Hasso-Plattner-Institut
University of Potsdam
Information Systems Group

Dependency Discovery for Data Integration

A thesis submitted for the degree of
"Doctor Rerum Naturalium"
(Dr. rer. nat.)
in Computer Sciences

Faculty of Mathematics and Natural Sciences
University of Potsdam

By: Dipl.-Inf. Jana Bauckmann

submitted on Potsdam, 2013/02/28

1st reviewer: Prof. Dr. Felix Naumann
2nd reviewer: Prof. Dr. Ulf Leser
3rd reviewer: Dr. Laure Berti-Équille (Ph.D, H.D.R)

defended on 2013/06/14

Published online at the
Institutional Repository of the University of Potsdam:
URL http://opus.kobv.de/ubp/volltexte/2013/6664/
URN urn:nbn:de:kobv:517-opus-66645
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-66645

Abstract

Data integration aims to combine data of different sources and to provide users with a unified view on these data. This task is as challenging as valuable. In this thesis we propose algorithms for dependency discovery to provide necessary information for data integration. We focus on inclusion dependencies (INDs) in general and a special form named conditional inclusion dependencies (CINDs): (i) INDs enable the discovery of structure in a given schema. (ii) INDs and CINDs support the discovery of cross-references or links between schemas.

An IND $A \subseteq B$ simply states that all values of attribute A are included in the set of values of attribute B. We propose an algorithm that discovers *all* inclusion dependencies in a relational data source. The challenge of this task is the complexity of testing all attribute pairs and further of comparing all of each attribute pair's values. The complexity of existing approaches depends on the number of attribute pairs, while ours depends only on the number of attributes. Thus, our algorithm enables to profile entirely unknown data sources with large schemas by discovering all INDs. Further, we provide an approach to extract foreign keys from the identified INDs.

We extend our IND discovery algorithm to also find three special types of INDs: (i) Composite INDs, such as $AB \subseteq CD$, (ii) approximate INDs that allow a certain amount of values of A to be not included in B, and (iii) prefix and suffix INDs that represent special cross-references between schemas.

Conditional inclusion dependencies are inclusion dependencies with a limited scope defined by conditions over several attributes. Only the matching part of the instance must adhere the dependency. We generalize the definition of CINDs distinguishing covering and completeness conditions and define quality measures for conditions. We propose efficient algorithms that identify covering and completeness conditions conforming to given quality thresholds. The challenge for this task is twofold: (i) Which (and how many) attributes should be used for the conditions? (ii) Which attribute values should be chosen for the conditions? Previous approaches rely on pre-selected condition attributes or can only discover conditions applying to quality thresholds of 100 %.

Our approaches were motivated by two application domains: data integration in the life sciences and link discovery for linked open data. We show the efficiency and the benefits of our approaches for use cases in these domains.

Zusammenfassung

Datenintegration hat das Ziel, Daten aus unterschiedlichen Quellen zu kombinieren und Nutzern eine einheitliche Sicht auf diese Daten zur Verfügung zu stellen. Diese Aufgabe ist gleichermaßen anspruchsvoll wie wertvoll. In dieser Dissertation werden Algorithmen zum Erkennen von Datenabhängigkeiten vorgestellt, die notwendige Informationen zur Datenintegration liefern. Der Schwerpunkt dieser Arbeit liegt auf Inklusionsabhängigkeiten (inclusion dependency, IND) im Allgemeinen und auf der speziellen Form der Bedingten Inklusionsabhängigkeiten (conditional IND, CIND): (i) INDs ermöglichen das Finden von Strukturen in einem gegebenen Schema. (ii) INDs und CINDs unterstützen das Finden von Referenzen zwischen Datenquellen.

Eine IND $A \subseteq B$ besagt, dass alle Werte des Attributs A in der Menge der Werte des Attributs B enthalten sind. Diese Arbeit liefert einen Algorithmus, der *alle* INDs in einer relationalen Datenquelle erkennt. Die Herausforderung dieser Aufgabe liegt in der Komplexität alle Attributpaare zu testen und dabei alle Werte dieser Attributpaare zu vergleichen. Die Komplexität bestehender Ansätze ist abhängig von der Anzahl der Attributpaare während der hier vorgestellte Ansatz lediglich von der Anzahl der Attribute abhängt. Damit ermöglicht der vorgestellte Algorithmus unbekannte Datenquellen mit großen Schemata zu untersuchen. Darüber hinaus wird der Algorithmus erweitert, um drei spezielle Formen von INDs zu finden, und ein Ansatz vorgestellt, der Fremdschlüssel aus den erkannten INDs filtert.

Bedingte Inklusionsabhängigkeiten (CINDs) sind Inklusionsabhängigkeiten deren Geltungsbereich durch Bedingungen über bestimmten Attributen beschränkt ist. Nur der zutreffende Teil der Instanz muss der Inklusionsabhängigkeit genügen. Die Definition für CINDs wird in der vorliegenden Arbeit generalisiert durch die Unterscheidung von überdeckenden und vollständigen Bedingungen. Ferner werden Qualitätsmaße für Bedingungen definiert. Es werden effiziente Algorithmen vorgestellt, die überdeckende und vollständige Bedingungen mit gegebenen Qualitätsmaßen auffinden. Dabei erfolgt die Auswahl der verwendeten Attribute und Attributkombinationen sowie der Attributwerte automatisch. Bestehende Ansätze beruhen auf einer Vorauswahl von Attributen für die Bedingungen oder erkennen nur Bedingungen mit Schwellwerten von 100 % für die Qualitätsmaße.

Die Ansätze der vorliegenden Arbeit wurden durch zwei Anwendungsbereiche motiviert: Datenintegration in den Life Sciences und das Erkennen von Links in Linked Open Data. Die Effizienz und der Nutzen der vorgestellten Ansätze werden anhand von Anwendungsfällen in diesen Bereichen aufgezeigt.

Acknowledgements

I thank my advisors Prof. Felix Naumann and Prof. Ulf Leser for their constant support and a great research environment. In our weekly meetings Prof. Felix Naumann always advanced my work with fruitful discussions, which provided new ideas and new insights. Prof. Ulf Leser always offered new perspectives on my work and food for thought on various problems.

I thank Ziawasch Abedjan and Heiko Müller for our inspiring research cooperation in the CIND discovery project.

I further thank all students that participated in different degrees in this research. In particular, I thank Jan Hegewald for our lively discussions and his work on LINKFINDER, Tobias Flach for implementing the Aladin tool, and Benjamin Emde for his ideas on CIND discovery using SQL.

I also thank my colleages, especially Melanie Herschel, Silke Trissl, Jens Bleiholder, Christoph Böhm, and Alexander Albrecht, who always provided support, suggestions, and fun in many lunch and coffee talks and discussions.

Finally, I thank my husband Hagen for being there, many talks, and lots of patience that backed me all along.

Contents

Chapter 1

Dependencies for Data Integration

Data integration aims to combine information of several data sources and thus provides the ability to gain new insights. Classical approaches define one global and several local schemas with mappings between them – distinguishing local-as-view, global-as-view, and global-local-as-view [46].

In the last years dataspaces were introduced to integrate several data sources in a "pay-as-you-go" fashion [29]. Dataspaces postpone integrating the data until integrated access is necessary and aim to provide this integration without the need of an complete and correct semantic mapping between data sources [36]. Another example for loosely coupled data sources are linked open data: Relationships between data sources are represented by links whose type and information degree can differ [37].

Despite the differences of all these approaches they share one basic property: Data dependencies offer the chance to gain necessary information to relate unknown sources and thus help to query these sources. In this thesis we focus on inclusion dependencies (INDs) as data dependencies that support data integration. An IND $A \subseteq B$ ensures that all values of the *dependent attribute* A are included in the set of values of the *referenced attribute* B. Thus, INDs provide information on data overlap, which makes them a good base for data integration. In the relational data model INDs between relations of one schema can be used to find foreign keys, while INDs between different schemas can be used to find inter-schema links.

Recently, a weaker form of dependencies, so called conditional dependencies, have gained attention, mostly due to their power in analyzing and improving data quality [26]. 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. Naturally, in this thesis we focus on discovering conditional inclusion dependencies (CINDs). CINDs provide the ability to find links between data sources that are restricted to a subset of the data. The identified conditions help to understand the characteristics of this subset and therefore help to better integrate both data sources.

Discovering Inclusion Dependencies. To test a single attribute pair A, B as IND $A \subseteq B$ we need to compare both attribute's values: Are all values of A included in the set of values of B? We could perform this check using SQL, e. g., using a join over A and B. $A \subseteq B$ holds, if the number of distinct values in A equals the number of distinct values of A in the join result. The challenge of identifying INDs in a given data source arises when we aim to identify *all* INDs: We have to test all attribute pairs and for each attribute pair we have to compare all of their values, which means a quadratic complexity in the number of attributes multiplied by the quadratic complexity in the number of values. We present our algorithm SPIDER, which efficiently identifies all INDs in a relational database. It leverages the sorting facilities of DBMS but performs the actual comparisons outside the database to save computation: All attribute pairs are tested as INDs in parallel using a special data structure, which reduces the number of necessary comparisons and I/O. This approach reduces the quadratic complexity in the number of attributes to a linearithmic time complexity (i. e., $n \log n$) and even a linear space complexity. SPIDER analyzes very large databases up to an order of magnitude faster than previous approaches. Thus, our algorithm enables to profile entirely unknown data sources with large schemas by looking for all INDs.

One of our test data sources is the well-known Protein Data Bank (PDB), a publicly available database containing essentially all known protein 3D-structures. PDB is distributed as a structured flat-file and can be imported into a relational schema using the OpenMMS software[1]. The OpenMMS schema consists of 176 tables with 2,713 attributes but does not specify a single foreign key constraint. This under-specification hinders the integration of this database with other life sciences data sources considerably [63]. Testing all attribute pairs with unique referenced attributes using one SQL statement per attribute pair did not finish within seven

[1] `openmms.sdsc.edu`

days. SPIDER finds all INDs in about six hours.

To use identified INDs as source of information on intra-schema structures, such as foreign keys, we want to distinguish spurious INDs from semantically meaningful INDs. Thus, we propose an approach to set spurious INDs apart from real foreign keys. We reach an overall F-measure of 93 % on our test databases.

We extend SPIDER in several ways to support several cases in data integration: First, we support the discovery of composite INDs, which are defined over composite attributes (such as $AB \subseteq CD$). Our algorithm Composite SPIDER leverages SPIDER's linearithmic complexity. Therefore, it is able to discover composite INDs efficiently up to level three in our test data sources, i. e., all INDs of type $A \subseteq B$ (level one), $AB \subseteq CD$ (level two), and $ABC \subseteq DEF$ (level three). It was intractable in discovering composite INDs of higher levels, because of the exponential growing search space. Existing algorithms for composite INDs are designed for discovery of INDs with very high levels (up to size 41 [44, 45, 54]). They are based on INDs of level one and two, but they lack efficient discovery of such INDs. Combining Composite SPIDER and these algorithms improves discovery of composite INDs altogether.

Our second extension of SPIDER discovers approximate INDs, which allow a certain amount of values in the dependent attribute to be not included in the set of values of the referenced attribute. Approximate INDs occur in dirty data, or they can indicate links between overlapping data sources. Again, Approximate SPIDER leverages SPIDER's linearithmic complexity.

Discovering a special type of inter-schema links is our third extension of SPIDER: LINKFINDER discovers INDs with prefixes or suffixes concatenated to the values of the dependent attribute. For instance, the protein classification data source CATH [57] links entries in the PDB using the actually linked protein's identifier concatenated with two characters – representing the applied protein domain. The challenge is to distinguish prefixes (or suffixes) from referencing values while allowing for each IND (i) differing lengths of referencing values and (ii) differing prefixes (or suffixes) with differing lengths. LINKFINDER reuses the idea of testing all attribute pairs in parallel to identify prefixed or suffixed INDs efficiently. It thus enables discovery of inter-schema links.

Discovering Conditional Inclusion Dependencies. CINDs are approximate INDs, enriched by conditions distinguishing the referenced attribute's included values

from non-included values. Discovering these conditions faces two challenges: The first challenge is to decide (and to describe) which conditions should be discovered, i.e., which conditions are good conditions? We characterize conditions as valid, complete, or covering and define quality measures over these properties. The new notion of covering conditions enables use cases in linked open data or in relational schemas over joined or denormalized relations.

The second challenge is to identify the desired conditions: How many and which attributes should be used for a condition? And which attribute values should be used for a condition? Our algorithms Cinderella and Pli discover such conditions efficiently. Their advantage is the ability to choose condition attributes and values – instead of only choosing condition values for pre-selected attributes as related work does.

Already in a relation with only 12 potential condition attributes there are $2^{12} = 4096$ possible attribute combinations. In our use case with these 12 potential condition attributes and about 280,000 tuples in the referencing relation we need 5.9 seconds to test *one* attribute combination for valid and covering conditions using Sql– summing up to about 6.5 hours to discover all valid and covering conditions with a given quality. Cinderella and Pli discover these conditions in about 8 seconds.

In the following we describe two use cases that show the relevance of our work. We use them in the entire thesis to show the effectiveness and efficiency of our approaches.

1.1 Data Integration in the Life Sciences

The "Nuclear Acids Research online Molecular Biology Database Collection" [30] currently lists 1,380 data sources. These sources provide heavily overlapping data with cross-references between data sources using so-called accession numbers [39] – which makes them an excellent resource for data integration applications. Unfortunately, usage of information spread over several sources is dominated by manual interaction: Searching one data source, finding a key-word or a cross-reference to another data-source, and finally searching the other data source with this information.

Obviously, it is hard to know for 1,380 data sources which data source cross-references which other sources in which way. Furthermore, each research community in the life sciences tends to set up its own data source instead of cooperating with other researchers in the same field. This leads to a steeply increasing number of heavily overlapping life science data sources [31].

There are data integration projects in the life sciences that combine known data sources with known structures – such as Columba, an integrated database of proteins, structures, and annotations [63], or the Gene Expression Data Warehouse (GEDAW) related to liver data [14, 34]. On the other hand there are frameworks that provide the ability to access and interlink known data sources and thus provide an integrated access to these data sources, such as TAMBIS [5], yOWL [64], or bio2RDF [12]. Note that information about how to interconnect these data sources always must be provided.

Opposed to these approaches our algorithms gather information to interconnect unknown data sources. We follow the idea of the Aladin approach [47] aiming to discover intra-schema and inter-schema relationships automatically. This idea seems confirmed by current life science research, which intertwines data integration and data analysis. The approach enables researchers to know exactly which data from which source is used – a main requirement in life science research [17].

Schemas of life science data sources are modeled most often insufficiently: Even large and successful systems, such as PDB [13] or Ensembl [40], both with hundreds of tables, are delivered without foreign key definitions. We apply our algorithm SPIDER and our approach to distinguish meaningful from spurious INDs to discover foreign keys.

Discovering cross-references between data sources supports the goal of data integration. We want to detect these references to define links at schema level and at object level. There are several methods used to link objects, i. e., to store and represent the referenced data source and the accession number. In our research we identified the following types:

A data source references only one data source using a single attribute, e. g., the protein classification data source SCOP [50] only references the Protein Data Bank PDB [13] using the attribute classification.pdb_id. In this case, the referenced data source is given in the attribute name. We identify this reference by discovering the IND using SPIDER or Approximate SPIDER.

Another type of cross-references to one other data source is the concatenation of an accession number and additional information. For instance, CATH [57] – a protein classification data source – uses a single attribute to reference PDB concatenating the accession number and two characters representing the protein domain. We apply LinkFinder to discover cross-references of this type.

Many data sources reference not only one data source, but reference several data sources. The usual way is to use a combination of a "referenced database code" and an accession number. This combination could be represented in two attributes or concatenated in one attribute. For instance, the BioSQL schema[2] – a shared database schema for storing sequence data – uses the attributes dbxref.dbname and dbxref.accessionnr to reference other life science data sources. We parsed the protein data source UniProt [4] into this schema. UniProt references 67 data sources. One of these referenced data sources is PDB. We discover and characterize this cross-reference in two steps: First, we apply approximate Spider to discover the approximate Ind to PDB with dbxref.accessionnr as dependent attribute. Second, we apply Cinderella or Pli to find conditions that characterize the linking values. This way, we were able to identify attribute dbxref.dbname with value PDB as condition. But joining relations in BioSQL (discovered using Spider) provided even better insights: About 50 % of all proteins in UniProt that reference PDB are described as proteins of human or Escherichia coli. This condition gives a semantic explanation why these proteins reference PDB, which then identifies proteins that probably should cross-reference PDB. Therefore, we are not only able to identify the cross-reference between both data sources, but are also able to propose data quality improvements.

