Covering or Complete? Discovering Conditional Inclusion Dependencies

Jana Bauckmann, Ziawasch Abedjan, Ulf Leser,
Heiko Müller, Felix Naumann

Technische Berichte Nr. 62

des Hasso-Plattner-Instituts für
Softwaresystemtechnik
an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam | 62

Jana Bauckmann | Ziawasch Abedjan | Ulf Leser |
Heiko Müller | Felix Naumann

Covering or Complete?

Discovering Conditional Inclusion Dependencies

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.de/ abrufbar.

Universitätsverlag Potsdam 2012
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URL http://pub.ub.uni-potsdam.de/volltexte/2012/6208/
URN urn:nbn:de:kobv:517-opus-62089
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-62089

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-212-4

Covering or Complete? Discovering Conditional Inclusion Dependencies

Jana Bauckmann[1] Ziawasch Abedjan[1] Ulf Leser[2]
Heiko Müller[3] Felix Naumann[1]
[1]firstname.lastname@hpi.uni-potsdam.de
[2]leser@informatik.hu-berlin.de
[3]heiko.mueller@csiro.au

[1]Hasso Plattner Institute, Potsdam, Germany
[2]Humboldt-Universität zu Berlin, Germany
[3]Intelligent Sensing and Systems Laboratory, Hobart, Australia

Abstract. Data dependencies, or integrity constraints, are used to improve the quality of a database schema, to optimize queries, and to ensure consistency in a database. In the last years conditional dependencies have been introduced to analyze and improve data quality. In short, a conditional dependency is a dependency with a limited scope defined by conditions over one or more attributes. Only the matching part of the instance must adhere to the dependency. In this paper we focus on conditional inclusion dependencies (CINDs).
We generalize the definition of CINDs, distinguishing covering and completeness conditions. We present a new use case for such CINDs showing their value for solving complex data quality tasks. Further, we define quality measures for conditions inspired by precision and recall. We propose efficient algorithms that identify covering and completeness conditions conforming to given quality thresholds. Our algorithms choose not only the condition values but also the condition attributes automatically. Finally, we show that our approach efficiently provides meaningful and helpful results for our use case.

1 Problem Statement

Studying data dependencies, or integrity constraints, has a long history in database research. Traditionally, integrity constraints are defined such that all tuples in a table must obey them, and they have mostly been used to preserve consistency in a database and as a hint to the query optimizer. Recently, a weaker form of dependencies, so called *conditional dependencies*, have gained attention, mostly due to their power in analyzing and improving data quality [9]. A conditional dependency is a dependency with a limited scope, where the scope typically is defined by conditions over several attributes. Only those tuples for which these conditions evaluate to true must adhere the dependency. Research has focussed on two types, i.e., conditional functional dependencies (CFDs) [4] and conditional inclusion dependencies (CINDs) [5]. Results have been published

on reasoning about consistency and implication [5, 6], validation of known conditional dependencies [6], or detection of conditional functional dependencies [11, 13].

Interestingly, detecting conditional inclusion dependencies has yet received little attention. Here we present an entirely new case for CINDs, which clearly shows their value for solving complex data quality tasks. To this end we generalize the established definition for CINDs to cover a subtle, yet important difference between different classes of CINDs. We present algorithms to detect all CINDs and evaluate the algorithms' efficiency.

1.1 Motivating Use Case

Our motivation for studying CINDs comes from the problem of describing links between objects on the web. Consider, as a concrete example, the problem of interlinking representations of persons in the English and German version of DBpedia [**?**]. Clearly, many persons have both an English and a German description in DBpedia. Relationships between entries in DBpedia are either represented by using the same URI or by "sameAs"-links; we refer to these relationships as *links*. The relationship between corresponding entries for the same person, however, is not made explicit in all cases. Having a German (English) description of a person without a link to an English (German) description of a person, two situations are possible: (1) This English (German) description does not exist; then the lack of the link truly reflects the database. (2) The English (German) description does exist; then the missing link is a data quality problem. Such problems are very common in scenarios where heterogeneous data sets have overlapping domains but no central authority takes care of properly and bi-directionally linked objects in this overlap. Many examples of such cases can be found in the world of Linked Open Data [15] and in dataspaces [12].

The key idea to identify candidates for missing links is to look for characteristics of those persons in the German DBpedia that are also included in the English DBpedia and vice versa. Most probably, a certain set of persons in the German DBpedia is interesting to US or English readers (and vice versa), but the question is how this set can be characterized. A quite reasonable guess is, for instance, that "German" persons with a birthplace or place of death in the United States are much more likely to also be represented in the English DBpedia. We propose a method to find such hypotheses automatically by computing a special kind of CINDs between the sets of interlinked objects. Objects that also have the same characteristics but are not yet linked are then good candidates for missing links. Note that our approach does not perform record linkage per se, but merely suggests good candidates of one dataset that might also be represented in the other data set. Focusing record linkage on a subset of entities is an important feature of efficient record linkage strategies. We show in Sec. 5 that our identified conditions indeed provide a reasonable (and interesting and unforeseen) semantic distinction between included and non-included persons.

We use the following relational schema to represent information about persons: `person(pid, cent, description)`, `birthplace(pid, bplace)`, and `deathplace(`

pid, dplace) with foreign key relationships from birthplace.pid to person.pid and from deathplace.pid to person.pid. Each person has an identifier (pid; mostly a person's name), century of birth (cent), and a description. The separate relations for place of birth and place of death result from the fact that persons in DBpedia can have several places of birth or death distinguishing for example the country, region, and city of birth or death. For example, for the actor Cecil Kellaway the places of birth are Kapstadt and Südafrika and places of death are Los Angeles, Kalifornien, and United States in the German version of DBpedia. Figure 1 shows (part of) the result of the full outer join over relations person, birthplace, and deathplace on the foreign key attributes in the English version of DBpedia (Person_EN) and the German version (Person_DE).

Links between persons in Person_EN and Person_DE in Fig. 1 are represented by an identical pid. For some persons in Person_EN, e. g., Sante Gaiardoni, there is no link to Person_DE (and vice versa). The inclusion dependency Person_EN.pid $\subseteq$ Person_DE.pid therefore only holds for part of Person_EN. The goal of discovering Cinds is to identify conditions within Person_EN that summarize properties of those persons that have a link to Person_DE. In the given example we can observe a condition deathplace = United States $\wedge$ cent = 18, which can be explained by the large number of European emigrants in the 19th century to the US.

1.2 CIND Discovery and Condition Features

We approach the problem of Cind detection in three steps: (i) detecting an approximate Ind, i. e., an Ind that is only satisfied by part of the database, (ii) detecting conditions that can turn an approximate Ind into a Cind, i. e., conditions that hold in the part of the database that satisfies the approximate Ind, and (iii) choosing a (sub-)set of discovered conditions to build the pattern tableau of the Cind. The first step can be solved using detection methods for approximate Inds, such as [2, 18], or it could be manually performed by an expert user. The problem of finding an optimal pattern tableau has been addressed for Cfds in [13]. In this paper we assume approximate Inds to be given and focus on the second step, namely on efficiently detecting "good" conditions that turn given approximate Inds to Cinds. We outline in related work how the third step can be realized by applying our algorithms to the ideas of [13].

Features of Conditions. To achieve this goal, we need to formulate desired features of identified conditions. In the following we reason over single conditions and their features. Given an approximate inclusion dependency $R_1[X] \subseteq R_2[Y]$ between attributes X in relation R_1 and attributes Y in relation R_2. A condition over the dependent relation R_1 should distinguish tuples of R_1 included in the referenced relation R_2 from tuples not included in R_2. A condition filtering *only* included tuples is called a *valid* condition. The degree of validity can be regarded as the "precision" of a condition. Furthermore, a condition should filter *all* included tuples; its degree can be regarded as the "recall" of a condition. However, our example in Fig. 1 shows that simply relying on counting the number of tuples that match a condition may not give the desired results. In our

pid	cent	birthplace	deathplace	description
<http:...Cecil_Kellaway>	18	<http:...South_Africa>	<http:...United_States>	"Actor"@en
<http:...Mel_Sheppard>	18	<http:...United_States>	<http:...United_States>	"American athlete"@en
<http:...Buddy_Roosevelt>	18	<http:...Meeker,_Colorado>	<http:...Meeker,_Colorado>	"Actor and stunt man"@en
<http:...Sante_Gaiardoni>	19	-	-	"2 Olympic cycling golds"@en

(a) Relation `Person_EN`

pid	cent	birthplace	deathplace	description
<http:...Cecil_Kellaway>	18	<http:...Kapstadt>	<http:...Los_Angeles>	"...Schauspieler"@de
<http:...Cecil_Kellaway>	18	<http:...Kapstadt>	<http:...Kalifornien>	"...Schauspieler"@de
<http:...Cecil_Kellaway>	18	<http:...Kapstadt>	<http:...United_States>	"...Schauspieler"@de
<http:...Cecil_Kellaway>	18	<http:...Südafrika>	<http:...Los_Angeles>	"...Schauspieler"@de
<http:...Cecil_Kellaway>	18	<http:...Südafrika>	<http:...Kalifornien>	"...Schauspieler"@de
<http:...Cecil_Kellaway>	18	<http:...Südafrika>	<http:...United_States>	"...Schauspieler"@de
<http:...Mel_Sheppard>	18	<http:...Almonesson_Lake>	<http:...Queens>	"...Leichtathlet"@de
<http:...Sam_Sheppard>	19	-	-	"...Mediziner, ..."@de
<http:...Isobel_Elsom>	18	<http:...Cambridge>	<http:...Los_Angeles>	"...Schauspielerin"@de
<http:...Isobel_Elsom>	18	<http:...Cambridge>	<http:...Kalifornien>	"...Schauspielerin"@de

(b) Relation `Person_DE`

Fig. 1. Selected data of relation `Person_EN` representing persons in the English DBpedia and relation `Person_DE` representing persons in the German DBpedia. Attribute cent provides the century of birth.

example there are multiple tuples for a single person. If we want to find a condition filtering all included persons, should all tuples for this person match the condition or does one matching tuple suffice? Consider the six tuples for `Cecil Kellaway` in `Person_DE`: `Cecil Kellaway` certainly matches condition `deathplace = Los Angeles`. Counting tuples, however, lets this condition look only one-third as good, because it covers only 2 out of 6 tuples. This problem is common when discovering CINDs over relations that are derived by joining relations in a normalized database. The problem is usually aggravated as the number of relations that are joined increases. In the full version of DBpedia persons that we use for our experiments, for example, we observe 1,458 tuples for `James Beaty Jr.`. Since none of these tuples matches condition `deathplace = Los Angeles` the overall tuple count for this condition does not properly reflect the number of persons having `Los Angeles` as their place of death. To account for these discrepancies we introduce a new feature to characterize the scope of conditions: We distinguish *covering* conditions for counting objects, e. g., persons, and *completeness* conditions for counting tuples. More general, a covering condition counts *groups* of tuples whose projection on the inclusion attributes is equal. Note, that completeness conditions suffice if the inclusion attributes form a key, i. e., in this case there is only one tuple per group (or person in our running example).

