Efficient Discovery of Temporal Inclusion Dependencies in Wikipedia Tables

Leon Bornemann
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
leon.bornemann@hpi.de

Tobias Bleifuß
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
tobias.bleifuss@hpi.de

Dmitri V. Kalashnikov
Unaffiliated
USA
dmitri.vk@acm.org

Fatemeh Nargesian
University of Rochester
Rochester, New York, USA
fnargesian@rochester.edu

Felix Naumann
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
felix.naumann@hpi.de

Divesh Srivastava
AT&T Chief Data Office
USA
divesh@research.att.com

ABSTRACT

Inclusion dependencies (INDs) demand that the value set that appears in one attribute is contained in the value set that appears in a different attribute. The automatic discovery of INDs in static data is a well-researched topic with many use-cases, such as foreign key discovery. However, data is usually not static, in fact data changes frequently, especially on Wikipedia. The availability of change data allows us to take a fresh look at the discovery of INDs in Wikipedia tables, by taking into account not only the current state of a dataset, but also its past versions.

In this work, we formally define the concept of temporal INDs (tINDs) and introduce several relaxations, allowing for the discovery of tINDs in dirty data. We present an efficient index structure for unary tIND search that returns all valid tINDs for a user query in 63 milliseconds on average, allowing users to interactively explore tIND relationships in Wikipedia tables. Furthermore, we can use our index to discover the set of all valid tINDs between 1.3 million attributes from Wikipedia tables in less than three hours. Finally, we show empirically, that tIND discovery can help to find genuine INDs much more reliably than IND discovery on static data.

1 INTRODUCTION

An inclusion dependency is a statement about the relationship between two attributes, usually of different relations. Given two relation instances R_1 and R_2 in a relational database, a traditional (unary) inclusion dependency (IND) $R_1[A] \subseteq R_2[B]$ states that the set of tuples in the projection of R_1 on attribute A is a subset of the set of tuples in the projection of R_2 on attribute B. The extension to n-ary INDs simply extends the projection to lists of attributes. Typically, $R_1[A]$ is called the left-hand-side and $R_2[B]$ the right-hand-side of the IND.

Knowledge about INDs in a dataset is useful, because INDs are a prerequisite for foreign key dependencies, which are instrumental in understanding and querying databases. Furthermore, INDs can be of use for data integration [12] or query optimization [15]. Thus, their automatic discovery is a well-known problem. The many different solution approaches need to deal with a search space, that grows quadratically for unary INDs and exponentially for n-ary INDs in the number of attributes that are considered [10].

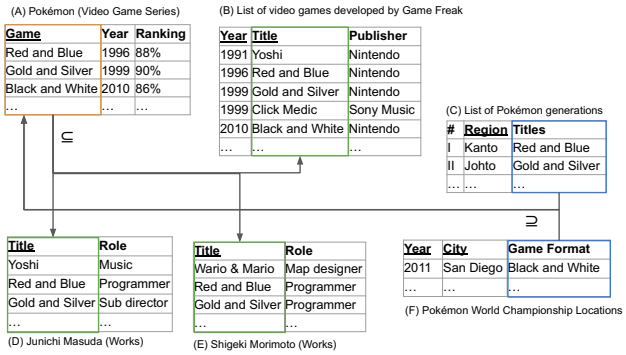

Figure 1: INDs found in tables on Wikipedia on the topic of Pokémon.

Several existing approaches are able to detect unary and n-ary INDs in relational databases [1]. Furthermore, the detection of INDs in other data sources, such as tables on Wikipedia [22] or open government data [25] has been studied. In data sources like Wikipedia, IND detection is particularly challenging due to the vast number of tables. Yet, it is also particularly useful, because in contrast to relational databases, users are unlikely to know on which pages relevant information about a particular entity is located. For example, a quick search reveals that six different relational tables about the Pokémon video games are distributed across six different Wikipedia pages. Figure 1 shows these tables and how they are linked by INDs. To discover such linkings between tables, we address the following *IND search* scenario:

Given a table, such as Table (A) from Figure 1, a user interactively explores the data and wants to expand the information for the set of entities in attribute Q, for example `Game`. This triggers the search for INDs in the dataset, where Q is the left-hand, included side. The results indicate which other tables may hold the information that they require. In the above example, this would be Tables B, D and E. The results should arrive rapidly, so that users do not have to wait to continue their exploration process. Given an efficient solution to the IND search problem, it is easy to see how the user experience in Wikipedia could be enhanced by providing links to the connected tables.

While there has been prior work on the discovery of inclusion dependencies in Wikipedia tables by Tschirschnitz et al. [22], that work suffers from two drawbacks. First, it assumes to be given a static snapshot of data, thus implicitly assuming that the data does not change, which is not the case in practice. Second, it yields many INDs that are not *genuine*, meaning they only hold by chance and do not express real-world constraints. Our experiments revealed that even on a prefiltered set of attributes, only 11% of the INDs found by Tschirschnitz et al. are genuine.

Our work addresses these shortcomings by discovering *temporal inclusion dependencies* (tINDs) in Wikipedia tables. We formally define different variants of tINDs in Section 3 but give a brief overview here. Intuitively, a (strict) temporal unary inclusion dependency between two attributes is valid if the inclusion is satisfied across all of the observed time. Because there are specific data quality issues that arise when analyzing temporal data, we also introduce several relaxed variants to tINDs. The introduced relaxations allow for briefly occurring errors in the data as well as temporal delays between updates. Naturally, if an IND is genuine, it can be expected to hold not just at the current snapshot, but also for most of the prior time. Thus, it is unsurprising that our experiments also show that (relaxed) tINDs can help to find genuine INDs with significantly higher precision. Like prior work [22], we focus explicitly on tables in Wikipedia, as our use-case and our approach is tailored to this setting: it can handle many tables containing only a few rows as opposed to a few tables containing many rows (like in database systems).

Our contributions are as follows:

- Definitions of several variants of relaxed temporal inclusion dependencies (tINDs).
- A novel index structure and discovery algorithm to efficiently search for valid relaxed tINDs for a given attribute (the search query).
- An experimental evaluation of our approach on tables extracted from Wikipedia containing 1.3 million attributes. Our approach returns results for a single search query in 63 milliseconds on average, and is able to discover all tINDs in the dataset in less than 3 hours.
- An evaluation on a manually annotated set of 900 discovered INDs, showing that tIND discovery can find genuine INDs with a precision of up to 50%, whereas static IND discovery only achieves 11% precision.

The rest of the paper is organized as follows. Section 2 discusses related work, and Section 3 formally introduces different variants of (relaxed) temporal INDs. Subsequently, Section 4 presents our algorithm for efficient discovery of relaxed temporal INDs. Finally, we evaluate our methods in Section 5 and conclude in Section 6.

2 RELATED WORK

Because the discovery of inclusion dependencies is a well established research area with broad applications, there are many related approaches that focus on different variants of the problem. There are several approaches that solve the classical problem of n-ary IND discovery in relational databases [4, 9, 19]. While most approaches are run on a single machine, Kruse et al. present a distributed algorithm to solve IND discovery [18]. There is also an approximate IND discovery approach called FAIDA which trades off correctness against speed [17]. The efficiency of IND discovery algorithms in relational databases has been experimentally compared in a study by Dürsch et al. [10]. All of the above mentioned approaches consider only static data and are thus related, but not directly applicable to our scenario.

In the context of temporal databases, the temporal extension of dependencies, such as functional dependencies [11] or INDs [7, 13] has been conceptually studied. However, their existance is assumed to be known and their discovery is not discussed. Shaabani et al. consider the problem of incrementally discovering and updating INDs [20, 21] as the data evolves. While the notion of dynamic data is particularly related to our work, Shaabani et al. only maintain INDs that are valid at the current state of the data and do not consider past data. Furthermore, the approaches by Shaabani et al. only discover strict INDs and thus do not allow any relaxation.

In static data, several relaxations to INDs have been introduced: Conditional INDs are INDs that do not need to hold for all tuples, but only for those that fulfill a certain predicate. Conditional INDs have been studied in both relational databases [3], as well as RDF data [16]. Partial INDs relax the degree to which the left-hand-side needs to be contained in the right-hand side. Zhu et al. refer to this problem as domain search and present an approach to discover unary partial INDs based on a threshold, indicating the degree of this relaxation, in open-government data [25]. Later, various solutions and index structures were proposed for solving the top-k version of this discovery problem [23, 24]. A recently suggested relaxation are similarity INDs (sINDs), which require that every value in the left-hand side needs a corresponding value on the right-hand side that it is similar enough to, according to a distance function. Kaminsky et al. present SAWFISH as a solution to discover sINDs in relational databases [14].

The most closely related work is MANY [22], which finds unary inclusion dependencies in tables on Wikipedia. Because of the focus on tables in Wikipedia, MANY is able to handle a particularly large set of attributes. To compactly represent all attributes and efficiently test for subset relationships, MANY employs a matrix of Bloom filters. While we face a similar number of attributes, MANY cannot be used out of the box because it does not consider temporal data and does not allow for any relaxation. We do however reuse some of the ideas of MANY, which is why we explain its core functionality in more detail in Section 4.1.

3 MODELLING TEMPORAL INCLUSION DEPENDENCIES

We first introduce the notation used in this paper to define temporal inclusion dependencies, after which we formally introduce the problem of discovering them efficiently.

3.1 Preliminaries and notation

We denote the input set of attributes of a database among which we want to discover temporal INDs as $\mathcal{D}$ and use capital letters, such as A, B, and Q to denote individual attributes of that set. We model time as a sequence of equidistant timestamps $\mathcal{T} = [t_1, t_2, ..., t_n]$. For brevity of notation, we also refer to timestamps using their index and use integers to specify durations. Additionally, we overload the interval notation of an interval $I = [s, e]$ to also denote the set of all timestamps in that interval: $I = \{t_s, \ldots, t_e\}$. For convenience, we denote $I.s = s$ and $I.e = e$ to refer to start and end times of an interval.