1.2 Link Discovery for Linked Open Data

Linked open data are publicly available, open licensed RDF data, which provide links between each other. Only these links enable the use of several data sources to gain new information. But discovering and maintaining such links is a hard task.

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 inter-

[2] http://biosql.org

linking representations of persons in the English and German version of DBpedia[3]. 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 linking objects in this overlap. Many examples of such cases can be found in the world of Linked Open Data [37] and in dataspaces [29].

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 covering conditions – a generalization 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.

1.3 Contributions and Outline

The contributions of this thesis are as follows:

- We propose our algorithm SPIDER, which discovers all INDs of a given schema efficiently. SPIDER is for small schemas at least twice as fast as previous approaches and for large schemas up to a magnitude of order faster than previous approaches. Thus, SPIDER enables profiling of large schemas – without demanding necessary information in advance [8, 9].
- To show SPIDER's efficiency we provide a complexity estimation and exper-

[3] `http://wiki.dbpedia.org/Downloads`

iments on real world and generated data of several domains – including life sciences data sources. Further, we compare SPIDER to related work revealing SPIDER's superiority [8, 9].

- We evaluate the usage of identified INDs as foreign keys, propose an approach to distinguish spurious INDs from foreign keys, and provide an approach to discover the "primary relation" – a structural characteristic of life sciences data sources. The contribution on distinguishing spurious INDs from foreign keys is based on the term paper [3], which was supervised by the thesis' author, and was extended in joint work with Rostin et al. [60].
- We extend SPIDER to discover approximate and composite INDs. Both algorithms leverage the linearithmic complexity of SPIDER, which is confirmed by our experimental results [8].
- We extend SPIDER to discover prefix or suffix INDs, which represent a special type of cross-references between data sources. We evaluate efficiency and effectiveness to discover cross-references among life sciences data sources. This contribution is based on the masters thesis [38], which was supervised by the thesis' author.
- We describe a novel *use case* for CIND discovery 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 complete conditions, which enables this use case and further enables discovering CINDs over denormalized relations [6, 7].
- 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 [6, 7].
- We present two *algorithms* – CINDERELLA and PLI– that efficiently identify valid and covering (or valid and complete) conditions, while choosing the condition attributes and their values automatically. We provide a thorough evaluation of the effectiveness and efficiency both of algorithms using two real-world data sets. This contribution is joint work with Ziawasch Abedjan and others [6, 7].

The remainder of this thesis is structured as follows: Chapter 2 provides terms and definitions and covers related topics for schema detection. In Chapter 3 we describe

approaches to discover unary INDs, i. e., SQL based approaches and SPIDER, and propose pruning strategies for CIND discovery. An evaluation of these approaches regarding effectiveness and efficiency is presented in Chapter 4. We extend SPIDER to discover approximate, composite, and prefix or suffix INDs and evaluate these algorithms in Chapter 5.

Chapter 6 covers the discovery of CINDs: from providing the ability to describe desired condition, over an SQL approach to two efficient algorithms, namely CINDERELLA and PLI. We evaluate and leverage CINDs in Chapter 7 and conclude the thesis in Chapter 8.

Chapter 2

Identifying Database Structure

In this chapter we consider two topics that could help in schema discovery for data integration at first sight. We show the commonalities with our application and the differences, which impede their usage in this thesis. Afterwards we provide terms and definitions and introduce our test data sources.

2.1 Related Topics for Schema Discovery

There are several approaches in the literature that cover schema discovery. We discuss two topics that are related to our application of discovering database structure: *Schema matching* and *discovering functional dependencies*. Both topics share the aim to discover dependencies between attributes with our work.

Schema matching identifies mappings between elements of two schemas. These mappings indicate semantically corresponding elements [59]. Schema matching presumes that the data sources provide the same information, i. e., store data on the same topic in syntactically and structurally different representations. Semantically equivalent parts are mapped to each other using the data source's structures or instances. Thus, schema matching looks for overlapping parts of both data sources. We look in our application at different, yet related data sources that interlink each other. Consider for example the protein classification data source SCOP, which references the protein 3D-structure data source PDB: Both data sources provide information on proteins, but their type of information on proteins differs. Thus, we find cross-references between them, but cannot find (larger) overlapping parts between both data sources. The prerequisites to apply schema matching are more

demanding than the requirements for IND discovery. Further, schema matching is based on similarities, while IND discovery provides the chance to discover exact dependencies (in few overlapping attributes). Because of these fundamental differences regarding the prerequisites and the intended purpose between schema matching and IND discovery we do not further consider schema matching in this thesis.

Functional dependencies are a class of data dependencies, as are inclusion dependencies. A functional dependency $X \rightarrow Y$ demands for two tuples with equal values in attributes X to also share equal values in attributes Y. Functional dependencies and inclusion dependencies are covered together in many works, which suggests to also share their approaches in discovering both dependencies. In detail, functional dependencies are equality generating dependencies and inclusion dependencies are tuple generating dependencies. Both are covered together when looking at axiomatization, decision problems, and implication [19, 21]. But both problems are completely different in the aim to discover these dependencies: Functional dependencies consider tuples in one relation with same values in the antecedent and consequent attributes. Thus, discovering functional dependencies needs to find horizontal partitions of several attributes sharing equal values in one relation [41]. Opposed to that, discovering inclusion dependencies needs to find pairs of attributes in several relations whose value sets are included in each other. Thus, we cannot use ideas of discovering functional dependencies for link discovery in data integration.

Instead of these topics, we focus in this thesis on discovery and usage of inclusion dependencies – in several modifications – to support data integration.

2.2 Terms and Definitions

Let R_1, R_2 be relations over a fixed set of attributes $A_1, A_2, \ldots, A_k$. Each attribute A has an associated domain $dom(A)$. We denote arbitrary 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 $A \in R$. We use $t[X]$ and $I[X]$ to denote the projection of t and I onto attributes X, respectively.

Def. 1: Unary Inclusion Dependency Given relations R_1, R_2 and their arbitrary instances I_1, I_2: A unary inclusion dependency (IND)
$R_1[A] \subseteq R_2[B]$, or for short $A \subseteq B$, is built of the *dependent* attribute $A \in R_1$ and

the *referenced* attribute $B \in R_2$. $R_1[A] \subseteq R_2[B]$ means that all values of A are included in the set of values of B, i. e., $\forall t_1 \in I_1 \exists t_2 \in I_2 : t_1[A] = t_2[B]$.

□

Obviously, the IND relationship is not symmetric. We call any pair of attributes A and B an IND candidate, which we denote by $A \subseteq^? B$. An IND is *satisfied* if the IND requirements are met and *unsatisfied* otherwise. We denote satisfied INDs as $A \subseteq B$ and unsatisfied INDs as $A \nsubseteq B$. A *value* $v \in I_1[A]$ *violates an* IND $A \subseteq B$ if $v \notin I_2[B]$. An attribute C is *covered by an IND (candidate)* if it is part of that IND (candidate) as dependent or as referenced attribute: $C \in \{A, B\}$.

Strictly speaking, INDs cannot be discovered: One can merely verify whether they are satisfied for two given instances of two relations. For sake of clarity we ignore this fine distinction throughout the thesis.

We now extend the definition of unary INDs to composite INDs. Therefore, we first define the term composite attribute.

Def. 2: Composite Attribute A composite attribute is a list of attributes. The composite attribute's values are built by concatenating for each tuple the single attribute's values. We denote a composite attribute by listing its single attributes.

□

The number of single attributes is called the *size* of a composite attribute. For example, the composite attribute AB concatenates the values of attributes A and B; the size of AB is 2.

Def. 3: Composite Inclusion Dependency A composite IND $R_1[A_1, \ldots, A_n] \subseteq R_2[B_1, \ldots, B_n]$, or for short $A_1 \ldots A_n \subseteq B_1 \ldots B_n$, is an IND that is defined over composite attributes. The dependent and referenced composite attributes must have the same size.

□

The *level* of a composite IND is the size of the dependent (and referenced) composite attributes. For example, the composite IND $AB \subseteq CD$ is built of the dependent composite attribute AB and the referenced composite attribute CD; the level of $AB \subseteq CD$ is 2. Analogously to INDs, a pair of composite attributes of the same size is a composite IND candidate $A_1 \ldots A_n \subseteq^? B_1 \ldots B_n$.

One purpose of discovering INDs in this thesis is discovering foreign keys:

Def. 4: Foreign Key A Foreign Key (FK) is a semantically meaningful, justified IND. For IND $A_1 \dots A_n \subseteq B_1 \dots B_n$ we call $A_1 \dots A_n$ the foreign key, which references key $B_1 \dots B_n$. □

Foreign keys are defined in a schema by a human expert, i. e., a database administrator.

We weaken the strict definition of INDs to allow further dependency discovery between data sources and in dirty data:

Def. 5: Approximate Inclusion Dependency An approximate IND $R_1[A] \sqsubseteq R_2[B]$, or for short $A \sqsubseteq B$, is an IND that allows a certain amount of violating values in the dependent attribute A. We call this amount error rate. It can be defined as an absolute number of values or as percentage of distinct dependent values. □

We introduce a further type of IND to support discovery of cross-references between data sources that concatenate the linked value by prefixes or suffixes.

Def. 6: Prefix Inclusion Dependency A Prefix IND $R_1[A] \subseteq_p R_2[B]$, or for short $A \subseteq_p B$, is an IND between the dependent attribute A and the referenced attribute B after removing a (fixed or variable) prefix from each value in A. □

Def. 7: Suffix Inclusion Dependency A Suffix IND $R_1[A] \subseteq_s R_2[B]$, or for short $A \subseteq_s B$, is an IND between the dependent attribute A and the referenced attribute B after removing a (fixed or variable) suffix from each value in A. □

In Chapter 1 we introduced the example of CATH referencing PDB: Attribute domain_name in relation domain_list of CATH references attribute entry_id of relation struct in PDB by using values of entry_id concatenated with a suffix representing the domain of this protein (assigned by CATH). We denote this suffix IND as domain_list[domain_name] $\subseteq_s$ struct[entry_id].

Any pair of attributes A and B is called a prefix IND candidate $A \subseteq_p^? B$, and a suffix IND candidate $A \subseteq_s^? B$. The referencing part of values in A, i. e., the value after removing the prefix or suffix, is called the *key*. Each prefix (or suffix) IND has a varying or fixed *key length* and a varying or fixed prefix (or suffix) length. We

denote a prefix IND between attributes A and B with key length k and prefix length l as $A \subseteq_p^{k,l} B$, a suffix IND analogously as $A \subseteq_s^{k,l} B$. Varying key and prefix (or suffix) lengths are denoted by a dot, i. e., a suffix IND with fixed key length k and varying suffix length is denoted as $A \subseteq_s^{k,\cdot} B$.

Analogously to INDs, we define approximate prefix and approximate suffix INDs:

Def. 8: Approximate Prefix Inclusion Dependency An approximate prefix IND $R_1[A] \sqsubseteq_p R_2[B]$, or for short $A \sqsubseteq_p B$, is a prefix IND that allows a certain amount of violating values in the dependent attribute A. We call this amount error rate. It can be defined as an absolute number of values or as percentage of distinct dependent values. □

Def. 9: Approximate Suffix Inclusion Dependency An approximate suffix IND $R_1[A] \sqsubseteq_s R_2[B]$, or for short $A \sqsubseteq_s B$, is a suffix IND that allows a certain amount of violating values in the dependent attribute A. We call this amount error rate. It can be defined as an absolute number of values or as percentage of distinct dependent values. □

Of course we could also define approximate composite INDs, composite prefix INDs, or composite suffix INDs, but these types of INDs are out of scope of this thesis.

Lastly, we define conditional inclusion dependencies (CINDs), which restrict the scope of an IND by conditions: Formally, a CIND is defined by an embedded approximate IND and a pattern tableau representing the conditions. The following definitions are based on [19] and [26], but we chose a different, yet in our mind more intuitive formulation.

Def. 10: 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 pattern $t_p \in T_P$, $t_p[A]$ 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 $\forall A \in X_P$: $t_p[A] =$ ('–'

$\vee t_1[A]$). 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 empty, i. e., X_P or Y_P can be empty sets; an empty side specifies no restriction on any attribute of the respective relation. We call attributes X_P and Y_P *condition attributes*.

Def. 11: Conditional inclusion dependency A conditional inclusion dependency (CIND)

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

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

In the context of CINDs, we call attributes X and Y *inclusion attributes*. In Chapter 1 we described the example of UniProt in the BioSQL schema referencing PDB: The approximate IND dbxref[accessionnr] $\sqsubseteq$ struct[entry_id] is restricted by condition dbxref[dbname] = PDB. We denote this CIND as follows:

$$\varphi\colon\ (\texttt{dbxref[accessionnr; dbname]} \subseteq \texttt{struct[entry\_id;]}, T_P)$$

$$T_P\colon \begin{array}{|c||} \hline \text{dbname} \\ \hline\hline \text{PDB} \\ \hline \end{array}$$

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.

2.3 Test Data Sources

In this section we introduce our test data sources, which are several life sciences data sources, one generated data source, one extremely large SAP/R3 data source, and DBpedia person data.

UniProt[1] is a database of annotated protein sequences available in several formats [4]. We chose the BioSQL[2] schema and parser, creating a database of 16 tables with 85 attributes and 21 defined foreign keys. The total size of the database is 900 MB, with the largest attribute having approximately 1 million different values.

SCOP[3] is a database of protein classifications available as a set of files [56]. We wrote our own parser, populating 4 tables with 22 attributes. The total size of the database is 17 MB, with the largest attribute having about 90,000 different values.

CATH is another database of protein classifications, which is available as set of files. We populated 4 tables with 25 attributes. The total size of the database is 20 MB, with the largest attribute having about 65,000 different values.

PDB is a large database of protein structures [13]. We used the OpenMMS software[4] for parsing PDB files into a relational database. PDB populates 116 tables with 1,297 non-empty attributes in the OpenMMS schema. No foreign keys are specified. The total database size is 32 GB, with the largest attribute having approximately 218 million different values. To achieve a better idea of the scalability of our approaches, we also used a 2.8 GB fraction of the PDB, obtained by removing 9 extremely large tables on atom coordinates for each atom in each protein.

Additionally, we used a generated instance of the *TPC-H* benchmark using scale factor one. The TPC-H schema consists of 8 tables with 61 attributes. It defines 7 unary foreign keys and 1 foreign key over composite attributes of size 2. Further, we used a SAP/R3 database instance[5] as a very large database with an enormous schema. It populates 25,002 non-empty tables with 237,836 attributes. The total size is 145 GB.

For testing our Cind discovery approaches we use person data in the German and English DBpedia [15]. In DBpedia, individual persons are represented by the same URI in both data sets. There are 296,454 persons in the English DBpedia 3.6 and 175,457 persons in the German DBpedia 3.6[6]; 74,496 persons are included in both data sets. We mapped the data sets into relations to enable detection of covering and complete conditions. We use one attribute per predicate, namely

[1] `www.pir.uniprot.org`

[2] `obda.open-bio.org`

[3] `http://scop.mrc-lmb.cam.ac.uk/scop`

[4] `openmms.sdsc.edu`

[5] We thank the SAP HCC at Otto-von-Guericke-Universität Magdeburg for providing this database instance.

[6] `http://wiki.dbpedia.org/Downloads`

`personID`, `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.

Chapter 3

Discovering Unary Inclusion Dependencies

In this chapter, we provide algorithms to discover unary INDs. The main challenge is *regarding all attribute pairs as IND candidates*, instead of pruning candidates, e. g., by data type or restricting to a subset of relations or attributes. This generality enables to profile entirely unknown data sources, even with weak schema definitions as in our life sciences use case. But this generality demands to cope with large numbers of IND candidates, and thus demands to perform the tests of IND candidates as efficiently as possible. We show that previous approaches do not scale up accordingly.