Quality of Conditions. Our use case requires to find valid and covering conditions with a certain quality. We not only search for valid conditions that perfectly choose only included persons. Such CINDs are interesting in and of themselves, but we could not propose any missing links. We are also interested in "almost valid" conditions with some non-included persons matching the condition. Furthermore, it is quite unlikely to find a condition covering all included persons. We need to rate conditions by their number of covered persons. In fact, we find in our test data no conditions with perfect validity, covering, or completeness (see Sec. 5). To measure the quality of a condition, i. e., the degree of its validity, covering, or completeness, we use precision and recall measures (see Sec. 3).

1.3 Challenges of Condition Discovery

Discovering valid and covering, or valid and completeness conditions of a given quality for given approximate INDs poses two major challenges: (i) Which (and how many) attributes should be used for the conditions? (ii) Which attribute values should be chosen for the conditions? Within this paper, we propose algorithms that address both of these challenges. Given an approximate IND, our algorithms find all selecting conditions above a given quality threshold for validity and covering (or completeness) without the need to specify the set of attributes over which the condition is generated.

The contributions of this paper are as follows:

- We describe a novel *use case* for CIND detection that is motivated by the increasing amount of linked data that is published on the Web.
- We define a *generalization* of CINDs to distinguish covering and completeness conditions for discovering CINDs over denormalized relations.

– We define *quality measures* for identified conditions inspired by precision and recall that are used to restrict the discovered set of conditions to those that are most likely to represent characteristic descriptions for existing links between databases.
– We propose a SQL-based discovery approach.
– We present two *algorithms* that efficiently identify valid and covering (or valid and completeness) conditions, while choosing the condition attributes and their values automatically.
– We provide a thorough evaluation of the effectivity and efficiency of algorithms using two real-world data sets.

The remainder of this paper is structured as follows: Section 2 formally defines CINDs. In Sec. 3, we define CIND condition features and quality measures for CINDs and propose a SQL approach. Sec. 4 presents our algorithms for discovering CINDs efficiently. We evaluate our algorithms in Sec. 5. Section 6 discusses related work. We conclude in Sec. 7 and briefly discuss future work.

2 CIND Definition

Formally, a CIND is defined by an embedded approximate IND and a pattern tableau representing the conditions. The following definitions are based on [5] and [9] but we chose a different, yet in our mind more intuitive formulation. Let R_1, R_2 be relational schemata over a fixed set of attributes $A_1, A_2, \ldots, A_k$. Each attribute A has an associated domain $dom(A)$. We denote instances of R_1 and R_2 by I_1 and I_2, respectively. Each instance I is a set of tuples t such that $t[A] \in dom(A)$ for each attribute $A \in R$. Let X, X_P and Y, Y_P be lists of attributes in R_1 and R_2, respectively. We use $t[X]$ to denote the projection of t onto attributes X.

Definition 1 (Approximate IND). *An approximate* IND $R_1[X] \subseteq R_2[Y]$ *is an* IND *that is satisfied for a non-empty subset of tuples in* I_1*, i. e.,* $\exists t_1 \in I_1, t_2 \in I_2 : t_1[X] = t_2[Y]$.

A tuple $t_1 \in I_1$ satisfies $R_1[X] \subseteq R_2[Y]$ if there exists a *referenced tuple* $t_2 \in I_2$ with $t_1[X] = t_2[Y]$. We call attributes X and Y *inclusion attributes*.

Definition 2 (Pattern tableau). *A pattern tableau* T_P *restricts tuples of* R_1 *over attributes* X_P *and tuples of* R_2 *over attributes* Y_P*. For each attribute* A *in* X_P *or* Y_P *and each tuple* $t_p \in T_P$*,* $t_p[A]$ *is either a constant 'a' in* $dom(A)$ *or a special value '-'.*

Each pattern tuple $t_p \in T_P$ defines a condition. A constant value for $t_p[A]$ restricts a matching tuple's attribute value to this constant, a dash represents an arbitrary attribute value. A tuple $t_1 \in I_1$ *matches* $t_p \in T_P$ ($t_1 \asymp t_p$) if $\forall A \in X_P$: $t_p[A] = (\text{'–'} \vee t_1[A])$. The definition for a tuple $t_2 \in I_2$ matching $t_p \in T_P$ follows analogously over attributes Y_P. The pattern tableau is divided into a left-hand side (with attributes X_P) and a right-hand side (with attributes

Y_P). Both sides of the tableau can be left empty; an empty side specifies no restriction on any attribute of the respective relation. We call attributes X_P and Y_P *condition attributes*.

Definition 3 (Conditional inclusion dependency (CIND)). *A conditional inclusion dependency*

$$\varphi\colon\ (R_1[X; X_P] \subseteq R_2[Y; Y_P], T_P)$$

consists of the embedded approximate IND $R_1[X] \subseteq R_2[Y]$ *and the pattern tableau* T_P *over attributes* X_P *and* Y_P *defining the restrictions. Sets* X *and* X_P *are disjoint, and* Y *and* Y_P *are disjoint.*

Our example CIND is denoted as follows:

$$\varphi\colon\ (\texttt{Person\_EN[pid; cent, deathplace]} \subseteq \texttt{Person\_DE[pid;]}, T_P)$$

$$T_P\colon \begin{array}{ll} \text{cent} & \text{deathplace} \\ \hline 18 & \text{<http:...United\_States>} \end{array}$$

A CIND φ holds for a pair of instances I_1 and I_2 if

1. *Selecting condition on* I_1*:* Let $t_1 \in I_1$ match any tuple $t_p \in T_P$. Then t_1 must satisfy the embedded IND.
2. *Demanding condition on* I_2*:* Let $t_1 \in I_1$ match tuple $t_p \in T_P$. Further, let t_1 satisfy the embedded IND with referenced tuple $t_2 \in I_2$, i. e., $t_1[X] = t_2[Y]$. Then t_2 also must match t_p.

Note that the CIND definition treats selecting conditions, i. e., the left-hand side of the pattern tableau, and demanding conditions, i. e., the right-hand side of the pattern tableau, separately and asymmetrically: Selecting conditions are required to be valid, i. e., to select only included tuples. Further, they should be complete or covering to be able to build concise pattern tableaux. In contrast, demanding conditions are required to be complete, i. e., all referenced tuples are required to match the condition. There is no equivalent notion for validity. In the following, we focus on selecting conditions, because their requirements subsume the demanding condition's requirements.

3 Condition Features

In this section we formally define the features *valid*, *completeness*, and *covering*, and define their degree through the precision and recall of a condition. These features are used to quantify the quality of individual conditions (i. e., pattern tuples) in a pattern tableau. All three features refer to selecting conditions only.

Given a conditional inclusion dependency φ and instances I_1 and I_2. Let I_φ denote the set of tuples from I_1 that satisfy the embedded IND, i. e., $I_\varphi = I_1 \ltimes_{X=Y} I_2$. We refer to I_φ as the set of *included tuples*. For denormalized relations like those in our motivating example we are also interested in groups of included

tuples that have equal values in attributes X, e. g., all tuples for `Cecil Kellaway`. Let g_x denote a *group* of tuples in I_1 having value x for $t[X]$, i. e., $g_x = \{t | t \in I_1 \wedge t[X] = x\}$. We call g_x an *included group* if all tuples are included tuples, i. e., $g_x \subseteq I_\varphi$. A group g_x matches a pattern tuple t_p, denoted by $g_x \asymp t_p$, if any of the tuples in g_x matches t_p, i. e., $g_x \asymp t_p \Leftrightarrow \exists t \in g_x : t \asymp t_p$. Let G_1 denote the set of groups in I_1 and G_φ denote the set of included groups. Finally, for a pattern tuple t_p let $I_1[t_p]$ and $I_\varphi[t_p]$ denote the set of tuples from I_1 and I_φ that match t_p, respectively. $G_1[t_p]$ and $G_\varphi[t_p]$ denote the groups in G_1 and G_φ that match t_p, respectively.

Definition 4 (Valid Condition). *A condition is* valid *if all tuples of I_1 that match t_p also satisfy the embedded IND, i. e., $I_1[t_p] \subseteq I_\varphi$.*

Definition 5 (Complete Condition). *A condition is* complete *if it matches all included tuples, i. e., $I_\varphi \subseteq I_1[t_p]$.*

Definition 6 (Covering Condition). *A condition is* covering *if it matches all included groups, i. e., $G_\varphi \subseteq G_1[t_p]$.*

3.1 Measuring Condition Features

One will rarely find single conditions that are valid, complete, or covering. Our use case, furthermore, actually requires to find conditions that are not perfectly valid. In the following, we define measures for validity, completeness, and covering that are used to constrain the conditions that are found by our condition discovery algorithms.

Valid Conditions. The validity of a condition can be measured by the precision of this condition, i. e., the number of matching *and* included tuples related to the number of all matching tuples:

$$valid(t_p) := \frac{|I_\varphi[t_p]|}{|I_1[t_p]|}$$

A validity of one means that all tuples that match t_p are included tuples, i. e., t_p is *valid*. If $valid(t_p) > \gamma$ we call t_p *γ-valid*.

