# Uncovering the Evolution History of Data Lakes

Meike Klettke, Hannes Awolin
*University of Rostock*
*Rostock, Germany*
*meike.klettke@uni-rostock.de*

Uta Störl, Daniel Müller
*University of Applied Sciences*
*Darmstadt, Germany*
*uta.stoerl@h-da.de*

Stefanie Scherzinger
*OTH Regensburg*
*Regensburg, Germany*
*stefanie.scherzinger@oth-regensburg.de*

***Abstract*—Data accumulating in data lakes can become inaccessible in the long run when its semantics are not available. The heterogeneity of data formats and the sheer volumes of data collections prohibit cleaning and unifying the data manually. Thus, tools for automated data lake analysis are of great interest. In this paper, we target the particular problem of reconstructing the schema evolution history from data lakes. Knowing how the data is structured, and how this structure has evolved over time, enables programmatic access to the lake. By deriving a sequence of schema versions, rather than a single schema, we take into account structural changes over time. Moreover, we address the challenge of detecting inclusion dependencies. This is a prerequisite for mapping between succeeding schema versions, and in particular, detecting nontrivial changes such as a property having been moved or copied. We evaluate our approach for detecting inclusion dependencies using the MovieLens dataset, as well an adaption of a dataset containing botanical descriptions, to cover specific edge cases.**

***Keywords*-NoSQL databases; schema version extraction; evolution operations; integrity constraints; inclusion dependencies**

## I. Introduction

These days, large amounts of data are being collected and stored in data lakes for later analysis. Often, the data is collected in a schema-flexible or schema-less NoSQL database, since these systems allow for convenient storage of heterogeneous datasets. However, when the structure and semantics of the data stored are not known, accessing the data can become expensive and complicated. Therefore, automated tools for analyzing and understanding data lakes, extracting integrity constraints or even a schema, are a lively research area, e.g. [1]–[5].

In this article, we target the analysis of data lakes accumulating in *NoSQL databases*. In particular, we focus on recognizing the structural changes that have occurred over time. These may be as simple as a property having been added, renamed, or deleted. However, we also consider changes affecting more than one type of entity, where properties are moved or copied. Such changes, effectively denormalizing the data, are particularly common with NoSQL data: Since many NoSQL database management systems do not implement join queries, NoSQL practitioners introduce redundancy in favor of faster data access [6].

As a specific example, let us consider an application recording observations of species in the Baltic sea.

*Example 1.1:* Figure 1 shows some sample entities in JSON format. The database stores entities of type `Protocols` as well as information about the observed `Species`. The `Species` entities describe different species of mussels. The data is heterogeneous: All entities carry the scientific names, as well as the timestamp `ts` denoting the last write to the entity. However, only one entity has a property `category` with the catalogue number of the species. In the entities with the most recent timestamps, the catalogue number is stored as `WoRMS` (short for World Register of Marine Species).

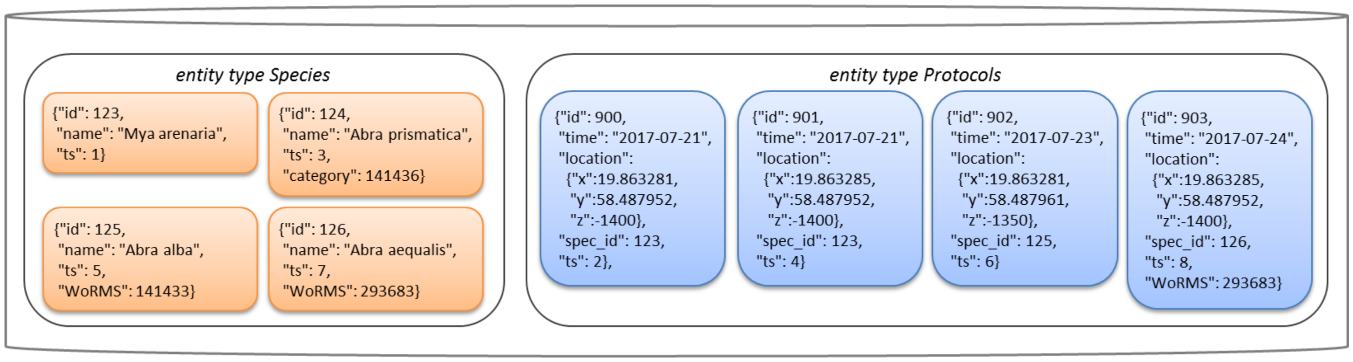
`Protocols` record the observation time, the location (longitude `x`, latitude `y`, and depth `z` in cm), as well as a reference to the species `spec_id`, and finally, a timestamp `ts`. Only one `Protocols` entity has a `WoRMS` property. □

Like other state-of-the-art schema extraction algorithms, our approach from earlier work [3] produces a JSON schema describing `Protocols`, as shown in Figure 2. However, this description does not take into account changes to the schema that occur over time. In our example, properties like `WoRMS` that have been added are merely annotated as having an occurrence less than 100%.

In this paper, we make a case for extracting a sequence of schema versions, rather than a global schema description. Moreover, we propose mappings between these schema versions. A data analyst with domain knowledge may then choose which mappings seem most plausible. In our running example, renaming property `category` in `Species` entities to `WoRMS` is such a mapping. Thus, the complete schema evolution history may be recovered. This process is sketched in Figure 3, and has been implemented within our *Darwin* tool (c.f. [7] for an earlier version of *Darwin* not yet equipped with this feature). Additionally, it is useful to gain insight into the inter-connectedness of the data.

*Example 1.2:* In our example, the property `spec_id` of `Protocols` is a foreign key, referencing the `id` of `Species` entities. Moreover, the `WoRMS` property of the `Species` entity with identifier 126 has been copied to the referencing protocol with identifier 903. □

In order to reliably recognize *copy* and *move* operations, we need to be able to detect which inclusion dependencies hold in the data. We therefore present and experimentally evaluate an algorithm that addresses this challenge.



```
{"id": 123,
 "name": "Mya arenaria",
 "ts": 1}

{"id": 124,
 "name": "Abra prismatica",
 "ts": 3,
 "category": 141436}

{"id": 125,
 "name": "Abra alba",
 "ts": 5,
 "WoRMS": 141433}

{"id": 126,
 "name": "Abra aequalis",
 "ts": 7,
 "WoRMS": 293683}
```

```
{"id": 900,
 "time": "2017-07-21",
 "location":
   {"x":19.863281,
    "y":58.487952,
    "z":-1400},
 "spec_id": 123,
 "ts": 2},

{"id": 901,
 "time": "2017-07-21",
 "location":
   {"x":19.863285,
    "y":58.487952,
    "z":-1400},
 "spec_id": 123,
 "ts": 4}

{"id": 902,
 "time": "2017-07-23",
 "location":
   {"x":19.863281,
    "y":58.487961,
    "z":-1350},
 "spec_id": 125,
 "ts": 6}

{"id": 903,
 "time": "2017-07-24",
 "location":
   {"x":19.863285,
    "y":58.487952,
    "z":-1400},
 "spec_id": 126,
 "ts": 8,
 "WoRMS": 293683}
```

Figure 1. NoSQL database instance with entities of type `Species` and `Protocols`.