Given a timestamp t, we denote the set of values contained in attribute A at that timestamp t by $A[t]$. If there is no change to attribute A at timestamp t, then $A[t] = A[t-1]$. If attribute A is not observable at timestamp t (for example because A or its

Notation	Meaning
$\mathcal{D}$	entire set of attributes
A, B, Q	attributes $\in \mathcal{D}$
$\mathcal{T} = [t_1, ..., t_n]$	entire observable time period
I	time interval
$I.s$ / $I.e$	start / end time of interval I
$A[t]$	values of A at timestamp t
$A[I]$	values of A in interval I
V_A	timestamps where A changed (versions of A)
ε	maximal violation of a valid tIND (in terms of timestamps or summed weight)
δ	maximum allowed temporal shift
$w(t)$	weight function for timestamps
α	base for an exponential decay function ($\alpha \in [0, 1]$)

Table 1: Key notations used in this paper

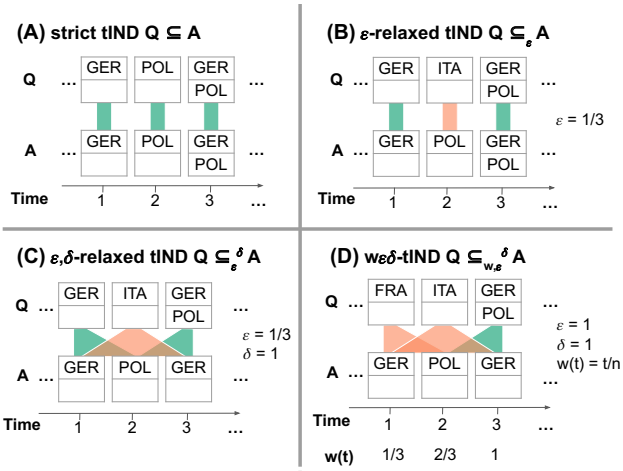

Figure 2: Different tIND variants. For timestamp t, the corresponding value sets $Q[t]$ and $A[t]$ are shown in the respective table columns above the timestamp.

relation was deleted), we define $A[t] = \emptyset$ (empty set). Whenever there is a change to A at a timestamp t (meaning $A[t] \neq A[t-1]$), then we say that there is a new version of A at timestamp t. We define versions of attribute A as $V_A = \{t | t \in \{1, .., n\} \wedge A[t] \neq A[t-1]\} \cup \{0\}$ to denote all timestamps at which a new version of A appears. Given a time interval $I = [s, e]$, we denote $A[I] = \bigcup_{t \in [s,e]} A[t]$. The most important of the above notations are summarized in Table 1.

3.2 Initial definitions

We first define a *static inclusion dependency* (IND) between two attributes A and B:

Definition 3.1 (static IND). Given two attributes A and B and a timestamp t, we say that A and B are in a static inclusion dependency (IND) at timestamp t if $A[t] \subseteq B[t]$.

A static IND is a traditional unary inclusion dependency on static data, as described in related work [1]. This can be extended to include the version history of attributes in a straightforward manner:

Definition 3.2 (strict temporal IND). Given two attributes A and B, we say that A and B are in a strict temporal inclusion dependency (strict tIND) if $\forall t \in \mathcal{T}\ A[t] \subseteq B[t]$, i.e., the inclusion dependency is valid for the entire observable time period.

Definition 3.2 is equivalent to that of temporal INDs as defined in literature on temporal databases, with the exception that Definition 3.2 consider only unary temporal INDs, whereas related work also considers n-ary temporal INDs [7]. An example of a strict tIND is visualized in Figure 2 (A). Inclusion must hold at all timestamps for a strict tIND to be valid. While some strict tINDs may be found in very well-behaved data, such as in a relational DBMS, it is unrealistic to expect that INDs will never be violated in practice, especially in data sources like Wikipedia where updates to different tables are not synchronized and do not underly transactional semantics. To prevent brief inconsistencies from invalidating INDs, the next section introduces relaxed variants of strict tINDs.

3.3 Relaxed temporal inclusion dependencies

There are two main types of data quality issues that we observed in Wikipedia and consider in this work: erroneous updates and temporal delays. Erroneous updates, such as the insertion of incorrect values into the left-hand side attribute, are usually fixed soon after their occurrence and occur very frequently on Wikipedia. For example, one can easily imagine that in Table A from Figure 1 a user mistakenly adds the Pokémon Trading Card Game[1], which is a spin-off title and *not* part of the series. Such data quality issues can be accounted for by allowing a candidate IND to be violated in a fraction of the timestamps ε:

Definition 3.3 (ε-relaxed tIND). Given two attributes A and B, we say that A and B are in an ε-relaxed tIND, denoted as $A \subseteq_\varepsilon B$, if the share of timestamps t where A is not contained in B is at most ε:

$$\frac{|\{t \in \mathcal{T} \mid A[t] \not\subseteq B[t]\}|}{|\mathcal{T}|} \leq \varepsilon \quad (1)$$

An example of an ε-relaxed tIND is visualized in Figure 2 (B). Even though $Q[2] \not\subseteq A[2]$, the tIND is still valid, because $\varepsilon = 1/3$ allows violation in a third of the timestamps.

The second issue that frequently occurs in temporal data is that of temporal delays. As an example, consider Tables A, D and E from Figure 1. If a new Pokémon game is announced it is reasonable to assume that it will be quickly added to Table A. However, it may take a few days or even weeks until it is clarified that both Junichi Masuda and Shigeki Morimoto have once again contributed to the new game, thus delaying the updates of Tables D and E and the restoration of the IND. In general, it is unrealistic to expect the data in Wikipedia to change perfectly in sync, as different pages are maintained by different users who may work in different intervals. To define a second relaxation that allows for such shifts in time, we first define the notion of δ-containment, which specifies a time-interval for the referenced column:

Definition 3.4 (δ-containment). Given two attributes A and B and a timestamp t, we say that $A[t]$ is δ-contained in B, denoted as $A[t] \subseteq^\delta B$, if $A[t] \subseteq B[I]$, where $I = [t - \delta, t + \delta]$.

With the help of δ-containment, we define relaxed temporal inclusion dependencies that also allow for temporal shifts:

Definition 3.5 (ε, δ-relaxed tIND). Given two attributes A and B, we say that A and B are in an ε, δ-relaxed tIND, denoted as

[1] https://en.wikipedia.org/wiki/PokÃĭmon_Trading_Card_Game_(video_game)

$A \subseteq^{\delta}_{\varepsilon} B$, if the share of timestamps t where A is not δ-contained in B is at most ε:

$$\frac{|\{t \in \mathcal{T} \mid A[t] \not\subseteq^{\delta} B\}|}{|\mathcal{T}|} \leq \varepsilon \tag{2}$$

An example of an ε, δ-relaxed tIND is visualized in Figure 2 (C). Even though $Q[3] \not\subseteq A[3]$ the tIND is not violated at timestamp 3, because A contains the necessary missing value POL in the temporally close position $A[2]$.

There is another potential data quality issue that neither ε nor δ addresses, which is the usage of different entity names for the same entity (such as `USA` and `United States`) in different attributes over long periods of time. Conceptually, this could be solved analogously to the work by Zhu et al. [25], which solves approximate set containment. This would add another parameter to specify the degree to which the left-hand side must be contained in the right-hand side at every timestamp for the tIND to hold. We refrain from doing so in this work for two reasons: First, there is a preprocessing step that can greatly reduce these issues in Wikipedia tables (see Section 5.1). Second, the main use-case for tIND discovery is to find genuine INDs with higher precision. While allowing approximate containment would boost recall, precision would likely suffer, because many new false positives would be found. Thus, we leave this additional relaxation to future work.

A remaining issue with Definitions 3.3 and 3.5 is that all timestamps are given equal importance. In practice, this approach is frequently undesirable, as data in the distant past is often less relevant than recent data. For example, one can observe that earlier revisions of Shigeki Morimoto's page (Table E in Figure 1) were maintained infrequently, which may have lead to the IND being invalid for longer periods of time in the past. To account for such issues, we introduce the notion of a weighting-function $w\colon \mathcal{T} \to \mathbb{R}$ that can assign individual weights to timestamps. Intuitively, a higher weight for a timestamp t means that t is more important, and a violation at t contributes more towards the overall violation budget (ε) than violations at timestamps with lower weights. This leads to our most general definition of temporal INDs:

Definition 3.6 ($w\varepsilon\delta$-tIND). Given two attributes A and B, we say that A and B are in a w-weighted ε, δ-relaxed tIND ($w\varepsilon\delta$-tIND), denoted as $A \subseteq^{\delta}_{w,\varepsilon} B$, if the sum of the weights of all timestamps where A is not δ-contained in B is at most ε:

$$\sum_{t \in T_V} w(t) \leq \varepsilon, \text{ where } T_V = \{t \in \mathcal{T} \mid A[t] \not\subseteq^{\delta} B\} \tag{3}$$

An example of a $w\varepsilon\delta$-tIND is visualized in Figure 2 (D). Even though the $w\varepsilon\delta$-tIND is violated at two timestamps, their summed weight remains at 1, which is still an allowed violation.