Although the definition for valid conditions over tuples also works in the presence of groups, it is useful to redefine the degree of the feature for groups: Consider a condition `cent = 18` for persons in `Person_DE` to be included in `Person_EN` (see Fig. 1). Counted over tuples, this condition would be 0.78-valid, but over groups (or persons) it is just 0.67-valid. That is, the six included and matching tuples for `Cecil Kellaway` made this condition look "more valid" than it is. The other way around, several matching tuples for a non-included group would make it look "less valid" than it really is. So we apply the idea of using precision as measure for validity to groups, i. e., we relate the number of matching and included groups to the number of all matching groups:

$$valid_g(t_p) := \frac{|G_\varphi[t_p]|}{|G_1[t_p]|}$$

We call a condition γ-valid$_g$ if $valid_g(t_p) > \gamma$. Note that validity can also be interpreted as the confidence of the rule "If a condition matches a tuple, then this tuple is an included tuple." The resulting ratio equals our quality measure.
Completeness Conditions. The completeness of a condition can be measured as recall of this condition counting the relation's tuples, i. e., the number of matching *and* included tuples related to the number of all included tuples:

$$complete(t_p) := \frac{|I_\varphi[t_p]|}{|I_\varphi|}$$

A completeness of one means that t_p matches all included tuples, i. e., t_p is complete. If $complete(t_p) > \delta$ we call t_p *δ-complete*. Completeness is also a measure for quality (i. e., confidence) for the rule "If a tuple is an included tuple, then the condition matches this tuple".
Covering Conditions. The quality of covering conditions can be measured by the recall of these conditions based on the relation's groups, i. e., the number of matching *and* included groups related to the number of all included groups:

$$covering(t_p) := \frac{|G_\varphi[t_p]|}{|G_\varphi|}$$

A covering of one means that t_p matches all included groups, i. e., t_p is covering. If $covering(t_p) > \lambda$ we call t_p *λ-covering*. Covering is a measure for quality (i. e., confidence) for the rule "If a group is an included group, then the condition matches at least one tuple in this group".

3.2 Detecting Conditions Using SQL

For a given approximate IND and a set of condition attributes, we can use SQL to detect conditions. The general idea is threefold: (i) Compute a left outer join over the dependent and referenced relation, (ii) use the referenced attributes as indicator for included or non-included tuples (or groups), (iii) group the result by the preselected condition attributes to examine each value combination as condition.

Recall our example on finding persons in the German DBpedia to be included in the English DBpedia. Consider the left outer join over (`Person_DE.pid`, `Person_EN.pid`) grouped by the condition attributes `deathplace` and `cent` of relation `Person_DE` (see Fig. 2). `Person_DE.pid` lists all persons in the German DBpedia and `Person_EN.pid` indicates if a person is included (i. e., a non-NULL value) or non-included (i. e., a NULL value). Counting values in `Person_DE.pid` gives the number of matching tuples, i. e., $|I_1[t_p]|$; counting values in `Person_EN.pid` gives the number of matching and included tuples, i. e., $|I_\varphi[t_p]|$. Using this observation we can compute the validity and completeness of a condition.

Fig. 3a shows the SQL statement to find γ-valid and δ-complete conditions and their quality measures. Note that the statement returns the absolute number of matching and included tuples. To compute completeness we have to divide this number by the total number of included tuples. In our example condition

lhs inclusion attribute	condition attributes **Person_DE.**		rhs inclusion attribute
Person_DE.personID	**cent**	**deathplace**	**Person_EN.personID**
Cecil_Kellaway	18	Los_Angeles	Cecil_Kellaway
Cecil_Kellaway	18	Los_Angeles	Cecil_Kellaway
Isobel_Elsom	18	Los_Angeles	NULL
Cecil_Kellaway	18	Kalifornien	Cecil_Kellaway
Cecil_Kellaway	18	Kalifornien	Cecil_Kellaway
Isobel_Elsom	18	Kalifornien	NULL
Cecil_Kellaway	18	United_States	Cecil_Kellaway
Cecil_Kellaway	18	United_States	Cecil_Kellaway
Mel_Sheppard	18	Queens	Mel_Sheppard
Sam_Sheppard	19	-	NULL

Fig. 2. Left outer join over relations `Person_DE` and `Person_EN` as given in Fig. 1, projected on inclusion and condition attributes and grouped by condition attributes. URL-specific parts of values are omitted for readability.

`Person_DE.cent = 18` and `Person_DE.deathplace = Los Angeles` is computed as 2/3-valid with an absolute value for completeness of 2 (out of 7 included tuples).

Figure 3b shows the modified statement to find γ-valid$_g$ and λ-covering conditions by counting the number of distinct values for `Person_DE.pid` and `Person_EN.pid` instead. The results are the number of matching groups ($|G_1[t_p]|$), and the number of matching and included groups ($|G_\varphi[t_p]|$). Both values can again be used to compute the quality measures, but now for valid$_g$ and covering conditions. Our example condition `Person_DE.cent = 18` $\wedge$ `Person_DE.deathplace = Los Angeles` achieves more interesting measures as it is computed to be 1/2-valid$_g$ with an absolute value for covering of 1 (out of 2 included persons).

```
SELECT de.cent, de.deathplace,
    cast (count(en.pid) as double)
        / count(de.pid) as valid,
    count(en.pid) as complete_abs
FROM Person_DE de left outer
      join Person_EN en
    on de.pid = en.pid
GROUP BY de.cent, de.deathplace
```

(a) γ-valid and δ-complete conditions

```
SELECT de.cent, de.deathplace,
    cast (count(distinct en.pid) as
          double)
        / count(distinct de.pid) as
              valid_g,
    count(distinct en.pid) as
          covering_abs
FROM Person_DE de left outer
      join Person_EN en
    on de.pid = en.pid
GROUP BY de.cent, de.deathplace
```

(b) γ-valid$_g$ and λ-covering conditions

Fig. 3. SQL statements to find conditions and their quality measures for embedded IND `Person_DE` $\subseteq$ `Person_EN` over preselected attributes `cent` and `deathplace`.

In summary, it is possible to detect valid and completeness, or valid$_g$ and covering conditions and their quality measures using SQL – for preselected condition

attributes. In our example there are 12 potential condition attributes leading to $2^{12} - 1$ combinations to test. Execution times for the given statements were 1.2 s for valid and completeness conditions, and in 5.9 s for valid$_g$ and covering conditions. Assuming this as the average runtime, the estimated runtime for all combinations is about 80 min and 6 h40 min, respectively. In the next section we present more efficient approaches to detect conditions without the need to preselect condition attributes.

4 Discovering General Conditions

In this section we describe algorithms to detect all γ-valid$_g$ and λ-covering conditions, as well as γ-valid and δ-complete conditions without restricting the attributes that should be used. We describe two different approaches – "Conditional INclusion DEpendency REcognition Leveraging deLimited Apriori" (CINDERELLA) uses an Apriori algorithm and is faster, while "Position List Intersection" (PLI) leverages value position lists and consumes less memory. We compare the complexity of both algorithms in Sec. 4.3. We first describe our algorithms to detect valid$_g$ and covering conditions, and modify them afterwards to detect valid and completeness conditions.

Both algorithms reuse the idea of a left outer join over the dependent and referenced relation with the referenced attributes as flag for included or non-included tuples (or groups) (Sec. 3.2). Our algorithms do not rely on the relational representation of the data. Instead, we choose a representation for the join result that allows handling multiple uses of one attribute or predicate for a single group. Each group is represented by three items: (i) the left-hand side inclusion attribute, i. e., the person identifier, (ii) a right-hand side inclusion indicator with values INCLUDED for included groups or NULL for non-included groups, and (iii) a list of (attribute : value)-pairs for potential condition attributes, i. e., all attributes of the dependent relation apart from the inclusion attributes. Figure 4 shows this representation for the embedded IND `Person_DE.pid` $\subseteq$ `Person_EN.pid`.

4.1 Using Association Rule Mining

Association rule mining was introduced for market basket analysis to find rules of type "Who buys X and Y often also buys Z". We apply this concept to identify conditions like "Whose century of birth is 18 and place of death is 'United States' often also is INCLUDED (in the English DBpedia)".

To leverage association rule mining we need to prepare our baskets in two steps: We use the modified representation of the left outer join result as shown in Fig. 4. Note that we only need the right-hand side inclusion indicator and the potential condition attributes to build the baskets, because we do not want to find conditions over the dependent inclusion attributes. Second, we must encode the affiliation of values to their attributes to form basket items. For our example, we want to be able to distinguish the two conditions `birthplace = Los Angeles` and `deathplace = Los Angeles`. Therefore, we prefix each value with an

	lhs inclusion attribute	rhs inclusion indicator	potential condition attributes and values
1	de.pid:Cecil_Kellaway	`en.pid:`INCLUDED	`cent:`18, `birthplace:`Kapstadt, `birthplace:`Südafrika, `deathplace:`Los_Angeles, `deathplace:`Kalifornien, `deathplace:`United_States, `description:`"...Schauspieler"@de
2	de.pid:Mel_Sheppard	`en.pid:`INCLUDED	`cent:`18, `birthplace:`Almonesson_Lake, `deathplace:`Queens, `description:`"...Leichtathlet"@de
3	de.pid:Sam_Sheppard	`en.pid:`NULL	`cent:`19, `description:`"...Mediziner, ..."@de
4	de.pid:Isobel_Elsom	`en.pid:`NULL	`cent:`18, `birthplace:`Cambridge, `deathplace:`Los_Angeles, `deathplace:`Kalifornien, `description:`"...Schauspielerin"@de

Fig. 4. Left outer join over `Person_DE` and `Person_EN` in Fig. 1; Modified representation handles multiple occurrences of one attribute for a single person. URL-specific parts of values are omitted for readability.

attribute identifier. Using prefixes A to D for our example yields the following basket for the first group of Fig. 4: { INCLUDED, A18, BKapstadt, BSüdafrika, CLos_Angeles, CKalifornien, CUnited_States, D"...Schauspieler"@de }. Now we are able to apply an Apriori algorithm to these baskets to find frequent itemsets and derive rules.