*Contributions:* We make the following contributions:

- We present a three-step process to extract a sequence of schema versions, rather than a global schema, from an evolving data lake of hierarchical, heterogeneous data.
- We propose candidate mappings between schema versions, to restore the schema evolution history. A data analyst may interactively resolve any ambiguities.
- We propose an algorithm that detects inclusion dependencies in NoSQL data lakes.
- We evaluate our algorithm experimentally on data derived from the MovieLens dataset, as well as a customized dataset with botanical descriptions. Furthermore, we explore edge cases that arise from redundant data. We are able to show that with highly redundant data, our algorithm does not scale well. We then sketch our ideas for improving our algorithm in this regard.

*Structure.* The rest of this article is organized as follows: We introduce the preliminaries in Section II. The method for extracting schema versions from evolving datasets is introduced in Section III. To be able to propose evolution operations between schema versions, we need to know which inclusion dependencies hold. In Section IV, we introduce our algorithm for deriving inclusion dependencies from NoSQL data which we evaluated in Section V. In Section VI, we sketch the idea of combining bottom-up and top-down search for datasets which are rich in replicated data. In Section VII, we review related work. We close with a summary and an outlook on future work.

## II. Preliminaries

We first introduce some preliminaries on NoSQL data collections, and then review a schema extraction algorithm which we build upon [3].

An *entity* refers to a persisted object in a NoSQL database. It carries a set of *properties*, each with a name and a value. A value can be atomic (e.g., a string or numeric), or structured (an object or an array). Structured properties can be nested. For instance, in a document oriented NoSQL database such as MongoDB, a JSON document constitutes an entity.

Entities with similar structure (typically generated by the same object-oriented class in the application code) belong to the same *entity type*. Depending on the NoSQL data store and the overall software stack (e.g., the usage of an object-NoSQL mapper), entities of the same entity type are stored in the same NoSQL collection or bucket (in document oriented NoSQL databases), or in the same table (in column-family databases). Also, a dedicated property can explicitly specify the entity type. Thus, we may safely assume that we are always able to identify the entity type given an entity.

*Example 2.1:* In our example in Figure 1, we distinguish entities of type `Species` from entities of type `Protocols`. If the database system is MongoDB, we may declare MongoDB *collections* correspondingly. □

Our schema extraction approach from [3] captures implicit structural information, such as property names, data types of the values, and the nesting of objects. This yields a schema which contains *all structural variants* of each entity type.

*Example 2.2:* For our running example, the JSON schema extracted for `Protocols` is shown in Figure 2. □

The schema captures occurrence of each structural component in a description property. In deriving a JSON schema, we assume that the components which occur in all entities are required, and optional otherwise.

*Example 2.3:* For our running example, line 22 of the JSON schema in Figure 2 records that property `WoRMS` is only available in one of the four `Protocols` entities. □

Overall, the JSON schema identifies regular structures, statistics on the occurrence of properties, and identifies structural outliers. However, the schema does not capture the schema evolution history.

*Example 2.4:* The JSON schema in Figure 2 does not reflect that property `WoRMS` was first introduced at timestamp 8, and has been part of the schema since then. □

This motivates us to extract schema versions, as discussed in the next section.

```
{"title": "Protocols",
  "type": "object",
  "properties": {
    "id": {
      "type": "number",
      "description": "Occurrence: 4/4, 100%"},
    "location": {
      "type": "object",
      "properties": {
       "x": {
        "type": "number",
         "description": "Occurrence: 4/4, 100%"},
       ...},
      "required": ["x", "y", "z"],
      "description": "Occurrence: 4/4, 100%"},
    "spec_id": {
      "type": "number",
      "description": "Occurrence: 4/4, 100%"},
    ...
    "WoRMS": {
      "type": "number",
      "description": "Occurrence: 1/4, 25%"}},
  "required": [ "id", "location", "spec_id",
               "time", "ts"] }
```

Figure 2. JSON schema (excerpt) derived for sample `Protocols`.

## III. Extracting Schema Versions

In this section, we present our approach for identifying consecutive schema versions, as well as operations for translating one schema version into the next. A necessary prerequisite is that each entity carries a timestamp, recording the last write to this entity. This allows us to restore a partial order of writes against entities. In fact, in our study of open source and NoSQL-based applications [8], we could observe that many software projects indeed maintain such timestamps. In addition, NoSQL data stores typically use timestamps or version numbers internally.

In describing our approach below, we further assume that optional properties are encoded as null values or empty values, rather than missing properties. This simplifies our approach and reduces the number of ambiguities in the mappings between schema versions. However, this is not an inherent limitation, as our approach can be extended to handle missing properties as well.

### A. Schema Versions and Schema Evolution Operations

We extract schema versions and evolution operations in a three-step process, as visualized in Figure 3.

*(1) Structure Extraction:* First, the entities types to be analyzed are chosen, for instance, all entities of a database or selected collections or tables.

Next, we load the chosen entities. Their structure is summarized by *schema version graphs*. For each detected entity type, a separate schema version graph is created. This graph has a *root* node with the name of the entity type and a *list* with the timestamps of all entities of this entity type. The graph contains nodes for each property of the entity type with information about the data types and the hierarchical structure. The timestamps of all entities containing a certain property are stored in the corresponding node. Thus, they record in which time intervals certain structures occur.