Note that for $w\varepsilon\delta$-tINDs, ε is now an absolute threshold, as opposed to a relative one, as in Definitions 3.3 and 3.5. In principle, w can be set arbitrarily. However, depending on the granularity at which timestamps are available, it may be computationally prohibitive to compute w for every timestamp. Thus, we recommend functions where the sum of weights in an interval $\sum_{t \in [i,j)} w(t)$ can be computed efficiently (ideally in $O(1)$). Popular choices for analysis of temporal or streaming data are exponential decay functions [8] to model that data in the distant past is less important than more recent data:

$$w(t_i) = \alpha^{n-i}, \text{ where } \alpha \in (0, 1] \text{ and } n = |\mathcal{T}| \tag{4}$$

For such functions, the sum of weights in an interval $[i, j)$ can be computed in $O(1)$ using the closed formula of the geometric sum.

$$\sum_{t \in [i,j)} w(t) = \sum_{t \in [i,j)} \alpha^{n-t} = \frac{\alpha^n(\alpha^{-j} - \alpha^{-i})}{1 - \alpha} \tag{5}$$

Naturally, users could also use a linear decay function or even a custom function that might disregard certain time periods entirely, for example because of known data quality issues during them. Thus the choice of w gives the users a lot of freedom on how different timestamps or time periods should be weighted.

It is noteworthy that each of the above temporal IND variants are a generalization of their predecessors and conversely a specialization of their successor. Specifically, ε, δ-relaxed tINDs model $w\varepsilon\delta$-tINDs with the special case of w being a constant function, that assigns a weight of $\frac{1}{n}$ to every timestamp, while ε-relaxed tINDs additionally fix $\delta = 0$. Lastly, strict tINDs also set $\varepsilon = 0$. This means that an approach that efficiently discovers $w\varepsilon\delta$-tIND can also discover all other types of tINDs that we defined above. Thus, the problems that we solve in this paper are the search and discovery problems for $w\varepsilon\delta$-tIND as detailed in Section 3.5. For the remainder of this paper, when we speak of temporal inclusion dependencies (tINDs), we mean $w\varepsilon\delta$-tIND unless stated otherwise. While it may be challenging for users to set the three parameters to suit their use-case, we mitigate this issue to some extent: first, we achieve very low latency for individual user queries as shown in Section 5. Second, a single index structure that we use to solve efficient tIND search can support a wide range of parameter settings. Thus, users can quickly try out many different parameter settings and converge to a useful setting for a specific scenario.

3.4 Theoretical properties of (relaxed) tINDs

While theoretical properties are not the focus of this paper, we briefly discuss them in the following. The work by Bronselaer et al. discusses reflexivity, projection, permutation and transitivity of strict n-ary tINDs [7]. While permutation and projection are irrelevant for unary tINDs as discussed in this paper, reflexivity and transitivity are directly applicable. It is easy to see that reflexivity also holds for all variants of relaxed temporal INDs defined above. However, standard transitivity no longer holds, because the violations do not need to be temporally aligned. As an example, consider again ε-relaxed tIND in Figure 2 (B). Now imagine a third attribute B with $B^1 = \{ITA\}$, $B^2 = \{POL\}$ and $B^3 = \{GER, POL\}$. It is clear to see that $Q \subseteq_{1/3} A$ and $A \subseteq_{1/3} B$ holds, but $Q \subseteq_{1/3} B$ does not hold, showing that relaxed tINDs are not transitive.

3.5 tIND discovery: problem variants

The interactive discovery of tINDs comes in two variants. In the first variant, which we call tIND search, we are searching for all attributes that a query attribute is contained in:

Definition 3.7 (tIND search). Given a query attribute Q, a set of attributes $\mathcal{D}$, as well as the parameters ε, δ and w, (efficiently) find all $A \in \mathcal{D}$ where $Q \subseteq^{\delta}_{w,\varepsilon} A$.

The second variant is the reverse tIND search, where we search for attributes that are contained in the query.

Definition 3.8 (reverse tIND search). Given a query attribute Q, a set of attributes $\mathcal{D}$, as well as the parameters ε, δ and w, (efficiently) find all $A \in \mathcal{D}$ where $A \subseteq^{\delta}_{w,\varepsilon} Q$.

Because the search for attributes in which a query attribute is contained is more common in related work [25], it is also the focus of our work. We will, however, show that our solution can also solve reverse tIND search with only a slight increase in runtime.

As the use-case for both tIND search variants is interactive exploration by a user, results for both variants should be returned within at most a few seconds. This is different for the *all-pairs* variant of the problem, in which one is given a set of attributes $\mathcal{D}$, as well as the parameters ε, δ and w, and the task is to efficiently find all tINDs $A \subseteq^{\delta}_{w,\varepsilon} B$ with $A, B \in \mathcal{D}$. Naturally, the all-pairs problem can be solved by iteratively querying all $A \in \mathcal{D}$ against $\mathcal{D}$ if an efficient solution to the search problem exists. Thus, we present an efficient algorithm to solve tIND search and reverse tIND search in the next section.

4 EFFICIENT SEARCH FOR TINDS

Our solution to efficiently search for tINDs based on a query attribute Q uses multiple index structures that reuse existing techniques from MANY [22], a solution to the static IND discovery problem on many tables. In the following, we assume, for simplicity, that ε, w and δ are known beforehand, and we can build the index optimally for these parameters. This does not need to be the case in practice: we can also build the index without knowledge about what values queries will use for ε or w. However, we do need to know the maximum δ that queries will use. The runtime impact of using suboptimal values for ε, w, and δ is shown to be low, as discussed in Section 5.3. We first briefly summarize the core idea of the candidate search of MANY [22] in Section 4.1. Subsequently, we present a solution for tIND search in Sections 4.2 to 4.4. Finally, we describe how the existing indices for tIND search can be used to also solve the reverse search problem in Section 4.5.

4.1 Candidate search in MANY

MANY [22] solves the problem of candidate search by using a hash function h that transforms the value set of an attribute A into a Bloom filter [6] of size m. A Bloom filter is a bit vector $h(A) = [b_1, ...b_m]$, $b_i \in \{0, 1\}$, designed for fast set membership and containment tests. Its main feature is that the hash function preserves subset relationships: If $A \subseteq B$ and the i-th bit in $h(A)$ is set, then the i-th bit must also be set in $h(B)$. To find all candidates for a particular query efficiently, MANY represents the set of attributes $\mathcal{D}$ as a bit-matrix, where the i-th column corresponds to the Bloom filter of the i-th attribute in $\mathcal{D}$. The example in Figure 3 illustrates the candidate search in this matrix. To find all potential supersets of the query attribute Q, one needs to consider only all rows where $h(Q) = 1$. The logical conjunction (bitwise-and) of these rows then forms the set of candidate columns that may contain Q. Note that one can also use this matrix to find all subsets of a query attribute: To find all attributes that are contained in a query attribute Q, one considers all rows where $h(Q) = 0$. The logical conjunction of the inverse (bitwise-negation) of these rows then forms the set of candidate columns that may be contained by Q.

4.2 An index structure for tIND search

While the problem of finding tINDs is more complex than checking for set containment, we can use the absence of certain set containments to prune candidates. Thus, we re-use the basic idea of MANY by creating several different value sets, for which we build Bloom filter matrices that we can then use to narrow down the candidates for a query. We propose two pruning mechanisms: *required values* and *time slice indices*.

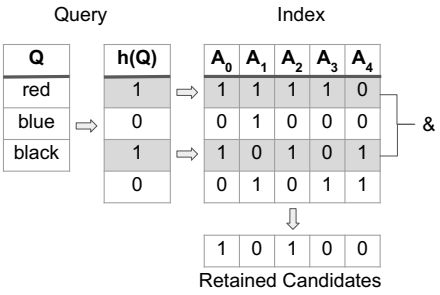

Figure 3: Candidate search in MANY.

4.2.1 Required values. If a value v appears in Q for a set of timestamps whose weights sum up to more than ε, then for the tIND $Q \subseteq^{\delta}_{\varepsilon,w} A$ to be valid, v needs to also be present in A. Let the summed occurrence weight (w_v) of a value v in attribute Q be defined as the sum of the weights of timestamps in which v occurs in Q:

$$w_v(Q) = \sum_{t \in \{t | v \in Q[t]\}} w(t) \tag{6}$$

Then the *required values* for an attribute Q are defined as:

$$RQ_{\varepsilon,w}(Q) = \{v | w_v(Q) > \varepsilon\} \tag{7}$$

It is clear to see that $Q \subseteq^{\delta}_{w,\varepsilon} A$ implies $RQ_{\varepsilon,w}(Q) \subseteq A[\mathcal{T}]$: for all $v \in RQ_{\varepsilon,w}(Q)$, there must be at least one version where A contains v. Thus, for every attribute $A \in \mathcal{D}$, we build a Bloom filter on $A[\mathcal{T}]$ and assemble them into a matrix $M_{\mathcal{T}}$ as detailed in Section 4.1. For any query attribute Q, we can then query $RQ_{\varepsilon,w}(Q)$ against this matrix and prune all candidates that do not contain the required values. Note that $M_{\mathcal{T}}$ can be constructed without knowledge about any of the three parameters w, ε, or δ.

4.2.2 Time slice indices. While the use of the matrix $M_{\mathcal{T}}$ is good for an initial pruning step, it checks subset containment only across the entire time period without taking the time at which values appear into account. To remedy this, we also index multiple smaller time slices to check for containment in a more localized fashion.

Given an interval $I = [t_s, t_e]$, let us for simplicity first assume that for a query attribute Q there is only one distinct version within an interval I, meaning $Q[t_s] = Q[t_s + 1] = ... = Q[t_e]$. Furthermore, let us assume that $\sum_{t \in I} w(t) > \varepsilon$ and let $I^{\delta} = [t_s - \delta, t_e + \delta]$. If for a candidate right-hand side A we observe that $Q[t] \notin A[I^{\delta}]$, then we can safely prune A as a candidate right-hand side, because the tIND must be violated in the entire interval I, whose summed weight is more than ε and thus more than the allowed violation.