The Apriori algorithm [1] consists of two steps: (i) Find all frequent itemsets that occur in at least a given number of baskets, and (ii) use these frequent itemsets to derive association rules. Apriori uses support and confidence of a rule to prune the search space. In our case the covering of a condition is a measure for the support of a condition in the set of included groups, and the validity of a rule corresponds to the confidence of the rule. Thus, we apply those measures for pruning in the Apriori algorithm. A frequent itemset then ensures λ-covering conditions, while the rule generation step filters γ-valid$_g$ conditions.

We could use the original Apriori algorithm to find conditions, but we would waste optimization possibilities based on the following observation: We need only a special case of association rules to identify conditions: We need only rules with right-hand side item INCLUDED, because left-hand side items of such rules build the selecting condition. Thus, we only need to find frequent itemsets containing item INCLUDED, i. e., we can largely reduce the number of itemsets that must be handled and therefore improve the efficiency of the algorithm. We describe our algorithm Conditional INclusion DEpendency REcognition Leveraging de-Limited Apriori (CINDERELLA) in the next section.

Cinderella Algorithm The CINDERELLA algorithm reduces the number of generated frequent itemsets by only considering itemsets that contain item INCLUDED. Algorithm MultipleJoins is a Apriori variation that finds rules containing (or not containing) specified items [20]. It proposes three joins for candidate generation depending on the position of the specified item in the basket. In our case we can simplify this approach. We reduce it to only one join, due to our strict constraint of exactly one fixed item (INCLUDED).

Algorithm 1 shows the detection of frequent itemsets with item INCLUDED. It assumes (as Apriori) that all items in a basket are sorted by a given order. Furthermore, it assumes that item INCLUDED is the first element in this order, i. e., INCLUDED is always the first item in any sorted basket or itemset. We therefore can reduce the three joins of the algorithm MultipleJoins in our case to only one join.

Let L_k denote the set of frequent itemsets of size k. The first set L_1 is retrieved by a single scan over all *included* baskets; L_2 is built by combining each frequent 1-itemset with item INCLUDED. All further sets L_k are built level-wise by combining sets of L_{k-1} using method `aprioriGen-Constrained` (see Alg. 2) to itemset candidates C_k and testing them afterwards. The algorithm stops if L_{k-1} is empty.

Algorithm 1: Apriori-Constrained: Find all frequent (i. e., λ-covering) itemsets with item INCLUDED.

input : Included tuples as baskets: baskets
output: frequent itemsets with item INCLUDED
`/* single scan over baskets to get` L_1 `*/`
$L_1 = \{\text{frequent 1-itemsets}\}$;
$L_2 = \{(\text{INCLUDED}, l_1) \mid l_1 \in L_1\}$;
for k=3; $L_{k-1} \neq \emptyset$; k++ **do**
  C_k =aprioriGen-Constrained(L_{k-1}) ;
  foreach *basket* $b \in$ *baskets* **do**
    $C_t = \text{subset}(C_k, b)$;
    foreach $c \in C_t$ **do**
      c.count++;
    $L_k = \{c \in C_k \mid c.\text{count} \geq \lambda * |\text{baskets}|\}$
return $\cup_k L_k \bigcup L_2$

Method `aprioriGen-Constrained` (Alg. 2) combines in the first step two itemsets of size $k-1$ to an itemset candidate if both itemsets are equal in the first $k-2$ items. In the second step it prunes such candidates with at least one subset of size $k-1$ that contains INCLUDED but that is not contained in L_{k-1}. Creating the candidates by a self-join of L_{k-1} is exactly the same as in Apriori. This procedure works for our constrained case, because we require INCLUDED to be smaller than any other item. Thus, each created candidate will contain INCLUDED. The difference to the original aprioriGen is that only such subsets are

considered for pruning that contain item INCLUDED, because only these itemsets can be contained in L_{k-1}.

After creating the candidate itemsets of size k, the number of occurrences in the baskets of each candidate is counted. We can apply method `subset` as described for Apriori: All candidates C_k are represented in a HashTree to find the subset of candidates C_t contained in a basket very fast. Then, all frequent (i. e., λ-covering) candidates build set L_k.

Algorithm 2: aprioriGen-Constrained

input : frequent itemsets of size $k-1$: L_{k-1}
output: candidates for frequent itemsets of size k: C_k

```
1 insert into C_k
2 select p.item_1, p.item_2, ..., p.item_{k-1}, q.item_{k-1}
3 from L_{k-1} p, L_{k-1} q
4 where p.item_1 = q.item_1 ∧ ... ∧ p.item_{k-2} = q.item_{k-2} ∧ p.item_{k-1} <
  q.item_{k-1},
5 foreach candidate c ∈ C_k do
6     foreach (k − 1)-subsets s of c containing item INCLUDED do
7         if s ∉ L_{k-1} then
8             delete c from C_k
9 return C_k
```

The rule generation step uses the identified frequent itemsets and computes the validity of conditions: The number of included groups matching the condition is the number of occurrences (support) of a frequent itemset; the number of all groups matching a condition is the support of the frequent itemset without item INCLUDED. This number of occurrences must be counted in an extra scan over all baskets, because we do not have this information up to this step. Again, all itemsets can be represented in a hash tree to count their occurrences fast. Using both values for each frequent itemset we can filter γ-valid$_g$ conditions.

Detecting Completeness Conditions with Cinderella We can apply the CINDERELLA algorithm to also detect γ-valid and δ-complete conditions by a single modification: We only need to build our baskets differently. So far, we built one basket per group to detect λ-covering conditions. Now, we build several baskets per group, i. e., we build one basket per tuple in the relational representation. In our running example we now have six baskets for Cecil Kellaway. Using this slight modification we can apply the CINDERELLA algorithm as described. Having only one basket per tuple, we now count tuples instead of groups and therefore detect γ-valid and δ-complete conditions.

4.2 Using Position List Intersection

The Position-List-Intersection (PLI) approach searches for conditions in a depth-first manner and uses an ID list representation for each value of an attribute. Using the *position list* representation for conditions the algorithm is able to prune irrelevant candidate conditions and is more memory efficient than CINDERELLA because of its depth-first approach. The position list representation of values has also been applied by the algorithm TANE for discovering functional dependencies [16]. While our approach looks for intersections of lists, the partition refinement of TANE is based on the discovery of subset relationships of position lists. In the following, we first introduce the concept of position lists and intersections and then describe the PLI algorithm.

Position Lists and Intersections The PLI algorithm is based on the idea that every distinct value in an attribute can be represented by the set of row numbers (or tuple IDs [16]) where the value occurs in the table. Those sets are referred to as position lists (or inverted lists). Thus, each attribute is associated with a set of position lists – one for each of its distinct values. In our case positions can be both tuple IDs when looking for completeness conditions and group-IDs (e.g., numbers 1-4 in Fig. 4) when looking for covering conditions. In the following, we refer only to group-IDs as we describe the algorithm for the discovery of λ-covering and γ-valid$_g$ conditions.

Table 1 illustrates the position lists for the attributes `cent` and `deathplace` from the example in Fig. 4. The frequency of each value is given by the cardinality of its position list. Values having a position list with fewer members than required by the covering threshold can be ignored for further analysis and are omitted. We use a special position list, called *includedPositions*, for all included groups. Intersecting the position list of a value with *includedPositions* returns the included subset of the groups that match the value. The list *includedPositions* is the position list for `en.pid`'s value INCLUDED in Table 1.

The position lists of an attribute combination can be calculated by the *cross-intersection* of the position lists of its contained attributes. Cross-intersection means each position list of one attribute is intersected with each position list of the other attribute. For example, detection of conditions from the attribute combination `cent, deathplace` requires to intersect each position list of attribute `cent` with the position lists of each value in attribute `deathplace`: The intersection of the position list of `cent:`18 with the position list of `deathplace:`Los_Angeles, for example, results in position list $\{1\}$. Intersecting position list `cent:`18 with position list `deathplace:`Kalifornien results in $\{1, 2\}$.

Workflow of the PLI Algorithm While the CINDERELLA algorithm traverses the powerset lattice of condition combinations *breadth-first* or level-wise, by checking all combinations of a certain size in the same pass, the recursive PLI algorithm processes the powerset lattice *depth-first* by checking all possible combinations that contain a certain condition. The algorithm is based on two

attribute	value	position list
cent	18	$\{1, 2, 4\}$
	19	$\{3\}$
deathplace	Los_Angeles	$\{1\}$
	Kalifornien	$\{1, 2\}$
	United_States	$\{1\}$
	Queens	$\{2\}$
en.pid	INCLUDED	$\{1, 2\}$
	NULL	$\{3, 4\}$

Table 1. Position List Example

phases: First, it retrieves the position lists for each single attribute and the set of included group-IDs. Second, it retrieves all combinations of conditions across multiple attributes by cross intersection of the position lists of the disjoint attribute combinations.

Position list retrieval. The algorithm needs to scan the table once for retrieving position lists of each potential condition attribute. In addition, position list *includedPositions* is retrieved that contains all group-IDs of included groups. This step is straightforward by iterating through all groups using several hashmaps per attribute that map each value to a list of group-IDs. At the same time position list *includedPositions* is maintained by adding group-IDs of each included group. In our running example, list retrieval includes *includedPositions* and position lists for the attributes `deathplace`, `birthplace`, `description`, and `cent`.

Multi-attribute analysis. After retrieving the position lists of potential inclusion attributes, the next step is to discover all γ-valid$_g$ and λ-covering conditions. In this step, each value of the current attribute is extended via cross intersection with values of other attributes. As far as the result of the cross intersection contains λ-covering position lists the algorithm continues to cross intersect the result with the position lists of the next attribute in line. Whenever a cross intersection results in an empty set, the recursive algorithm backtracks one step in its recursion and substitutes the last added attribute with the next alternative attribute. The beginning of the recursion is always a single attribute that contains λ-covering conditions.

The PLI algorithm (Algorithm 3) iterates over all potential condition attributes ensuring that all attribute combinations are considered. In each iteration function *analyze* is called. Using an arbitrary but fixed numerical order over the attributes, the *analyze* function traverses all combinations that involve the current attribute and attributes of higher order. The numerical order prevents that the same combination is analyzed repeatedly. For convenience, we use numerical identifiers $\{1, 2, \dots\}$ for attributes. Function *analyze* returns all γ-valid$_g$ and λ-covering conditions of these attribute combinations.

