Relaxed Functional Dependency Discovery in Heterogeneous Data Lakes

Rihan Hai[1], Christoph Quix[2,3], Dan Wang[1]

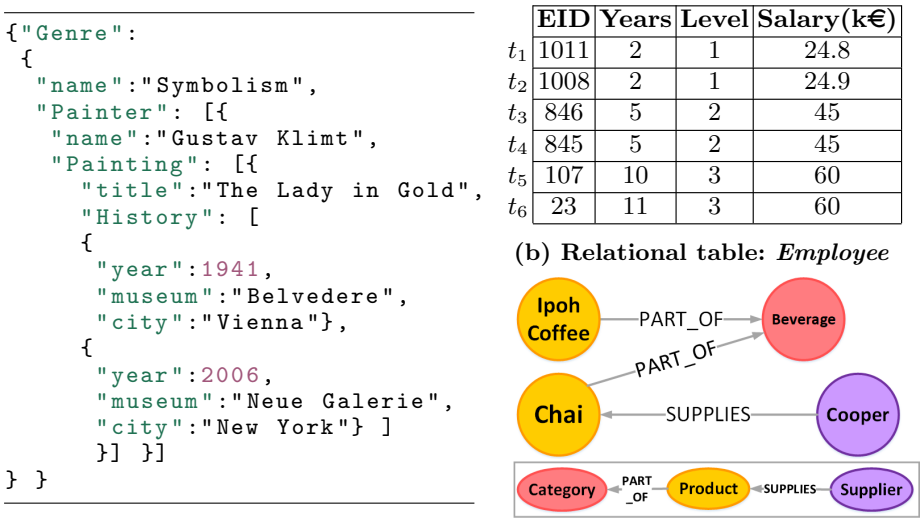
[1] RWTH Aachen University, Germany
[2] Hochschule Niederrhein, University of Applied Sciences, Germany
[3] Fraunhofer Institute for Applied Information Technology FIT, Germany
{hai,wang}@dbis.rwth-aachen.de, christoph.quix@hs-niederrhein.de

Abstract. Functional dependencies are important for the definition of constraints and relationships that have to be satisfied by every database instance. Relaxed functional dependencies (RFDs) can be used for data exploration and profiling in datasets with lower data quality. In this work, we present an approach for RFD discovery in heterogeneous data lakes. More specifically, the goal of this work is to find RFDs from structured, semi-structured, and graph data. Our solution brings novelty to this problem in the following aspects: (1) We introduce a generic meta-model to the problem of RFD discovery, which allows us to define and detect RFDs for data stored in heterogeneous sources in an integrated manner. (2) We apply clustering techniques during RFD discovery for partitioning and pruning. (3) We performed an intensive evaluation with nine datasets, which shows that our approach is effective for discovering meaningful RFDs, reducing redundancy, and detecting inconsistent data.

1 Introduction

Data lakes (DLs) have been proposed to tackle the problem of data access by providing a comprehensive repository, in which the raw data from heterogeneous sources is ingested in its original format [5]. Although DLs have been generally considered as a promising solution, they face the challenges that the ingested raw data often lack sufficient metadata or have inadequate data quality.

Functional dependencies (FDs) specify that attributes functionally depend on some other attributes, e.g., in Fig. 1b, the working years of employees determine their levels: $Years \rightarrow Level$. In contrast to such an *exact* FD, we might also be interested in discovering *relaxed* functional dependencies (RFDs). RFDs are relaxed in the sense that they do not apply to all tuples of a relation, or that *similar* attribute values are also considered to be equal [2]. In Fig. 1b we can have a RFD: $Years \rightarrow Salary$, if we consider 24.8 and 24.9 to be similar. RFDs are especially useful in cases where the source data have lower quality with inconsistencies and incorrect values. By using RFDs, we can detect additional relationships among data items for data exploration or profiling; on the other hand, we can identify the data objects that violate the detected constraints, which can be useful for data cleaning.



```
{"Genre":
 {
  "name":"Symbolism",
  "Painter": [{
   "name":"Gustav Klimt",
   "Painting": [{
     "title":"The Lady in Gold",
     "History": [
     {
      "year":1941,
      "museum":"Belvedere",
      "city":"Vienna"},
     {
      "year":2006,
      "museum":"Neue Galerie",
      "city":"New York"} ]
      }] }]
} }
```

(a) JSON document: Art genre

	EID	Years	Level	Salary(k€)
t_1	1011	2	1	24.8
t_2	1008	2	1	24.9
t_3	846	5	2	45
t_4	845	5	2	45
t_5	107	10	3	60
t_6	23	11	3	60

(b) Relational table: *Employee*

(c) Graph: Supply chain

Fig. 1: Running example: heterogeneous data in a data lake

Problem definition. We tackle the problem of discovering RFDs in DLs with structured data (relations), semi-structured data (JSON) and graphs. Fig. 1 shows our running example with JSON, relational, and graph data. JSON is a popular data model for semi-structured data, but it has not yet been investigated for RFD discovery. Discovery of FDs or RFDs in graphs often focuses on linked data in RDF (Resource Description Framework) [18], but not for labeled property graphs of Neo4j, which we address in our work.

Related works. Most existing solutions for FD discovery [12] focus on handling only one type of data, e.g., relational [9, 16], XML [4, 17], or graphs [18]. In this work we propose an approach based on a generic metadata model, which handles heterogeneous data in a unified manner and thus avoids the need for specific algorithmic solutions for each data type. It also leads to a clearer RFD definition, which simplifies the understanding over different data structures.