Naturally, there may be several distinct versions of Q within I, some of which may be contained in $A[I^{\delta}]$ and some of which might not. Furthermore, depending on the settings of ε or w at query time, the summed weight of all timestamps in I may be smaller than ε. In both cases, we cannot immediately prune A as a candidate. Therefore, we use multiple time slice indices built on selected disjoint intervals $I_1, ..., I_k$ and track the total violations that we have seen per candidate right-hand side. Whenever a new violation is found, we check whether we have exceeded ε and can thus prune the candidate.

Tracking violations for all candidate right-hand sides is feasible, because querying $M_{\mathcal{T}}$ already prunes most of the search space, so that not too many candidates remain. An example of how the pruning via time-slice indices functions is visualized in Figure 4. In the example, attribute A is indexed on two time intervals. When querying Q against the time slice indices, two separate (partial) violations are detected at timestamps 3 and 7, because in both cases the string USA is not contained in any version of A within δ timestamps. After the second violation, A can be pruned as a candidate, because the maximally allowed violation is exceeded.

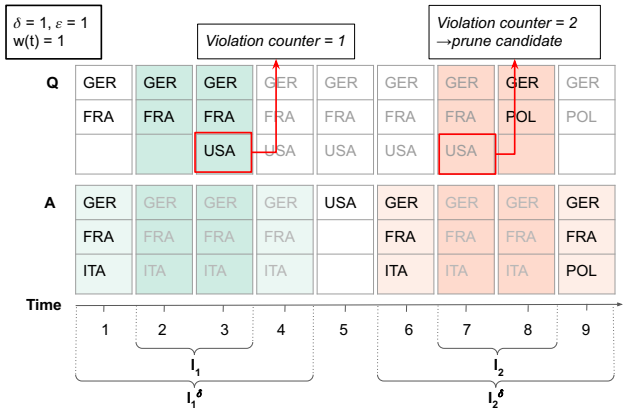

Figure 4: Pruning using time slice indices on intervals I_1 (left, green) and I_2 (right, red). If the state of Q or A is unchanged compared to the previous timestamp, the font is in grey.

The final index structure is then created by chaining $M_{\mathcal{T}}$ with the time slice indices $M_{I_1}, ..., M_{I_k}$. The total memory used by the index structure (in bytes) is $(k{+}1){\cdot}|\mathcal{D}|{\cdot}m{\cdot}1/8$, which means there is a tradeoff between m and k. In general, m should be chosen based on the (average) attribute cardinality, as more values in the attributes require larger Bloom filters for the pruning to work. A larger k is beneficial if the attributes frequently undergo significant changes, because only then will more time slice indices yield more pruning power. We empirically evaluate different settings for both m and k in Section 5.4.

The process of candidate pruning for a query attribute Q is depicted in Figure 5 and the algorithm for tIND search using the indices is given in Algorithm 1. First, the required values for Q are determined and queried against $M_{\mathcal{T}}$ to obtain an initial set of candidates. Then, for every interval I_i, all distinct versions of Q within that interval are determined (line 6) and queried against M_{I_i} (lines 7-10). Partial violations are tracked and remaining candidates are pruned if the summed violation exceeds the allowed violation (lines 11-15). Because Bloom filters are a probabilistic data structure where false positives are possible, we subsequently validate for all candidates that the subset relationships of the query Bloom filters with the Bloom filters in the index also hold true for the actual data (line 16). Afterwards, all remaining tIND candidates need to be validated, which can be done in parallel. Note that, in case one solves the all-pairs problem, it is superior to parallelize the calls to *tINDSearch*, meaning to evaluate multiple queries in parallel. We now discuss how to efficiently validate tINDs.

Input : Query attribute Q, indexed attributes $A_1 \cdots A_{|\mathcal{D}|}$, indices $M_{\mathcal{T}}$ and $M_{I1}, ...M_{Ik}$, ε, w, δ
Output: All indexed attributes A where $Q \subseteq^{\delta}_{\varepsilon,w}$ holds

1 **PROCEDURE** *tINDSearch()*:
2 C_0 = BitVector(size=$|\mathcal{D}|$, *fill* = 1);
3 C = *query_index*($M_{\mathcal{T}}, RQ_{\varepsilon,w}(Q), C_0$);
4 VIO = empty_map() ; /* tracks violations */
5 **forall** $i \in \{1, \ldots, k\}$ **do**
6 V_{QI} = *sorted*($\{t | t \in V_Q \wedge t \in I\} \cup \{I_i.end\}$);
7 **forall** $j \in \{0, \ldots, len(V_{QI}) - 1\}$ **do**
8 $begin = V_{QI}[j]$;
9 $end = V_{QI}[j+1]$;
10 C_{ij} = *query_index*($M_{Ii}, Q[begin], C$);
11 $PV = C \wedge \neg C_{ij}$;
12 **forall** $c \in PV$ *where* $PV[c] == 1$ **do**
13 $VIO[c] = VIO[c] + w([begin, end))$;
14 **if** $VIO[c] \geq \varepsilon$ **then**
15 $C[c] = 0$; /* pruned */
16 C = *validate_subset_relationships*(Q, C);
17 **forall** i *where* $C[i] = 1$ ***in parallel*** **do**
18 **if** tINDIsValid($Q \subseteq^{\delta}_{\varepsilon,w} A_i$) **then**
19 $C[c] = 0$
20 **return** C;
21 **PROCEDURE** *query_index*($M, values, C$):
22 C_M = BitVector(size=$|\mathcal{D}|$,*fill* = 1) ;
23 $bf = h(values)$; /* creates Bloom filter */
24 **forall** *rows* r *in* M *where* $bf[r] = 1$ **do**
25 $C_M = C_M \wedge M[r]$
26 **return** $C \wedge C_M$;

Algorithm 1: tIND candidate pruning

4.3 Efficient tIND validation

After using the index to prune the search space, we need to validate if a tIND candidate $Q \subseteq^{\delta}_{\varepsilon,w} A$ holds. A trivial algorithm to validate a tIND is to check δ-containment for every timestamp t and sum up the weights of timestamps where it does not hold. There are two problems with this approach. First, the number of timestamps n may be very large and second, if the attributes Q or A do not change much, a lot of computations are repeated needlessly even though nothing has changed.

We avoid this by iterating over only exactly those timestamps where the δ-containment may have changed, thus avoiding computations if the state of both A and Q did not change. In the following, we assume that the history of an attribute A is stored as a list of versions of A that is sorted by their insert time. Given a candidate tIND $Q \subseteq^{\delta}_{\varepsilon,w} A$, our algorithm can validate it in $\Theta(TV(Q) + TV(A))$, where $TV(A) = \sum_{t \in V_A} |A[t]|$, meaning we only need to look at every version of Q and A once. The algorithm for tIND validation is given in Algorithm 2.

The basic idea of the algorithm is to partition the entire time period $\mathcal{T}$ into intervals $\mathcal{I}$, so that for every $I \in \mathcal{I}$, the following properties hold:

- There is only a single version of Q in I
- $\forall t, t' \in I\ A[[t-\delta, t+\delta]] = A[[t'-\delta, t'+\delta]]$. This implies that if Q is δ-contained in A at any $t \in I$, it is δ-contained in all $t \in I$ and vice-versa.

These two properties ensure that for an interval $I \in \mathcal{I}$, we only have to check δ-containment for the first timestamp, reducing

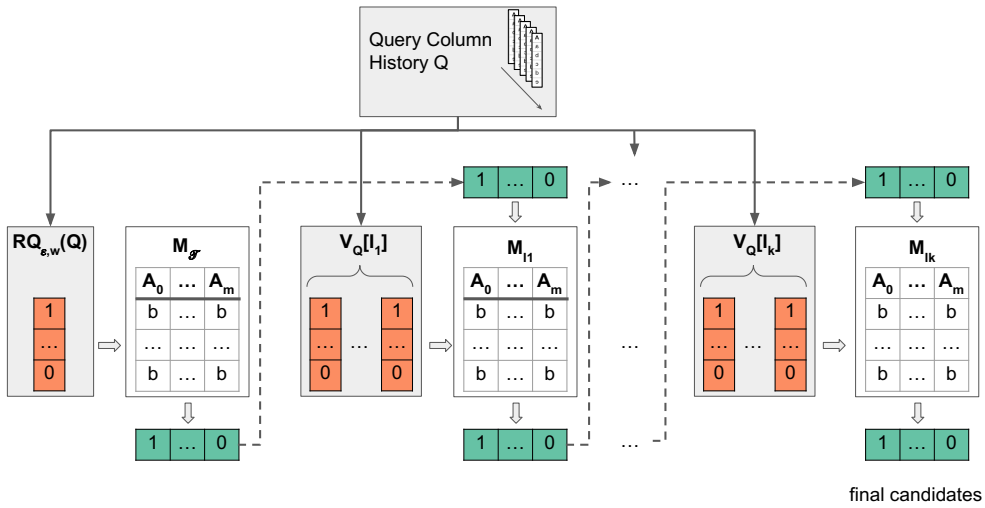

Figure 5: Querying of the index structure. The query Bloom filters are depicted in orange and (intermediate) candidates in green. $V_Q[I_j]$ denotes the Bloom filters of all versions of Q in interval I_j. The indexed attributes are denoted by $A_1 \cdots A_m$.