Function *analyze* is shown in Algorithm 4. The method traverses the powerset lattice of the condition attributes depth-first. Its parameters are the currently added attribute (*attrNo*), the position lists of the attribute (*attrNoPLs*), the

Algorithm 3: PLI: pli()

```
   input  : potential condition attributes
   output: all γ-valid and λ-covering conditions
   /* Start recursive call for each attribute                    */
1  for i = 1 to |attributes| do
2  |  analyze (i, getPLs (i), ∅, ∅) ;
3  return conditions
```

current combination of attributes (*currComb*) (without *attrNo*), and the position lists of *currComb* (*combPLs*).

The method *analyze* is initially called with the numerical identifier of the first attribute *attrNo*, its position lists *attrNoPLs* and $\emptyset$ for the current attribute combination *currComb* and its position lists, respectively. At first the method builds a new combination *newAttrComb* by adding the given *attrNo* to the current set of attributes *currentComb* and creates the new set of position lists *newCombPLs* by cross-intersecting the position lists of the given *attrNo* and the current combination *currComb*.

If *currComb* is $\emptyset$, the new combination contains just the position lists *attrNoPLs* of the current single attribute *attrNo* (line 2). Next, each position list in *newCombPLs* is checked for λ-covering and γ-validity (lines 7-12). For this purpose each of the position lists *PL* is intersected with the position list of the inclusion attribute *includedPositions*. If the corresponding condition of a position list is λ-covering it is added to the new list of position lists *coveringCombPLs* that can be extended by further attributes in the next recursion step. If the condition is also γ-valid$_g$, it is added to the result set *conditions*.

As far as *coveringCombPLs* contains any position list the algorithm can continue to select the next attribute that can be added to the new set of attributes *newColComb*. So the method *analyze* is called for the next attribute in line ($i >$ *attrNo*), the new attribute combination *newColComb* and their position lists respectively. The method terminates as soon as no further attribute can be added to the current combination either because there are no λ-covering position lists or the current attribute was the last attribute. In both cases, the set of generated conditions is returned.

Detecting Completeness Conditions with PLI For the discovery of λ-covering conditions we considered group-IDs as positions. For the discovery of δ-complete conditions on key inclusion attributes the algorithm can be adapted by using tuple-IDs of the relational representation instead of group-IDs as positions.

4.3 Cinderella vs. PLI

We now compare our two algorithms in terms of complexity. The search space of both algorithms is in worst case exponential in the number of attributes: Given

Algorithm 4: Analyze Attribute Combinations: analyze

```
   input  : attrNo, attrNoPLs, currComb, combPLs
   output: all γ-valid_g and λ-covering conditions for given attribute combination
           and extensions with attribute number larger than attrNo

   /* Build new attribute combination.                                    */
 1 newAttrCombs ← currComb ∪ attrNo ;
 2 if currComb ≠ ∅ then
 3     newCombPLs ← crossIntersect (
 4         attrNoPLs, combPLs);
 5 else
 6     newCombPLs ←attrNoPLs;
   /* Check each position list.                                           */
 7 foreach pl ∈ newCombPLs do
 8     plIncluded ← (pl ∩ includedPositions) ;
       /* Compute measures of covering, valid_g.                          */
 9     if |plIncluded| / |includedPositions| > λ then
10         coveringCombPLs.add (pl) ;
11         if |plIncluded| / |pl| > γ then
12             conditions.add(
13                 newCombAttrNos, pl.values (),
14                     covering, valid) ;
15 if ¬ coveringCombPLs.isEmpty () then
16     for i = attrNo + 1 to |attributes| do
17         conditions.addAll (analyze (i, getPLs (i),
18             newColCombs, coveringCombPLs ))) ;
19 return conditions
```

n attributes and the average number of values per attribute v, both algorithms have to check $O(2^n \cdot v^n)$ potential conditions.

Using the apriori paradigm of pruning CINDERELLA is able to reduce the search space drastically, depending on the δ and λ thresholds for respectively completeness and covering conditions. The PLI algorithm works depth-first and is not able to apply this pruning strategy. Therefore the search space of the PLI algorithm is always at least as large as for the CINDERELLA algorithm. Both algorithms scan the database only once and therefore require the same disk IO.

With regard to memory usage the PLI algorithm outperforms CINDERELLA, since it needs only the position lists for each single attribute and the position lists that have been created during one complete recursion branch. The upper bound for the total number of position lists in memory is at most $O(2 \cdot n \cdot v)$ resulting from $n \cdot v$ position lists for the single attributes and additional $n \cdot v$ position lists that might be created in one recursion branch. The breadth-first search strategy of the CINDERELLA algorithm, however, requires to store all $\binom{n}{l} \cdot v^l$ generated candidates of each level l in memory. In the worst case, the number of

candidates corresponds to $\binom{n}{\frac{n}{2}} \cdot v^{\frac{n}{2}}$ for itemsets of size $\frac{n}{2}$. Our implementation of Cinderella holds all baskets in memory, such that we have a resulting memory complexity of $O(\binom{n}{\frac{n}{2}} \cdot v^{\frac{n}{2}} \cdot \frac{n}{2} + v \cdot n)$. Although a position list requires (in most cases) more memory than an itemset, PLI still should outperform Cinderella with regard to memory usage, because of its quadratic complexity compared to Cinderella's exponential complexity. In the next section, we confirm our complexity analysis using actual experimental results.

4.4 Detecting Demanding Conditions

Until now we have solely focused on discovering selecting conditions. Our algorithms could directly be used to discover demanding conditions by indicating referenced tuples instead of included tuples. The definitions for completeness and covering could easily be adapted for right-hand side conditions. There is, however, no equivalent notion for validity, i. e., matching unreferenced tuples is not considered a violation for a demanding conditions. Thus, we can omit the extra check for validity in our algorithms.

5 Evaluation

We evaluate our methods with two real-life data sets: Persons in the English and German DBpedia 3.6 (see introduction) and using Wikipedia image data from [14]. Section 5.1 shows results on the former case, namely on discovering conditions for persons in the German DBpedia to be included in the English DBpedia, and vice versa. In Section 5.2 we evaluate and compare the efficiency our algorithms experimentally on the DBpedia data sets. Results for applying our algorithms on the Wikipedia data set are shown in Section 5.3.

In DBpedia, individual persons are represented by the same URI in both data sets. There are 296, 454 persons in the English DBpedia and 175, 457 persons in the German DBpedia; 74, 496 persons are included in both data sets. We mapped the data sets into relations to enable detection of covering and completeness conditions. We use one attribute per predicate, namely `pid`, `name`, `givenname`, `surname`, `birthdate`, `birthplace`, `deathdate`, `deathplace`, and `description`. Further, we extracted the century and the year of birth and death into additional attributes from attributes `birthdate` and `deathdate`, respectively. The resulting relations contain 474, 630 tuples for the English DBpedia, and 280, 913 tuples for the German DBpedia with an intersection of 133, 208 tuples.

5.1 Identified Conditions

In this section we point out discovered conditions to show the value of applying the concept of Cinds to our use case of detecting missing links. Figure 5 shows a scatter plot over all covering conditions with an absolute threshold of at least one person for persons in the German DBpedia to be included in the English DBpedia, i. e., 566, 830 conditions. We can see that the conditions spread over

the entire range of γ for validity. The majority of conditions has a λ-covering of less than 0.01, which corresponds to 744 persons.

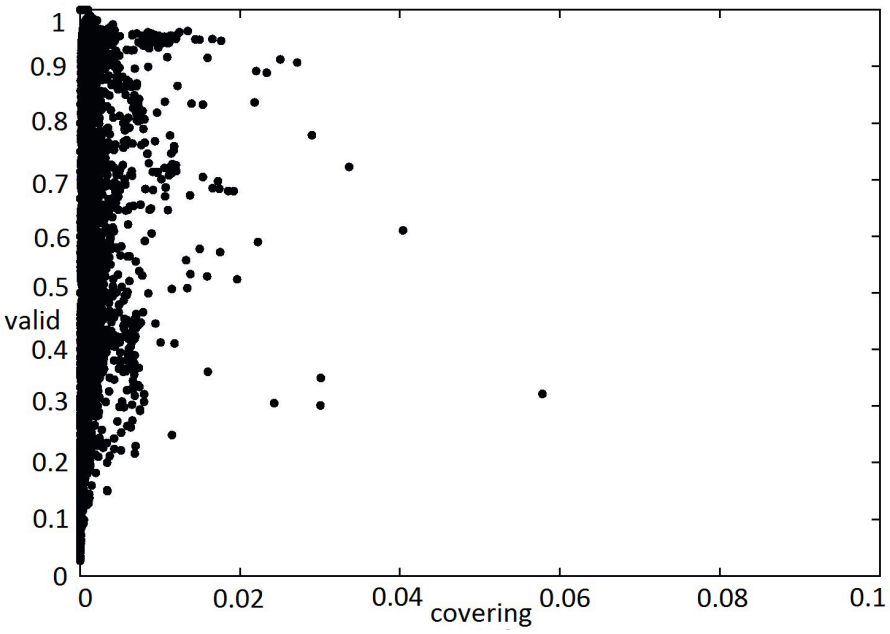


Fig. 5. Identified conditions.

We decided to set the validity threshold depending on the instances. We use the validity of an empty condition as reference value: As validity is computed as the fraction of matching and included persons to all matching persons, the empty condition's validity is the fraction of included persons to all persons in the dependent data set. We use twice this validity value as threshold for γ-valid conditions.

German DBpedia persons included in the English DBpedia. The validity of the empty condition is 0.42, i. e., 42% of persons in the German DBpedia are also included in the English DBpedia. We used a covering threshold of 0.008 (i. e., 600 persons), which leads to a useful amount of conditions. We identify 85 conditions with a γ-validity of above 0.84, including 16 conditions with $\gamma > 0.95$.

The two conditions with the largest covering measure are `description = American actor`[1] (γ-valid$_g$ = 0.91, λ-covering = 0.029, i. e., 2173 persons) or `description = American actress`(γ-valid$_g$ = 0.89, λ-covering = 0.024, i. e., 1791 persons). We also found both conditions in conjunction with the condition `birth-`

[1] Note that we provide translated condition values as the actual value is in German.