A number of RFD definitions and discovery solutions have been proposed [2]; they can be classified into three categories. To tolerate inconsistent data, the first type of RFDs [9] require that "almost" all tuples satisfy the dependency; usually an error threshold ε indicates the degree to which the dependency is allowed to be violated. The second category of RFDs [1] relax on how the attribute values are compared. Instead of grouping identical values, they group similar attribute values by using similarity functions (e.g., $Years \rightarrow Salary$ in our example). The third category of RFDs [4] are the hybrids of the previous two types. A recent approach [11] discovers RFDs based on sampling and a novel search space traversal strategy, which is efficient, yet the possibilities of supporting heterogeneous data and applying clustering techniques are not considered as in this work.

Our solution. We propose a clustering-based RFD discovery method for heterogeneous data such that we can find the "hidden" relationships among attributes in various datasets. To support inconsistent or incomplete data, our approach tolerates a certain degree of tuple violation. Our main contributions are: **(1)** We propose a generic metamodel and RFD definition, which facilitates an integrated method to perform RFD discovery over heterogeneous data. **(2)** We provide clustering-based algorithms to discover RFDs. **(3)** We propose a pruning procedure based on agglomerative clustering, which can effectively reduce the search space and thereby improve the performance. **(4)** We show experimentally that our approach produces more meaningful RFDs with less redundancy, and the discovered RFDs can be effectively used for error detection.

The remainder of the paper is organized as follows: first we introduce preliminary concepts in Sec. 2; we explain our proposed generic metadata model and definition of RFD in Sec. 3; then we discuss the overall approach in Sec. 4; we evaluate our solutions in Sec. 5, before we conclude the paper in Sec. 6.

2 Preliminaries

We define FDs and relevant concepts as in [9, 12]. Given a relational schema R and its instance r, for a tuple $t_i \in r$ and a set of attributes $X \subseteq R$, $t_i[X]$ denotes the projection of t_i on X. For a set of attributes $X \subseteq R$ and an attribute $A \in R$, the FD $X \rightarrow A$ *holds* iff for all pairs of tuples $t_i, t_j \in r$, if $t_i[X] = t_j[X]$, then $t_i[A] = t_j[A]$. We refer to X as the left-hand side (LHS) and A as the right-hand side (RHS).

A FD $X \rightarrow A$ is *minimal* if by removing any attribute from X, the FD is no longer satisfied. A FD $X \rightarrow A$ is *non-trivial* if $A \notin X$. Given two attributes $A, B \in R$, if we have $X \rightarrow A$ and $X \rightarrow B$, then $X \rightarrow AB$. Thus, it is sufficient to detect dependencies with singleton RHS. Like most existing works, our approach generates *minimal, non-trivial* RFDs with *singleton* RHS.

Partition. Given a set of attributes X, we group tuples with identical values projected on X, and obtain *equivalent classes*. In Fig. 1b, $\{t_1, t_2\}$ is an equivalence class regarding $Years$. By grouping all tuples in r into equivalent classes, we obtain a *partition* π_X of r regarding X. The number of equivalent classes $|\pi_X|$ is its *rank*. In Fig. 1b, $\pi_{Years} = \{\{t_1, t_2\}, \{t_3, t_4\}, \{t_5\}, \{t_6\}\}$, $\pi_{Level} = \{\{t_1, t_2\}, \{t_3, t_4\}, \{t_5, t_6\}\}$, and $|\pi_{Years}| = 4$, $|\pi_{Level}| = 3$. If we already have the partitions of π_X and π_Y, we can obtain a new partition of the attribute set $X \cup Y$ by computing the *product* of π_X and π_Y, i.e., $\pi_{X \cup Y} = \pi_X \cdot \pi_Y$ (Lemma 3.6, [9]). For instance, $\pi_{\{Years, Level\}} = \pi_{Years} \cdot \pi_{Level} = \{\{t_1, t_2\}, \{t_3, t_4\}, \{t_5\}, \{t_6\}\}$. If a set of attributes X has no two identical tuple values, we call X a *superkey*. If none of the strict subsets of X is a superkey, X is a *key*, e.g., EID in Fig. 1b. By removing equivalent classes whose sizes are one, we obtain *stripped partitions*, denoted as $\widehat{\pi}$. For example, $\widehat{\pi}_{Years} = \{\{t_1, t_2\}, \{t_3, t_4\}\}$. $||\widehat{\pi_X}||$ is the sum of the sizes of equivalent classes in $\widehat{\pi}$, e.g., $||\widehat{\pi_{Years}}|| = 4$.

FD inference. Given $\widehat{\pi_X}$, the error measure $e(X) = (||\widehat{\pi_X}|| - |\widehat{\pi_X}|)/|r|$, indicates the minimum fraction of tuples to be removed from r such that X

becomes a superkey. A FD $X \rightarrow A$ holds iff $e(X) = e(X \cup A)$ (Lemma 3.5, [9]). For example, $e(Years) = e(\{Years, Level\}) = \frac{4-2}{6}$, thus we have the FD: $Years \rightarrow Level$. Given a set of FDs (denoted as Σ) over schema S, a *cover* of Σ (denoted as σ) is a set of FDs satisfying: 1) $\sigma \subseteq \Sigma$; 2) any FD in Σ is either also in σ, or can be implied by σ, denoted as $\sigma \models \Sigma$.