Input : tIND candidate $Q \subseteq^{\delta}_{\varepsilon,w} A$
Output: True if $Q \subseteq^{\delta}_{\varepsilon,w} A$ is valid, false otherwise
1 **PROCEDURE** *tINDIsValid*($Q \subseteq^{\delta}_{\varepsilon,w} A$):
2 $V^{\delta}_A = \{t | (t + \delta \in V_A \vee t - \delta \in V_A) \wedge t \in \mathcal{T}\}$;
3 $T_{\mathcal{I}} = sorted(V_Q \cup V^{\delta}_A \cup \{n\})$; /* $|\mathcal{T}| = n$ */
4 violation = 0;
5 **forall** $i \in \{0, ..|T_{\mathcal{I}}| - 1\}$ **do**
6 $I = [T_{\mathcal{I}}[i], T_{\mathcal{I}}[i+1])$;
7 $I^{\delta} = [I.begin - \delta, I.end + \delta]$;
8 **if** $Q[I] \not\subseteq A[I^{\delta}]$ **then**
9 violation = violation + $w(I)$;
10 **if** *violation* $> \varepsilon$ **then**
11 **return** False;
12 **return** True;

Algorithm 2: tIND validation

the necessary number of δ-containment checks to $|\mathcal{I}|$. These intervals are constructed by assembling and sorting all timestamps at which δ-containment may change (lines 2-3). The subset-check in line 8 can be implemented without creating $Q[I]$ and $A[I^{\delta}]$ from scratch for every check, by using two sliding windows over the versions of Q and A. This works, because the intervals are traversed in a sorted manner, which means that versions of Q or A that leave their respective sliding window are never needed again.

We now discuss how the parameters of a tIND impact index construction and how to choose suitable time intervals to index on.

4.4 Index parameters and interval selection

While tIND discovery has the three parameters ε, w, and δ, not all of these parameters need to be known exactly during the index construction. This is of course beneficial, because a once constructed index can thus be used for many queries with different parameters. However, queries with strongly deviating parameters may take longer to complete, as the index can prune less efficiently. As already mentioned in Section 4.2.2, the maximum δ that queries will use needs to be known during index construction. That is because, if a query were to use a $\delta' > \delta$, then the time slice indices are built on intervals that consider a smaller δ than what the query allows. This means that a violation detected in those indices no longer guarantees a violation of $A \subseteq^{\delta'}_{w,\varepsilon} B$ which means that none of the time slice indices could be used to prune candidates without possibly introducing false negatives. However, if a query uses a $\delta' \leq \delta$ the time slice indices can still be used to prune candidates, although if $\delta' << \delta$, then the pruning is not as effective as it could be. The example in Figure 4 showcases this. If during index creation δ would have been set to two instead of one, the value USA from timestamp five would have been present in both I^{δ}_1 and I^{δ}_2 for attribute A. This would have left the index unable to prune A as a candidate, even though for $\delta = 1$ (which is what the query uses), A is not a valid right-hand side for Q.

As seen in the previous section, knowledge about future ε, and w parameters of queries are neither required for $M_{\mathcal{T}}$, nor when building the time slice indices on pre-chosen intervals. However, it is beneficial to take potential knowledge about these parameters into account when choosing the intervals to build the index on. When selecting intervals, there are two factors to consider: the interval length and the temporal location of the interval (determined by the interval starting time). We discuss them in the following.

4.4.1 Interval length. The optimal interval length is difficult to determine in theory, because there are many effects to consider: Firstly, smaller intervals lead to smaller sets for the right-hand sides which may lead to better pruning power. Smaller intervals are also more accurate in terms of the temporal containment: Recall that the Bloom filters for the candidates for the right-hand side are created based on the entire value set $A[I^{\delta}]$. Thus, the larger I, the more values will also be in $A[I^{\delta}]$. This makes it more likely that the values of a version of Q ($Q[t]$ with $t \in I$) will be contained in $A[I^{\delta}]$ even though $Q[t]$ is not actually δ-contained in A. This can happen if a value $v \in A[I^{\delta}]$ that is also in $Q[t]$ is only contained in a version of A that is too far away from t (more than δ). In such cases, we cannot prune A based on the information in the index, even though it would be correct to do so.

The example in Figure 4 showcases this effect. If I_2 were enlarged by starting at timestamp six instead of seven, then I_2^δ would start at timestamp five and the value USA would now be contained in $A[I_2^\delta]$. This results in the index not detecting a violation at I_2, leaving the index unable to prune A.

However, smaller intervals are not necessarily superior, because the number of indices that we can build can be expected to be limited due to memory constraints. Thus, if the interval lengths are small, but the total time period is large, a lot of information is not present in the indices. Versions of Q that are critical for pruning some candidates are more likely to not be present within any of the indexed time slices if the time slices are small, leading to worse pruning power and thus worse runtime. Again, we can see this in Figure 4. If the intervals I_1 and I_2 were both of length one instead of two while starting at the same timestamps, then I_1 would no longer contain the version of Q that contains USA, which is crucial for pruning A.

What should always be avoided is the case of interval I being so small, that its summed weight $w(I)$ is smaller than the ε used by a query. This would mean that M_I can only detect partial violations but never prune away candidates with no known prior violation. This is especially disadvantageous if M_I is the first time slice index that is queried, because in that case we need to track violations for all initial candidates obtained from $M_\mathcal{T}$. Doing so would have a significant negative impact on runtime.

Barring this extreme case, both small and large intervals can be either beneficial or disadvantageous. Thus, we have evaluated different intervals lengths empirically. Fortunately, our experiments in Section 5.3 have shown that the index is not particularly sensitive to either large or small intervals. We thus use the setting $w(I) = \varepsilon + 1$ as our standard setting for interval size.

4.4.2 Interval starting time. While choosing the starting time of an interval is more straightforward than finding a suitable interval length, there are still a few effects to consider. First, the weighting function w may lead to the starting time determining the minimal duration that an interval should have. As mentioned above, for any chosen interval I to index on, $w(I) > \varepsilon$ should hold. This means, that if w assigns less weight to timestamps in the past, then an interval that starts at an early timestamp will also need to be longer than an interval that starts at a more recent timestamp.

Independent of that effect, we want to select a set of k intervals, that maximizes the overall pruning power for future queries. Our experiments in Section 5.4 show that selecting a random set of intervals achieves a satisfying performance, especially if k is large. For small k however, one can do better by estimating the pruning power of individual intervals. In general, the pruning power of an interval I heavily depends on two factors:

- How distinguishable are the future queries in I?
- How distinguishable are the indexed attributes $A \in \mathcal{D}$ in I?

While we do not assume to know anything about the value distribution of future queries, we do have this knowledge about the indexed attributes in $\mathcal{D}$. Thus, for an interval $I = [t_s, t_e]$, we can estimate the pruning power of I by counting the number of distinct values across all $A \in \mathcal{D}$ in I and dividing by the length of the interval: $p(I) = \frac{\sum_{A \in \mathcal{D}} |A[I]|}{|I|}$. The higher $p(I)$, the more likely it is that I can prune a lot of candidates.

Given p as a function to assign weights to intervals, we now need to select k intervals. Assuming the interval length is determined as described in Section 4.4.1, we can use p to assign weights to all timestamps: $p(t) = p(I)$ where $I.begin = t$. Note that if $\mathcal{T}$ is too large to compute $p(t)$ for every $t \in \mathcal{T}$, it is always possible to sample from $\mathcal{T}$ at a lower granularity. Given the timestamps $t \in \mathcal{T}$ and their weights $p(t)$, we can draw a weighted random sample of k timestamps by iteratively sampling timestamps from $\mathcal{T}$. The probability $P(t)$ of timestamp t being selected is proportional to its weight: $P(t) = p(t)/\sum_{t' \in \mathcal{T}} p(t')$. We refer to this method of choosing time slices as *weighted random*.

Weighted random favors timestamps at which a lot of data is available. While this is generally beneficial, it can backfire for large values of k by creating redundant indices. Our experiments in section 5.4 show that a weighted random choice is superior to a random choice for small k. However, for larger k, the increased variance of a random choice benefits the pruning, because it makes the creation of redundant indices less likely.

4.5 Reverse tIND search

In the above sections, we have described how to solve tIND search, meaning the search for attributes that a query attribute Q is contained in. As discussed in Section 3.5 it can also be of interest to reverse the direction and search attributes that Q contains. Fortunately, this problem can also be solved with the techniques we outlined above, with some adjustments. As mentioned in Section 4.1, the Bloom filter matrices can also be used to find all potential subsets of a query, as opposed to all supersets. However, we still need to slightly adapt our index structures to make them usable for reverse tIND search queries.

Recall that $M_\mathcal{T}$ indexes the entire value set $A[\mathcal{T}]$ of every attribute $A \in \mathcal{D}$. However, $A \subseteq^\delta_{w,\varepsilon} Q$ implies $RQ_{\varepsilon,w}(A) \subseteq Q[\mathcal{T}]$, which means that $M_\mathcal{T}$ is unhelpful for reverse tIND search. Instead, we build another index M_{RQ}, which indexes the required values $RQ_{\varepsilon,w}(A)$ of every $A \in \mathcal{D}$ given assumed query parameters ε and w. Note that in contrast to the normal tIND search, the maximum ε that queries intend to use now needs to be known at index time. Queries may choose a lower ε than assumed during indexing, but not a higher one, because otherwise valid candidates may be erroneously pruned.

The time slice indices $M_{I_1}, \ldots, M_{I_k}$ can be re-used for reverse tIND search, as long as not just $I_1, ..., I_k$, but also $I_1^\delta, \cdots, I_k^\delta$ are disjoint. Recall that in a time slice index M_I, all attributes $A \in \mathcal{D}$ have been indexed on $A[I^\delta]$. Thus, we can now query the time slice indices for value sets that are contained in $Q[I^{2\delta}]$, meaning we expand I^δ by additional δ timestamps to the left and right for the query. By doing so, we check for all A whether $A[I^\delta] \subseteq Q[I^{2\delta}]$. If that is the case, we cannot detect any violation. If that is not the case, we know that at least one of the versions of A in I^δ is not δ-contained in Q and thus partially violates the tIND.