*Example 3.1:* Figure 4 shows the schema version graphs for our running example. Looking at the schema version graph for entity type `Species`, the timestamp lists in the property nodes reflect that property `name` occurred in the entities with timestamp `1`, `3`, `5`, and `7`, and property `WoRMS` in the entities with timestamp `5` and `7`. Thus, this property did not exist before timestamp `5`. □

*(2) Deriving schema evolution operations:* We next determine possible candidates for schema evolution operations. We cover the set of schema evolution operations presented in [9], comprising adding, deleting, and renaming properties from all entities of an entity type, as well as copying and moving properties between entities of different types.

Proposing schema evolution operations is the most challenging step of the process: First, all timestamp lists in each node of the schema version graph are sorted. Afterwards, we identify the structural differences between entities based on the schema version graphs and the sorted timestamp lists along the timeline. We can then derive the single-type operations (*add*, *delete*, and *rename*), affecting properties of the *same* entity type.

By comparing schema version graphs of different entity types, we are able to detect multi-type operations that *copy* or *move* properties between entity types.

Note that there may be alternative schema evolution operations between two consecutive schema versions.

*Example 3.2:* In our example, we detect a new property `WoRMS` in entity type `Protocols` at timestamp `8`. This structural change may have been caused by an *add* operation. However, it may also have been caused by a *copy* operation, originating from entity type `Species` and joining on the `Species id` and `Protocols spec_id`. □

*(3) Resolution of Ambiguities:* In the last step, the alternative schema evolution operations are resolved by a user. Furthermore, in case of a *copy* or *move*, a *join condition* has to be specified. For the time being, we apply a semiautomatic approach which preformulates a join condition and leaves it to the user to specify the join predicate.

*Example 3.3:* Continuing with the previous example, only a data analyst with domain knowledge can disambiguate the situation. We therefore compile a *decision table*, listing alternatives for the data analyst to choose from, as discussed next. Figure 5 shows the decision table produced by the *Darwin* tool. □

In Section IV we present our algorithm for deriving inclusion dependencies, which allows us to disambiguate some cases automatically. For instance, a *copy* or *move* operation between entities of different types is unlikely when there are no inclusion dependencies between the participating entity types. After all, each *copy* or *move* operation requires a join between the source and target entities, and inclusion

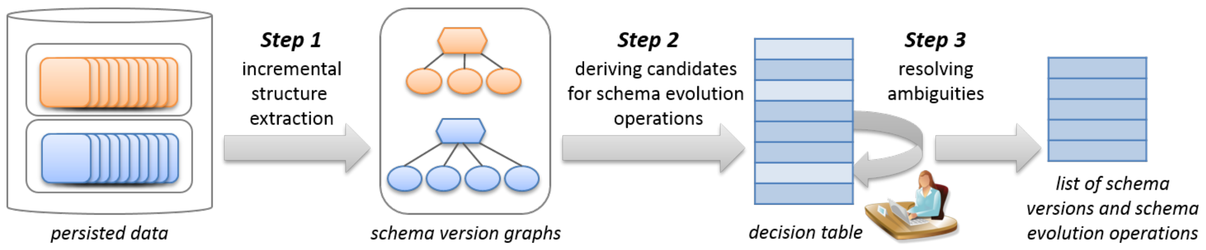
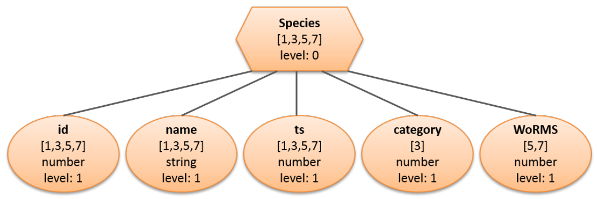
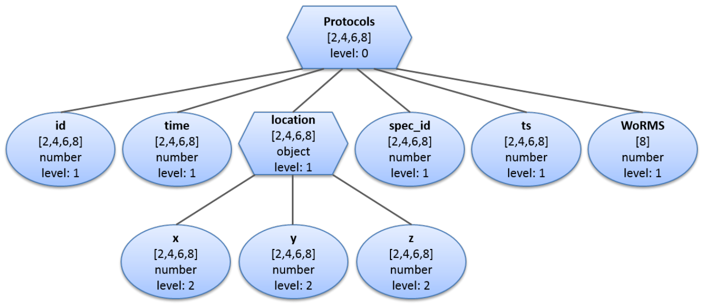

persisted data
Step 1
incremental structure extraction
schema version graphs
Step 2
deriving candidates for schema evolution operations
decision table
Step 3
resolving ambiguities
list of schema versions and schema evolution operations

Figure 3. Extracting schema versions and evolution operations.

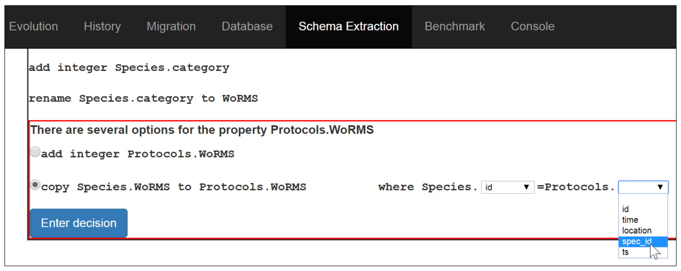


(a) Schema version graph for entity type `Species`.



(b) Schema version graph for entity type `Protocols`.

Figure 4. Schema version graphs for entity types `Species` and `Protocols` extracted from the entities in Figure 1.



```
add integer Species.category

rename Species.category to WoRMS
```

There are several options for the property Protocols.WoRMS

```
( ) add integer Protocols.WoRMS

(●) copy Species.WoRMS to Protocols.WoRMS        where Species. id =Protocols.
```

Figure 5. *Darwin* screenshot: In the *decision table*, the data analyst may interactively resolve ambiguous schema evolution operations and supplement the join predicate.

dependencies typically hold for join properties.

As a result of this three-step process, we have restored a plausible schema evolution history.

*Example 3.4:* For our example we derive four schema versions and the corresponding schema evolution operations. Figures 6 and 7 show relevant excerpts of the JSON schemas for entity type `Protocols` in versions 3 and 4. The following schema evolution operation, specified in our syntax from [9], transforms version 3 to version 4:

```
copy Species.WoRMS to Protocols.WoRMS
where Species.id = Protocols.spec_id
```
□

## B. Scalability

As NoSQL databases often back data-intensive applications, our approach must scale. In general, the time complexity of the approach proposed above is $O(n \log n)$, due to sorting timestamp lists in the schema version graphs.

For extracting the schema version graphs, our implemented approach does not load the entire data instance into main memory, but proceeds in batches.