3 Metadata Model and RFD Definition

In order to discover the RFDs for heterogeneous data sources, a primary step is to have a proper representation of the schemas. FD or RFD discovery usually works on relational structures. The tree-structured data models such as JSON, could be represented in a universal relation that contains all attributes of the JSON document with normalized atomic values. However, such a normalization causes failures to detect dependencies including array-based elements, and massively increases the number of tuples, which puts a huge burden on performance [17]. Therefore, our approach for RFD discovery preserves the hierarchical relationships among schema elements by applying a generic metadata model, which can also represent the schema of relational tables and Neo4j graphs.

Our metadata model is inspired by our previous model *GeRoMe* [10], but the present model is much simpler as we focus on RFD discovery. The model is similar to extended entity-relationship models with explicit types for attributes which might also be complex types to represent nested objects.

Definition 1. *A generic schema S is a tuple $\langle T, C, P, A\rangle$ with*

- *T is a set of atomic types, e.g., {number, string, boolean, ... };*
- *C is a set of complex types $\{c_1, \ldots, c_n\}$ where each c_i has a set of properties $prop(c_i) \subseteq P$ and super types $super(c_i) \subseteq C$;*
- *P is a set of properties $\{p_1, \ldots, p_m\}$ where each p_i has a type, i.e., $type(p_i) \in C \cup T$, and each p_i might be unique ($unique(p_i) \in \{true, false\}$) or multi-valued ($multi(p_i) \in \{true, false\}$);*
- *A is a set of binary association types $\{a_1, \ldots, a_k\}$ where each a_i has a set of properties $prop(a_i) \subseteq P$, a source and a target type ($source(a_i) \in C$ and $target(a_i) \in C$).*

A dataset as an instance of a schema is logically represented by a set of tuples of different arities. For example, if the schema has a complex type c with n properties, then an instance of c is a tuple with $n+1$ attributes: $c(id, v_1, \ldots, v_n)$ where id is a unique object identifier and each v_i represents a property value. Instances of an association type a_i with m properties are represented as tuples $a_i(id, o_1, o_2, v_1, \ldots, v_m)$ with id being a unique identifier for this association, o_1/o_2 being the identifier of the source/target object and each v_i is a property value. Note that tuples of complex or association types might have nested structures as the properties might have complex types and/or be multi-valued.

Definition 2. *A generic functional dependency (gFD) for a generic schema $S = \langle T, C, P, A\rangle$ is an expression of the form: $[X_0]X_1, \ldots, X_n \rightarrow Y_1, \ldots, Y_m$*

- X_0 *is an absolute path expression starting with a complex or association type* $t \in C \cup A$*, defining the* context *of the gFD, resulting in a set of objects* O*;*
- $X_1, \ldots, X_n, Y_1, \ldots, Y_m$ *are relative path expressions that are evaluated relative to the results of* X_0 *and select for each* $o \in O$ *a single value, i.e., generating a virtual relation* R *with tuples of the form* $r(o, x_1, \ldots, x_n, y_1, \ldots, y_m)$*;*
- *a path expression has the form* $s_0.s_1.\ldots.s_k$*, where* $s_0 \in C \cup A$ *is the initial step for an absolute path expression to select the starting type, and* $s_1, \ldots, s_k$ *are steps in absolute or relative path expressions that refer either to properties or association types to navigate in the schema;*
- *the semantics is as for plain FDs:* $\forall t_i, t_j \in R : t_i[X] = t_j[X] \Rightarrow t_i[Y] = t_j[Y]$.

In the following, we briefly explain how different types of data sources are virtually mapped to this generic representation of schemas and FDs.

Representation of JSON. The types of objects in JSON are represented by complex types in our model, as they might have properties and associations. JSON arrays are represented as multi-valued properties.

Example 1 *In Fig. 1a,* Genre *is a* complex type *with two properties:* name *with an atomic type and* Painter *with a complex type. Some properties (e.g.,* Painting*) are multi-valued. This gFD states that each painting is owned by one museum in the corresponding year:* $[Genre.Painter.Painting]title, History.year \rightarrow History.museum$

Representation of Neo4j graphs. A Neo4j graph separates data into nodes and relationships. A node has a collection of properties in key-value pairs and a set of labels that specify the node type. A relationship is a directed edge connecting two entity nodes. In our metadata model, nodes and their properties are represented as complex types. A node may have multiple labels indicating its domain roles, which we specify as *super types*. We represent a relationship as *association*, whose *source* and *target* identify the corresponding in- and outgoing complex types. A relationship in Neo4j usually also has a set of *properties* that can be attached to the association in our model.

Example 2 *In Fig. 1c, the nodes* Chai *and* Ipoh Coffee *both have the label* Product*, which is a* complex type*, as well as* Category *and* Supplier*. The relationships* PART_OF *and* SUPPLIES *are* association types*. In addition to Fig. 1c,* Product *has the properties* productName*,* unitInStock *and* unitPrice*, while* Supplier *has* ID *and* name*, which all have atomic types. The association* SUPPLIES *has a property* purchasePrice*. The gFD below defines that for each supplied product, the purchase price depends on the supplier ID and the product name:* $[Supplier.SUPPLIES]source.ID, target.productName \rightarrow purchasePrice.$

Subsequently, gFDs that relate properties from several complex or association types are referred to as **inter-FDs**, in contrast to **intra-FDs** defined within the scope of one type. Usual FDs in relational schemas are intra-FDs.

