 (2022), 932–945.

[25] A. Koeller and E.A. Rundensteiner. 2003. Discovery of high-dimensional inclusion dependencies. In *ICDE*. 683–685.

[26] Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. 2022. Data dependencies for query optimization: a survey. *VLDB J.* 31, 1 (2022), 1–22.

[27] Sebastian Kruse and Felix Naumann. 2018. Efficient Discovery of Approximate Dependencies. *PVLDB* 11, 7 (2018), 759–772.

[28] Sebastian Kruse, Thorsten Papenbrock, Christian Dullweber, Moritz Finke, Manuel Hegner, Martin Zabel, Christian Zöllner, and Felix Naumann. 2017. Fast Approximate Discovery of Inclusion Dependencies. In *BTW*. 207–226.

[29] Sebastian Kruse, Thorsten Papenbrock, and Felix Naumann. 2015. Scaling Out the Discovery of Inclusion Dependencies. In *BTW*, Vol. P-241. 445–454.

[30] Mark Levene and Millist W. Vincent. 2000. Justification for Inclusion Dependency Normal Form. *IEEE Trans. Knowl. Data Eng.* 12, 2 (2000), 281–291.

[31] Ester Livshits, Alireza Heidari, Ihab F. Ilyas, and Benny Kimelfeld. 2020. Approximate Denial Constraints. *PVLDB* 13, 10 (2020), 1682–1695.

[32] Shuai Ma, Wenfei Fan, and Loreto Bravo. 2014. Extending inclusion dependencies with conditions. *Theor. Comput. Sci.* 515 (2014), 64–95.

[33] Fabien De Marchi, Stéphane Lopes, and Jean-Marc Petit. 2009. Unary and n-ary inclusion dependency discovery in relational databases. *J. Intell. Inf. Syst.* 32, 1 (2009), 53–73.

[34] Fabien De Marchi and Jean-Marc Petit. 2003. Zigzag: a new algorithm for mining large inclusion dependencies in database. In *ICDM*. 27–34.

[35] Renée J. Miller, Mauricio A. Hernández, Laura M. Haas, Ling-Ling Yan, C. T. Howard Ho, Ronald Fagin, and Lucian Popa. 2001. The Clio Project: Managing Heterogeneity. *SIGMOD Rec.* 30, 1 (2001), 78–83.

[36] Shaabani Nuhad and Christoph Meinel. 2016. Detecting Maximum Inclusion Dependencies without Candidate Generation. In *Database and Expert Systems Applications*. 118–133.

[37] Thorsten Papenbrock, Tanja Bergmann, Moritz Finke, Jakob Zwiener, and Felix Naumann. 2015. Data Profiling with Metanome. *Proc. VLDB Endow.* 8, 12 (2015), 1860–1863.

[38] Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. 2015. Divide & Conquer-based Inclusion Dependency Discovery. *Proc. VLDB Endow.* 8, 7 (2015), 774–785.

[39] Eduardo H. M. Pena, Eduardo Cunha de Almeida, and Felix Naumann. 2019. Discovery of Approximate (and Exact) Denial Constraints. *PVLDB* 13, 3 (2019), 266–278.

[40] Chaoqin Qian, Menglu Li, Zijing Tan, Ai Ran, and Shuai Ma. 2023. Incremental discovery of denial constraints. *VLDB J.* 32, 6 (2023), 1289–1313.

[41] Hemant Saxena, Lukasz Golab, and Ihab F. Ilyas. 2019. Distributed Implementations of Dependency Discovery Algorithms. *PVLDB* 12, 11 (2019), 1624–1636.

[42] Nuhad Shaabani and Christoph Meinel. 2015. Scalable Inclusion Dependency Discovery. In *DASFAA*. 425–440.

[43] Nuhad Shaabani and Christoph Meinel. 2017. Incremental Discovery of Inclusion Dependencies. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management, Chicago, IL, USA, June 27-29, 2017*. ACM, 2:1–2:12.

[44] Nuhad Shaabani and Christoph Meinel. 2018. Improving the Efficiency of Inclusion Dependency Detection. In *CIKM*. 207–216.

[45] Nuhad Shaabani and Christoph Meinel. 2019. Incrementally updating unary inclusion dependencies in dynamic data. *Distributed Parallel Databases* 37, 1 (2019), 133–176.

[46] Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. 2018. Effective and complete discovery of bidirectional order dependencies via set-based axioms. *VLDB J.* 27, 4 (2018), 573–591.

[47] Fabian Tschirschnitz, Thorsten Papenbrock, and Felix Naumann. 2017. Detecting Inclusion Dependencies on Very Many Tables. *ACM Trans. Database Syst.* 42, 3 (2017), 18:1–18:29.

[48] Jesse Vig, Shilad Sen, and John Riedl. 2012. The Tag Genome: Encoding Community Knowledge to Support Novel Interaction. *ACM Trans. Interact. Intell. Syst.* 2, 3 (2012), 13:1–13:44.

[49] Renjie Xiao, Zijing Tan, Haojin Wang, and Shuai Ma. 2022. Fast approximate denial constraint discovery. *Proc. VLDB Endow.* 16, 2 (2022), 269–281.

[50] Yi Zhang and Zachary G. Ives. 2020. Finding Related Tables in Data Lakes for Interactive Data Science. In *SIGMOD*. 1951–1966.

[51] Erkang Zhu, Fatemeh Nargesian, Ken Q. Pu, and Renée J. Miller. 2016. LSH Ensemble: Internet-Scale Domain Search. *Proc. VLDB Endow.* 9, 12 (2016), 1185–1196.