However, in Step 2, the length of the timestamp lists may become a critical factor with respect to memory limitations. We therefore implemented an *incremental* approach as alternative: Instead of extracting and analyzing the dataset as a whole, we divide it into *subsets* that we process incrementally. The division of the entity set into consecutive ranges is based on the timestamps. First, we extract and analyze the entities of the first subset. The decision table is initially populated. After each step, we may discard all information in the schema version graphs on the already analyzed subset except information on the last schema version. Afterwards, the process continues with the next data subset. The decision table is sequentially updated after processing a subset. Finally, ambiguities are resolved as described above.

The drawback of this incremental method is that in Step 1, data can no longer be loaded in arbitrary order. We have to ensure that we process entities with consecutive timestamp ranges. However, the time complexity remains in $O(n \log n)$, so we may safely handle large volumes of data.

## IV. Inclusion Dependencies

Our main motivation for deriving integrity constraints is to detect schema changes due to *move* or *copy* operations. As in Example 3.4, these operations require a join between entity types. By identifying inclusion dependencies in the data lake, we are able to propose meaningful join conditions.

```
{ "title": "Protocols",
  "description": "schema version 3",
  "type": "object",
  "properties": {
    "id": { "type": "number" },
    "location": {
      "type": "object",
      "properties": {
       ...} },
    "spec_id": { "type": "number" } } }
```

Figure 6. JSON schema (excerpt) for entity type `Protocols` in version 3.

```
{ "title": "Protocols",
  "description": "schema version 4",
  "type": "object",
  "properties": {
    "id": { "type": "number" },
    "location": {
      "type": "object",
      "properties": {
       ...} },
    "spec_id": { "type": "number" },
    "WoRMS": { "type": "number" } } }
```

Figure 7. JSON schema (excerpt) for entity type `Protocols` in version 4.

In relational databases, inclusion dependencies are often unary, consisting of only one attribute. Yet in working with NoSQL databases, schema denormalization is common practice. Redundancies are introduced intentionally, to work around the limitations of the query languages supported by NoSQL databases (such as no join operations in some NoSQL database systems [6]).

We next propose an algorithm for deriving $k$-ary inclusion dependencies. We start with the definition of these dependencies for NoSQL data.

## A. Definition

We extend the definition of inclusion dependencies from the relational model [10] to NoSQL data. Given an entity $e$ and a property name $P$, we write $e.P$ to obtain the value of property $P$ in this entity. For nested entities, we use the dot-notation accordingly.

*Definition 4.1:* An *inclusion dependency* over entity types $E_1$ and $E_2$ is an expression of the form

$$\sigma = E_1[A_1, \ldots, A_k] \subseteq E_2[B_1, \ldots, B_k],$$

where

- $A_1, \ldots, A_k$ is a sequence of distinct properties of $E_1$,
- $B_1, \ldots, B_k$ is a sequence of distinct properties of $E_2$.

Then a database instance $\mathbf{I}$ satisfies $\sigma$, denoted $\mathbf{I} \models \sigma$, if

$$\{\langle e.A_1, \ldots, e.A_k\rangle \mid e \text{ is an entity of type } E_1\} \subseteq \{\langle e.B_1, \ldots, e.B_k\rangle \mid e \text{ is an entity of type } E_2\}.$$

In the special case that $k = 1$, we say an inclusion dependency is *unary*. □

During the detection of inclusion dependencies, we distinguish three states: First, an inclusion dependency is known to be valid, denoted $E_1[A_1, \ldots, A_k] \subseteq E_2[B_1, \ldots, B_k]$ according to the definition above. Second, it may be still unknown whether an inclusion dependency holds, as it has not been tested yet. We talk of *inclusion dependency candidates* in this case, and denote this state as $E_1[A_1, \ldots, A_k] \overset{?}{\subseteq} E_2[B_1, \ldots, B_k]$. Third, it can be known that an inclusion dependency is not valid, which we denote as $E_1[A_1, \ldots, A_k] \nsubseteq E_2[B_1, \ldots, B_k]$.

## B. Basic Idea and Complexity

Inclusion dependency candidates have to be tested for all subsets of properties of two entity types $E_1$ and $E_2$. Let $m_1$ be the number of properties in $E_1$, and $m_2$ the number of properties in $E_2$. The total number of candidates for inclusion dependencies over entity types $E_1$ and $E_2$ is:

$$\sum_{k=1}^{min(m_1,m_2)} \frac{m_1!}{(m_1-k)! * k!} * \frac{m_2!}{(m_2-k)!} \qquad (1)$$

For the explanation of the formula, we use two concepts from combinatorial analysis: variation[1] and combination[2]. The number of inclusion dependency candidates is the number of all *combinations* of properties from $E_1$ multiplied with the number of all *variations* of properties from $E_2$.

*Example 4.1:* Let us consider two entity types $E_1$ and $E_2$, with the properties $A_1, A_2$ and $B_1, B_2$ respectively. For finding the valid inclusion dependencies over $E_1$ and $E_2$, we test the following candidates:

- The unary candidates $E_1[A_1] \overset{?}{\subseteq} E_2[B_1]$, $E_1[A_1] \overset{?}{\subseteq} E_2[B_2]$, $E_1[A_2] \overset{?}{\subseteq} E_2[B_1]$, and $E_1[A_2] \overset{?}{\subseteq} E_2[B_2]$.
- The binary candidates $E_1[A_1, A_2] \overset{?}{\subseteq} E_2[B_1, B_2]$ and $E_1[A_1, A_2] \overset{?}{\subseteq} E_2[B_2, B_1]$.

Additional variations of the properties on the left-hand side do not have to be tested (the candidates $E_1[A_2, A_1] \overset{?}{\subseteq} E_2[B_2, B_1]$ and $E_1[A_2, A_1] \overset{?}{\subseteq} E_2[B_1, B_2]$ are already covered by the binary candidates above). In our example, this yields

$$\sum_{k=1}^{2} \frac{2!}{(2-k)! * k!} * \frac{2!}{(2-k)!} = 6$$

different candidates. □

Formula 1 yields the number of inclusion dependencies that have to be tested in the worst case. For each unknown inclusion dependency $E_1[A_1, \ldots, A_k] \overset{?}{\subseteq} E_2[B_1, \ldots, B_k]$, we

[1]The *variation* is the ordered selection of $k$ elements from a set of $m$ elements. It generates $\frac{m!}{(m-k)!}$ distinct results.