We define the problem of unary IND discovery as follows:

Problem Statement 1: Discovering unary INDs Given a schema with an arbitrary number of relations and n attributes: Discover all satisfied unary INDs between attributes of these relations, i. e., consider all $n(n-1)$ attribute pairs as IND candidates. □

Note that our problem statement does not demand different relations for the attributes in an IND candidate. We could restrict the problem definition in this point: Let be given k relations with m attributes in each relation. The number of IND candidates for the attributes of one relation R is then $m \cdot (k-1)m$, i. e., all attributes of R compared to all other $(k-1)$ relation's attributes. Thus the overall number of IND candidates is $k \cdot m \cdot (k-1)m$. As this problem definition also results in a quadratic number of IND candidates depending on the number of attributes (and relations), we decided to use the unrestricted, more general problem definition.

Section 3.1 presents ideas on testing IND candidates inside the DBMS using SQL. Section 3.2 introduces our algorithm SPIDER, which tests IND candidates outside the DBMS to allow parallel tests while saving I/O and comparisons. We give a complexity analysis for both approaches showing (i) the major disadvantage of the SQL approach – the dependency on the number of IND candidates, i. e., the quadratic dependency on the number of attributes – and (ii) the major advantage of SPIDER – its only linearithmic dependency on the number of attributes.

Further, we show in Section 3.3 pruning strategies based on the data values, aiming to reduce the number of IND candidates to test without losing any IND, i. e., we only prune non-satisfied INDs.

3.1 SQL approaches

Our first approaches use SQL for performing set inclusion tests. We propose four alternative statements using join, except, not in, and not exists to evaluate the ability of SQL to test all IND candidates of a given schema. In each case, only one query is necessary to perform the actual test for an IND candidate. We show the statements in the following paragraphs assuming an IND candidate $A \subseteq^? B$ between attribute A in relation R and attribute B in relation S.

Utilizing Join The first statement utilizes a join as proposed by [11]. We perform a join on the inclusion dependency candidate and compare the number of returned tuples with distinct dependent values against the number of distinct non-null values in the dependent attribute. The IND is satisfied if both values are equal. The join statement can be seen in Figure 3.1a. Note that this statement can be simplified for IND candidates with a unique referenced attribute (see Fig. 3.1b): It suffices to compare the number of all returned tuples with the number of all non-null values in the dependent attribute. This step replaces the costly counting of distinct values after projecting on the dependent attribute by a simple count of all returned tuples. Note that the number of (distinct) non-null values of each attribute can be queried once for each attribute.

These statements simply compute a join over two sets, which is a different problem than the IND test. We can use a join to verify INDs, but we cannot tell the RDBMS engine that the procedure can be stopped as soon as any tuple is detected for which

```
select count(distinct A)
       as matchedDependents
from   R JOIN S on R.A = S.B
```

(a) general

```
select count(*)
       as matchedDependents
from   R JOIN S on R.A = S.B
```

(b) for unique referenced attributes

Figure 3.1: Join statements to test IND candidate $R.A \subseteq^? S.B$ utilizing join.

no join partner exists. Therefore, we formulate three other statements that aim to assist the DBMS to recognize this point. The idea is to find a statement that returns no tuples if the IND candidate is satisfied and one or more tuples otherwise. This way, we can stop the computation after the first tuple in the result set, hoping that first tuples can be produced without computing the entire result.

Utilizing Except The idea of utilizing except is to subtract values of the referenced attribute from values of the dependent attribute; if there are tuples in the result set then the IND candidate is not satisfied. The statement can be seen in Figure 3.2. Note that the fetch first n rows only clause is used to enable the query engine to stop execution early.

```
select *
from ( (select A from R where A is not null)
       EXCEPT (select B from S) )
fetch first 1 rows only
```

Figure 3.2: Statement to test IND candidate $R.A \subseteq^? S.B$ utilizing except.

Utilizing an Anti-Join: Not In and Not Exists Another possibility to obtain an empty result set if the tested IND candidate is satisfied is to utilize an anti-join. An anti-join returns all tuples that are not included in a semi-join result. There are two possibilities to formulate this anti-join: Using Not In and Not Exists. The idea is to ask for values in the dependent attribute that are not included in the referenced attribute. Again, we can restrict the result set to a single tuple as can be seen in Figure 3.3.

Note that there may be further possible statements to test each IND candidate or vendor-specific tricks and tweaks to speed up the execution. But we aim to

```
select *
from R
where A is not null
    and A NOT IN
        (select B from S)
fetch first 1 rows only
```

(a) Using not in

```
select *
from R
where A is not null
    and NOT EXISTS
        (select * from S where R.A = S.B)
fetch first 1 rows only
```

(b) Using not exists

Figure 3.3: Statements to test IND candidate $R.A \subseteq^? S.B$ utilizing an anti-join.

profile unknown data sources without any requirements to the database system or the database structure. We tested the above statements with indexes as they were provided by the original schemas and additionally with sorted indexes on every single attribute. All SQL approaches were much slower than the class of algorithms presented in the next section. Even worse, for large databases all SQL approaches were infeasible, because of their immense runtime. The fastest approach was using not exists statements. See Chapter 4.1.1 for our experimental results.

Complexity Analysis We repeat the complexity analysis in terms of comparisons of [42] and provide a proof assuming a sort merge join on both attributes with n attributes and maximally t values in each attribute. We do not assume any given sortation on the attribute's values. Note that an anti-join can be computed as a join that just outputs the non-joining tuples. Thus, our complexity analysis covers the join and anti-join approaches.

Theorem 1. *The time complexity to identify all unary* INDs *using one (anti-)join query per* IND *candidate in a database, i. e., the necessary number of comparisons, is* $O(n^2 t \log t)$ *assuming a sort merge join on both attributes.*

Proof. We need $t \log t$ comparisons to sort an attribute's values and $O(t)$ comparisons to merge two attributes. Thus, for the $n(n-1)$ IND candidates we need $O(n^2 t \log t + n^2 t) = O(n^2 t \log t)$ comparisons for the (anti-)join queries.

For the test utilizing a join query, we need additionally the numbers of (distinct) non-null values. These numbers can be queried once for each attribute, e. g., using a sort. Thus, we need $O(n \cdot t \log t)$ comparisons to query the numbers of (distinct) non-null values of all attributes.

Altogether, we need $O(n^2 t \log t)$ comparisons to test all IND candidates utilizing one (anti-)join query per IND candidate. □

Assuming an sorted index on each attribute reduces the complexity for sorting to a single sort per attribute, thus the time complexity reduces to $O(nt \log t + n^2 t)$.

The complexity in terms of I/O is additionally dependent on the internal memory size M given as number of items (attribute values in our case) and the block transfer size B, i. e., the number of items in one I/O block.

Theorem 2. *The number of I/O required to identify all unary* INDs *using one (anti-)join query per* IND *candidate in a database is* $O(n^2 \frac{t}{B} \log_{\frac{M}{B}} \frac{t}{B})$ *assuming a sort merge join on both attributes.*

Proof. We need $\frac{t}{B} \log_{\frac{M}{B}} \frac{t}{B}$ I/O to sort an attribute's values and $O(\frac{t}{B})$ I/O to merge two attributes [65]. Thus, for the $n(n-1)$ IND candidates we need $O(n^2 \frac{t}{B} \log_{\frac{M}{B}} \frac{t}{B})$ I/O operations for the (anti-)join queries.

For the test utilizing a join query, we need additionally the numbers of (distinct) non-null values. These numbers can be queried once for each attribute, e. g., using a sort. Thus, we need $O(n \cdot \frac{t}{B} \log_{\frac{M}{B}} \frac{t}{B})$ I/O operations to query all attributes.

Altogether, we need $O(n^2 \frac{t}{B} \log_{\frac{M}{B}} \frac{t}{B})$ I/O operations to test all IND candidates utilizing one (anti-)join query per IND candidate. □

With sorted indexes on each attribute the I/O complexity reduces to $O(n \frac{t}{B} \log_{\frac{M}{B}} \frac{t}{B} + \frac{n^2 t}{B})$, because each attribute must be sorted just once.

In case of an unsatisfied IND, the test of the IND candidate using an anti-join stops earlier and needs less I/O than the test using join, because of the fetch first n rows only clause: The number of necessary comparisons and I/O to merge both attributes reduces (to 1 in best case). Only for satisfied INDs $O(t)$ comparisons and $O(\frac{t}{B})$ I/O are necessary to merge both attributes. But in any case, this merge must be performed for the $O(n^2)$ IND candidates, i. e., the quadratic complexity in the number of attributes remains.

Note that all possible SQL approaches have one major disadvantage in common: Each single IND candidate is tested separately, which causes a quadratic time and I/O complexity in the number of attributes. That is why we introduce SPIDER in the next section. SPIDER's complexity does not depend on the number of IND candidates, but only depends on the number of attributes.

3.2 Single Pass Inclusion DEpendency Recognition (SPIDER)

There are two major ideas explaining the speed of SPIDER: (i) We abort the test of single IND candidates as soon as we find the first value violating the IND. This idea is described in Section 3.2.1. (ii) We test all IND candidates in parallel – reducing the complexity in I/O and in the number of comparisons. We describe the idea in Section 3.2.2 and the overall algorithm and its complexity in Section 3.2.3.

3.2.1 Basic test for single IND candidates

The basic test can be performed using the following procedure, which is a variation of a sort merge join: First, sort the value sets of all attributes using any common sort order. From this point on we have the choice to regard only distinct values. Second, iterate over the sorted value sets of each IND candidate $A \subseteq^? B$, starting from the smallest items using cursors. Let $\mathsf{dep} \in A$ be the current dependent value and $\mathsf{refs} \in B$ be the current referenced value of an IND candidate. There are three possible cases: (i) If $\mathsf{dep} = \mathsf{refs}$ move both cursors one position further, because the dependent value was found in the set of referenced values. (ii) If $\mathsf{dep} > \mathsf{refs}$ move only the referenced cursor, i. e., look for the current dependent value in the remaining referenced values. (iii) Otherwise, if $\mathsf{dep} < \mathsf{refs}$, dep is not included in the referenced value set and we can immediately stop the test for this candidate. A candidate satisfies an IND if all dependent values were found in the referenced value set.

The brute force algorithm directly implements this procedure and creates and tests all IND candidates sequentially, i. e., one by one. The sorted attribute value lists are stored as files on disk. Compared to a SQL join, the main advantages are (i) the early stop for unsatisfied INDs, as usually most IND candidates are unsatisfied and (ii) the single sort of all attribute's values (instead of once per IND if there are no sorted indexes on all single attributes). Accordingly, most tests stop after comparing only a few or even only a single value pair, while a SQL join always computes all unmatched values (or all matching values).

Theorem 3. *The time complexity to identify all unary* INDs *using the brute force algorithm, i.e., the necessary number of comparisons, is* $O(nt \log t + n^2 t)$.

Proof. We need $O(nt \log t)$ comparisons to sort all n attributes inside the database. Furthermore, we need $O(n^2 t)$ comparisons to test all $n(n-1)$ IND candidates. □

This complexity equals the complexity of the SQL approach with sorted indexes on all single attributes. But in contrast to the SQL approach, this worst case of t comparisons per IND candidate is rarely necessary on average, assuming that for most IND candidates only very few values must be compared before a violating value is found. But there are yet two disadvantages: First, each attribute's values are read as often as the attribute is part of an IND candidate, i. e., $2(n-1)$ times.

Theorem 4. *The number of I/Os required to identify all unary* INDs *using the brute force algorithm is* $O(n \cdot \frac{t}{B} \log_{\frac{M}{B}} \frac{t}{B} + n^2 \cdot \frac{t}{B})$.

Proof. We need $n \cdot \frac{t}{B} \log_{\frac{M}{B}} \frac{t}{B}$ I/O to sort the attribute's values [65], because each attribute is sorted just once. To test each of the $n(n-1)$ candidates we must read at most once all of the two attribute's values, i. e., $n^2 \cdot \frac{2t}{B}$ I/O. □

Second, the values of all attributes are compared pairwise for the IND tests. Note that the brute force approach is a lower bound for the performance of the SQL approach: Both approaches share the I/O cost of checking each IND candidate separately. Further, any SQL approach needs at least as many comparisons as brute force does to check a single IND candidate.

When testing all IND candidates in parallel we are able to combine all these tests. Thus, we reduce the quadratic complexity of I/Os and comparisons to linearithmic complexity, as we show next.

3.2.2 Parallel test for all IND candidates with SPIDER

We now show how the simple brute force approach can be improved considerably. We retain the idea of using sorted data sets allowing an early stop of execution. However, the new algorithm, SPIDER, eliminates the need to read data multiple times from disk. Instead, it creates and tests all IND candidates in parallel with a single read over the data. Further, it eliminates the need to compare attribute values pairwise by sorting all attribute's values as needed using a special data structure. This procedure greatly reduces the complexity in terms of I/O and of comparisons and thus the run time (by a factor of up to 10 for the tested data sets).

SPIDER first opens all sorted attribute's lists (stored as files on disk) and starts reading values using one cursor per attribute. The challenge is to decide when the cursor for each attribute can be moved: All potentially dependent attributes affect the point in time when the cursor of a referenced attribute can be moved. But also all referenced attributes control when the cursor of a dependent attribute can be moved. Finally, any particular attribute can be a referenced and dependent attribute simultaneously.

Consider the following attributes and their values: $A = \{1, 2, 3, 4\}$, $B = \{2, 4\}$, $C = \{1, 2, 4, 5\}$ (see Figure 3.4). In all, $n(n-1) = 6$ IND candidates have to be tested: $A \subseteq^? B, A \subseteq^? C, B \subseteq^? A, B \subseteq^? C, C \subseteq^? A$, and $C \subseteq^? B$. Only two out of these six IND candidates are in fact satisfied: $B \subseteq^? A$ and $B \subseteq^? C$. Let all cursors initially point to the first value of each attribute.

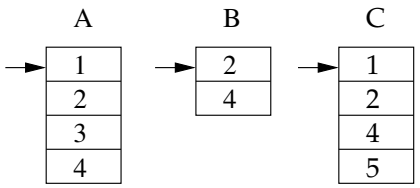


Figure 3.4: Three attributes with sorted data sets.

For example, to test $B \subseteq^? C$ the cursor in C has to be moved, because $2 > 1$. But before this movement the current value of C is needed to test $A \subseteq^? C$. Further, the current value in C is needed to test $C \subseteq^? A$ and $C \subseteq^? B$. Despite this mutual dependency, it is possible to synchronize the cursor movements without running into deadlocks or missing IND candidate tests, because we use *sorted* data sets. The main idea is to process attributes blockwise sorted by their values while maintaining all IND candidates covered by these attributes. E. g., in our example we first process all attributes (and their IND candidates) that include the value 1, then all attributes that include the value 2, and so on. Before formally describing the algorithm we give an intuitive idea of SPIDER using the example above.

SPIDER is based on a data structure that sorts all attribute's values and represents equal values grouped together. In Figure 3.5 the attributes are given as columns and same values are grouped in rows.

To represent the IND candidates we use the following observation: IND candidates can be grouped into disjoint sets by their dependent attribute, i. e., all IND candidates covering a given dependent attribute define a set. Thus, each attribute

attributes A,B,C			att and refs in SPIDER iteration steps			
				att	A.refs	B.refs	C.refs
A	B	C	Initialization:		B,C	A,C	A,B
1		1	Step 1:	A,C	C	A,C	A
2	2	2	Step 2:	A,B,C	C	A,C	A
3			Step 3:	A	∅	A,C	A
4	4	4	Step 4:	A,B,C	∅	A,C	A
		5	Step 5:	C	∅	A,C	∅

Figure 3.5: Attributes of Figure 3.4 represented in SPIDER's data structure: Attribute values are shown as columns; equal values are grouped in rows. IND candidates are given as lists 'refs' for each attribute.