Unfortunately, we have no way of knowing which exact versions of A in I^δ partially violate the tIND, because we cannot efficiently reconstruct the individual versions in I^δ from its Bloom filter $h(A[I^\delta])$. Thus, we compute the weight for every subinterval of I^δ in which exactly one version of A exists and can only add the minimum of these weights to the violations we have seen so far.

An example of how violations for reverse tIND search are detected in time slice indices is visualized in Figure 6. Despite the original violation being caused by A_2, we cannot infer this from the Bloom filters, and thus we can only infer a minimum violation

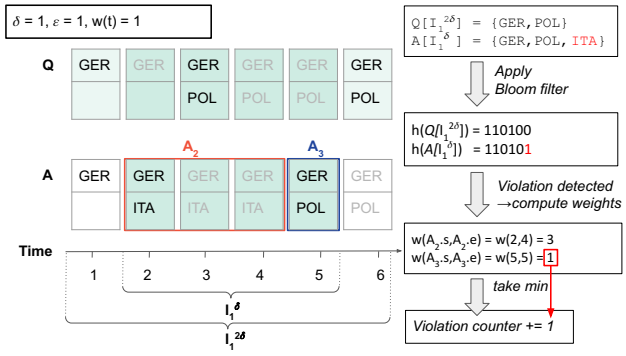

Figure 6: Violation detection for reverse tIND search in a time slice index.

of 1 instead of 3 in I^{δ}. Clearly, this effect can lead to worse pruning power of the time-slice indices than for normal tIND search. Additionally, larger and thus sparser Bloom filters lead to more necessary operations for tIND reverse search, because as explained in Section 4.1 every row where the query Bloom filter is equal to zero needs to be checked. However, dense Bloom filters also lead to more false positives being retained, which means that subset validation (line 16) will take more time.

However, our experiments show that for data on Wikipedia, the overall time that a reverse tIND query takes when using pre-existing indices for normal tIND search is still low enough for interactive exploration.

5 EVALUATION

We evaluate our approach on a set of tables extracted from Wikipedia pages. Our implementations and datasets are publicly available[2]. We give a brief overview of our dataset extraction, preparation and filtering steps in Section 5.1. Subsequently, Section 5.2 presents a general evaluation of our approaches for both the *search* and *reverse search*, as well as the *all-pairs* problem. Section 5.3 presents experiments where the query parameters deviate from those that the index is optimized for. The impact of hyperparameter choices is explored in Section 5.4. Finally, Section 5.5 presents experiments using manually labelled data, that showcase how tIND discovery can be used to detect genuine INDs with high precision.

5.1 Dataset and experimental setup

We evaluate our approach on the historical data that is provided by the Wikimedia Foundation[3], choosing more than 16 years between early 2001 and late 2017. Given the many different Wikipedia page revisions, we used existing work [5] to first extract and match all tables to create table histories and subsequently match different versions of the same attribute to create historical data for each attribute in a table. To reduce the impact of vandalism, which frequently appears in Wikipedia [2], we aggregated the time granularity of the observed data to daily snapshots by setting the version of an attribute at a day to the version that was valid for the longest time on that day.

Similar to prior work on IND discovery in Wikipedia tables [22], we then filtered out attributes that are mostly numeric, and unified commonly used symbols for the null value [22]. Furthermore, for values with hyperlinks to other Wikipedia pages, we replaced the text of the link with the title of the linked page to uniformly represent such links across all tables. Because many entities in Wikipedia tables are linked, this partially solves the data quality issue of differing entity representations as discussed in Section 3.3. Additionally, we filtered out any attributes that do not have at least five different versions, meaning four changes, because we require at least some temporal data to properly evaluate tINDs. Lastly, with our use-case in mind, an explorative evaluation of our dataset has shown that attributes with very small value sets very rarely appear in genuine INDs, which is why we also require attribute histories to have a median value set size of five or more.

These filtering steps result in a dataset of more than 1.3 million attribute histories. On average, an attribute history of our dataset has 13 changes and exists for 5.6 years in total. The average column version has a cardinality of 28. All experiments presented in this section (including the related work approach [22]) were executed on this dataset. Thus, the results are directly comparable.

Unless otherwise specified, we use the following parameter settings for the tIND relaxation in our experiments: $\varepsilon = 3days$, $w(t) = 1$ (constant function) and $\delta = 7days$. We chose these settings, because they achieve the highest recall for a fixed precision of 50% when looking for genuine INDs (see Section 5.5). With the exception of the experiments in Section 5.3, we always use these settings for both indexing and querying, thus assuming accurate knowledge of query needs at index time.

There are three hyperparameters that do not impact the set of discovered tINDs, but affect the runtime of our approach. These are the size of the Bloom filters m, the number of time slice indices k and the method of choosing time slices. As detailed in Section 5.4, setting $m = 4096$ and $k = 16$, as well as randomly choosing time slices achieves the best runtime for tIND search, which is why we use these as our default settings. For reverse tIND search, the highest runtime is achieved by setting $m = 512$ and $k = 2$ and using the weighted random approach to choose time slices. Therefore, we use this as our default setting for reverse tIND search.

We compare our method against the direct application of the closest related work on static data, namely MANY [22]. To adapt MANY for the temporal use-case, we modify it to build k different Bloom filter matrices on randomly chosen snapshots and use these to prune the candidate space. We call this baseline k-MANY. To enable a fair comparison, we always set the k for k-MANY to the number of time slice indices used by tIND search.

All experiments were executed using 32 threads on a server running Ubuntu 20.04 LTS with two Intel Xeon E5-2650 2.00 GHz CPUs and 256 GB RAM.

5.2 General evaluation

Because interactive exploration of datasets via tIND search requires fast execution times, we evaluate the runtime of our approach for different scenarios.

Figure 7 shows the distribution of query times for 30,000 randomly chosen queries for different numbers of indexed attributes. The figure shows that the median runtime for a tIND search query is below 100 milliseconds for all input sizes. While the average runtime does increase with the number of indexed attributes, that increase is rather slow, showing that our index prunes invalid candidates well. In fact, even for the full number of more than 1.3 million indexed attributes, the mean execution time of a query is 63 milliseconds. Furthermore, 86.3% of all queries are

[2]https://github.com/HPI-Information-Systems/tindResources
[3]https://dumps.wikimedia.org/

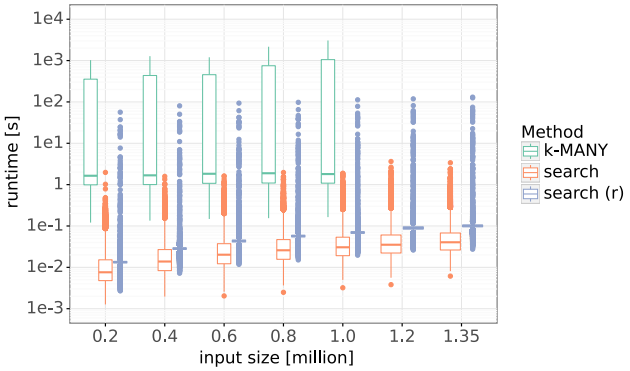

Figure 7: Runtimes for different numbers of indexed attributes. *Search (r)* displays reverse search. k-MANY ran out of memory, starting at 1.2 million attributes.

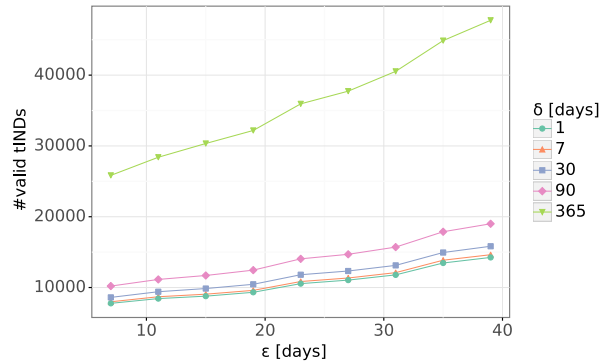

Figure 8: Impact of ε and δ on the number of tINDs found for 30,000 search queries.

answered in under 100 milliseconds and 99.8% are answered in under 1 second.

While, as expected, the overall runtime for reverse tIND search is higher, it scales with the input size in a similar manner, and the mean runtime of 142 milliseconds is still suitable for interactive exploration.

The figure also shows that a straightforward application of MANY cannot solve tIND search efficiently. Even without considering the extreme outliers where k-MANY cannot prune the search space at all, the median runtime for a query is still more than one order of magnitude greater than what our algorithm achieves. Furthermore, from 1.2 million attributes onwards, k-MANY runs out of memory, because it always has to track the violations for all candidates.

Figure 8 shows that, as expected, more tINDs are discovered if either of the two relaxation parameters ε or δ are increased.

Figure 9 shows the average runtime for a tIND search query with varying ε and δ. We can see that overall, a higher ε leads to a linear increase in runtime. The same is true for higher δ values, but the effect is much less pronounced, except for $\delta = 365d$, which is also a particularly large setting. This makes sense, because a higher δ can merely address larger temporal delays and not erroneous values: For a tIND candidate $A \subseteq^{\delta}_{\varepsilon,w} B$, if $A[t]$ contains a value that is not contained in B at all, then no matter how large δ becomes the tIND will always be violated at timestamp t.