Based on our metadata model, we now define RFDs.

Definition 3. *Given a schema $S = \langle T, C, P, A \rangle$ and its instance r, a RFD f is in the form:* $[X_0]X_1, \ldots, X_n \xrightarrow[c_X, c_Y]{\varepsilon} Y_1, \ldots, Y_m$

- *the basic structure of the RFD is as for gFDs defined in Definition 2;*
- *ε is the error threshold that indicates the minimal percentage of objects to be removed from O (the result of X_0) such that the RFD becomes valid, i.e., $\Psi(f) = min\{|O_1| \; |O_1 \subseteq O \text{ and } f \text{ holds for } O \backslash O_1\}/|O| \leq \varepsilon$ [9];*
- *c_X and c_Y are values that indicate the clustering over the values of the LHS and RHS of f; values within the same cluster are considered to be equal.*

Example 3 *The following examples define RFDs given the relation in Fig. 1b:*
(a) $[Employee]Level \xrightarrow[1.0,1.0]{0.2} Years$ *(b)* $[Employee]Salary \xrightarrow[0.998,1.0]{0} Level$
The first example states that the dependency holds if we remove no more than 20% tuples (e.g., t_5 or t_6) from the relation; the second example requires all the tuples to satisfy the dependency.

The values c_X, c_Y are indicators for the clustering applied to the instance values. A higher value indicates that elements are well matched to their own cluster, while lower values indicate that more dissimilar elements are grouped in one cluster. In Sec. 4, we will use the silhouette coefficient (SC) [14] as indicator, which has not been applied in existing RFD discovery approaches. In Example 3b, the salary values 24.8 and 24.9 are grouped in the same cluster, and with a SC of 0.998 we can say *Salary* is "well-clustered".

4 Our Approach

4.1 Approach Overview

Algorithm 1 describes our approach with a dataset D and error threshold ε as input. We first extract the schema of D and represent it in our generic metamodel. The dataset is transformed into a decomposed form, which basically corresponds to the first normal form of the logical instance representation discussed in Sec. 3 (all properties are transformed to atomic values). The original hierarchical structures and relationships are maintained as the schema information (see Sec. 4.2). We refer to the resulting tables as *property tables*, and they are preprocessed in two steps. First, we filter long textual attributes which lead to less interesting RFDs without semantic significance (long texts are often unique). The second step is to group the same values of an attribute using hash partition.

We prune and remove certain attributes, if they are found to be *equivalent* by our feature-based agglomerative clustering (see Sec. 4.3). Then, we discover the intra-RFDs by *X-means* clustering (Sec. 4.4). For JSON or graphs, a RFD may also exist among properties of different complex types or association types. Therefore, we also try to discover whether there exist inter-RFDs (Sec. 4.5). The union of intra-RFDs and inter-RFDs is returned as the final result.

Algorithm 1: RFD Discovery

Input: Relational/JSON/graph dataset D, threshold ε; **Output**: Set of RFDs

$R \leftarrow ExtractSchema(D)$; $D' \leftarrow 1NF\_Decomposition(D, R)$; $\Sigma \leftarrow \emptyset$; $P_r \leftarrow \emptyset$
foreach *property table* $d \in D'$ **do**
  $P_0 \leftarrow HashPartition(FilterLongTextualAttrs(d))$; $P \leftarrow \emptyset$ `// Preproc.`
  $C \leftarrow FeatureAgglomeration(P_0)$ `// Prune equivalent attributes`
  foreach *attribute* $c \in C$ **do**
    $(P_c, s_c) \leftarrow ClusterPartition(P_0, c)$ `// Obtain equivalent classes`
    Add P_c to P, add s_c to Φ `// Φ: silhouette coefficient`
  $\Sigma \leftarrow \Sigma \cup IntraRFDDiscovery(P, R, \varepsilon, \Phi)$; $P_r \leftarrow P_r \cup P$
$\Sigma' \leftarrow InterRFDDiscovery(P_r, R, \varepsilon, \Phi, \Sigma)$
return $\Sigma \cup \Sigma'$

4.2 Schema Inference and Dataset Decomposition

A relational dataset can be directly processed for RFD discovery; a JSON or graph dataset needs some preprocessing to generate tables with atomic values only. Our DL system [5, 8] loads a JSON document and extracts its schema using Apache Spark[4]. Based on the extracted schema, we decompose the JSON documents into a set of *property tables*. We perform decomposition as follows: **(1)** For each complex type, we create a property table for it with an identifier ID and each property as a column. **(2)** If a complex type c_2 is a property of another complex type c_1, we create a column *parent identifier (PID)* with the identifier of c_1 in the table of c_2. **(3)** If there is an association type (e.g., a relationship in Neo4j graph), we create a new association property table with columns storing its properties, e.g., identifiers of source and target objects. **(4)** If a *Property* is multi-valued, we generate a value table.