A maintains in SPIDER's data structure a list $\mathsf{A.refs}$ of attributes that build a (still satisfied) IND candidate $A \subseteq^? B, B \in \mathsf{A.refs}$. In our running example these lists are initially set to $A.\mathsf{refs} = \{B, C\}$, $B.\mathsf{refs} = \{A, C\}$, and $C.\mathsf{refs} = \{A, B\}$.

SPIDER iterates row by row over the data structure shown in Figure 3.5 performing the following procedure:

1. Get the set att of all attributes that contain this row's value.
2. For all attributes in list att: Update their lists refs by intersecting refs and att.

In our running example this procedure yields the following results as visualized in Figure 3.5:

1. In the first iteration step for the data value '1' we have $\mathsf{att} = \{A, C\}$. Thus, the lists refs of A and C are updated as follows:

$$A.\mathsf{refs} := A.\mathsf{refs} \cap \mathsf{att} = \{B, C\} \cap \{A, C\} = \{C\}$$
$$C.\mathsf{refs} := C.\mathsf{refs} \cap \mathsf{att} = \{A, B\} \cap \{A, C\} = \{A\}$$

   Thus we now know the IND candidates $A \subseteq^? B$ and $C \subseteq^? B$ are unsatisfied, which is correct, because the value 1 of A and C is not contained in B.

2. In the second iteration step $\mathsf{att} = \{A, B, C\}$. This means the intersections with their lists refs do not change these lists, which is correct, because value 2 is contained in all attributes.

3. The next iteration step defines $\mathsf{att} = \{A\}$, because the value '3' appears only in A. Thus, we update $A.\mathsf{refs} := A.\mathsf{refs} \cap \{A\} = \emptyset$. This – correctly – means A is

not contained in any other attribute. Note that the values of A are still needed in the data structure, because A is still a potentially referenced attribute of attributes B and C.

4. The fourth step is analogous to step two, because all attributes are contained in att.

5. The last iteration step sets $C.\mathsf{refs} := C.\mathsf{refs} \cap \{C\} = \emptyset$, because $\mathsf{att} = \{C\}$.

After processing all values the data structure provides only the satisfied INDs given by all attribute's lists refs. In our running example $B \subseteq A$ and $B \subseteq C$ are satisfied INDs. There are no INDs with A or C as referenced attribute, because their lists refs are empty.

3.2.3 The SPIDER algorithm

The data structure of SPIDER represents all attributes and their values as well as the IND candidates. Further, the attributes are sorted by all values in all attributes. This structure can be achieved by the following two representations:

Each attribute is represented as an *attribute object* providing the attribute's sorted values and a cursor to the current value. As IND candidates can be divided into distinct sets by their dependent attribute, the IND candidates are represented as the list A.refs in each attribute object A, i. e., if $A \subseteq^? B$ then $B \in \mathsf{A.refs}$. Initially these sets are filled with all IND candidates that are to be tested. During a SPIDER run the attributes in A.refs are known to contain all previously viewed values of A and the currently viewed value of A, i.e., the IND yet holds.

Note that an attribute can be covered in multiple IND candidates as referenced attribute and / or as dependent attribute. The dependent role is represented as attribute object, the referenced role is represented in another attribute object's list refs.

The main idea of SPIDER is to process all attributes with equal values as a block. Thus, the data structure provides all attributes sorted by all values in all attributes. This sorting can be efficiently achieved, because the values in each individual attribute are already sorted. We hold all attribute objects in a min-heap sorted by their current values.

The SPIDER algorithm is given in Algorithm 1. SPIDER iterates blockwise over the heap data structure by (1) receiving the set att of attribute objects with the currently

minimal but equal value, and (2) updating the lists refs of attribute objects in att by intersecting refs and att. This way, all attributes B not containing the current value, i. e., $B \notin$ att, are discarded from the set of referenced attributes for attributes $A \in$ att. (3) If an attribute object A in att has a next value then the cursor is moved on and the attribute object is re-inserted into the min-heap. Otherwise, all INDs given by A and its list A.refs are proven satisfied.

Algorithm 1: SPIDER.

Input: attributes: set of attribute objects with their sorted values and their respective refs sets (the IND candidates)

Output: Set of satisfied INDs.

```
1  Min-Heap heap := new Min-Heap( attributes ) ;
2  while heap != ∅ do
       /* get attributes with equal minimal value                    */
3      att := heap.removeMinAttributes() ;
4      foreach A ∈ att do
           /* update list A.refs                                     */
5          A.refs := A.refs ∩ att ;
           /* process next value                                     */
6          if A has next value then
7              A.moveCursor() ;
8              heap.add(A) ;
9          else
10             foreach B ∈ A.refs do
11                 INDs := INDs ∪ { A ⊆ B } ;
12 return INDs
```

Theorem 5. *The time complexity to identify all unary* INDs *using* SPIDER*, i.e. the necessary number of comparisons, is* $O(nt \log t)$ *with* n *attributes and maximally* t *values in each attribute, assuming* $t > n$*.*

Proof. To sort all attribute's values we need $O(nt \log t)$ comparisons.

The cost to test the IND candidates is as follows: We need $O(\log n)$ comparisons to insert one attribute object into the heap depending on its currently viewed value,

and thus $O(nt \log n)$ to insert all attributes. To pop attributes from the heap we need $O(nt \log n)$ comparisons for the heap operations and $O(nt)$ comparisons for identifying the attributes in the minimum value set (Min). The list intersection of A.refs and att is $O(1)$ assuming both lists are represented as bit vectors of fixed size[1], i. e., we need $O(nt)$ operations for all needed intersections. Thus, the complexity of SPIDER to test the IND candidates (i. e., without sorting) is $O(nt \log n)$.

Assuming $t > n$, we need $O(nt \log t)$ comparisons for the complete execution of SPIDER, i. e., for sorting and testing. □

This analysis shows that the complexity to test the IND candidates is lower than the complexity to sort all attribute's values (if $t > n$). This is a considerable improvement over the SQL and brute force approaches, which require more effort for testing than for sorting.

Theorem 6. *The number of I/Os required to identify all* INDs *using* SPIDER *is* $O(n \cdot \frac{t}{B} \log_{\frac{M}{B}} \frac{t}{B})$.

Proof. We need $n \cdot \frac{t}{B} \log_{\frac{M}{B}} \frac{t}{B}$ I/O to sort the attribute's values [65], because each attribute is sorted just once. To test all IND candidates in parallel each value is read at most once, i.e., $n \cdot \frac{t}{B}$: Each attribute's values are represented in a single attribute object and each item in the list of its values is read at most once. Both tasks together result in $O(n \cdot \frac{t}{B} \log_{\frac{M}{B}} \frac{t}{B})$. □

Note that also in I/O testing all IND candidates is cheaper than sorting all attribute's values. In comparison to time complexity the cost of testing the candidates reduces even stronger: The time complexity to test all IND candidates is linearithmic in the number of attributes ($O(nt \log n)$), while the I/O complexity is only linear in the number of attributes ($O(n \cdot \frac{t}{B})$).

The experiments in Section 4.1.2 validate the statement that the complexity of SPIDER depends only on the number of attributes and the number of their values, but not on the number of tested IND candidates.

SPIDER uses a database to sort and "distinct" the values of all attributes, and then writes the sorted lists to disk. Thus, the total time of a run consists of sorting inside the database, shipping the sets to a client, writing them to disk, and reading

[1]Note that the complexity of operations at bit vectors is O(1) if the maximum number of items is constant, and O(MaxItems) if the bit vector must be implemented with variable size. As we don't need to vary the size during a SPIDER run, the complexity in our case is O(1). (see e.g. [24])

them in parallel for the tests. In Section 4.1.2 we analyze in detail which of these components dominate the runtime of the algorithm. Before that we evaluate in Sec. 3.3 several strategies to prune IND candidates.

3.3 Pruning IND candidates

In this section we discuss various strategies to prune IND candidates. The strategies are safe in that they do not prune candidates unless they are unsatisfied. For each strategy we evaluate the impact on various test sets. The pruning strategies are also applicable to other algorithms for IND detection (see Sections 4.1 and 3.4).

We also and particularly examine the exclusion of entire attributes from consideration. This is possible when all IND candidates that are covered by an attribute are excluded. Excluding attributes is particularly interesting for SPIDER, because its complexity directly depends on the number of involved attributes.

We note already here that we found that the efficiency improvement of pruning to be less favorable than one might expect on first glance. This effect was not mentioned by related work and also came as a surprise to us.

3.3.1 Simple strategies

A simple pruning strategy (called distinct in Table 3.1) is to compare the number of distinct values of each IND candidate: Let $v(A)$ denote the number of distinct values of attribute A. If $v(\mathrm{A}) > v(\mathrm{B})$ then there is at least one value in the attribute A that is not included in attribute B. Thus, the IND candidate $A \subseteq^? B$ can be excluded (but not $B \subseteq^? A$).

An equally simple test – suggested by [11] – is to compare the maximum and minimum values of attributes. An IND candidate can be excluded (i) if the maximum dependent value is greater than the maximum referenced value (we call this strategy max) or (ii) if the minimum dependent value is lower than the minimum referenced value (called min).

All three tests are inexpensive, because we can piggy-back the computation of distinct, max, and min to the procedure of sorting the attributes in the database and writing them to disk.

The selectivity of these filters on our test databases is shown in Tab. 3.1. In all tested databases, the number of IND candidates is reduced by at least a factor of 6

when combining all three tests. While these savings seem impressive we analyze in Sec. 4.1.3 how the actual runtime is affected.

	UniProt	TPC-H	PDB	
	900 MB	1.3 GB	2.8 GB	32 GB
# attributes	68	61	1,215	1,297
# IND candidates	1,393	877	219,106	245,562
# satisfied INDs	36	33	4,972	5,431
# attributes in INDs	31	20	448	478
distinct				
# IND candidates	910	477	139,807	157,818
# attributes in IND candidates	68	58	1,208	1,297
distinct & max				
# IND candidates	541	295	72,016	83,321
# attributes in IND candidates	59	54	997	1,080
distinct & min				
# IND candidates	345	275	61,920	68,664
# attributes in IND candidates	54	57	990	1,042
distinct & max & min				
# IND candidates	174	137	22,655	25,821
# attributes in IND candidates	49	52	670	709

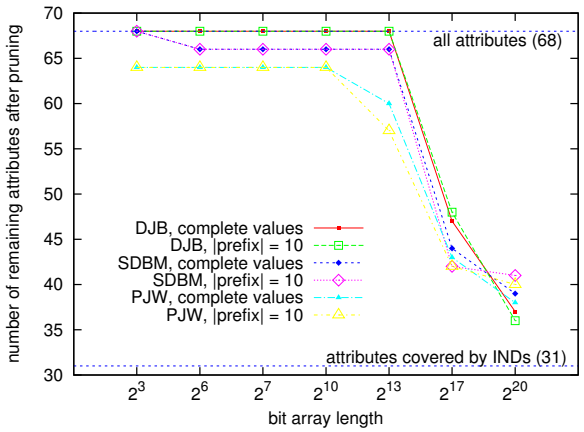
Table 3.1: Number of remaining IND candidates and attributes after pruning using distinct, max, and min.

3.3.2 Bloom filter

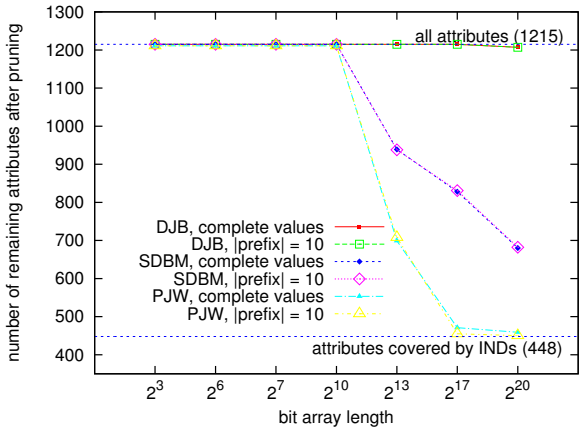
The simple filters described above use only very little information about the attributes. In particular, these filters are insensitive to the distribution of values between the minimum and maximum values. Bloom filters offer a better adaptation of the filter to the data [16]: We hash each attribute's values into a bit-array such that each bit represents several values. To prune IND candidates we compare the bit arrays of the two attributes looking for bits that are 1 in the dependent array, but 0 in the referenced array. If one such bit is found, the candidate is not satisfied; otherwise, we still need to test all values. The test can be performed efficiently by

a bitwise dep $\wedge \neg$ref operation, such that in the resulting array a bit is 1 iff the IND candidate can be excluded.

To achieve an optimal impact on filtering entire attributes, i. e., filtering all IND candidates that cover these attributes, we experimented with the size of the bit array and the hash function. Further, we examined whether hashing only prefixes of certain length instead of hashing complete attribute values affects filtering. We tested the PJW, DJB, and SDBM hash functions[2]. The results of our experiments regarding the selectivity of the Bloom filter are shown in Figure 3.6.



(a) UniProt



(b) PDB 2.8 GB

Figure 3.6: Impact of varying Bloom filter parameters on number of remaining attributes for (a) UniProt and (b) PDB.

[2]See General Purpose Hash Function Algorithms, `www.partow.net`

The DJB shows worst filtering impact on all tested databases. SDBM and PJW are comparable on UniProt, but PJW prunes clearly better on PDB data.

The results confirm the intuition that increasing the size of the bit array leads to a higher amount of pruned IND candidates and attributes due to an improved spread of values. On the other hand, large arrays require more memory and more time for their comparison. The experiments show that a length of 2^{17} bit (which requires 20 MB memory for 1000 attributes) already is very effective for the PDB with respect to pruned attributes and also has very good effects on UniProt (see Fig. 3.6). Longer arrays improve pruning on UniProt only marginally. Although these results highly depend on the data sets, they show that reasonable reductions can be achieved with modest memory consumption.

To reduce filter creation costs we tested the idea not to hash entire values into the bit array but just prefixes of a certain length. The hash value of a small prefix can be computed much faster and we already expect high selectivity in the first few characters. We tested on prefix lengths between 1 and 10. Interestingly, hashing prefixes of fixed length already leads to impressive results on very small bit arrays. Larger hashed prefixes result in larger numbers of pruned IND candidates and attributes – as one would expect. However, we found that hashing prefixes of length 10 already behaves nearly identical to hashing the complete values with regard to filtering attributes; in UniProt data it is even slightly better than hashing complete values due to the different distribution of the shorter values to hash buckets (see Fig. 3.6).

In the context of integration of unknown data sources one cannot determine "the" optimal setting of parameters without a detailed (and costly) analysis. Nevertheless, hashing prefixes of length 10 into a 2^{17} bit array with the PJW hash function seems to be the combination that covers all tested databases best. The filter results for TPC-H and the entire PDB confirm this statement (see Table 3.2).

The effects of combining Bloom filters (parameters as above) with the simple pruning strategies are shown in Table 3.2. The Bloom filter is in almost all cases more selective than the filter on number of distinct items, maximum, and minimum (compare to Tab. 3.1, last line). But the simple filters catch some IND candidates that are not pruned by the Bloom filter. Therefore, all filters together reach the best performance.

	UniProt	TPC-H	PDB	
	900 MB	1.3 GB	2.8 GB	32 GB
# attributes	68	61	1,215	1,297
# IND candidates	1,393	877	219,106	245,562
# INDs	36	33	4,972	5,431
# attributes in INDs	31	20	448	478
distinct & bloom				
# IND candidates	245	43	10,462	12,130
# attributes in IND candidates	42	26	454	587
dist., max, min, bloom				
# IND candidates	125	40	9,006	10,149
# attributes in IND candidates	35	25	450	518

Table 3.2: Number of remaining IND candidates and attributes using Bloom filter and simple pruning.

3.3.3 Filtering and Performance

The overall runtime of SPIDER is composed of the costs of sorting data, reading it from the database, writing it to disk, and reading it again for the IND tests. Therefore, filters are most useful if they can be applied within the database, thus reducing the amount of data to be shipped to and considered by a client outside the DBMS. However, applying the filters inside the database also costs time. For instance, to obtain the minimum and maximum value and the number of different values for an attribute the database must read the entire bag of values of this attribute – a read that is repeated later for all attributes that cannot be excluded completely. Note that in general it is impossible to force a SQL database to save the previously "distinct" attribute's values for reuse. For Bloom filters expensive computations, for which database programming languages, such as Transact-SQL or PL/SQL, are not well prepared (array manipulation, XOR operations), must take place that might outweigh the simple read-and-compare style of the SPIDER algorithm. On the other hand, applying the filters while we read the sorted columns from the database comes at almost no additional cost, as by then data shipping has already taken place. Thus, it is not at all clear where filters should be applied. We analyze this trade-off in Section 4.1.3.