Overall, Figure 9 shows that even for the most lenient setting that we tested ($\varepsilon = 39d$ and $\delta = 365d$), the average runtime of a

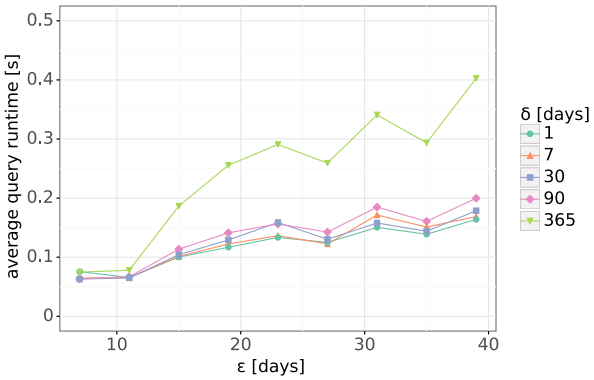

Figure 9: Impact of ε and δ on the mean runtime.

tIND search query is still below 500 milliseconds. Notably, still 78.5% of all queries are answered in less than 100 milliseconds and 99.3% in less than 1 second.

We also executed the all-pairs tIND discovery by querying every $A \in \mathcal{D}$ against our index. In this setting, it took less than three hours (including index construction times) to find the complete set of 306,047 tINDs. As a comparison: static IND discovery on the latest snapshot results in 883,506 INDs. Roughly a third of the tINDs is not discovered by executing static IND discovery on the latest snapshot, meaning we find an additional 50% of valid tINDs in comparison to the static variant. Additionally, we observe that 77% of the INDs discovered by the static approach are invalid tINDs. This indicates that INDs valid at only a single point in time are often spurious.

5.3 Impact of tIND parameter deviations

As discussed in Section 4.4, users may want to deviate from the parameter settings for ε or δ that the index is optimized for. Figures 10 and 11 show the runtime impact on tIND search queries when building indices for larger, more generous parameter values for ε and δ than the queries actually use. Figure 10 shows that when building the index for higher ε values, the mean runtime is largely unaffected, but the runtimes for some outliers increase. Figure 11 shows that if the index uses a setting for δ that is up to 16 times larger than what the actual queries use, there is again no significant impact on runtime. For settings that are even larger, there is a slight dip in performance, but the majority of the queries can still be answered in under 100 milliseconds. We conclude from this that the index structure is not sensitive to ε and δ. Thus, when building an index for an interactive setting, it is advisable to use high settings for ε and δ, because this also allows the usage of the index for queries with such settings, while having only a small impact on runtime for queries with even significantly smaller settings for ε or δ.

5.4 Hyperparameter impact on runtime

The most important parameter for the runtime is the size of the Bloom filters m. Figure 12 clearly shows that the larger the Bloom filters are, the better the average runtime for tIND search. As expected, this effect is reversed for reverse tIND search, because larger and thus less dense Bloom filters directly lead to more *AND*-operations per query. However, it is interesting to note that while the average runtime for reverse queries increases with the Bloom filter size, there are also fewer severe outliers, which may

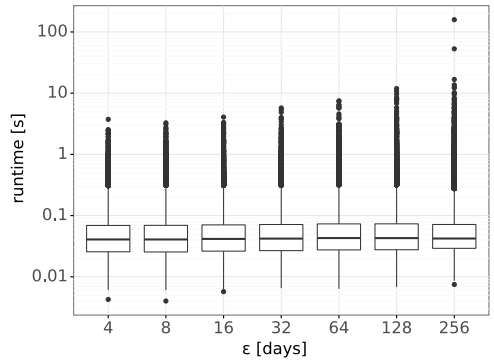

Figure 10: Runtimes for different settings for ε in the index for queries with a fixed ε=3 (standard setting).

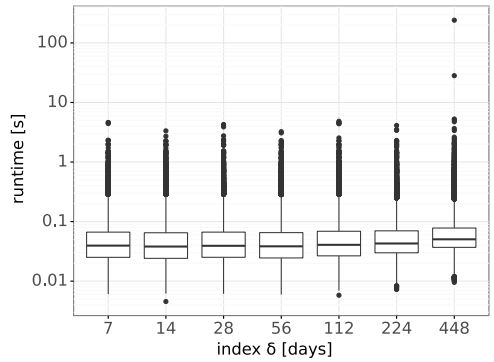

Figure 11: Runtimes for different settings for δ in the index for queries with a fixed δ=7 (standard setting).

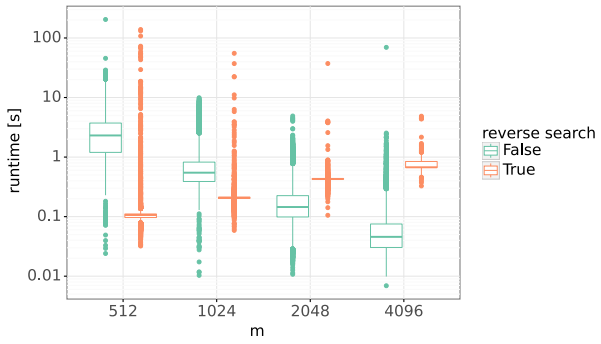

Figure 12: Runtimes of individual queries for different Bloom filter sizes.

be desirable in some circumstances. If we want to answer both types of queries with the same index, setting m to either 1024 or 2048 is suitable. With m=2048, we are able to discover 98.6% of all tIND queries and 99.9% of all reverse tIND queries in under 1 second.

To compare the two methods for choosing time slices to index on, we randomly chose three different sets of queries of 10,000 attributes each, and ran both algorithms with three different seeds. Figures 13 and 14 show the average time it takes to complete a query for different numbers of chosen time slices k. The boxplots show that a larger number of time slice indices also leads to a

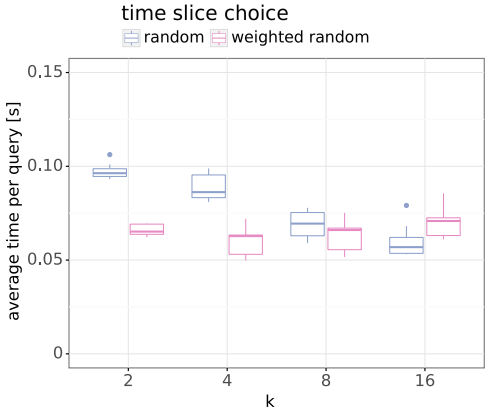

Figure 13: Average runtimes for different number of time slice indices (k) for tIND search

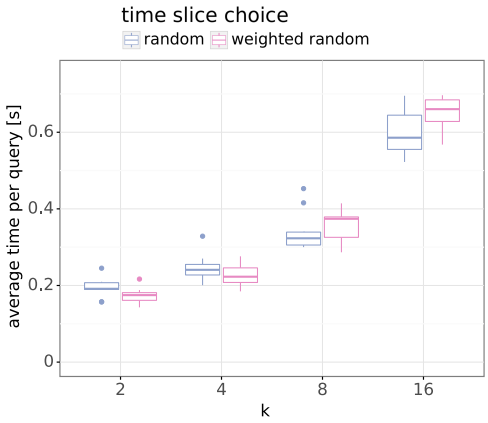

Figure 14: Average runtimes for different number of time slice indices (k) for *reverse* tIND search.

superior runtime for tIND search, but the effect is not as pronounced as for the size of the Bloom filters. Figure 13 shows that for tIND search, the weighted random approach is superior for a smaller k, but at around $k = 8$ this approach starts to stagnate and subsequently gets worse for $k = 16$. Contrary to that, choosing time slices randomly without weights continues to benefit from more time slices. This behavior is not unexpected, because *weighted random* tends to favor the same temporal areas, which means the more indices there are, the more likely it is that some of them yield the same result for many queries.

Interestingly, Figure 14 shows that more than two time slice indices actually increase the runtime for reverse tIND search. This shows that indeed the candidate pruning is not as effective for reverse search as it is for normal tIND search. However, this behavior is not problematic, even if we want to answer both types of queries with the same index: One can still build 16 indices to minimize the response time for normal tIND search, but only use two of them for reverse search queries.

5.5 Genuine IND discovery using tINDs

The (temporal) inclusion of one column in another can have many meanings, and knowledge about them has different uses. A discovered inclusion can indicate a key-foreign key relationship that could be enforced by the data management system and that

Bucket	TP [%]
$[4, 8) \subseteq [4, 8)$	7
$[4, 8) \subseteq [8, 16)$	10
$[4, 8) \subseteq [16, \infty)$	12
$[8, 16) \subseteq [4, 8)$	7
$[8, 16) \subseteq [8, 16)$	12
$[8, 16) \subseteq [16, \infty)$	9
$[16, \infty) \subseteq [4, 8)$	4
$[16, \infty) \subseteq [8, 16)$	14
$[16, \infty) \subseteq [16, \infty)$	24

Table 2: Basic statistics about the labelled INDs. TP (true positives) denotes the percentage of genuine INDs in a bucket.

could be used to join the two tables. For instance, the country-column of a table of athletes should be included in the name column of a table of UN countries. A discovered inclusion can indicate a joinability that does not depend on strict foreign key relationships. For instance, a key column of one table might be included in the key of another table, such as the name column of an EU-country table being included in that of a UN-country table. Users looking to extend information about such entities can use INDs to discover joinable tables. Finally, INDs can uncover hidden but interesting constraints. For instance, the name-column of the Soviet Union's male figure skating medalists is included in that of the Russian medalists, indicating legal successorship. We call such discovered tINDs *genuine*, as opposed to spurious ones.

At the beginning of our work, we hypothesized that INDs that are valid, despite frequent changes to one or both attributes, are more likely to be genuine. To test this hypothesis, we used static INDs discovered on the latest snapshot and distributed them into buckets according to the number of changes to the left- and right-hand side. Subsequently, we manually annotated a sample of 100 INDs per bucket as genuine or not. With the joinability use-case in mind, we labeled such INDs as genuine that should hold if the respective tables were complete and both columns have the same semantic type. The annotated INDs are available in the paper artifacts.