Example 4 *Table 1 shows the decomposed property tables, one for each complex type. Except the root node* Genre*, each property table has a column PID. Note that we use* ***path-based*** *table names, thus we can uniquely find an attribute in a property table. Graphs of Neo4j are processed similarly, but we retrieve the schema using the APOC library*[5]*. In addition to the steps above, we create* path *tables for inter-RFD discovery. In the evaluation, we limit the length of paths to 3, as longer paths often do not lead to meaningful results. Table 2 shows three examples of property tables: the left one for the complex type* Supplier*, the middle one for the association type* SUPPLIES*, and the last one which is a path table including IDs of all nodes and relationships on the path.*

4.3 Pruning Rules

Given a dataset whose number of attributes is m, and number of tuples is $|r|$, the lattice-based FD discovery has the complexity $\mathcal{O}(|r|^2 \left(\frac{m}{2}\right)^2 2^m)$ [12]. Thus,

[4] `https://spark.apache.org/`
[5] `https://neo4j-contrib.github.io/neo4j-apoc-procedures/`

Table 1: Decomposed result of Fig. 1a

Genre

ID	name
1	Symbolism

Genre.Painter

ID	PID	name
11	1	Gustav Klimt

Genre.Painter.Painting

ID	PID	title
101	11	The Lady in Gold

Genre.Painter.Painting.History

ID	PID	year	museum	city
1001	101	1941	Belvedere	Vienna
1002	101	2006	Neue Galerie	New York

Table 2: Partial decomposed result of Fig. 1c

Supplier

ID	name
12	Cooper

Supplier-SUPPLIES-Product

ID	SrcID	TgtID	purchasePrice
101	12	35	35

Supplier-SUPPLIES-Product-PART_OF-Category

node1_ID	rel1_ID	node2_ID	rel2_ID	node3_ID
12	101	35	102	18

pruning candidate FDs has been intensively conducted in existing works for efficiency. Existing pruning rules [9] mainly check whether an attribute is a key, or whether a candidate FD can be inferred from existing dependencies. Besides implementing the existing pruning rules [9,12,17], we have designed the following pruning procedure based on agglomerative clustering.

Equivalent attributes. We call two attributes A and B *equivalent* if two RFDs $A \to B$ and $B \to A$ both hold [16]. We can replace A with B in the LHS or RHS of a dependency. In this way, we have less RFDs; yet, it preserves the same information, which leads to a smaller cover of the final RFD set.

Pruning based on agglomerative clustering. To detect equivalent attributes, we use agglomerative (hierarchical) clustering; for generality, we calculate the distances using Ward's method [15] as criterion. The first step is feature selection. Recall in Sec. 2 we introduced that a FD can be inferred from the calculation of $e(X)$, which depends on the values of the stripped partition rank $|\widehat{\pi_X}|$ and its sum $||\widehat{\pi_X}||$. Thus, we calculate the values of $|\widehat{\pi}_A|$ and $||\widehat{\pi}_A||$ for each attribute to form the feature matrix for agglomerative clustering. Given two attributes A and B, their squared Euclidean distance is $d_{AB} = \sqrt{(|\widehat{\pi}_A| - |\widehat{\pi}_B|)^2 + (||\widehat{\pi}_A|| - ||\widehat{\pi}_B||)^2}$. During the hierarchical clustering process, the values of distance among current clusters are monotonic. Thus, we can determine the optimal number of clusters by detecting the change point[6] in the distance slope, and use it as the termination condition. Note that [16] also discovers equivalent attributes, yet it examines strict value equality. In this work we detect equivalence by applying agglomerative clustering, which provides the relaxation. Finally, we set attributes in each cluster as equivalent attributes for pruning.

Example 5 *Fig. 2a shows a few additional tuples which we added to Fig. 1b. Since* EID *is a key and already pruned, we calculate the feature matrix for the remaining attributes (cf. Fig. 2b), e.g., by comparing similar values of* Salary *we obtain stripped partition* $\widehat{\pi}_{Salary} = \{\{t_1, t_2\}, \{t_3, t_4\}, \{t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}\}\}$ *(t_{12} is stripped out as its equivalent class size is one). Thus, in the third row of the matrix in Fig. 2b we insert* $|\widehat{\pi}_{Salary}| = 3,\ ||\widehat{\pi}_{Salary}|| = 11$. *Fig. 2c shows the hierarchical clustering result using the matrix, and the red line indicates that the clustering terminates with two clusters when it reaches the change point. Since*

[6] Library used in implementation: `https://github.com/deepcharles/ruptures`

	EID	Years	Level	Salary
t_7	108	10	3	60
t_8	25	11	3	60
t_9	11	12	3	60
t_{10}	10	12	3	60.2
t_{11}	9	13	3	60.2
t_{12}	3	13	3	200

$$\begin{array}{c} \\ \text{Years} \\ \text{Level} \\ \text{Salary} \end{array} \begin{array}{c} |\widehat{\pi}| \ ||\widehat{\pi}|| \\ \begin{bmatrix} 6 & 12 \\ 3 & 12 \\ 3 & 11 \end{bmatrix} \end{array}$$

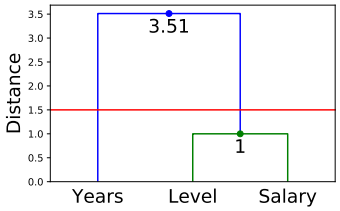


Fig. 2: (a) Added tuples; (b) Feature matrix; (c) Clustering result

Level *and* Salary *are in the same cluster, we have them as equivalent attributes. We can prune* Salary *and only examine RFD candidates including* Level*.*

4.4 Clustering-based RFD Discovery

To provide the relaxation in attribute value comparison, for partition we use clustering instead of exact value matching.

Our partition method *ClusterPartition* is invoked from the main algorithm with the previous partition P_0 and the current attribute c as input, and produces the new clustering-based partition P_c. The second output s_c is the silhouette coefficient (SC) for c, which constitutes c_X and c_Y in Definition 3.