3.4 Related Work

Kantola et al. [42] give a complexity estimation of discovering all unary INDs. They propose to prune the IND candidates by data type. They further argue that the checks for most IND candidates should terminate quickly as only few inclusion dependencies hold for data sources, but do not propose any approach. We confirm this assumption by our brute force approach as we show in Sec. 4.1, but also show that separately testing all unary IND candidates is infeasible for large schemas.

Bell and Brockhausen [10, 11] propose to create all unary IND candidates and test them sequentially by utilizing an SQL join statement. The tested (satisfied and not satisfied) INDs are used to exclude further tests and therefore to reduce the number of IND candidates to test. Furthermore, the number of IND candidates is reduced by constraints on the data types and maximal and minimal values. The SQL join statement performs a join on the attributes A and B of the IND candidate and compares the number of returned distinct values in A to the number of distinct values in A and B therefore verifying $A \subseteq^? B$ and $B \subseteq^? A$. We use a similar join statement in our join approach in Sec. 3.1. SPIDER considerably outperforms this approach as we show in Sec. 4.1.2.

Marchi et al. [52, 53] detect unary INDs among same data types by preprocessing all data and then testing all IND candidates in parallel. The preprocessing assigns to each value in the database the list of attributes that include this value. This step is very costly, because all values in all attributes must be combined into one data structure. We show in Sec. 4.1.2 that SPIDER outperforms this approach by orders of magnitude.

Koeller [43] aims to discover composite INDs between two relations. As prerequisite he tests unary and binary INDs using a minus approach. This approach equals our except approach as minus is the Oracle syntax for the Standard SQL except. In his environment the minus approach runs faster than the join approach. We cannot repeat these results in our differing environment. As we show in Sec. 4.1 in our environment join outperforms except. Note that Koeller runs experiments only on relations with $6,000$ to $200,000$ tuples, compared to our data sources with up to approximately $219 * 10^6$ tuples.

Dasu et al. [25] apply data summaries to detect join paths approximately, i. e., to detect INDs. They use set resemblance and multiset resemblance – measures of sim-

ilarity between two (multi-)sets estimated by using their hash (multi-)set signatures – to find a join path, its size, and its direction. The authors use this approach as a first approximate step in schema discovery to help a human expert. In our scenario we want to avoid false positive and false negative INDs. Our re-implementation of this approach did not produce such exact results.

Brown and Haas [20] study algebraic constraints between pairs of attributes to use them in query optimization. They create IND candidates by heuristics and data samples and might therefore miss some INDs. Our aim is explicitly to find all INDs. SQL join statements are used to test the IND candidates.

Finally, Petit et al. [58] extract IND candidates from existing applications on a database by analyzing a workload searching for frequently used equi-joins. These joins are then tested against the database and rated by a human expert. Again, this is a heuristic approach that might miss INDs.

Chapter 4

Evaluating and Leveraging Unary Inclusion Dependency Discovery

In this chapter we evaluate the discovery of unary INDs using our life sciences data sources and the generated TPC-H instance presented in Section 2.3. We first evaluate the algorithms regarding efficiency. Effectiveness is evaluated by their use in discovering intra-schema and inter-schema relationships.

4.1 Efficiency of Unary IND Discovery

For our efficiency experiments on unary IND discovery we run two different set-ups. Our complexity analysis shows that the proposed SQL approaches as well as the brute force approach depend on the number of IND candidates. Thus, we decided to vary this parameter:

- Restricted IND candidates: In the first setting we create only IND candidates that have a referenced attribute with only unique non-null values (a potential primary key). This setting is useful for detecting unary foreign key relationships (through INDs).

- Exhaustive IND candidates: The second setting removes this restriction, i. e., we test all pairs of attributes.

We exclude candidates with both attributes from the same table (intra-table references) and candidates with attributes of data type Large Object (LOB) in both settings. All tests were performed including the distinct filter. The runtime effects of the other filters were tested individually in Section 4.1.3.

We tested all algorithms using a Linux system with 2 Intel Xeon processors (2.80 GHz) and 12 GB RAM running a commercial object-relational database management system.

4.1.1 Evaluating the SQL approaches

We run the experiments for the proposed SQL approaches only on the restricted IND candidates. As we show, this set-up already reveals the limits of the SQL approaches.

Conforming to the set-up, we used the refined, less costly join statement assuming a unique referenced attribute. Further, we applied the statements to match the syntax of our used DBMS.

We measured the required time for computing all unary INDs for each of the four methods on our life science data sets CATH, SCOP, UniProt, and PDB, and on the generated TPC-H data set. The measured times together with the numbers of attributes, IND candidates, and satisfied INDs are given in Tab. 4.1: We first discuss the experiments on our smaller data sets CATH, SCOP, UniProt, and TPC-H. The join approach delivers good results, but is quite slow on the TPC-H data. The except approach shows in all cases the best runtime. The two versions for anti-join differ substantially in their runtime: The not exists approach is in all cases the fastest approach. The not in approach is equally fast only on the CATH and TPC-H data sets, but in orders of magnitude slower than the not exists approach for SCOP and UniProt. Note that the evaluation of not in and not exists statements differs even between different DBMS versions [61], and most probably also between different DBMS. However, in our test environment the not exists approach is the fastest alternative. As the join approach showed also good results, and as the evaluation of join queries should be the best optimized operation in most DBMS, we decided to use the not exists and join approaches for further experiments.

All four approaches are not applicable to discover all INDs in a database of the size of the PDB. We first ran tests on the entire PDB, but stopped after two days because the RDBMS estimated a particular table-scan to last several more days. We then eliminated the 9 biggest PDB tables, containing atom coordinates for each

	CATH	SCOP	UniProt	TPC-H	PDB
DB size	20 MB	17,5 MB	900 MB	1.3 GB	2.8 GB
# attributes	25	22	68	61	1,215
# IND candidates	68	43	910	477	139,807
# INDs	0	11	36	33	4,972
join	6 s	7 s	9 m 04 s	25 m 02 s	16 h 14 m
except	15 m 27 s	16 m 05 s	27 m 35 s	1 h 09 m	–
not in	5 s	52 m 11 s	6 h 33 m	7 m 45 s	–
not exists	5 s	6 s	3 m 57 s	7 m 51 s	10 h 20 m

Table 4.1: Runtime performance of the SQL approaches. IND candidates are restricted to cover unique referenced attributes. We used only a fraction of PDB.

atom in each protein, and thus reduced the database size from 32 GB to 2.8 GB. The discovery procedure using not exists finished within 10 h20 m for this reduced data set, the join approach within 16 h14 m.

In an additional experiment we built sorted indexes on any attribute in UniProt to support two tasks of the query execution: (i) to sort each attribute only once instead for every tested IND candidate, and (ii) to support the execution of each single IND test. The saved time in testing the IND candidates was consumed by the time to generate the indexes. Note that additionally this approach largely increases the necessary storage for each data set, and is therefore not applicable to large data sources.

The problems with using SQL for set inclusion are twofold. First, one cannot tell the optimizer what the real question is, and that, as a consequence, there are optimization strategies that are clearly better than those used for "ordinary" SQL statements. Further, the aimed early stop after the first violating value in the not exists, not in, and except approaches depends on the ability of the DBMS to use the fetch first clause for an early abort of the execution. Second, we have to run a single SQL statement for each IND candidate. Thus, the attribute's values are read and compared repeatedly for each IND candidate.

4.1.2 Evaluating SPIDER

We evaluate SPIDER on our larger data sources UniProt, TPC-H, and PDB as these sources show a potential for improvement in IND discovery. We first evaluate restricted and exhaustive IND candidates. Afterwards, we compare to related work that discovers exact INDs.

Restricted IND candidates Experimental runtime results are shown in Table 4.2. We directly compare SPIDER to the fastest SQL approaches (join and not exists), to the brute force algorithm, and to two re-implemented approaches from related work – Bell and Brockhausen [11] and Marchi et al. [53] (described later in this section). Both brute force and SPIDER greatly outperform the SQL approaches, which are infeasible for large schemas.

For low numbers of IND candidates, i.e., UniProt and TPC-H, there is only a small difference between the brute force and the SPIDER algorithm. For large schemas with high numbers of IND candidates the improvement of SPIDER over brute force is considerable.

In our example databases, the number of satisfied INDs depends on the number of attributes. It does not depend on the database size as can be seen when comparing the number of satisfied INDs in TPC-H and the 2.8 GB part of PDB. We expect this behavior for most databases.

Brute force and SPIDER share the same cost (and time) to sort and "distinct" the data inside the database, to ship them to the client, and to write them to the file system. The real difference is the IND candidate test. The shared overhead for UniProt and TPC-H is 1 m 32 s and 6 m 15 s, respectively. Thus, the IND candidate test speeds up for these small schemas by factor of 2 for UniProt and by factor 1.5 for TPC-H. For PDB the overhead is 21 m for the 2.8 GB part and 5 h 56 m for the complete PDB. Thus, in both cases the speed-up for testing IND candidates is 75-fold.

To test our algorithm on very large databases with enormous schemas, we use a SAP/R3 database instance (see Sec. 2.3), which populates 25,002 non-empty tables with 237,836 attributes. The total size is 145 GB. We had to reduce the number of used attributes due to (i) the allowed number of open files on our system and (ii) main memory constraints of the Java virtual machine, which made it impossible to open more than $\sim$ 25,000 attribute files at once. These problems could be

	UniProt	TPC-H	PDB	
DB size	900 MB	1.3 GB	2.8 GB	32 GB
# attributes	68	61	1,215	1,297
# IND candidates	910	477	139,807	157,818
# INDs	36	33	4,972	5,431
Best SQL approaches				
join	9 m 04 s	25 m 02 s	16 h 14 m	–
not exists	3 m 57 s	7 m 51 s	10 h 20 m	–
Our approaches				
Brute Force	2 m 11 s	6 m 30 s	3 h 29 m	19 h 51 m
SPIDER	1 m 51 s	6 m 25 s	23 m 36 s	6 h 07 m
Re-implementations of related work				
Bell & Brockhausen [11]	4 m 39 s	–	1 h 32 m	–
Marchi et al. [53]	9 h 58 m	–	–	–

Table 4.2: Run-time performance of the algorithms. IND candidates are restricted to only cover unique referenced attributes.

circumvented by using partitioning techniques on the candidate sets, but we have not explored this direction. Instead, we applied a filter to the original 237,972 attributes in the SAP/R3 database: We considered only attributes with at least 50 distinct values (aiming to exclude those attributes that cause only low costs in the IND test). This restriction resulted in about 230,000 IND candidates covering 22,887 attributes. The size of this excerpt is about 40 GB. We identified 3,927 INDs within 2 h 38 m. These measurements show that SPIDER is well capable of handling very large databases with a large number of IND candidates. Further, together with our results on the complete PDB (32 GB, tested within $\sim$ 6 hours) it shows that the runtime depends not directly on the size of the database, but mostly on the number and size of distinct values in the database.

Exhaustive IND candidates The results of exhaustively testing all IND candidates using SPIDER on our various test data sets are given in Table 4.3. In comparison to the restricted test (Table 4.2; repeated for readability in Tab. 4.3), the runtime increases slightly and more satisfied INDs are obtained.

The amount of work to sort the data inside the DBMS, to ship them to the client

	UniProt	TPC-H	PDB	
DB size	900 MB	1.3 GB	2.8 GB	32 GB
restricted IND candidates				
# IND candidates	910	477	139,807	157,818
# INDs	36	33	4,972	5,431
SPIDER	1 m 51 s	6 m 25 s	23 m 36 s	6 h 07 m
unrestricted IND candidates				
# IND candidates	2,235	1,616	734,851	837,358
# INDs	116	86	35,752	40,415
SPIDER	2 m 07 s	6 m 35 s	24 m 34 s	6 h 10 m

Table 4.3: Experimental results of testing IND candidates restricted to cover unique referenced attributes (repeated from Tab. 4.2) and unrestricted IND candidates using SPIDER.

and to write them to files is identical for both set-ups. The difference is the test of IND candidates: The increase of satisfied INDs implies that more values have to be handled, because attributes cannot be excluded from the SPIDER heap as early. Thus, the runtime increases. But note that this is only a slightly increase as opposed to the increase in the number of IND candidates and in the number of satisfied INDs. For instance, for the entire PDB the number of IND candidates increases by factor 5.3, the number of satisfied INDs even by factor 7.4, but the runtime increases only by factor 1.008. These results confirm our complexity analysis stating that SPIDER is independent of the number of IND candidates, but only dependent on the number of attributes.

Comparison to other approaches We compared our algorithms with the approaches of Bell and Brockhausen [11] and Marchi et al. [53] using our own, careful re-implementation (see Table 4.2). We made one adaptation to both algorithms: As a simple filtering, both methods test only candidates of the same data type. In our life sciences setting, we usually find schemas with only `string` attributes or no type definition at all, and thus must test all pairs of attributes. For a fair comparison we assigned the same data types to all attributes; obviously, it would be very simple to extend SPIDER to also filter for data types.

The algorithm of Bell and Brockhausen leverages filters on maxima and minima

to prune IND candidates and utilizes already finished IND tests for further pruning, thus exploiting the transitivity of the IND relationship. IND candidates are tested by SQL join statements. It runs 4 m 39 s on UniProt data and 1 h 32 m on the smaller part of the PDB, which is three times slower than SPIDER. Furthermore, the runtime of this approach strongly depends on the number of IND candidates – as opposed to SPIDER. To show this dependency, we run an experiment exhaustive IND candidates. The analysis of the UniProt data set (2,235 IND candidates) did not stop within 5 hours and the analysis of the 2.8 GB part of PDB (734,851 IND candidates) did not finish within 19 hours. SPIDER tests these sets within $\sim$ 2 m and $\sim$ 25 m, respectively (see also Tab. 4.3).

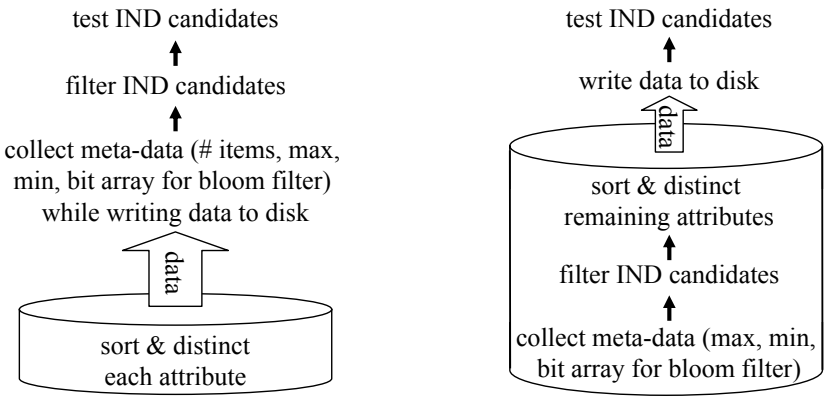
The approach of Marchi et al. preprocesses all data by assigning to each value in the database all attribute's names that contain this value. The results are stored in a table. Afterwards, all IND candidates are tested in parallel exploiting the sets of attribute names. We implemented the preprocessing as a PL/SQL script and the test as a Java program. The preprocessing on UniProt already consumes 9 h 58 m, the actual IND test only about four minutes, compared to the total run time of SPIDER of 1 m 51 s. Note that Marchi et al. tested their approach only on small databases with an in-memory implementation; their largest database comes in a 77 MB dump file, compared to our smallest database (UniProt) with 900 MB.

4.1.3 Evaluating the Effects of Pruning

Table 4.4 shows run times when we apply filters before running SPIDER. For the experiment, we read all data out of the database and wrote it to disk. After this step we filtered the IND candidates and applied the SPIDER test to the remaining IND candidates (see Fig. 4.1 a). In the next paragraph we examine the alternative of filtering within the database (Fig. 4.1 b).