[2]The *combination* is the selection of $k$ elements from a set of $m$ elements. In contrast to the variations, the order does not matter. The combination generates $\frac{m!}{(m-k)! * k!}$ distinct results.

have to test if all values from $E_1[A_1, \ldots, A_k]$ also occur in $E_2[B_1, \ldots, B_k]$. Let us capture the effort involved.

Let the number of entities of entity type $E_1$ be $n_1$, and let the number of entities of entity type $E_2$ be $n_2$ accordingly. We get the following worst case for the number of tests required to derive inclusion dependencies.

$$\left(\sum_{k=1}^{min(m_1,m_2)} \frac{m_1!}{(m_1-k)! * k!} * \frac{m_2!}{(m_2-k)!}\right) * (n_1 * n_2) \quad (2)$$

Evidently, a large number of properties impacts performance. We also observe this effect in our experiments in Section V.

We next introduce two rules which are the building blocks of our algorithm. Given inclusion dependencies that are valid, we can conclude that the inclusion dependencies for *subsets* of properties are also valid. Let $i < k$, then

$$\begin{aligned} E_1[A_1, \ldots, A_k] \subseteq E_2[B_1, \ldots, B_k] \Rightarrow \\ E_1[A_1, \ldots, A_i] \subseteq E_2[B_1, \ldots, B_i] \\ \wedge E_1[A_{i+1}, \ldots, A_k] \subseteq E_2[B_{i+1}, \ldots, B_k]. \end{aligned} \quad (3)$$

If we know that inclusion dependencies are not valid, we can further conclude that the inclusion dependency for the *superset* is not valid either:

$$\begin{aligned} (E_1[A_i] \nsubseteq E_2[B_i]) \vee (E_1[A_j] \nsubseteq E_2[B_j]) \\ \Rightarrow (E_1[A_i, A_j] \nsubseteq E_2[B_i, B_j]) \end{aligned} \quad (4)$$

Next, we introduce an algorithm which uses rules (3) and (4) to prune the search space.

## C. Algorithm

Our algorithm for detecting inclusion dependencies proceeds bottom-up, starting with unary inclusion dependencies. Based on the unary inclusion dependencies found, the candidates for binary inclusion dependencies are tested next, and so on, until the $k$-ary candidates are tested based on the $(k-1)$-ary inclusion dependencies.

For testing all $k$-ary inclusion dependency candidates, we generate all combinations of properties for the left-hand side ($C_k(U_1)$) and all variations ($P_k(U_2)$) of properties for the right-hand side (as introduced in Formula 1). For the purpose of our algorithm, we assume that $C_k$ produces tuples, preserving the order of the properties as syntactically listed in $U_1$.

For each inclusion dependency candidate ($l \overset{?}{\subseteq} r$), with $l = E_1[A_1, \ldots, A_k]$ and $r = E_2[B_1, \ldots, B_k]$, we test if the inclusion dependency holds. The idea is similar to the well established *apriori algorithm* for mining association rules [11]. Thus, in accordance with rule 4, a candidate for a $k$-ary inclusion dependency is only tested when all $(k-1)$-ary inclusion dependencies defined over subsets of the properties are known to be valid. For this test, we introduce

**Algorithm 1** Finding Inclusion Dependencies over Entity Types $E_1$ and $E_2$

```
1: $U_1$ = {properties of $E_1$}
2: $U_2$ = {properties of $E_2$}
3: $ID = \emptyset$ /* inclusion dependencies */
4: $k = 0$
5: repeat
6:     $k = k + 1$
7:     changes = false
8:     for all $l \in C_k(U_1)$ do
9:         /* all k-ary combinations of properties from $U_1$ */
10:        for all $r \in P_k(U_2)$ do
11:            /* all k-ary variations of properties from $U_2$ */
12:            apriori = true
13:            for all $i = 1, \ldots, k$ do
14:                if (subtuple($l$, $i$), subtuple($r$, $i$)) $\notin ID$ then
15:                    apriori = false
16:                    break
17:                end if
18:            end for
19:            if ((apriori) and tablescan($l, r$)) then
20:                $ID = ID \cup \{(l, r)\}$
21:                changes = true
22:            end if
23:        end for
24:    end for
25: until (changes == false)
```

a function subtuple($t, i$), that takes a $k$-ary tuple $t$ and returns a $(k-1)$-ary tuple with all but the $i$th component.

The apriori condition in Algorithm 1, lines 12–18 is similar to the apriori step for mining association rules: Only candidates which satisfy the apriori condition are tested during a tablescan. The tablescan for an inclusion dependency candidate $E_1[A_1, \ldots, A_k] \overset{?}{\subseteq} E_2[B_1, \ldots, B_k]$ tests for each entity of type $E_1$ whether the values for properties $A_1, \ldots, A_k$ match the property values $B_1, \ldots, B_k$ of some entity of type $E_2$ (lines 19–22). If so, this function returns that the inclusion dependency candidate is valid. Consequently, it is added to the set of valid inclusion dependencies $ID$. Each valid inclusion dependency is encoded as a pair of tuples $(l, r)$, containing the sequence of properties from the left-hand side ($l$) and the right-hand side ($r$) of the dependency.

The algorithm terminates when no new inclusion dependencies can be found (lines 5, 7, and 25).

*Example 4.2:* Let us return to our running example: In Figure 5, we have seen that an *add* or a *copy* operation could have caused the data migration into the current schema version. If we do not find an inclusion dependency over the two entity types, we can settle on the *add* operation.