Our approach automatically determines the optimal number of clusters by applying the clustering method *X-means* [13], which conducts an initial clustering, then repeatedly performs local *K-means* to split each cluster, and selects the most promising subset of clusters using Bayesian Information Criterion.

The upper bound u for the number of clusters is the rank of the initial partition P_0, the lower bound l is 2. To make *X-means* more efficient for a potentially large number of clusters, we do a preprocessing to narrow down the potential values for the number of clusters. The preprocessing uses a binary search strategy to refine the upper and lower bounds. In each step, a *K-means* clustering is performed with $K = \frac{u+l}{2}$ and the search continues in that part with better SCs. From the intensive experiments over real world datasets, we found that the clustering results become less meaningful with SC below 0.8; thus, we will ignore clusterings with a SC less than 0.8. If the initial rank of P_0 is low (e.g., less than 50), then the preprocessing step is skipped as *X-means* is already quite efficient in this case. The final result of the *ClusterPartition* procedure is a new partition and the corresponding SC.

In [18], an algorithm for automatically determining the number of clusters for discovering dependencies in RDF graphs was also proposed. However, [18] is based on gap statistics and *K-means*, and increases the value of K by one per step, which is inefficient for a large value of K. In contrast, *X-means* selects the most promising subset of clusters for refinement per *K-means* sweep, and we apply binary search to fasten the procedure with a large K.

After obtaining the partition, the function *IntraRFDDiscovery* (Alg. 1, line 8) computes the stripped partition and infers the RFD using the error measure in

Definition 3. We also generate the RFD candidates based on the attribute lattice similar to TANE [9].

4.5 Inter-RFD Discovery

For JSON documents, a RFD may exist among attributes of different types. Thus, we also need to examine such *inter-RFDs*, whose LHS/RHS are attribute sets from different tables after decomposition. The key challenge in inter-RFD discovery is how to efficiently "group" property tables to avoid an exhaustive search. More specially, regarding partition and RFD inference, discovering inter-RFD is similar to intra-RFD which we have introduced. However, for inter-RFD we need to combine the attributes from a set of property tables, which implies much more RFD candidates than the intra-RFD discovery for a single property table. We mainly developed the following rules extended from [17] to make the inter-RFD discovery more efficient:
(1) If a property table T_2 is generated from a multi-valued property, we group it with its parent table T_1, e.g., $(History, Painting)$ in Fig. 1a.
(2) If a RFD created from rule (1) is valid for tuples with the same parents (PIDs), but does not hold in all the tuples, we add the next ancestor table farthest from the root. We repeat this step until we find a valid inter-RFD or we reach the root table. For instance, if the RFD candidates generated from $(History, Painting)$ do not hold, we examine $(History, Painting, Painter)$.
(3) We group the parent table with its child tables, or two sibling tables if they have similar number of records. The purpose of this rule is to group property tables with similar number of records ($|r|$ for computing $e(X)$), otherwise there will be a number of null values, and it rarely leads to valid RFDs.

Note that for (1) and (2), [17] has proposed algorithms to compute inter-RFDs from invalid intra-RFDs, which hold for the instance values of each individual complex type, but might not for the whole JSON document or graph. For instance, in the $History$ table in the right of Table 1, the intra-RFD: $year \rightarrow museum$ holds only for the same painting (e.g., PID as 101). It is very likely that there exist other painting records with year 2006 but in other museums, i.e., $year \rightarrow museum$ is not valid. However, within the grouping $(History, Painting)$ if we add the attribute *title* in LHS, we might find a valid inter-RFD, e.g., $[Genre.Painter.Painting]title, History.year \rightarrow History.museum$.

After obtaining the attributes in the same group, the search space construction, pruning and RFD discovery are similar with intra-RFDs. In particular, for Neo4j property graphs, the inter-RFD discovery procedure is similar but among node/relationship/path tables from the shortest paths to the longest paths.

5 Evaluation

We have evaluated our approach over nine datasets to compare with a baseline approach (Sec. 5.1), and demonstrate that our approach discovers more meaningful dependencies with less redundancy. In Sec. 5.2, we show that our approach is fault-tolerant, and can be used to detect inconsistent data.

Table 3: Relational datasets

Name	Rows#	Attr#
Adult	32561	15
Bio	184292	9
Wiki Image	777676	12

Table 4: JSON datasets

Name	Records#	Objects#	Attr#	L
TVshows	20	18	47	5
Restaurant	25357	4	13	3
Profile	1561	3	15	2

Table 5: Graph datasets

Name	Nodes#	Rel#	NodeTypes#	NodeProp#	RelTypes#	RelProp#
Movie	171	250	2	5	4	1
Northwind	1035	3139	5	51	4	5
WorldCup	83382	156673	13	30	16	0

Table 6: Number of FDs/RFDs: baseline and our approach ($\varepsilon = 0$)

Name	Adult	Bio	Wiki Image	TVshows	Restaurant	Profile	Movie	Northwind	WorldCup
Baseline	78	30	66	603	44	75	27	1749	826
Our approach	46	10	66	259	44	40	27	2105	715

Experimental setting. We have conducted the experiments in a server running Ubuntu 14.04 LTS, with two Intel Xeon X5647@2.93GHz CPUs (8 logical cores per CPU) and 16G RAM. The tested relational, JSON, and graph datasets are stored in MySQL 5.5.62, MongoDB 3.4.16, and Neo4j 3.4.5, respectively. The decomposed tables are stored in MySQL and the discovered RFDs are stored in MongoDB. We have implemented the decomposition in Java and the other algorithms in Python 3.5, which are embedded in our DL system *Constance* [5].
Tested datasets[7]**.** Table 3 shows the name, number of rows and attributes of the relational datasets: *Adult*, *Bio* and *Wiki Image*. Table 4 provides the number of JSON records, number of objects/attributes and nesting levels (L) in JSON schema for three JSON datasets. Table 5 reports the number of entity nodes and relationships in the data, the number of nodes types and relationship types, and their corresponding properties in the metadata of graph datasets.