`century = 19` with slightly increased validity and slightly decreased covering measures.

The above conditions are intuitive and hardly surprising. But we also found the following unforeseen conditions while similar probable conditions were not found: We identified the conditions

- `birthcentury = 18` $\wedge$ `description = American politician`
  (γ-valid$_g = 0.94$, λ-covering $= 0.015$)
- `birthcentury = 19` $\wedge$ `deathplace = Los Angeles`
  (γ-valid$_g = 0.91$, λ-covering$= 0.010$)
- `birthcentury = 19` $\wedge$ `deathplace = New York City`
  (γ-valid$_g = 0.86$, λ-covering $= 0.012$)
- `birthcentury = 19` $\wedge$ `deathplace = California`
  (γ-valid$_g = 0.91$, λ-covering $= 0.015$)

Interestingly, we found a class of conditions using only the year of birth, e. g., `birthyear = 1900`, for the years 1900 to 1928 and 1945 to 1947. Each of these conditions has a γ-validity of above 93 % and a λ-covering between 0.8 % to 1 %, i. e., 595 to 744 persons. Combining all these conditions using a disjunction results in a overall validity of 93 % and a covering of 28.8 % (or 21,454 persons). The reason for these conditions can be seen in Fig. 6: The English DBpedia contains overall more persons born in 1900 to 1928 and 1945 to 1947 compared to other years, while there is no special behavior for these years in the German DBpedia. Thus, persons born in these years are more likely to be included than others. If we had known this data skew in advance, we could have guessed these conditions. But detecting conditions led us to detect this data skew instead of imagining and checking all possible, guessable variations in the data.

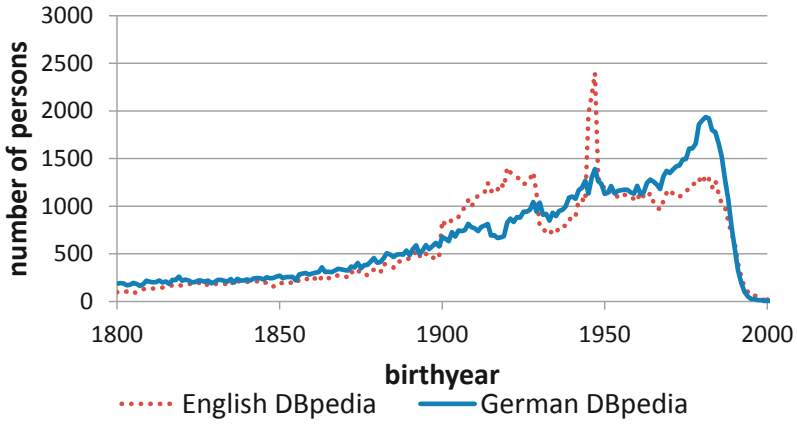


Fig. 6. Persons per year of birth.

English DBpedia persons included in the German DBpedia. The validity of the empty condition is 0.25, i. e., 25% of persons in the German DBpedia are also included in the English DBpedia. We choose a covering threshold of 0.007. We identified 14 conditions with a γ-validity of above 0.5 with the maximum γ-validity of 0.66.

We find obvious conditions, such as `birthplace = Germany` (γ-valid$_g$ = 0.59, λ-covering = 0.024) or `deathplace = Germany` (γ-valid$_g$ = 0.55, λ-covering = 0.011), or again `description = actor` $\wedge$ `birthcentury = 19` or `description = actress` (both with γ-valid$_g$ = 0.60 and λ-covering = 0.01)).

But we also find the surprising conditions `deathplace = United States` $\wedge$ `birthcentury = 18` (that we already introduced in Sec. 1) with γ-valid$_g$ = 0.51 and λ-covering = 0.008, or a `description = road bicycle racer` (γ-valid$_g$ = 0.66, λ-covering = 0.012).

5.2 Performance of Algorithms

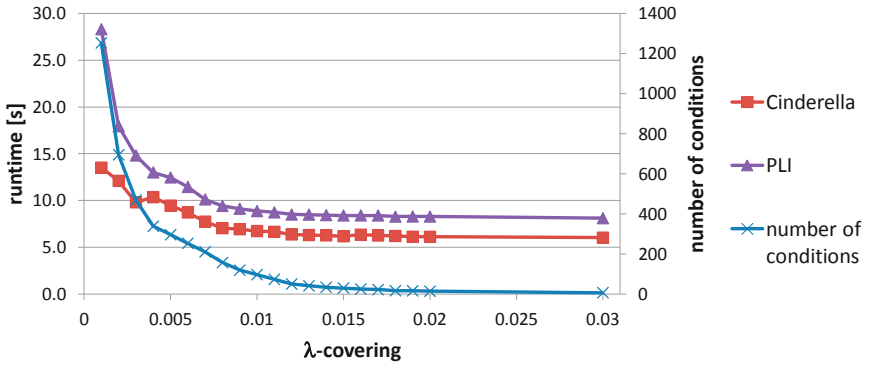
We set up two experiments on the DBpedia data sets to evaluate (i) the effect of the number of identified conditions, and (ii) the effect of the number of tuples in the dependent data set. We consider runtime and memory consumption in both experiments.

We implemented our algorithms Cinderella and PLI in Java6, and store the data sets in a commercial DBMS. We run our experiments on a 2x Xeon quad-core server with 16 GB RAM running a 64bit Linux.

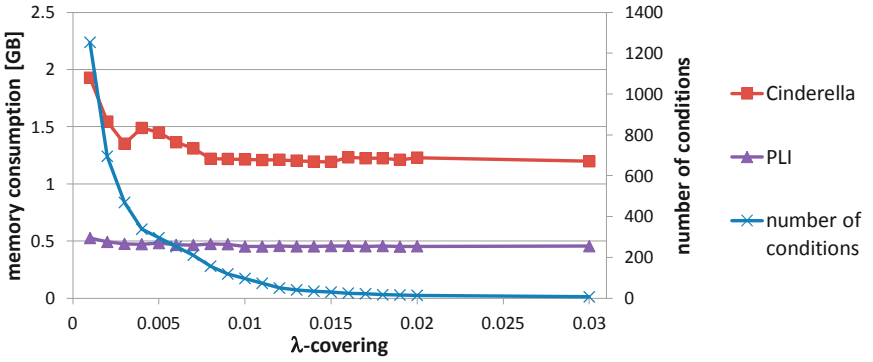
Varying Number of Conditions. In this experiment we want to test the effect of the number of identified conditions. We use the German DBpedia person data set and its included persons in the English DBpedia. We vary the number of identified conditions by varying λ or δ for detecting covering or completeness conditions, respectively.

Figures 7(a) and 7(b) show the runtime and memory consumption for varying λ, i. e., for detecting covering conditions. In both diagrams we also show the number of identified conditions using a secondary y-axis on the right. The runtime of both algorithms correlates with the number of identified conditions, but Cinderella is less sensitive to increasing numbers of identified conditions. Generally, Cinderella is faster than PLI. The memory consumption of Cinderella correlates with the number of identified conditions, while PLI is much less sensitive to larger numbers of conditions and generally needs less memory. These observations confirm our complexity estimations in Sec. 4.3.

Experiments on completeness conditions reveal equivalent results as can be seen in Fig. 8. In comparison to discovering covering conditions increases the number of identified conditions strongly for very low thresholds, which is caused by larger conditions (i. e., over more attributes) satisfying the thresholds. The runtime increases for both algorithms where more conditions are identified, i. e., for low thresholds. Memory consumption increases for both algorithms for all thresholds, which results from an increased amount of work: Cinderella must handle more baskets as each tuple forms a basket instead of each group. PLI must handle longer position lists, which result from using tupleIDs instead of



(a) runtime

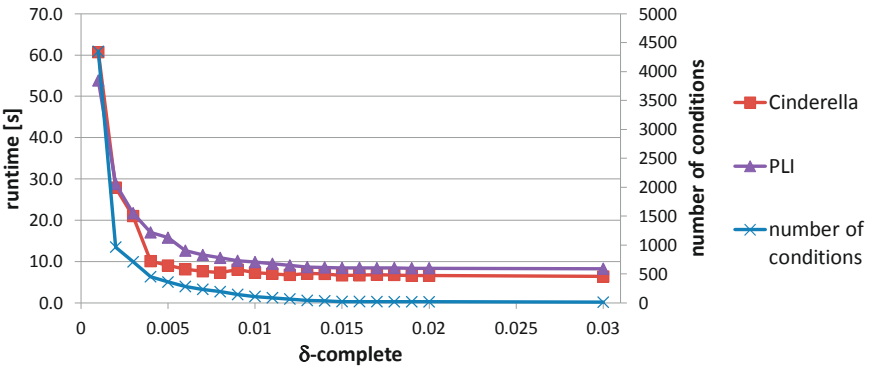


(b) memory consumption

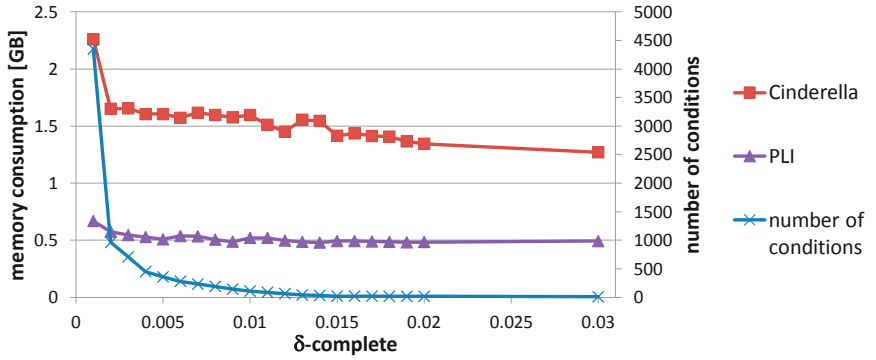
Fig. 7. Results for varying covering thresholds.

groupIDs. Again CINDERELLA increases less in runtime while PLI increases less in memory consumption, as expected by our complexity analysis.