The results of the annotation can be seen in Table 2. It shows that a higher change frequency indeed increases the density of genuine INDs. This holds for both the left- and right-hand side, except for bucket $[16, \infty) \subseteq [4, 7]$, where only few genuine INDs are found.

While Table 2 shows that the frequency of changes in the past can already serve as an indicator for the genuineness of an IND, precision can be further increased by using the notion of relaxed tINDs as presented in this paper. To evaluate the usefulness of the different tIND variants for the discovery of genuine INDs, their parameters ε, δ and α (base for the exponential decay function w) were varied in a grid-search. Figure 15 shows the (micro-average) precision recall curves of all parametrized tIND variants discussed in this paper. Notably, every specialization of tINDs that we presented brings some improvement. Simple ε-relaxed tINDs perform worse than ε, δ-relaxed tINDs, which are worse than $w\varepsilon\delta$-tINDs for higher recall settings (and equivalent for low recall settings). Static INDs discovered on the latest snapshot (the basis for the ground truth annotation) only manage a low precision of 11% across all buckets.

The desired tradeoff for many applications, such as letting experts confirm discovered INDs as genuine before further processing, is to achieve high precision even at the expense of recall, so that less time of human annotators is wasted. Relaxed tINDs can provide this tradeoff. Notably, strict tINDs perform much worse than the relaxed variants, only achieving 25% precision and 4% recall. This shows that relaxation is indeed necessary, even for settings where high precision is desired.

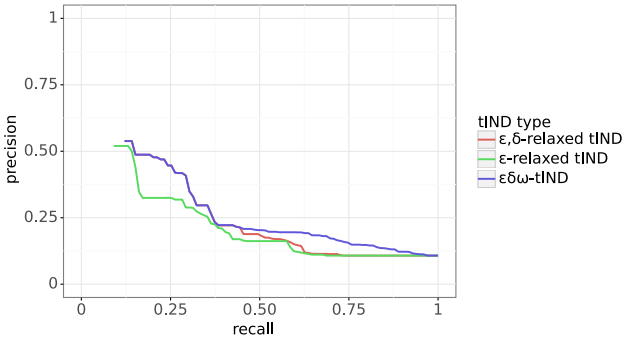

Figure 15: Micro-average precision-recall curves for the different variants of tINDs.

6 CONCLUSION

This work defined the notion of temporal inclusion dependencies (tINDs) – INDs that hold across all temporal versions of a dataset. To work with real-world data, we introduced several relaxations that allow the discovery of tINDs in dirty data. The first relaxes the amount of time that an attribute must be contained in another: ε denotes the fraction or summed weight of timestamps at which a tIND is allowed to be violated. The second relaxation allows for the updates to the data to be shifted for up to δ time units. The last relaxation allows for the user to specify an arbitrary weighting-function on the timestamps, for example to give a higher weight to recent ones.

To efficiently solve unary tIND search, we presented an index to efficiently discover for a query attribute Q, those attributes that Q is contained in ($Q \subseteq^{\delta}_{w,\varepsilon} A$), as well as those attributes contained in Q ($B \subseteq^{\delta}_{w,\varepsilon} Q$). Our experiments indicate that the index is able to answer almost all queries in under one second, allowing users to interactively explore datasets with tINDs. Furthermore, we were able to show that temporal INDs can help to find genuine, meaningful INDs with greater precision than the discovery of INDs on static data.

Future work could combine the existing $w\varepsilon\delta$-tINDs with already known IND-relaxations, such as partial [25], conditional [3] or similarity INDs [14]. Furthermore, the discovery of n-ary tINDs could be studied, potentially also on large relational databases, as such a setting will likely require different methods. Finally, it would be interesting to investigate whether the approaches presented in this paper are also applicable to general webtables or open-government data.

REFERENCES

[1] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2015. Profiling relational data: a survey. *VLDB Journal* 24, 4 (2015), 557–581.

[2] B. Thomas Adler, Luca De Alfaro, Santiago M Mola-Velasco, Paolo Rosso, and Andrew G West. 2011. Wikipedia vandalism detection: Combining natural language, metadata, and reputation features. In *International Conference on Intelligent Text Processing and Computational Linguistics*. Springer, 277–288.

[3] Jana Bauckmann, Ziawasch Abedjan, Ulf Leser, Heiko Müller, and Felix Naumann. 2012. Discovering conditional inclusion dependencies. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. ACM, 2094–2098. https://doi.org/10.1145/2396761.2398580

[4] Jana Bauckmann, Ulf Leser, Felix Naumann, and Veronique Tietz. 2007. Efficiently Detecting Inclusion Dependencies. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 1448–1450.

[5] Tobias Bleifuß, Leon Bornemann, Dmitri V. Kalashnikov, Felix Naumann, and Divesh Srivastava. 2021. Structured Object Matching across Web Page Revisions. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 1284–1295.

[6] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[7] Antoon Bronselaer, Christophe Billiet, Robin De Mol, Joachim Nielandt, and Guy De Tré. 2019. Compact representations of temporal databases. *VLDB Journal* 28 (2019), 473–496.

[8] Graham Cormode, Vladislav Shkapenyuk, Divesh Srivastava, and Bojian Xu. 2009. Forward Decay: A Practical Time Decay Model for Streaming Systems. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 138–149. https://doi.org/10.1109/ICDE.2009.65

[9] Fabien De Marchi, Stéphane Lopes, and Jean-Marc Petit. 2009. Unary and n-ary inclusion dependency discovery in relational databases. *Journal of Intelligent Information Systems* 32, 1 (2009), 53–73.

[10] 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 *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. ACM, 219–228. https://doi.org/10.1145/3357384.3357916

[11] Curtis Dyreson, Fabio Grandi, Wolfgang Käfer, Nick Kline, Nikos Lorentzos, Yannis Mitsopoulos, Angelo Montanari, Daniel Nonen, Elisa Peressi, Barbara Pernici, et al. 1994. A consensus glossary of temporal database concepts. *SIGMOD Record* 23, 1 (1994), 52–64.

[12] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. 2005. Data exchange: semantics and query answering. *Theoretical Computer Science* 336, 1 (2005), 89–124. https://doi.org/10.1016/j.tcs.2004.10.033

[13] João Marcelo Borovina Josko. 2018. A Formal Taxonomy of Temporal Data Defects. In *Data Quality and Trust in Big Data - 5th International Workshop, QUAT 2018, Held in Conjunction with WISE 2018, Dubai, UAE, November 12-15, 2018, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11235)*, Hakim Hacid, Quan Z. Sheng, Tetsuya Yoshida, Azadeh Sarkheyli, and Rui Zhou (Eds.). Springer, 94–110. https://doi.org/10.1007/978-3-030-19143-6_7

[14] Youri Kaminsky, Eduardo H. M. Pena, and Felix Naumann. 2023. Discovering Similarity Inclusion Dependencies. *Proceedings of the International Conference on Management of Data (SIGMOD)* 1, 1 (2023). https://doi.org/10.1145/3588929

[15] Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. 2022. Data dependencies for query optimization: a survey. *VLDB Journal* 31, 1 (2022), 1–22. https://doi.org/10.1007/s00778-021-00676-3

[16] Sebastian Kruse, Anja Jentzsch, Thorsten Papenbrock, Zoi Kaoudi, Jorge-Arnulfo Quiane-Ruiz, and Felix Naumann. 2016. RDFind: Scalable Conditional Inclusion Dependency Discovery in RDF Datasets. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

[17] Sebastian Kruse, Thorsten Papenbrock, Christian Dullweber, Moritz Finke, Manuel Hegner, Martin Zabel, Christian Zöllner, and Felix Naumann. 2017. Fast Approximate Discovery of Inclusion Dependencies *(LNI, Vol. P-265)*, Bernhard Mitschang, Daniela Nicklas, Frank Leymann, Harald Schöning, Melanie Herschel, Jens Teubner, Theo Härder, Oliver Kopp, and Matthias Wieland (Eds.). GI, 207–226. https://dl.gi.de/20.500.12116/629

[18] Sebastian Kruse, Thorsten Papenbrock, and Felix Naumann. 2015. Scaling Out the Discovery of Inclusion Dependencies. In *Proceedings of the Conference Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*.

[19] Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. 2015. Divide & conquer-based inclusion dependency discovery. *PVLDB* 8, 7 (2015), 774–785.

[20] Nuhad Shaabani and Christoph Meinel. 2017. Incremental Discovery of Inclusion Dependencies. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*. ACM, 2:1–2:12. https://doi.org/10.1145/3085504.3085506

[21] Nuhad Shaabani and Christoph Meinel. 2019. Incrementally updating unary inclusion dependencies in dynamic data. *Distributed Parallel Databases* 37, 1 (2019), 133–176. https://doi.org/10.1007/s10619-018-7233-5

[22] Fabian Tschirschnitz, Thorsten Papenbrock, and Felix Naumann. 2017. Detecting inclusion dependencies on very many tables. *ACM Transactions on Database Systems (TODS)* 42, 3 (2017), 18.

[23] Zhong Yang, Bolong Zheng, GuoHui Li, Xi Zhao, Xiaofang Zhou, and Christian S. Jensen. 2020. Adaptive Top-k Overlap Set Similarity Joins. In *ICDE*. 1081–1092.

[24] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In *SIGMOD*. ACM, 847–864.

[25] Erkang Zhu, Fatemeh Nargesian, Ken Q Pu, and Renée J Miller. 2016. LSH Ensemble: Internet-scale domain search. *PVLDB* 9, 12 (2016), 1185–1196.