5.1 Baseline Comparison of Discovered Dependencies

The main goal of our clustering-based method and pruning procedure in Sec. 4 is to discover meaningful RFDs and reduce redundancy. Thus, we first report the results of discovered RFDs from all the datasets. For comparison, we have implemented a baseline algorithm using the TANE approach [9]. Since TANE only supports relational data, in the baseline we also apply our rules for inter-FD discovery from JSON/graph datasets. Thus, the comparison mainly reflects the effect of relaxation brought by *ClusterPartition* procedure and pruning using agglomerative clustering.

Reduce redundancy. By allowing a ratio ε of violating tuples, it usually leads to a larger number of RFDs than the exact FDs. To reduce this effect such

[7] Links: `http://dbis.rwth-aachen.de/cms/staff/hai/RFDDiscovery/datasets`

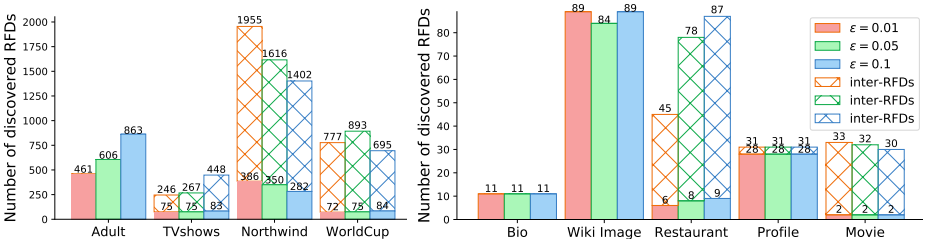


Fig. 3: Number of discovered RFDs in nine datasets with different ε

that we can observe the impact of our pruning method, we set error threshold ε as 0 in this test (Table 6). We can observe that in most datasets, our approach has a smaller or equal number of discovered dependencies, because with the agglomerative clustering we detect equivalent attributes and prune the dependencies accordingly. Thus, our results have less redundancy and imply all the FDs generated by the baseline.

Discover more meaningful dependencies. Now we assign ε with typical error threshold values: 0.01, 0.05, and 0.1. Fig. 3 depicts the impact of ε on the number of discovered RFDs. For JSON/graph datasets, the upper hatched parts indicate the number of inter-RFDs. For instance, for *TVshows* with $\varepsilon = 0.01$, there are 75 intra-RFDs and 246 RFDs in total (171 inter-RFDs). Combining Fig. 3 and Table 6, we observe that in most datasets there are more RFDs when $\varepsilon > 0$ than $\varepsilon = 0$; these dependencies do not exist in the FD results of the baseline approach. For instance, for *TVshows* with ε as 0.1 we obtain the RFD: $[root.embedded.episodes]name \rightarrow airtime$. It indicates that the episode name (not a key attribute) can functionally determine the episode airtime. Such meaningful dependencies can be found by our RFD discovery approach but not by the baseline approach. Moreover, with a larger value of ε, we often find more RFDs, e.g., the results of datasets *Adult*, *TVshows* and *Restaurant* in Fig. 3.

Moreover, in some cases we found that the discovered RFDs are "more compact" than FDs, i.e., with less attributes in LHS. For example, in the *Northwind* dataset, there exists a FD f_1 generated by the baseline: $Customer.postalCode$, $Order.shipRegion \rightarrow Order.shipCountry$. In our approach we produce the RFD f_2: $Customer.postalCode \rightarrow Order.shipCountry$ with ε as 0.05. By clustering similar values instead of exact value matching, and allowing a degree of tuple violation, we find RFDs like f_2. In addition, more RFDs will be pruned as we keep only minimal RFDs. Thus, in Fig. 3 we observe that with a higher value of ε, the number of discovered RFDs decreases in datasets *Northwind* and *Movie*.

5.2 Noise Tolerance and Error Detection

Besides finding interesting RFDs to enrich metadata in data lakes, another goal in this work is to apply our approach over dirty data, then use the discovered RFDs for error detection. We designed the below experiments to examine whether our discovered RFDs meet such a requirement.