Database external pruning In Table 4.4, we distinguish two components: (i) the time to sort and read (inside the RDBMS), ship (to the client), and write the data to files ("*SRSW*"), and (ii) the time to read and test the data at the client ("*test*").

As expected, the runtime for reading the data and writing them to disk increases slightly when applying the filters, because the bit array for the Bloom filter has to be computed. However, it is on first sight surprising that, despite the great reduction in IND candidates to be tested, the runtime for testing decreases only minimally.



(a) Database-external pruning. (b) Database-internal pruning.

Figure 4.1: Steps required for database-internal and database-external pruning.

	UniProt	TPC-H	PDB	
DB size	900 MB	1.3 GB	2.8 GB	32 GB
distinct	1 m 51 s	6 m 25 s	23 m 36 s	6 h 07 m
SRSW	1 m 32 s	6 m 15 s	20 m 59 s	5 h 56 m
test	17 s	8 s	2 m 23 s	11 m 21 s
distinct & max & min & bloom	1 m 55 s	6 m 53 s	24 m 06 s	6 h 18 m
SRSW	1 m 34 s	6 m 33 s	21 m 42 s	6 h 08 m
test	20 s	7 s	2 m 15 s	9 m 24 s

Table 4.4: SPIDER with and without filtering of IND candidates. IND candidates are restricted to cover unique referenced attributes.

This observation reconfirms our claim from Section 3.2 that the runtime of SPIDER is independent of the number of IND candidates, i. e., representing and tracking the IND candidates incurs almost no cost.

It is more surprising that even the exclusion of entire attributes does not yield much speed-up – even though the complexity of SPIDER depends on the number of attributes. This observation can be explained as follows: Most of the time for testing is required by satisfied INDs, because in those cases the entire value sets need to be read and compared until the end – there is no early stopping. However, the filtering removes only candidates that are certainly not satisfied and which would require only a few comparisons by SPIDER anyway. Thus, filtering on average removes only candidates that would not incur much effort, anyway.

Together, excluding unsatisfied INDs by filtering outside the database does not save much time, but costs time for computing the filters. The next paragraph evaluates possible approaches of filtering IND candidates (and with it attributes) inside the DBMS. This way, data of attributes not covered by IND candidates do not have to be shipped out of the database incurring high I/O cost.

Database internal pruning Given the observations described in the previous paragraph, it is clear that further optimization of SPIDER should concentrate on the SRSW phase (sorting, reading, shipping, and writing into filesystem). Of the 24 minutes to test the 2.8 GB part of PDB, approx. 21 minutes are spent in the SRSW phase and less than 3 minutes in the test phase (see Tab. 4.4). Thus, we strived to reduce the number and complexity of queries inside the database and/or reduce the amount of data to be shipped to the client by applying filtering already inside the DBMS.

Note that we consciously decided not to use information on maxima, minima, and the number of distinct values supplied by the database statistics. We could demand up-to-date statistics, but this requirement would just mask the necessary costs. Further, reading those values from the database catalogue opens the door to incorrect results due to outdated statistics.

Reaching the goal of reducing the costs in the SRSW phase is not as straightforward as one might think: The 21 minutes of the SRSW phase split down to $\frac{2}{3}$ for reading and sorting the data inside the database, and $\frac{1}{3}$ for reading and shipping them out of the database and writing them to disk. Thus, any filtering that requires to scan or sort all values, such as min, max, or distinct, very likely does not improve the overall performance, because the time saved during shipping and testing is dominated by the time required to scan/sort the data twice.

Another procedure for efficient filtering inside the database is to read all tables once and collect the needed metadata (maximum, minimum, and the Bloom filter bit array) for all their attributes on the fly (instead of reading them attribute-by-attribute). The reading operations can be done efficiently by a full table scan. After applying the filter only the remaining attributes must be sorted and further processed. Note that the number of distinct values cannot be collected by this approach (see Fig. 4.1 b).

There are two problems with this idea: (i) *All* values have to be processed to

collect the needed metadata instead of only distinct values. The ratio of distinct values to all values of the complete database varies for our test databases from 4 % to 24 %, i. e., at least 4 times as many items have to be processed for collecting metadata. (ii) The interface between the DBMS and the metadata collector must be fast enough to process this amount of data. We tested two setups: a Java interface and a dynamic PL/SQL interface. Both implementations showed that already reading the data over the interface (without any further processing) takes more time than sorting the attributes, shipping the sorted "distinct" values out of the database, and writing them to disk. If this overhead were eliminated by a faster interface (using for instance a natively implemented method inside the DB kernel), the overall performance of SPIDER should speed up greatly because of the large number of pruned attributes (as shown in Sec. 3.3).

4.2 Leveraging *Intra*-Schema INDs

In this section we evaluate the identified INDs used directly as foreign keys (Sec. 4.2.1 and propose and evaluate heuristics to filter foreign keys from INDs to improve the precision of foreign key discovery (Sec. 4.2.2). Further, we use the identified INDs to identify the "primary relation" of a data source – a domain specific characteristic of life sciences data sources (Sec. 4.2.3).

4.2.1 Effectiveness for Real World Data

We now evaluate the quality of detected INDs for one of their ultimate purposes, their indication of foreign keys. We evaluate UniProt and TPC-H, because their schema definitions come along with foreign keys and thus provide a gold standard.

The BioSQL schema, into which we parsed UniProt, defines 21 foreign keys. Of those we find 19 as INDs. The remaining two constraints are defined on empty tables and thus cannot be discovered by any instance-based approach. Another 11 INDs found by SPIDER are not defined as foreign keys in BioSQL but provide an interesting insight: They result from situations where there are two foreign key attributes A, B in different tables referencing a primary key attribute C. In addition to the defined FKs $A \subseteq C$ and $B \subseteq C$, SPIDER also detects the IND $A \subseteq B$ and sometimes additionally $B \subseteq A$ (i. e., $A = B$). SPIDER found three INDs in 1:1 relationships where only one direction was defined as foreign key constraint. Note

that all these "false positive" INDs actually are semantically correct and constitute helpful metadata to understand unknown schemas. They are omitted from the database definitions only for technical reasons. For instance, if a 1:1 relationship is defined by two foreign keys, then systems without the ability to defer constraint checking cannot allow the insertion of tuples.

Further, SPIDER detected three false positives that each relate a dependent attribute with only a single distinct value to a referenced attribute with about 10,000 distinct values. Altogether, these results imply a recall of 90% and a precision of $\sim 92\%$ for the detection of unary foreign key constraints in this particular example. Filtering already at the unary level is very important when it comes to composite INDs, as shown in Section 5.1. For instance, the BioSQL schema defines no composite keys. All the 13 detected composite INDs derive from the three false unary INDs.

For SCOP we found 11 INDs of which 4 are semantically correct (precision $\sim 36\%$). The other INDs base on a key built from numerical values only. But nevertheless, we also aimed at using the INDs for detecting the primary relation of a data source. This task is very well supported by the detected INDs as we will see in Section 4.2.3.

TPC-H defines seven unary foreign keys and one binary foreign key, which were all found (recall 100%). We further detected 24 INDs that are false positives (precision $\sim 23\%$). They all derive from the fact that the database uses only surrogate keys generated using a counter starting from the same initial number '1'. Thus, we often detect INDs either between two surrogate keys or between a numeric attribute and a surrogate key. Such cases, i. e., artificial keys generated by sequences ranging over the same namespace in all tables, are an apparent problem for all instance-based metadata discovery methods.

Overall, this evaluation shows that not all detected INDs represent foreign keys. The precision of simply taking every IND as foreign key would be $\sim 92\%$ for UniProt, $\sim 18\%$ for SCOP, and $\sim 23\%$ for TPC-H. Thus, we show in the next section methods to filter foreign keys from discovered INDs.

The OpenMMS schema—into which we imported the PDB data—does not define any foreign keys. On the one hand this is a good example for the necessity of identifying foreign keys, it is on the other hand difficult to verify the identified satisfied INDs. As the OpenMMS schema is very large, we could not perform a

systematic test. However, we observed that the OpenMMS schema often utilizes surrogate IDs, i.e., semantic-free integers whose ranges all begin at 1, as primary keys. This is a case where INDs fail to identify foreign keys. There are INDs between almost all of these ID attributes, leading to the observed 5,431 satisfied INDs with a unique referenced attribute. We applied the two following simple heuristics to prune probably uninteresting, yet satisfied INDs: (i) The referenced attribute must have more than one value; and (ii) At least 1 % of all distinct values of the referenced attribute must be covered by values of the dependent attribute. Both restrictions are very weak and should not exclude interesting possible foreign keys, but they reduced the number of INDs to 2,480. If we roughly estimate that each of the 116 tables in this schema defines one foreign key constraint, we see that further heuristics are necessary to support the step from INDs to foreign keys.

In the next section we introduce such methods to derive FKs from INDs and evaluate our results on PDB.

4.2.2 Deriving Foreign Keys

In this section we propose several heuristics to filter foreign keys from INDs. These heuristics are based on typical characterizations of foreign keys; we give an intuition for each heuristic. We combine the heuristics with weights to evaluate INDs. These results are based on work in [3]. In joint work with Rostin et al. [60] we extend this approach: we treat the heuristics as features and use machine learning to select useful heuristics and to combine features to evaluate INDs. We briefly report on those results.

Data Sources We use seven data sources for evaluation: three of our life science data sources (namely SCOP, UniProt, and PDB), three movie data sources (Movielens[1], the German data source Filmdienst[2], and the Internet Movie Database IMDB[3]), and our generated TPC-H instance with scale factor one. Table 4.5 lists the details on size, number of INDs, and number of foreign keys of each data source.

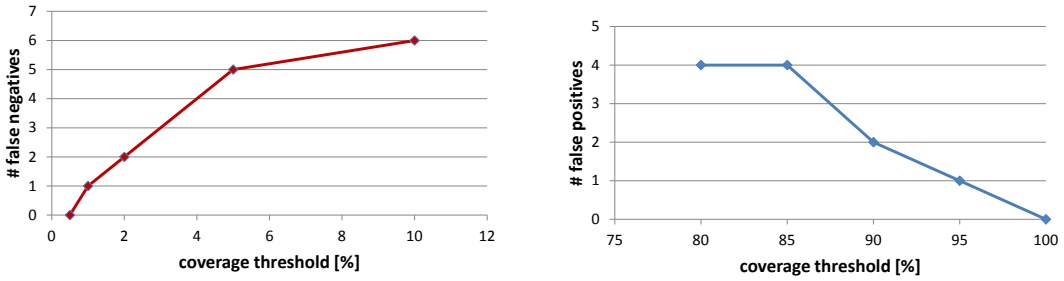
[1] www.movielens.org
[2] film-dienst.kim-info.de
[3] www.imdb.com

Data source	size	# tables	# attributes	# INDs	# FKs
SCOP	17,5 MB	4	22	11	5
UniProt	900 MB	16	68	36	33
PDB	32 GB	116	1,301	5,431	unknown
Movielens	73 MB	7	20	19	6
Filmdienst	135 MB	14	83	79	15
IMDB	776 MB	22	76	34	13
TPC-H	1,3 GB	8	61	33	9

Table 4.5: Data sources for evaluation of deriving foreign keys from INDs.

Heuristics The first characteristic of foreign keys regards the *coverage* of the referenced key's values. In many situations a foreign key attaches information to all values in the primary key.

We use this observation for two heuristics: The large coverage heuristic confirms INDs as foreign keys if more than 95 % of the referenced values are covered by dependent values. On the other hand, the small coverage heuristic rejects INDs when less than 1 % of the referenced values are covered by dependent values. Figure 4.2 shows the number of misclassifications for differing coverage thresholds for all evaluation data sources except PDB that led us to our choice of thresholds.



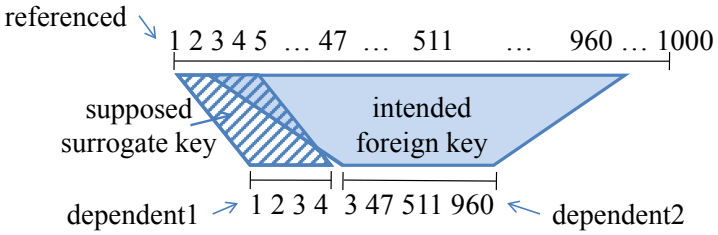
(a) small coverage (b) large coverage

Figure 4.2: Effects of different thresholds for heuristics small and large coverage over all evaluation data sources except PDB.

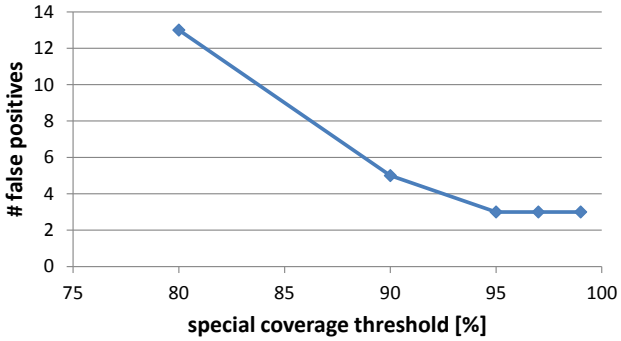
Of course, there are also frequent exceptions from this coverage rule, such as a small set of premium customers relating to a huge set of customers. That is why we set the threshold for rejection conservatively.

Real INDs that intend to relate a small subset to a larger set lead us to the second characteristic of foreign keys: The observation is that these intended subsets

distribute over the entire set instead of covering a special data range, as surrogate keys might do. Figure 4.3a illustrates this difference between foreign keys and surrogate keys showing two dependent attributes referencing one referenced attribute with different value distributions.



(a) intuition



(b) effects of different thresholds

Figure 4.3: Special coverage heuristic: (a) Intuitive difference between an intended subset and a surrogate key. (b) Effects of different thresholds over all evaluation data sources except PDB.

The special coverage heuristic considers the set of referenced values that is spanned by the minimal and maximal dependent value. Based on our results for our evaluation data sources in Fig. 4.3b we confirm an IND as a foreign key if this spanned set covers more than 95 % of the referenced values, and reject it otherwise.

The third characteristic regards the *number of distinct dependent values*: Usually a considerable number of different values is used in the foreign key to reference the key. This observation is backed by Koeller et al. [45] showing that IND candidates with fewer than six or seven different dependent values have a high chance to be satisfied only by statistical coincidence. We choose a more conservative minimum threshold to avoid exclusion of true foreign keys.

The number of values heuristic rejects an IND as a foreign key if there are fewer

than three distinct dependent values. This heuristic aims at rejecting binary attributes (such as 0 / 1 or male / female) relating another attribute.

Many schema designers use similar attribute names for keys and foreign keys. That is why our column name heuristic supports INDs with a dependent attribute name that equals or contains the referenced attribute name or the referenced relation name.

We considered three more heuristics, but discarded them during our experiments as they did not provide valuable information that were not covered by other heuristics. However, we introduce them here as they are later used as feature in [60]. The multi dependent heuristic is based on the observation that a *single foreign key references only one key*. Thus we reject an IND if the dependent attribute references several referenced attributes. Another observation for foreign keys is a *similar length of values* in the dependent and the referenced attributes, because they share in part the same values. This is in fact a simplification of the coverage heuristics. The last observation is that there are database schemas using controlled vocabulary via "category relations", i. e., assigning an id (mostly an integer) to each named category and using this id instead of the category name as value in the data set. Then there are semantically meaningful foreign keys between the usages of the described id and the category relation. But there are also many INDs between such category tables without any semantic relationship. To reject such INDs between category relations our small tables heuristic rejects INDs between relations with only few tuples.

Combining Heuristics Using Weights We assign weights to each heuristic to control their contribution on the overall evaluation of an IND. Note that most of our heuristics can only confirm an IND, but not reject it (e. g., large coverage), or it can reject an IND, but not confirm it (e. g., small coverage). That means, only a subset of our heuristics (that differs for single INDs) can contribute to the evaluation. To regard this aspect, we sum up two amounts to evaluate an IND: (i) the weights for all contributing, i. e., confirming or rejecting, heuristics as *base* and (i) the weights of all heuristics confirming the IND as *affirmation*. The ratio of affirmation to base is used to evaluate the IND.