However, our algorithm detects the inclusion dependency

`Protocols[spec_id]` ⊆ `Species[id]`. Thus, we are able to identify a *copy* operation and its join condition between these two entity types:

```
copy Species.WoRMS to Protocols.WoRMS
where Species.id = Protocols.spec_id
```
□

## D. Optimizations

All optimizations stated in the following exploit metadata to avoid the expensive table scans. We evaluate these optimizations experimentally in the upcoming section:

1) *Smart iterator*: The smart iterator is an optimization over the brute-force bottom-up method. It does not generate the $k$-ary candidates from all properties of the NoSQL datasets, for testing the apriori condition. Instead, it joins the valid $k-1$-ary inclusion dependencies in $ID$ to $k$-ary inclusion dependency candidates. This can effectively reduce the number of candidates for which we test the apriori condition.
2) Additionally, we may consider the *datatypes* of properties: We can ignore inclusion dependency candidates if the datatypes of the properties on the left-hand side and the right-hand side do not match.
3) Further, we may take into account *metadata*, such as the minimum and maximum values of each property, when the property value is atomic. This information can be collected as part of schema extraction and allows us to decide that an inclusion dependency does not hold, as expressed in rule (5) below:
   Let $E_1$ and $E_2$ be entity types. Let $min_A$ be the minimum value of property $A$ in all entities of type $E_1$, i.e. $min_A = min(\{e.A \mid e \text{ is an entity of type } E_1\})$. Likewise, we define $max_A$, as well as $min_B$ and $max_B$ for the entities of type $E_2$. Thus,

$$(min_A < min_B) \vee (max_A > max_B) \Rightarrow E_1[A] \not\subseteq E_2[B]. \quad (5)$$

These three optimizations do not influence the number of valid inclusion dependencies found. In all cases, we detect all inclusion dependencies that are valid in the dataset. In the upcoming section, we experimentally evaluate the impact of these optimizations on the runtime.

## V. Experimental Evaluation

We now evaluate our approach for detecting inclusion dependencies. Our experiments were run on an Intel Core i7-461000U CPU @ 2.70GHz machine with 8.00 GB RAM. The algorithm was implemented in Java. As a NoSQL database, we use MongoDB version 3.4. We realize entity types as MongoDB collections. All reported runtimes are averaged over 10 runs.

In Subsection V-A, we evaluate the algorithm, as well as the benefits of our optimizations on the MovieLens dataset [12]. In Subsections V-B and V-C, we perform experiments with a dataset based on the iris dataset [13]. We have engineered this dataset to represent artificial edge cases that reveal the limitations of our algorithm. We regard these edge cases as rare, since they require highly duplicated data between two entity types. Nevertheless, we consider it worthwhile to understand the limitations of our approach. We then propose optimizations motivated by the insights gained from examining the edge cases in Section VI.

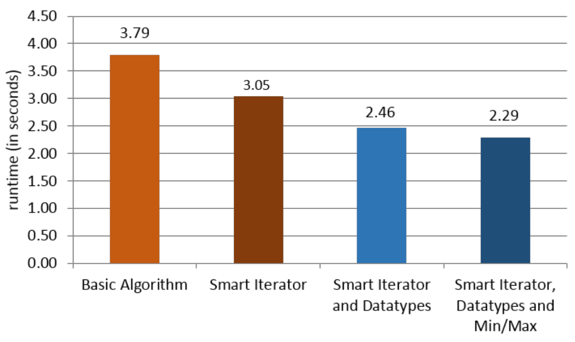


Figure 8. The basic algorithm for finding inclusion dependencies and the impact of optimizations for reducing the search space and runtime.

## A. The Algorithm and its Optimizations

We use the MovieLens dataset to evaluate our algorithm and to explore the benefits gained by the optimizations. Originally, this is a relational dataset. We have migrated parts of the MovieLens dataset into JSON documents.

The original benchmark contains a table with movie data and another table linking movies to other movie databases. Correspondingly, we have generated a MongoDB collection `movies` with about 27,000 JSON documents. Each entity has the properties `movieId`, `title`, and `genres`. The MongoDB collection `links` also contains about 27,000 JSON entities with the properties `movieId`, `imdbId`, and `tmdbId`. For each entry in `movies` there exists exactly one entry in `links`. In the MovieLens dataset, the inclusion dependencies `movies[movieId]` ⊆ `links[movieId]` and `links[movieId]` ⊆ `movies[movieId]` hold.

Figure 8 shows the runtimes of four variants of the algorithm. We see that the *smart iterator* (*second bar*) is an improvement over the basic algorithm. In the *third bar*, we see that additionally considering the datatypes of properties improves the runtime further. The *fourth bar* shows the additional improvement when we consider minimum and maximum values.

The entities in the MovieLens dataset contain relatively few properties. In the following, we experiment on datasets that have been artificially engineered to be more challenging.

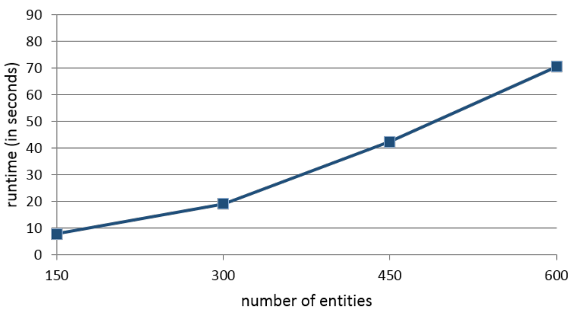


Figure 9. Edge case: Deteriorating Runtime of the *smart iterator* algorithm for 10 properties and 1,023 valid inclusion dependencies, increasing the number of entities.

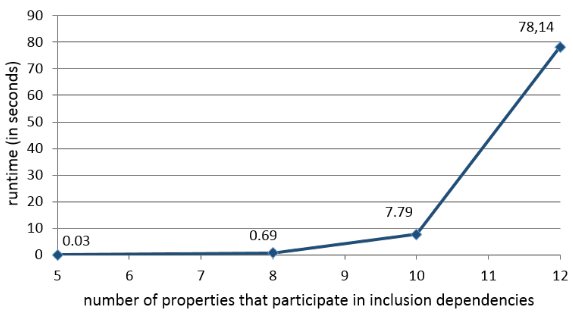


Figure 10. Edge case: Deteriorating runtime of the *smart iterator* algorithm for 150 entities, increasing the number of properties and valid inclusion dependencies.

### *B. Edge Case: Increasing the Number of Entities in the Presence of many Inclusion Dependencies*

We next examine which characteristics of the input data have an impact on the runtime. We make use of the iris dataset which contains classifications of the iris plant family. We have migrated this data into JSON format and extended it to 600 entities with 10 properties each.

We duplicate the dataset to obtain two groups of entities, and search for inclusion dependencies between both groups. Due to these redundancies, we may identify $2^{10} - 1$ =1,023 valid inclusion dependencies. This includes unary, binary, and up to 10-ary inclusion dependencies. We evaluate the algorithm over datasets containing up to 600 entities.

Figure 9 shows the results for the *smart iterator* implementation. As expected, this scenario is challenging for our algorithm, since we have 1023 valid inclusion dependencies and the number of inclusion dependency candidates to be tested is even higher. The runtime for checking each inclusion dependency candidate is quadratic in the number of entities.