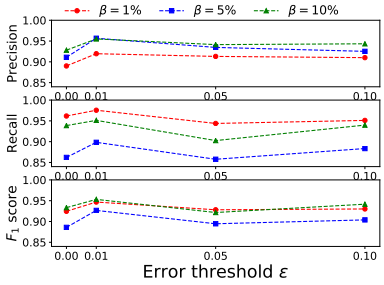


Fig. 4: Fault tolerance results

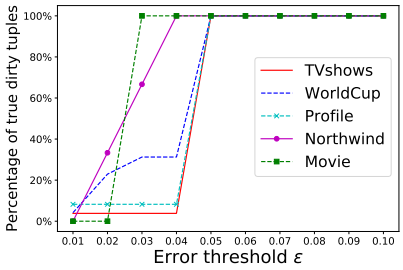


Fig. 5: Error detection results

Noise tolerance. In line with RFD-based data quality works [3], we obtain *dirty data* by adding Gaussian noise ($mean = 0.0$, standard deviation $\beta = [1\%, 5\%, 10\%]$) to the values of 10% of the existing numerical attribute tuples in every dataset. If a RFD discovered from the dirty data can also be found from the clean data, we consider it as the *true positive*. We divide the amount of true positives by the total numbers of RFDs discovered in the dirty data, and obtain *precision*; for *recall* we divide the number of true positives by the total number of RFDs in the original clean data. Fig. 4 shows the precision/recall/F_1 score results (y-axis) of our approach with error thresholds (x-axis) of the dataset *TVshows*.[8] We can observe that with different values of noises and error thresholds, our approach maintains a satisfying accuracy ($F_1 > 0.85$). Thus, even for a dataset with inaccuracy, our approach can effectively discover RFDs.

Error detection. In this experiment for each dataset, we insert additional dirty tuples whose values are inconsistent with the original data. Then we use the discovered RFDs to find violating tuples, and examine whether they are the inserted dirty tuples. Fig. 5 shows the results of 5 datasets with the percentage of dirty tuples as 5%. The x-axis shows the value of ε used to generate the RFDs, and the y-axis is the percentage of detected true dirty tuples by using these RFDs. In all datasets, we can observe that the percentage of detected error data has a significant increase when ε is getting close to the actual error data rate, although with different varying trends. This indicates that the choice of error threshold value plays a crucial role in finding all inconsistent data. For practical use when the error rate is unknown in a newly imported dataset, we recommend to run our approach with increasing values of error threshold, until the detected error data becomes stable.

6 Conclusion

We have addressed the problem of RFD discovery for heterogeneous data lakes. Our clustering-based approach groups similar attribute values. With our generic metadata model, we provide a unified definition for RFDs, thereby enabling integrated methods for processing different types of data, e.g., relational, JSON, and

[8] Full results: `http://dbis.rwth-aachen.de/cms/staff/hai/RFDDiscovery/res`.

graph data. We have designed a pruning procedure using agglomerative clustering, which can effectively prune RFD candidates. We have shown experimentally with nine datasets that our approach can find semantically interesting RFDs, which are less redundant compared to the classical approach. Our approach is also fault tolerant, and the discovered RFDs can be used for detecting dirty data. In the future, we plan to use the obtained RFDs for other tasks in data lakes such as schema mapping [6, 7].

Acknowledgements: The authors would like to thank the German Research Foundation DFG for the kind support within the Cluster of Excellence "Internet of Production" (Project ID: EXC 2023/390621612).

References

1. R. Bassée and J. Wijsen. Neighborhood Dependencies for Prediction. In *Proc. PAKDD*, pages 562–567, 2001.
2. L. Caruccio, V. Deufemia, and G. Polese. Relaxed functional dependencies - a survey of approaches. *IEEE Trans. Knowl. Data Eng.*, 28(1):147–165, 2016.
3. G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *Proc. VLDB*, pages 315–326, 2007.
4. F. Fassetti and B. Fazzinga. Approximate functional dependencies for XML data. In *Proc. ADBIS*, 2007.
5. R. Hai, S. Geisler, and C. Quix. Constance: An Intelligent Data Lake System. In *Proc. SIGMOD*, pages 2097–2100. ACM, 2016.
6. R. Hai and C. Quix. Rewriting of plain so tgds into nested tgds. *Proceedings of the VLDB Endowment*, 2019.
7. R. Hai, C. Quix, and D. Kensche. Nested Schema Mappings for Integrating JSON. In *Proc. ER*, pages 397–405, 2018.
8. R. Hai, C. Quix, and C. Zhou. Query rewriting for heterogeneous data lakes. In *Proc. ADBIS*, pages 35–49, 2018.
9. Y. Huhtala et al. TANE: an efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42(2):100–111, 1999.
10. D. Kensche, C. Quix, X. Li, Y. Li, and M. Jarke. Generic schema mappings for composition and query answering. *Data & Knowledge Eng.*, 68(7):599–621, 2009.
11. S. Kruse and F. Naumann. Efficient discovery of approximate dependencies. *Proceedings of the VLDB Endowment*, 11(7):759–772, 2018.
12. J. Liu, J. Li, C. Liu, and Y. Chen. Discover Dependencies from Data - A Review. *IEEE Trans. Knowl. Data Eng.*, 24(2):251–264, 2012.
13. D. Pelleg, A. W. Moore, and Others. X-means: Extending k-means with efficient estimation of the number of clusters. In *Proc. ICML*, pages 727–734, 2000.
14. P. J. Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.*, 20:53–65, 1987.
15. J. H. Ward Jr. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association*, 58(301):236–244, 1963.
16. H. Yao, H. J. Hamilton, and C. J. Butz. FD_Mine: Discovering functional dependencies in a database using equivalences. In *Proc. ICDM*, pages 729–732, 2002.
17. C. Yu and H. V. Jagadish. XML schema refinement through redundancy detection and normalization. *VLDB J.*, 17(2):203–223, 2008.
18. Y. Yu and J. Heflin. Extending Functional Dependency to Detect Abnormal Data in RDF Graphs. In *Proc. ISWC*, pages 794–809, 2011.