We chose experimentally the following weights: As the number of values heuristic is very reliable, we assign the highest weight of 100, which cannot be overruled by the other heuristics. As the special coverage heuristic complements the small and

Data source	# INDs	# FKs	# true positives	# false positives	# false negatives
SCOP	11	5	5	0	0
UniProt	36	33	29	0	4
Movielens	19	6	6	0	0
Filmdienst	79	15	12	1	3
IMDB	34	14	14	1	0
TPC-H	33	9	9	1	0
Overall	212	82	75	3	7

Table 4.6: Effectiveness of combined heuristics for the evaluation data sources.

large coverage heuristics we assign all three of them a weight of 50. Finally, we assign a weight of 30 to the column name heuristic. An IND is evaluated as FK if the ratio of affirmation to base is 100 %.

Evaluation We evaluate our data sources using the combined heuristics as described above. The results are given in Tab. 4.6. Overall, our method reaches a precision of 96.1 %, a recall of 91.4 %, and thus an F-measure of 93.6 %.

We were able to exclude all false positives from UniProt. Unfortunately, we also exclude four real foreign keys. This leads to a precision of 100 % and a recall of 87 %. Three false positives relate to FKs with only one to four distinct dependent values (which reference a relation of controlled vocabulary with 44 referenced values). The fourth false positive relates to a FK referencing a key with a single tuple. All four cases are very hard to evaluate as true FK for automatic approaches.

For Filmdienst, we miss three foreign keys and declare one IND as FK by mistake. Two false negatives relate an intended subset to the entire set, i. e., both cases match our intuition of the special coverage heuristic. Unfortunately, this heuristic misses the foreign keys, because their special coverage value lies slightly below the threshold. A slight increase of the threshold would introduce misclassifications for other data sources. The third false negative again concerns an FK with a dependent attribute having only a single value. The false positive relates to two surrogate keys over an almost identical range of values.

The single false positive for IMDB covers exactly the same problem of surrogate keys over a similar range. Also the false positive of TPC-H relates to surrogate

keys, but here in a special case: A surrogate key is referenced by an attribute storing counts of available items, which was generated by selecting values out of a range.

In summary, our heuristics provide good results for foreign keys over "real" data values. Problems occur for surrogate keys and as a special case for surrogate keys of controlled vocabulary. We are able to evaluate such INDs to non-FKs as long as the value ranges differ. INDs of surrogate keys with similar ranges are very hard to identify. Probably, the structure of all FKs in a data source could be used to exclude such false positives. We are not aware of any approach tackling this problem. Another class of problems relates to identifying FKs with only a single distinct dependent or referenced value. But we do not consider them as severe misclassifications, because the benefit of their knowledge is at least questionable.

Note that we do not know the correct number of foreign keys in PDB as there are no foreign key definitions. Thus, we estimated precision (but not recall) by manually checking the 529 INDs (out of 5,431) that were identified as FK by our combined heuristics. 446 INDs represented meaningful foreign keys. In 18 cases the INDs seem to reference tables storing controlled vocabularies, but we are not sure if these are real foreign keys. 65 INDs are false positives, again mostly generated by overlapping surrogate keys. Taken altogether, we estimate our precision in the range of 84.3 % and 87.7 %.

Using Machine Learning For the approach described so far we selected the used heuristics, their thresholds, and the weights to combine them manually. In joint work with Rostin et al. [60], we use machine learning to select the useful heuristics and to learn a classifier that combines heuristics and sets their thresholds. Therefore, we defined our heuristics as features:

- Heuristics large coverage and small coverage are combined to one feature Coverage, which just returns the coverage of an IND.
- Heuristic special coverage is represented as feature OutOfRange, but is defined as its complement: It returns the ratio of referenced values outside the range spanned by the minimum and maximum dependent values to all referenced values.
- The number of values heuristic is transformed to feature DistinctDependentValues, which simply returns the number of distinct dependent values.

- Heuristic multi dependent is extended to three features MultiDependent, MultiReferenced, DependentAndReferenced: MultiDependent returns the number of dependent values, MultiReferenced returns the number of referenced values, and DependentAndReferenced counts how often the dependent attribute is also used as referenced attribute aiming to consider the structure of INDs in a simple way.
- Heuristic column name directly transforms to feature ColumnName, which returns nominal values for different similarities (given by equal or contained referenced attribute name or referenced relation name).
- Heuristic ValueLengthDiff also directly transforms to feature ValueLengthDiff, which returns the difference between average value length of the dependent and referenced attribute.
- The small tables heuristic is changed to a similar feature TableSizeRatio, which returns the ratio of the number of tuples in the dependent and referenced attribute.
- The additional feature TypicalNameSuffix checks if the dependent or referenced attribute names end with a substring indicating a foreign key and returns nominal values. We used "id", "key", and "nr" (German for "no" – number).

First, we use different feature selection methods to find the most valuable features (and thus heuristics). The results are shown in Tab. 4.7. All four methods confirm our manual choice of Coverage, OutOfRange (i. e., the special coverage heuristic), and ColumnName. Both ranking methods also recommend to use feature ValueLengthDiff, which we discarded as heuristic. The reason was that ValueLengthDiff is just a simplification of Coverage – which is indeed ranked higher by both methods. Additionally, MultiDependent is recommended by all methods. DistinctDependentValues, which we selected manually as very trustable, is selected only by the best subset method with ranked search.

We further evaluate four different classifiers. Using the J48 classification algorithm and cross-validation at the level of data sources our approach consistently reaches F-measures above 80 % and often close to 100 % (average 93 %). These results confirm our findings using manually chosen combined heuristics – at our evaluation data sources. Obviously, the advantage of the machine learning approach is its ability to adapt to characteristics of new data sources.

	Coverage	OutOfRange	DistinctDependent-Values	MultiDependent	MultiReferenced	DependentAnd-Referenced	ColumnName	ValueLengthDiff	TableSizeRatio	TypicalNameSuffix
consistency subset evaluation										
ranked search	x	x		x	x		x	x	x	
randomized search	x	x	x	x		x	x			x
Rank according to										
InfoGain	2	1	8	5	7	10	3	4	6	9
χ^2-statistics	3	1	8	6	7	10	2	4	5	9

Table 4.7: Results for different feature selection methods. Features selected by consistency subset evaluation are marked with 'x'.

Since publication of that work, a related approach has been published [67], which discovers single-column and multi-column foreign keys. The general idea is based on the conjecture that the values in a foreign key build a uniform random sample of the referenced key. The authors define and discover the distance between the value distributions of the assumed foreign key and the referenced key as Earth Movers Distance. This approach provides on the TPC-H schema slightly better results in effectiveness than our work. A comprehensive comparison of both approaches remains for future work.

4.2.3 Deriving Primary Relations

A "primary relation" is a domain specific structural characteristic of many life science data sources. This relation represents the objects described in the data source, such as proteins or genes. All other relations provide secondary information on these objects.

We use two classes of information to choose the primary relation: the inclusion dependencies and accession number candidates, i. e., attributes with a structure of an accession number. An accession number is often characterized by the following properties [47]: The attribute containing the accession number is unique, the values are typically build of at least one non-digit character, are at least four characters

long, and the length of all accession numbers in one data source differs by at most 20 %.

To identify the primary relation of a database, we use the following heuristics:

1. One of the attributes of a primary relation has to be an accession number candidate.
2. The number of INDs referencing any attribute in a relation containing an accession number candidate is maximal for the primary relation.

Applying these heuristics to BioSQL we identified three accession number candidates (sg_bioentry.accession, sg_reference.crc and sg_ontology.name). Out of them, Heuristic 2 identifies unambiguously the correct primary relation, namely sg_bioentry.

For the OpenMMS schema we find nine accession number candidates, and 19 accession number candidates when softening the rules such that only 99.98 % of an attribute's values must fulfill the criteria of minimum length and containment of at least one non-digit character. Heuristic 2 leads to three primary relation candidates (exptl, struct, struct_keywords). Of these, struct is the correct solution, whereas struct_keywords could be considered as a second primary relation, as it is a table containing controlled vocabulary. Furthermore, the accession number candidates in these relations contain exactly the same values, which means they also relate to equivalent INDs. Thus, an automatic procedure cannot distinguish these relations. But a distinction is not necessary for the following steps on detecting inter-schema relationships, because the chosen accession numbers contain exactly equal values. Additionally, selecting these three relations is a very effective pre-selection (three tables out of 115), which helps a human expert to manually choose the primary relation.

For SCOP, we identified one accession number candidate classification.scop_id, which is the old accession number used in SCOP. It is still correct, but the newer version of SCOP uses an accession number built from numbers only [50]. Thus, we cannot detect it as an accession number with our current heuristic. Consequently, we are able to detect cross-references to SCOP using the old accession number when restricting the detection to the primary relation, but unable to detect cross-references using the new accession number.

4.3 Leveraging *Inter*-Schema INDs

As the goal of this thesis is the integration of several data sources, this section looks at the usefulness of unary IND discovery for data integration. For our life sciences data sources we followed the idea to find cross-references between data sources using accession numbers. Therefore, we restricted IND candidates to candidates with the accession number as referenced attribute, which we identified automatically as shown in Sec. 4.2.3. This way we aimed to reduce the number of identified spurious INDs. Thus, discovering intra-schema INDs can help to prune the search space for cross-references between data sources.

For actual discovery of cross-references, the strict requirement of INDs that all values of the dependent attribute must be included in the referenced attribute prevents discovery of partial, slightly modified, or dirty cross-references. In fact, we did not find any cross-reference between our life science data sources with exact IND discovery. This observation affirms the necessity of weakening the IND requirement to discover inter-schema references. We propose several modifications of SPIDER in the next Chapter 5.

Chapter 5

Extending Inclusion Dependency Discovery

In this chapter we extend SPIDER in several ways to discover special forms of INDs: We need only minor modifications to discover composite INDs (Sec. 5.1) and approximate INDs (Sec. 5.2). Discovering suffix or prefix INDs reuses general ideas of SPIDER, but demands its own algorithm - LINKFINDER (Sec. 5.3).

5.1 Composite SPIDER: Discovering Composite INDs

In the previous chapters we are able to discover only unary INDs. But in real life we also find foreign keys defined over composite attributes such as a foreign key AB in relation R_1 referencing a key CD in relation R_2. We reuse SPIDER to enable composite IND discovery.

Problem Statement 2: Discovering composite INDs Given relations $R_1, \ldots, R_n$. Discover all satisfied composite INDs between composite attributes of these relations.

□

Composite INDs can be identified by creating and testing composite IND candidates level-wise. Unsatisfied INDs at lower levels can be used to prune the space of potential candidates at higher levels, because satisfied INDs of lower levels are a precondition for a composite IND: If $A \not\subseteq C$, then there exist no attributes B and D such that $AB \subseteq CD$.

5.1.1 Extending SPIDER

We can use SPIDER with only minor modifications to test IND candidates of each level. SPIDER's test procedure is especially suitable for the detection of composite INDs, because its runtime is independent of the number of IND candidates. Therefore, the possibly large number of created IND candidates are analyzed efficiently.

Exhaustive Run of SPIDER To use SPIDER for composite INDs, we must test all IND candidates, i. e., all attribute pairs. Restricting the tests to IND candidates covering a unique referenced attribute, could lead to miss INDs that might be a precondition for a IND of a higher level. We showed in Section 4.1.2 that an exhaustive run increases SPIDER's runtime only slightly – as opposed to the numbers of IND candidates and satisfied INDs, which (may) increase largely.

Enumerating Composite IND Candidates The detection of INDs of level $l \geq 2$ is divided into two phases: (i) enumeration of all IND candidates that could be satisfied regarding the satisfied INDs of level $l-1$ and (ii) test of those IND candidates.

We use the GenNext algorithm presented in [52] to create the IND candidates level-wise. This algorithm is an adapted AprioriGen algorithm[2] using an order on attributes: The IND candidates of level l are generated by sorting all satisfied INDs of level $l-1$ by the first $l-2$ attribute pairs. An IND candidate is generated from the union of two level $l-1$ INDs with the same first $l-2$ corresponding attributes and different attributes at position $l-1$. Furthermore, only if all INDs of level $l-1$ that are implied by the generated IND candidate are satisfied must the IND candidate be tested. For example, if we verified the INDs $AB \subseteq DE$ and $AC \subseteq DF$, we generate the IND candidate $ABC \subseteq^? DEF$ at the next level. We test this IND candidate only if all implied INDs $AB \subseteq DE$, $AC \subseteq DF$, and $BC \subseteq EF$ are satisfied. In [52] the authors show that this algorithm indeed enumerates all possible composite INDs.

Testing Composite IND Candidates We modified SPIDER in the following aspects to test composite IND candidates, preserving the advantage of testing all IND candidates (of a given level) simultaneously: We hold composite attributes and their values in the min-heap, instead of single attribute values. For each level, we query the sorted composite data sets from the database and write them to disk. This step is necessary, because we need the correct associations of the attribute values to

tuples, which cannot be recovered from the single attribute value files.

Unfortunately, the number of candidates grows exponentially – not only in the number of IND candidates, but also in the number of composite attributes. Thus, we expect the runtime to increase largely for larger levels. Our experiments in the next section shall validate this estimation.

5.1.2 Evaluating Composite SPIDER

We run Composite SPIDER on our test data sources UniProt, TPC-H, and PDB. The results are shown in Table 5.1. No composite IND with level greater than two was found for UniProt and TPC-H, i. e., no IND candidate of level 3 was generated. The results show that the total runtime over all levels is dominated by the requirement to read and write tuples for each level (denoted as SRSW in Tab. 5.1). This observation shows again that a database internal filter on IND candidates could speed up the runtime.

When analyzing the 2.8 GB part of the PDB we had to resort to a heuristic to constrain the set of satisfied unary INDs. When all unary INDs are used to create composite INDs of level 2, then 368,997 IND candidates are created covering 15,798 composite attributes. The run took $\sim$ 33 hours and resulted in 227,028 satisfied INDs. These INDs implied about $5 * 10^6$ possible IND candidates at level 3, which cannot be tested in reasonable time.

However, recall that in the end we are hunting for foreign keys. Since we are confident that the number of actual composite foreign keys at level 2 or 3 in the data is rather small, we applied two heuristics to prune probably uninteresting, yet satisfied unary INDs: (i) The referenced attribute must have more than one distinct value; and (ii) At least 1 % of all distinct values of the referenced attribute must be covered by values of the dependent attribute. Both restrictions are very weak and should not exclude interesting foreign keys, but they reduce the number of unary INDs to 8,830 and the number of IND candidates in level 2 to 12,535 covering 2,576 composite attributes (see Table 5.1). Thus, we identified 7,442 INDs of level 2 in a total runtime of 6 h40 m. These INDs implied 13,647 IND candidates of level 3 covering 4,336 composite attributes.

These experiments confirm that the exponential explosion in the number of candidates is real, even when pruning with satisfied INDs from lower levels is fully exploited. In the literature there are approaches that especially aim to find compos-

	UniProt	TPC-H	PDB 2.8GB[a]
Composite SPIDER	5 m 25 s	27 m 29 s	6 h 40 m
level 1			
# IND candidates	2,235	1,616	734,851
# attributes	68	61	1,215
# INDs	116	86	35,752
SRSW	1 m 43 s	6 m 13 s	20 m 59 s
test	37 s	17 s	2 m 41 s
level 2			
# IND candidates	20	59	12,535
# composite attributes	14	36	2,576
# INDs	13	21	7,442
SRSW	8 m 24 s	25 m 31 s	6 h 00 m
test	44 s	54 s	16 m

[a] We filtered unary INDs before running level 2 and stopped execution after this level.

Table 5.1: Experimental results for discovering composite INDs.

ite INDs at larger levels (up to level 41) [44, 54]. These approaches use procedures as described in Section 3.4 to find unary and binary INDs. Using SPIDER for lower levels and the approaches of [44, 54] for larger levels would improve composite IND discovery at all.