### *C. Edge Case: High Number of Inclusion Dependencies*

We next examine the impact of the number of valid inclusion dependencies. Again, we modify the iris dataset: We keep the number of entities fixed to 150 in both datasets. We then run the algorithm with an increasing number of properties that are identical between both datasets. The number of inclusion dependencies grows accordingly: For $m$ identical properties, we obtain $2^m - 1$ valid inclusion dependencies.

We evaluate the algorithm for 5 properties (31 valid inclusion dependencies), 8 (255 valid inclusion dependencies), 10 (1,023 valid inclusion dependencies), and 12 properties (4,095 valid inclusion dependencies). For the last test, we added further artificial properties into the dataset.

Figure 10 shows that the algorithm does not scale well when the number of identical properties grows beyond 10.

Unfortunately, our optimizations from earlier are not helpful in the presence of this many inclusion dependencies: If an inclusion dependency holds, applying the optimizations adds additional tests, but does not reduce the search space. For datasets with many valid inclusion dependencies, we need different optimization ideas, as discussed next.

## VI. Outlook on Further Optimizations

Our experiments in Section V-C reveal that the bottom-up approach for deriving *all* inclusion dependencies does not scale when inclusion dependencies of arity 10 and higher are involved. This motivates us to modify our approach.

When we have successfully detected the unary inclusion dependencies in the bottom-up approach, we change our strategy. Let $k$ be the number of valid unary inclusion dependencies found. Rather than testing the binary inclusion dependency candidates next, we switch to a top-down search. That is, we test the cover of all $k$ valid unary inclusion dependencies. If this $k$-ary inclusion dependency is valid, we have effectively derived all inclusion dependencies between the datasets. Otherwise we can continue with $k/2$-ary candidates, and continue in this manner.

In the scenarios portrayed by Figures 9 and 10, this approach would reduce the number of inclusion dependency candidates to be tested to the 10 unary candidates, and in the next step the 10-ary inclusion dependency candidate. Thus, we need to execute only 11 table scans in contrast to the 1,023 table scans of the bottom-up approach.

## VII. Related Work

Our work is related to various earlier contributions in both database theory and applied research, ranging from schema extraction to deriving integrity constraints from relational, XML, and NoSQL data. Due to the long-standing history of this research, our review can be by no means exhaustive. We therefore focus on a sample of representative contributions.

*Schema Extraction:* Extracting a schema from data *a posteriori* is an established research field. In the context of this paper, we focus on schema extraction from semistructured or heterogeneous data. One of the first methods for deriving DTDs from XML documents was suggested by

Moh, Lim, and Ng in [14]. A graph datastructure captures all structural variants. The graph can be translated to a schema describing all heterogeneities of the input data. A similar task is addressed in [15]. Bex et. al. give a comprehensive survey over the extraction of DTDs and regular expressions from XML documents. In both efforts, it is assumed that all XML documents adhere to the same schema version. Evolutionary changes over time, the topic of the present paper, are not considered.

Recently, schema extraction for NoSQL data has gained interest [1]–[5]. Again, these approaches extract a single schema, and do not focus on schema changes over time.

*Schema Extraction and Schema Evolution:* The differential snapshot algorithm from Labio and Garcia-Molina [16] is an instance of an evolution-aware method for schema extraction. Originally, this algorithm was developed for supporting ETL processes in data warehouses. It matches two snapshots of a data source and derives the changes between these snapshots. The snapshots are assumed to be lists of records, and each data source is handled separately. The changes between two snapshots are described by the operations *add*, *delete*, and *update*, which are single-type operations that change one data source at a time.

However, this approach is designed for relational data and therefore not immediately transferrable to our use case, where we support a more complex data format. Additionally, we focus on detecting denormalization tasks such as *copy* and *move* operations between entities of different types.

Evolutionary changes to an XML schema are the focus of [17] by Baqasah et al. Whereas most methods start by analyzing XML documents, the XS-Diff algorithm receives two schemas as input. The authors derive the changes between XSD schemas in form of *insert*, *update*, *delete*, and *move* operations. The *move* operation, in contrast to our approach, is limited to moves along the element hierarchy, within a single XSD. In contrast, we aim at detecting *move* operations between different entity types, where we require a value-based join (not an implicit structural join).

For NoSQL databases, Ruiz, Morales, and Molina propose in [18] an approach based on model driven engineering for inferring the schema of aggregate oriented NoSQL databases. However, they only consider structural changes affecting the same entity type. Multi-type operations, important in our context, are not supported.

*Deriving Integrity Constraints from Data:* Detecting integrity constraints is vital for extracting a meaningful schema. In database theory research, there have been various studies on the feasibility of transferring decidability results for integrity constraints from the relational model to XML data. This concerns the concepts of keys (Buneman et. al. in [19]), functional dependencies (Kot et. al. in [20]), and foreign keys and thus, inclusion dependencies (Vincent et. al. in [21]).

There is a range of commercial tools mining constraints from relational databases. Most aim at deriving keys and functional dependencies. In this article, we specifically focus on inclusion dependencies. We therefore restrict our discussion of related work to this class of integrity constraints.

Fajt et. al. mine integrity constraints in XML data [22]. They define keys and foreign keys for XML, and start with the derivation of non-keys, which are easier and faster to detect than keys. Then, based on the non-keys, all $n$-ary keys are derived. To find a foreign key, the authors restrict their approach to *unary* foreign keys, while we consider $k$-ary inclusion dependencies. However, their restriction allows them to parallelize the algorithm elegantly.

With increasing data volumes, the development of scalable algorithms for deriving integrity constraints from relational databases has gained interest. Papenbrock et. al. introduce scalable methods for finding keys [23] and inclusion dependencies [24]. In both contributions, the authors apply sophisticated and efficient solutions for deriving all integrity constraints that hold in relational datasets. Our own approach differs in several aspects. First, we work on a different data model and derive integrity constraints from NoSQL datasets. We further use metadata (such as minimum and maximum values) to detect invalid inclusion dependencies.

In [1], Farid et. al. find logical constraints in data lakes. Based on an internal RDF format, they use directed hypergraphs for finding certain logical constraints in data as well as violations of these constraints. Their motivating use case is data cleaning.