Due to PLIs low memory requirements we were able to detect all conditions covering at least one person (with validity threshold set to zero). This run took 41 min using 2.4 GB memory and returned 566, 830 conditions. A run detecting completeness conditions with an absolute completeness of at least one tuple took 112 min using 5.8 GB memory and delivering 9, 214, 406 conditions. With CINDERELLAs higher memory requirements, we could perform the same experiment only up to at least 10 covered persons. The run took 3 min20 s using 7.8 GB RAM and returned 12, 587 conditions. Clearly, this entire set of conditions is not useful to find individual interesting conditions. But we can use it for a scatter plot as given in Fig. 5, which gives an intuition about the distribution of conditions and helps to set profitable thresholds.



(a) runtime

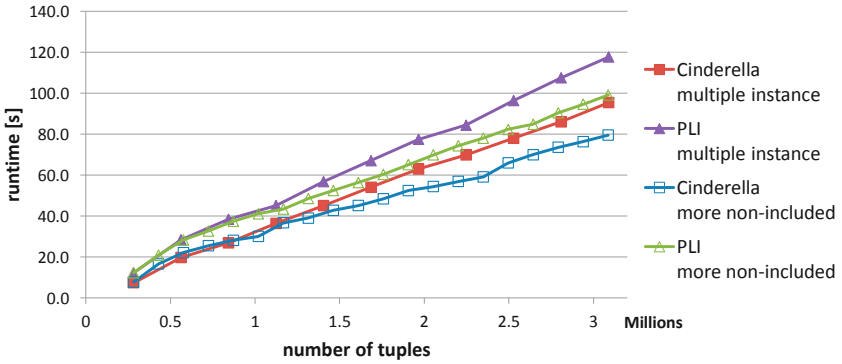


(b) memory consumption

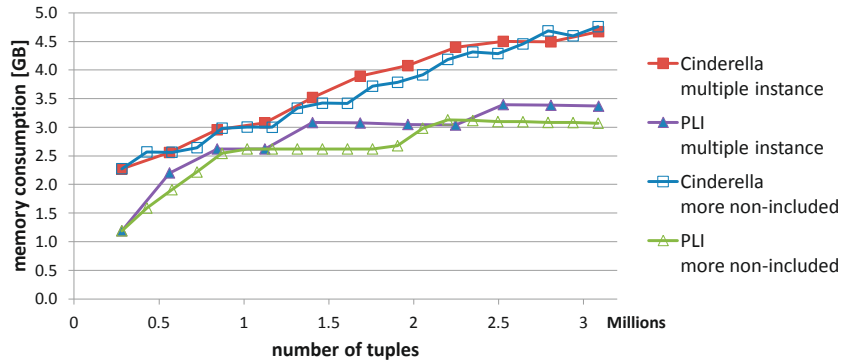
Fig. 8. Results for varying completeness thresholds.

Varying Size of Data Set. This experiment evaluates two aspects of the data set size: (i) the effect of the absolute size of the data set and (ii) the ratio of included and non-included groups (or tuples). Therefore, we concatenate multiple instances of the German DBpedia data set. To increase the absolute size of the data set with a constant ratio of included-to-non-included tuples we consider multiples of the entire data set. A second class of data sets is created by adding multiples of non-included persons to decrease the ratio of included-to-non-included persons. In both setups we ensured to add new persons instead of adding new tuples to the same group (person) by adding suffixes to values of attribute `de.person`. Note that the number of identified conditions is constant in both setups: If we multiply the entire data set, then all conditions and the ratios remain the same. Multiplying only the non-included persons has no impact on the covering (or completeness) threshold, because it relates to the constant number of included persons (or tuples).

Figures 9(a) and 9(b) show the runtime and memory behavior for both setups and both algorithms. Generally, runtime and memory consumption increase with increasing data size as expected from our complexity analysis. Multiplying only the non-included persons results in a softer increase of the runtime and memory consumption than multiplying the entire data set. This means, the amount of included tuples is the decisive factor for both algorithms, not so much the size of the entire data set. Again, Cinderella is faster, while PLI needs less memory.



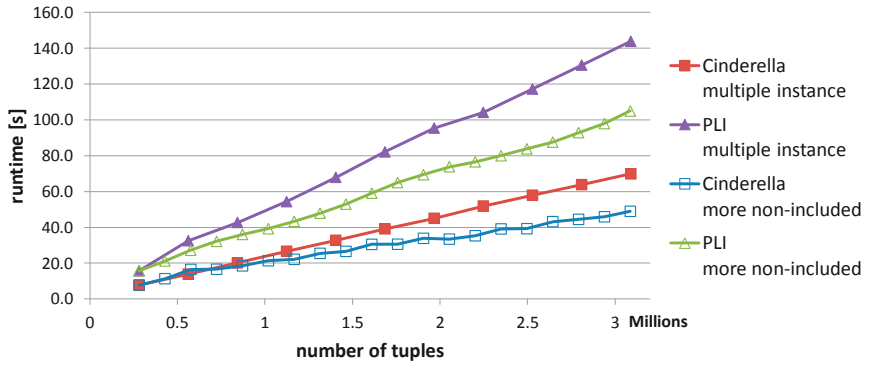
(a) runtime



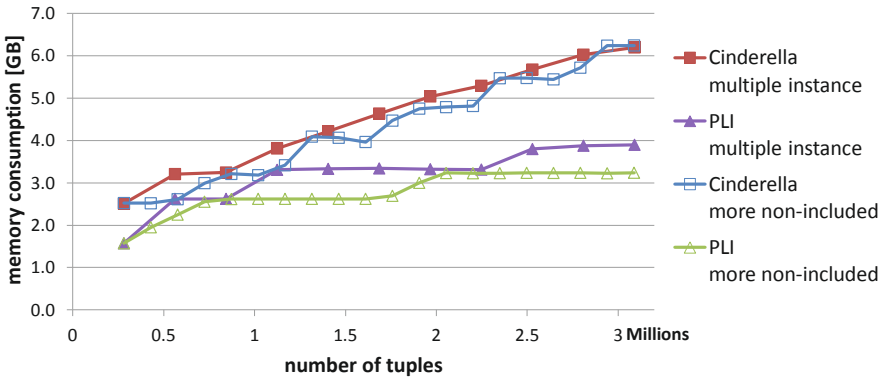
(b) memory consumption

Fig. 9. Results for discovering covering conditions over varying number of tuples.

Experiments for detecting completeness conditions using equivalent setups show comparable results (Fig. 10). All observations appear even stronger pronounced, which result again from the increased amount of work for discovering completeness conditions instead of covering conditions on the same data set.



(a) runtime



(b) memory consumption

Fig. 10. Results for discovering completeness conditions over varying number of tuples.

Our experimental results confirm our comparison of the algorithms in Sec. 4.3, as CINDERELLA is more runtime efficient due to its additional pruning possibilities, but needs always more memory than PLI, because of its breadth-first search manner and its in-memory baskets.

5.3 Evaluating the Wikipedia Use Case

A work closely related to ours is [14], which also discovers conditions for CINDs and builds additionally a pattern tableau, but pre-selects condition attributes and restricts considered conditions by parsimony. We use the same dataset to compare the conditions discovered by both approaches.

Golab et al. [14] use two tables of Wikipedia data, namely table `Image` with attributes `name`, `size`, `width`, `height`, `bits`, `media_type`, `major_mime`, `minor_mime`, `user`, `user_text`, `timestamp`, `sha1` and the table `Imagelinks` denoting links from

webpages to image files (attributes `il_from` and `il_to`). They assert the embedded IND `image.name` $\subseteq$ `imagelinks.il_to` and build a pattern tableau with completeness conditions of the pre-selected attributes `bits`, `media_type`, and `user_text`.

If we restrict our algorithms to the same attribute set with the same validity threshold of 0.85 and a completeness of at least 0.003, we discover the same conditions as [14], except the one on `user_text = ProteinBoxBot`, which shows a lower validity in our experiments. We cannot explain this slight difference, as we have used the original dataset pointed to by the authors. CINDERELLA runs 23s compared to 18s reported by [14] (on presumably different hardware).

However, our algorithms also discover more detailed conditions, which cannot be found by [14]: For condition `media_type = AUDIO` there is another condition `media_type = AUDIO` $\wedge$ `bits = 0` with the exact same validity and completeness. In the same manner, the five conditions `bits = 5`, `bits = 6`, `bits = 7`, `user_text = Blofeld of SPECTRE`, and `user_text = Melesse` can all be combined with `media_type = BITMAP` while covering the same tuples. These stricter conditions give more insight into the dataset and prevent wrongly generalizing the identified conditions for similar datasets.

The main advantage of our approach over [14] is that the condition attributes need not be pre-selected. Running our algorithms without restricting attributes yields even more interesting results: Unexpectedly, attributes `width` and `height` provide conditions with higher completeness than all other attributes. Conditions `width = 200` $\wedge$ `major_mime = image` and `width = 300` $\wedge$ `major_mime = image` both reach a completeness of 0.04, instead of the completeness measures of the above conditions between 0.003 and 0.008. Conditions `height = 300` and `height = 200` (each with completeness = 0.02), `height = 240` and `width = 240`, (each with completeness = 0.01) also have higher completeness. These conditions are non-trivial: other widths and heights also appear in the dataset with similar frequency. Another interesting identified condition regarding audio data is `bits = 0` $\wedge$ `major_mime = application` $\wedge$ `minor_mime = ogg` with completeness 0.008 and validity 0.9. CINDERELLA run 78 s to identify 188 conditions with γ-valid > 0.85 and δ-complete > 0.008.

In summary, the ability to select the condition attributes automatically led to the discovery of more complete conditions satisfying the same validity requirements, which in turn provide a base to build better pattern tableaux. [14] report an overall support of 0.0636, while we discover already *individual* conditions with a completeness of 0.04, which corresponds to a support of 0.03. Simply choosing our top two conditions yields a tableau with a completeness of 0.0824 (support 0.0641).