Pruning the search space for inclusion dependencies by considering types is also proposed from Bauckmann et. al. in [25]. However, the authors of [25] argue that in scientific databases, the types often cannot be trusted. This may also hold true for generic data lakes, and it is up to the data scientist to ascertain that the type information is reliable. For this reason, we derive the actual datatypes from the NoSQL data, to ensure that mixed types are also recorded in the derived schema.

## VIII. Conclusion and Future Work

In this paper, we focus on data gathering in data lakes, managed in a schema-free or schema-flexible NoSQL database. Even when the database itself does not enforce a schema, data that is generated automatically (e.g., from sensors or exported from data repositories), is likely to adhere to a schema, even if this schema is not explicit.

However, when the data is being collected over long periods of time, the schema is likely to change eventually. In this work, we focus on restoring the schema evolution history, since being aware of the evolution process is a prerequisite for any meaningful data analysis: If the data lake is accessed programmatically, yet without knowledge about its structure, semantics, and history, the analytical results may turn out to be rather useless.

In particular, we highlight two key tasks, namely the extraction of a sequence of schema versions (rather than a single, global schema), as well as mappings between these versions. We have a particular interest in recognizing non-trivial changes due to copying or moving properties between entity types, which are typical denormalization tasks in handling NoSQL data stores.

This requires us to detect inclusion dependencies. We have proposed a basic algorithm and several optimizations. We have further evaluated our algorithm, and are able to point out its strengths, as well as room for improvement. In addition, we can execute this algorithm within *Darwin*, our middleware for NoSQL schema management tasks.

In future work, we plan to move in two directions. (1) First, in order to propose meaningful join conditions for *copy* and *move* operations, we need not derive the entire set of inclusion dependencies. A focus on inclusion dependencies that are suitable for specifying join conditions would effectively reduce the search space. (2) Moreover, we intend to derive further kinds of integrity constraints, in order to better describe the schema evolution history of data lakes. Again, any algorithms will have to be able to handle versioned, noisy, and complex structured NoSQL data.

## Acknowledgements

This project was funded by the *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation), grant #385808805. The authors thank the anonymous reviewers of SCDM'17 for their careful reading and their constructive input.

## References

[1] M. Farid, A. Roatis, I. F. Ilyas, H.-F. Hoffmann, and X. Chu, "CLAMS: Bringing Quality to Data Lakes," in *Proc. SIGMOD'16*, 2016.

[2] S. Cebiric, F. Goasdoué, and I. Manolescu, "Query-Oriented Summarization of RDF Graphs," *PVLDB*, vol. 8, no. 12, pp. 2012–2015, 2015.

[3] M. Klettke, U. Störl, and S. Scherzinger, "Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores," in *Proc. BTW'15*, 2015.

[4] L. Wang, O. Hassanzadeh, S. Zhang *et al.*, "Schema Management for Document Stores," *PVLDB*, vol. 8, no. 9, pp. 922–933, 2015.

[5] M.-A. Baazizi, G. G. Dario Colazzo, and C. Sartiani, "Counting Types for Massive JSON Datasets," in *Proc. DBPL'17*, 2017.

[6] P. J. Sadalage and M. Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison Wesley, 2012.

[7] U. Störl, D. Müller, M. Klettke, and S. Scherzinger, "Enabling Efficient Agile Software Development of NoSQL-backed Applications," in *Proc. BTW'17*, 2017.

[8] A. Ringlstetter, S. Scherzinger, and T. F. Bissyandé, "Data Model Evolution using Object-NoSQL Mappers: Folklore or State-of-the-Art?" in *Proc. BIGDSE'16*, 2016.

[9] S. Scherzinger, U. Störl, and M. Klettke, "Managing Schema Evolution in NoSQL Data Stores," in *Proc. DBPL'13*, 2013.

[10] S. Abiteboul, R. Hull, and V. Vianu, Eds., *Foundations of Databases*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[11] R. Agrawal, T. Imieliński, and A. Swami, "Mining Association Rules between Sets of Items in Large Databases," in *Proc. SIGMOD'93*, 1993.

[12] F. M. Harper and J. A. Konstan, "The MovieLens Datasets: History and Context," *ACM Transactions on Interactive Intelligent Systems (TiiS)*, vol. 5, no. 4, pp. 19:1–19:19, 2016.

[13] M. Lichman, "UCI Machine Learning Repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[14] C.-H. Moh, E.-P. Lim, and W. K. Ng, "DTD-Miner: A Tool for Mining DTD from XML Documents," in *WECWIS'00*, 2000.

[15] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren, "Inference of concise regular expressions and DTDs," *ACM TODS*, vol. 35, no. 2, pp. 11:1–11:47, 2010.

[16] W. Labio and H. Garcia-Molina, "Efficient Snapshot Differential Algorithms for Data Warehousing," in *Proc. VLDB'96*, 1996.

[17] A. Baqasah, E. Pardede, J. W. Rahayu, and I. Holubová, "XS-Diff: XML schema change detection algorithm," *IJWGS*, vol. 11, no. 2, pp. 160–192, 2015.

[18] D. S. Ruiz, S. F. Morales, and J. G. Molina, "Inferring Versioned Schemas from NoSQL Databases and Its Applications," in *Proc. ER'15*, 2015.

[19] P. Buneman, S. Davidson, W. Fan, C. Hara, and W.-C. Tan, "Keys for XML," in *Proc. WWW'01*, 2001.

[20] L. Kot and W. White, "Characterization of the Interaction of XML Functional Dependencies with DTDs," in *Proc. ICDT'07*, 2007.

[21] M. W. Vincent, M. Schrefl, J. Liu, C. Liu, and S. Dogen, "Generalized inclusion dependencies in XML," in *Proc. APWeb'04*, 2004.

[22] S. Fajt, I. Mlynkova, and M. Necaský, "On Mining XML Integrity Constraints," in *Proc. ICDIM'11*, 2011.

[23] T. Papenbrock and F. Naumann, "A Hybrid Approach for Efficient Unique Column Combination Discovery," in *Proc. BTW'17*, 2017.

[24] T. Papenbrock, S. Kruse, J. Quiané-Ruiz, and F. Naumann, "Divide & Conquer-based Inclusion Dependency Discovery," *PVLDB*, vol. 8, no. 7, pp. 774–785, 2015.

[25] J. Bauckmann, U. Leser, and F. Naumann, "Efficient and Exact Computation of Inclusion Dependencies for Data Integration," Universität Potsdam, Tech. Rep., 2010.