6 Related Work

Conditional inclusion dependencies (CINDs) were proposed by Bravo et al. [5] for data cleaning and contextual schema matching. In [5], complexity bounds for reasoning about CINDs and a sound and complete inference system for CINDs are provided. The problem of discovering CINDs from a given database instance, how-

ever, is not addressed. Different aspects of CIND discovery have been addressed in [2, 8, 14, 18]. De Marchi et al. propose data mining algorithms to discover approximate INDs, i. e., INDs that are satisfied by part of a given database [18]. To allow mining of approximate INDs an error measure is introduced based on the number of tuples that one has to remove from a database to obtain a database for which a IND is satisfied. Likewise, our IND discovery algorithm SPIDER has been adopted to discover approximate INDs [2]. Approximate INDs are input to the algorithms presented in this paper. The work in [2, 18] is therefore orthogonal to our work. Algorithms for generating pattern tableaux for given INDs are proposed in [8, 14]. The algorithm in [8], however, assumes that the given IND is fully satisfied by the database, i. e., it does not ensure or check validity of conditions. Golab et al. present *Data Miner*, a system for analyzing data quality [14]. Given an approximate IND, the system generates a pattern tableau that as a concise summary of those subsets of the database that a) satisfy, and b) fail the IND. Golab et al. assume that the set of attributes over which the tableau patterns is generated (i. e., X_P), is given as input to the algorithm. The fact that pre-selecting X_P is not necessary is one of the major differences to our work. A second difference is that we introduce the new concept of covering INDs which is essential for the type of data and use case that we consider.

The algorithm in [14] is an extension of the algorithm proposed in [13] for generating pattern tableaux for conditional functional dependencies (CFDs). CFDs were introduced in [10] for data cleaning. Similar to CINDs, a CFD augments an embedded functional dependency (FD) with a pattern tableau that defines the subset of the database in which the FD is satisfied. In [13] Golab et al. characterize the quality of a pattern tableau based on properties of support, confidence, and parsimony. The authors show that generating an optimal tableau for a given FD is NP-complete but can be approximated in polynomial time via a greedy algorithm. Here, we consider the problem of generating conditions for a pattern tableau. Deriving an optimal tableau from the discovered set of conditions is similar to the basic greedy algorithm proposed in [13]. To regard marginal local support and confidence defined in [13], which are necessary to build concise pattern tableaux, we can adapt our algorithms slightly: CINDERELLA can compute these measures by re-counting the itemset frequencies as in the rule generation step. PLI must preserve the position list for each identified condition (instead of saving only meta-data). Then the marginal local supports and confidences can be calculated for each condition after choosing conditions for the pattern tableau.

Algorithms for discovering CFDs are also considered in [7, 11]. In contrast to other approaches, the work in [7] does not assume that the FD is given in advance. Discovering FDs, however, is significantly different from discovering approximate INDs and it therefore is not clear how the algorithms in [7] can be applied to CIND discovery. Fan et al. propose an algorithm for discovering constant CFDs based on closed itemset mining [11]. A minimal constant CFD is a special form of CFD for which the pattern tableau contains only constant values for the attribute in the right-hand side of the embedded FD. Thus, minimal constant CFDs correspond

to association rules with single attribute in their antecedent with confidence 100 %, i. e., to selecting conditions with γ-validity one. Algorithm *CFDMiner* in [11] uses closed itemset mining to find such association rules. Algorithms for mining association rules with fixed or constrained antecedent that are based on Apriori were proposed in [17, 20]. These algorithms were the motivation for our's in Sec. 4.

Contradiction pattern are also a form of association rules with fixed antecedent [19]. Contradiction patterns were proposed to discover conditions that are frequent within a subset of a database but not frequent within the remainder of the database. The definitions of conflict relevance and conflict potential are similar to our definitions of valid and completeness conditions. Covering conditions, however, cannot be discovered using the algorithms presented in [19].

7 Conclusion and Future Work

We generalize the definition of CINDs by distinguishing covering and completeness conditions. This distinction is important when discovering CINDs over denormalized relations. To discover CINDs we present algorithms CINDERELLA and PLI. In contrast to existing approaches, both algorithms not only select the condition values but also the condition attributes automatically. CINDERELLA is faster than PLI, but consumes more memory. In our experimental evaluation we identified comprehensible, but unforeseen conditions that highlight characteristics of persons for which there exists a link between the English and German version of DBpedia.

In future work we plan to adapt the distinction of covering and completeness conditions to the right-hand side of the pattern tableau. That is, if a person in the English DBpedia matches condition `birthplace = California`, we could require the referenced person in the German DBpedia to match `birthplace = Kalifornien`. This modification is valuable to improve data quality. There is also an even more interesting application in linked data. We currently can identify persons that are candidates for links without being able to state to which corresponding person they should be linked to. Right-hand side conditions can be used to generate rules that are input to systems like SILK [3] for automatic generation of links between pairs of datasets in the Web of Data.

References

1. R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1994.
2. J. Bauckmann, U. Leser, and F. Naumann. Efficient and exact computation of inclusion dependencies for data integration. Technical Report 34, Hasso-Plattner-Institut, Potsdam, 2010.
3. C. Bizer, J. Volz, G. Kobilarov, and M. Gaedke. Silk - a link discovery framework for the web of data. In *Workshop about Linked Data on the Web (LDOW)*, 2009.

4. P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2007.
5. L. Bravo, W. Fan, and S. Ma. Extending dependencies with conditions. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2007.
6. W. Chen, W. Fan, and S. Ma. Analyses and validation of conditional dependencies with built-in predicates. In *Database and Expert Systems Applications*, 2009.
7. F. Chiang and R. J. Miller. Discovering data quality rules. *Proceedings of the VLDB Endowment*, 1:1166–1177, 2008.
8. O. Curé. Conditional inclusion dependencies for data cleansing: Discovery and violation detection issues. In *Proceedings of the International Workshop on Quality in Databases (QDB)*, 2009.
9. W. Fan. Dependencies revisited for improving data quality. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, 2008.
10. W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Transactions on Database Systems (TODS)*, 33(2):1–48, 2008.
11. W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 23(4):683–698, 2011.
12. M. J. Franklin, A. Y. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Record*, 34(4):27–33, 2005.
13. L. Golab, H. Karloff, F. Korn, D. Srivastava, and B. Yu. On generating near-optimal tableaux for conditional functional dependencies. *Proceedings of the VLDB Endowment*, 1:376–390, 2008.
14. L. Golab, F. Korn, and D. Srivastava. Efficient and effective analysis of data quality using pattern tableaux. *IEEE Data Engineering Bulletin*, 34(3):26–33, 2011.
15. H. Halpin, P. Hayes, J. P. McCusker, D. McGuinness, and H. S. Thompson. When owl:sameas isn't the same: An analysis of identity in linked data. In *Proceedings of the International Semantic Web Conference (ISWC)*, 2010.
16. Y. Huhtala, J. Kaerkkaeinen, P. Porkka, and H. Toivonen. TANE: an efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100–111, 1999.
17. B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *Proceedings of the ACM International Conference on Knowledge discovery and data mining (KDD)*, pages 80–86, 1998.
18. F. D. Marchi, S. Lopes, and J.-M. Petit. Unary and n-ary inclusion dependency discovery in relational databases. *J. Intell. Inf. Syst.*, 32:53–73, 2009.
19. H. Müller, U. Leser, and J.-C. Freytag. Mining for patterns in contradictory data. In *Proceedings of the SIGMOD International Workshop on Information Quality for Information Systems (IQIS)*, 2004.
20. R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *Proceedings of the ACM International Conference on Knowledge discovery and data mining (KDD)*, pages 67–73, 1997.

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
61	978-3-86956-194-3	**Vierter Deutscher IPv6 Gipfel 2011**	Christoph Meinel, Harald Sack (Hrsg.)
60	978-3-86956-201-8	**Understanding Cryptic Schemata in Large Extract-Transform-Load Systems**	Alexander Albrecht, Felix Naumann
59	978-3-86956-193-6	**The JCop Language Specification**	Malte Appeltauer, Robert Hirschfeld
58	978-3-86956-192-9	**MDE Settings in SAP: A Descriptive Field Study**	Regina Hebig, Holger Giese
57	978-3-86956-191-2	**Industrial Case Study on the Integration of SysML and AUTOSAR with Triple Graph Grammars**	Holger Giese, Stephan Hildebrandt, Stefan Neumann, Sebastian Wätzoldt
56	978-3-86956-171-4	**Quantitative Modeling and Analysis of Service-Oriented Real-Time Systems using Interval Probabilistic Timed Automata**	Christian Krause, Holger Giese
55	978-3-86956-169-1	**Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium**	Peter Tröger, Andreas Polze (Eds.)
54	978-3-86956-158-5	**An Abstraction for Version Control Systems**	Matthias Kleine, Robert Hirschfeld, Gilad Bracha
53	978-3-86956-160-8	**Web-based Development in the Lively Kernel**	Jens Lincke, Robert Hirschfeld (Eds.)
52	978-3-86956-156-1	**Einführung von IPv6 in Unternehmensnetzen: Ein Leitfaden**	Wilhelm Boeddinghaus, Christoph Meinel, Harald Sack
51	978-3-86956-148-6	**Advancing the Discovery of Unique Column Combinations**	Ziawasch Abedjan, Felix Naumann
50	978-3-86956-144-8	**Data in Business Processes**	Andreas Meyer, Sergey Smirnov, Mathias Weske
49	978-3-86956-143-1	**Adaptive Windows for Duplicate Detection**	Uwe Draisbach, Felix Naumann, Sascha Szott, Oliver Wonneberg
48	978-3-86956-134-9	**CSOM/PL: A Virtual Machine Product Line**	Michael Haupt, Stefan Marr, Robert Hirschfeld
47	978-3-86956-130-1	**State Propagation in Abstracted Business Processes**	Sergey Smirnov, Armin Zamani Farahani, Mathias Weske
46	978-3-86956-129-5	**Proceedings of the 5th Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering**	Hrsg. von den Professoren des HPI
45	978-3-86956-128-8	**Survey on Healthcare IT systems: Standards, Regulations and Security**	Christian Neuhaus, Andreas Polze, Mohammad M. R. Chowdhuryy
44	978-3-86956-113-4	**Virtualisierung und Cloud Computing: Konzepte, Technologiestudie, Marktübersicht**	Christoph Meinel, Christian Willems, Sebastian Roschke, Maxim Schnjakin

ISBN 978-3-86956-212-4
ISSN 1613-5652