5.1.3 Related Work

Marchi et al. [53] reuse their approach of discovering unary INDs and give a level-wise approach for discovering composite INDs. We employ their approach for composite IND candidate creation but test the candidates with our SPIDER algorithm. Marchi et al. extended the level-wise approach for detecting composite INDs in [54]. Their main idea is to reduce the number of IND candidates by switching between a top-down and a bottom-up approach using an optimistic positive border.

Koeller and Rundensteiner proposed to create composite IND candidates by finding cliques in k-uniform hypergraphs [44]. These hypergraphs are made of satisfied INDs of lower levels. Unary and binary INDs are tested by an exhaustive approach

similar to [11]. Further, the authors extend their approach in [45] by defining heuristics to reduce the search space.

[54], [44], and [45] use tests for single IND candidates on diverse levels. The strength of SPIDER is – in contrast – its independence in runtime of the number of IND candidates. Note that [54], [44], and [45] aim at finding composite INDs of higher levels and run tests either on real world databases with INDs of maximally level three or on artificial databases with higher level INDs (up to level 41). In our test databases we did not observe such INDs and are not aware of real world databases with these characteristics. However, SPIDER could be leveraged to find composite INDs of lower levels followed by [54], [44], or [45] to find INDs of higher levels. Furthermore, note that both projects provide experimental data only for rather small schemas with only few tables and attributes.

5.2 Approximate SPIDER: Discovering Approximate INDs on Dirty Data

In most real-world databases one finds dirty data whenever foreign key constraints are not enforced by the system. Potential reasons are faulty parsers for importing data or simply errors in the data. Thus, we aim to discover approximate INDs. Recall from Sec. 2.2 that the amount of allowed violating, i. e., not included, values can be specified in two ways: (i) the number of all distinct, not included values expressed as a percentage of distinct values or (ii) the absolute number of not included values (as suggested by [51]). Both values are helpful to rate an approximate IND.

Problem Statement 3: Discovering Approximate INDs Given relations R_1, $\ldots, R_n$ and a threshold for the amount of allowed violating values. Discover all satisfied approximate INDs between attributes of these relations regarding the given threshold.

□

5.2.1 Extending SPIDER

The SPIDER algorithm – with some minor modifications – is able to discover approximate INDs very efficiently (see Alg. 2). In the following we describe modifications

Algorithm 2: Approximate SPIDER.

Input: attributes: set of attribute objects with their sorted values and their respective refs sets (the IND candidates); threshold for approximate INDs (absolute number of allowed distinct violating values)

Output: Set of satisfied approximate INDs.

```
1  Min-Heap heap := new Min-Heap( attributes ) ;
2  while heap != ∅ do
       /* get attributes with equal minimal value                    */
3      att := heap.removeMinAttributes() ;
4      foreach A ∈ att do
           /* update list A.refs                                     */
5          A.unsatRefs := A.refs \ att ;
6          A.refs := A.refs ∩ att ;
           /* Refs with counter ≤ threshold remain.                  */
7          foreach ref ∈ A.unsatRefs do
8              ref.counter++ ;
9              if ref.counter ≤ threshold then
10                 A.refs := A.refs ∪ {ref} ;
           /* process next value                                     */
11         if A has next value then
12             A.moveCursor() ;
13             heap.add(A) ;
14         else
15             foreach B ∈ A.refs do
16                 INDs := INDs ∪ { A ⊆ B } ;
17 return INDs
```

that are necessary to collect for each IND candidate the number of all distinct, violating values during the test.

Note that the absolute number of allowed distinct violating values can be easily derived from a given percentage of allowed violating values by multiplying the given percentage with the overall number of distinct values in the dependent attribute, which is known after reading the attribute's (sorted) distinct values from the database.

We add a counter to the references in each attribute object in the list refs. These counters represent the number of violating values of this dependent attribute object. When updating the list refs of an attribute object we do not discard attributes from refs directly. Instead we buffer them in a list unsatRefs and increase the counter of these objects. If the given threshold is not exceeded, the attribute is re-added to the list refs. Only once a given threshold of violating values is exceeded, is the referenced attribute object discarded.

To obtain the absolute number of all violating values, i.e., also counting duplicates, we need the number of occurrences for each value in the dependent attributes. These can be extracted from the database using a SQL group by statement on the dependent attribute with count as aggregation function, which does not incur more work for the database compared to a single sort and distinct. For counting absolute numbers, every not-included dependent value needs to be multiplied by its number of occurrences.

5.2.2 Evaluating Approximate SPIDER

We give experimental results on our test data sources UniProt, TPC-H, and PDB for the first alternative for specifying the tolerated level of dirtyness, i.e., allowing a certain percentage of distinct values to be not contained. Results are shown in Table 5.2. We allowed 5% violating distinct values in the dependent attribute, which we believe is a very high error rate. Thus, the results should be considered as rather conservative runtime estimations.

At this level there are indeed a considerable amount of approximate INDs in three of the four data sets. However, compared to the results of exact tests (Table 4.2; repeated in Tab. 5.2), the runtime increases only minimally. The difference stems from the additional costs for counting and for comparing this number to the given threshold. Furthermore, more values have to be processed, because attributes are

	UniProt	TPC-H	PDB	
DB size	900 MB	1.5 GB	2.8 GB	32 GB
# ind candidates	1,393	877	219,106	245,562
# INDs	36	33	4,972	5,431
# approximate INDs	36	40	10,737	12,081
SPIDER	1 m 51 s	6 m 25 s	23 m 36 s	6 h 07 m
approximate SPIDER	1 m 57 s	6 m 26 s	27 m 25 s	6 h 27 m

Table 5.2: Results for discovering approximate INDs. 5 % of violating dependent values were allowed. IND candidates are restricted to cover unique referenced attributes.

excluded later from all IND candidates and thus from the SPIDER heap. But the experiments show that SPIDER is very efficient also for discovering approximate INDs. Note that for a lower error rate the difference to a run without allowed errors would be even smaller.

5.2.3 Related Work

Marchi et al. [53] extend their approach of discovering unary INDs to find approximate INDs. As the preprocessing step is reused, the disadvantage of this approach remains.

Furthermore, approaches for discovering unary INDs approximately [20, 25, 45] are related work for Approximate SPIDER. We already described the first two approaches in Section 3.4 and the last approach in Section 5.1.3. SPIDER is able to give the exact number of not included values, which distinguishes it from all of these algorithms.

5.3 LINKFINDER: Discovering Prefix and Suffix INDs

The following section considers discovering prefix and suffix INDs. Recall from Sec. 2.2 that prefix (or suffix) INDs are INDs after removing a fixed or variable prefix (or suffix) from each attribute in the dependent attribute. This type of INDs is intended to discover links between data sources. The example of CATH cross-referencing PDB motivated our definition and discovery of prefix and suffix INDs:

CATH cross-references PDB accession numbers concatenated with a suffix of two varying characters, the encoded domain of the considered protein. Thus, we discover this cross-reference if we discover the suffix IND domain_list[domain_name] $\subseteq_s^{4,2}$ struct[entry_id], i. e., the suffix IND with dependent attribute domain_name in relation domain_list of CATH, referenced attribute entry_id of relation struct in PDB, a fixed key length of 4 characters, and a fixed suffix length of 2 characters.

This section is based on work in [38] supervised by the thesis' author. But we choose here a different running example that shows all difficulties in suffix (prefix) IND discovery and a different representation of the algorithm that includes all aspects of the algorithm, i. e., discovering the suffix INDs and their key and suffix lengths.

We describe in the following the problem of discovering prefix or suffix INDs and show the similarities and differences to the problem of discovering INDs. We elaborate the similarity to SPIDER and show the necessary changes. In the result we provide a completely new algorithm – LINKFINDER.

Based on the aim to discover cross-references between data sources we can distinguish attributes as exclusively dependent or exclusively referenced attributes for discovering prefix (suffix) INDs – as opposed to the problem of discovering INDs where each attribute can be covered by IND candidates as dependent or referenced attribute. Thus, we must consider any combination of an arbitrary dependent attribute with an arbitrary referenced attribute as prefix (suffix) IND candidate. We describe the problem of suffix IND discovery as follows. The problem of prefix IND discovery describes analogously.

Problem Statement 4: Discovering suffix INDs Given a set of dependent and a set of referenced attributes; Discover all satisfied suffix INDs with their key and suffix lengths, i. e., test any combination of an arbitrary dependent attribute with an arbitrary referenced attribute as suffix IND candidate. □

5.3.1 Similarities and Differences to IND Discovery

The problem statement shows the similarity to SPIDER: Each attribute (dependent or referenced) is part of several prefix or suffix IND candidates. Just like IND candidates, prefix or suffix IND candidates can be represented by a list of referenced attributes in a dependent attribute object. Further, any value in the dependent attribute must be contained in the referenced attribute (but with a different notion

of containment compared to IND discovery). Thus, we aim to reuse the ideas of SPIDER to save I/O and comparisons: We sort each attribute's distinct values and save them to disk. Afterwards, we read these values just once to test all prefix (or suffix) IND candidates in parallel. Therefore, we hold attribute objects in a min-heap and compare their values in a suitable order.

But as there are noteworthy similarities there are even more remarkable differences: For INDs we need to find for each dependent value an equal referenced value. For prefix (or suffix) INDs we have to find referenced values that are suffixes (or prefixes) for each dependent value. We say a dependent value *matches* a referenced value iff the referenced value is a suffix (or prefix) of the dependent value. Given the data structure of SPIDER with all sorted attributes sorted in a min-heap, it is obviously easier to test if a (referenced) value is the prefix of another (dependent) value. Thus, we first consider and explain discovering suffix INDs. Afterwards, we reduce the discovery of prefix INDs to discovering suffix INDs (Sec. 5.3.4).

Note, that this relaxation of finding matching prefixes (i. e., keys) instead of equal values results in a difficulty: Opposed to SPIDER **each value in a dependent attribute can match several values in the referenced attribute** – depending on the prefix or suffix length. Consider for example a suffix IND candidate $A \subseteq_s^? B$ with

- $A = \{bbc\}$
- $B = \{b, bb\}$

The dependent value *bbc* matches the referenced value *b* with suffix length two and matches referenced value *bb* with suffix length one. This simple example shows a major difference to SPIDER: We cannot release a dependent value as soon as we found a matching referenced value, because following referenced values (of the same or another attribute) could also match this dependent value. We need to know all matches as we can illustrate with a slight extension of our example:

- $A = \{bbc, bcd\}$
- $B = \{b, bb\}$

For $A \subseteq_s^? B$, again *bbc* matches *b* and *bb*, but *bcd* matches only *b*. All together we discover a suffix IND $A \subseteq_s^{1,2} B$ with key length one and suffix length two.

The above example shows even more differences to SPIDER: We need more than the minimal value of all attribute objects at a time – we **need to compare the minimal dependent with the minimal referenced value**. This way, we can compare e. g., *bbc* and *b* in our example. This observation leads us to hold dependent attributes and referenced attributes in different min-heaps.

A further extension of our example shows a difference to SPIDER regarding the lists maintained for each suffix IND candidate. Consider suffix IND candidates $A \subseteq_s^? B$ and $A \subseteq_s^? C$ with

- $A = \{bbc, bcd\}$
- $B = \{b, bb\}$
- $C = \{bb\}$

For $A \subseteq_s^? C$, we cannot find the match between *bbc* and *bb* in the first comparison of the overall minimal dependent (*bbc*) and referenced values (*b*). We get the opportunity to compare *bbc* and *bb* after the first cursor movement in B. Thus, we **need two lists with referenced attributes representing suffix IND candidates**: those that could be suffix INDs after checking all previous dependent values refs, and those that already matched the current dependent value matchedCurrent.

Furthermore, not only each single dependent value can match several referenced values. Additionally, **each single referenced value can be matched by several dependent values** – as *b* is matched by *bbc* and *bcd* in our example. The problem is now to decide when we move the cursors in dependent and referenced attributes: After comparing *bbc* and *b*, should we move the cursor in A to compare (and find the match between) *bcd* and *b*? This way we would miss the opportunity to compare *bbc* and *bb*. Or should we move the cursor in B to compare (and find the match between) *bbc* and *bb*? But now we miss the opportunity to compare *bcd* and *b*.

One important observation helps to solve this problem: **If several dependent values match the same referenced value, they share this referenced value as prefix** (otherwise they would not match this referenced value). Thus, we can deduce this match by administrating common prefixes of dependent attributes. As all values are sorted, values with same prefixes follow each other, i. e., we cannot miss matches. In our example, *bbc* and *bcd* both match *b* – and both share *b* as prefix. We move the cursor in A and deduce the match between *bcd* and *b*.

Taken all together, we can describe the general rules of comparisons and cursor movements in LinkFinder as follows:

- As with Spider hold the attribute objects in a min-heap, but now divided in two min-heaps for dependent attribute objects and referenced attribute objects. Choose in each step those attribute objects with the minimal current values, i. e., compare the current minimum dependent value *minDep* and current minimum referenced value *minRef*. Note that there might be several (dependent or referenced) attribute objects sharing the same value.

- If *minDep* matches *minRef* then we found a match: Note this match and move the cursor in the referenced attribute objects with current value *minRef* to find further matches for *minDep*.

- If otherwise *minDep* > *minRef* then we did not find a match for *minDep* yet. Thus, move the cursor in the referenced attribute objects with current value *minRef* to find possible matches.

- If *minDep* < *minRef* then *minDep* cannot match any following *minRef*. Thus, we update the information on matches in the dependent attribute objects with value *minDep* and move their cursor.

- The algorithm stops if no dependent or referenced values are left in any attribute object.

5.3.2 LinkFinder By Example

We show in this Section a general run of LinkFinder using an example with two potentially dependent attributes D_1, D_2 and two potentially referenced attributes R_1, R_2:

- $D_1 = \{baa, bbaa, bbac\}$
- $D_2 = \{baa, baab, bab\}$
- $R_1 = \{b, baa, bba, bbb\}$
- $R_2 = \{ba, bba\}$

We have to test four suffix IND candidates: $D_1 \subseteq_s^? R_1$, $D_1 \subseteq_s^? R_2$, $D_2 \subseteq_s^? R_1$, and $D_2 \subseteq_s^? R_2$. We chose the example to show difficulties in suffix IND discovery and their solution: All three values in D_1 share prefix b, but the second and third value share the extended prefix bba. LINKFINDER must be able to handle these overlapping prefixes to not only discover suffix IND $D_1 \subseteq_s^{1,\cdot} R_1$ with key length one and variable suffix length (i. e., all referencing value b), but also to discover $D_1 \subseteq_s^{3,\cdot} R_1$ with key length three and variable suffix length (i. e., baa matching baa and $bbaa$, $bbac$ matching bba). Further, LINKFINDER must find the suffix IND $D_1 \subseteq_s^{\cdot,1} R_2$ with variable key length and a suffix length of one (i. e., baa matching ba, and $bbaa$, $bbac$ matching bba). Another difficulty can be seen at $D_2 \subseteq_s^? R_1$: In D_2 the first two values share prefix baa and the third value shares the shortened prefix ba. LINKFINDER must handle this shortened prefix to exclude matches for value bab to referenced value baa in R_1, i. e., dismiss suffix IND $D_2 \subseteq_s^{3,\cdot} R_1$ and confirm suffix IND $D_2 \subseteq_s^{1,\cdot} R_1$. Altogether, we observe 5 suffix INDs: $D_1 \subseteq_s^{1,\cdot} R_1$, $D_1 \subseteq_s^{3,\cdot} R_1$, $D_1 \subseteq_s^{\cdot,1} R_2$, $D_2 \subseteq_s^{1,\cdot} R_1$, and $D_2 \subseteq_s^{2,\cdot} R_2$.

We first show a run of LINKFINDER handling common prefixes in dependent attributes, which shows the major characteristics of LINKFINDER. Afterwards we show in a second run handling additionally the possible key and suffix lengths.

Figure 5.1 illustrates this first run of LINKFINDER. The left hand side shows two min-heaps with dependent and referenced attribute objects. Dependent attribute objects track the suffix IND candidates with their lists refs and matchedCurrent. We initialize the lists refs of attributes D_1 and D_2 with R_1, R_2 indicating the four suffix IND candidates.

1. In the first step LINKFINDER compares the minimum referenced value, i. e., b of R_1, with the minimum dependent value, i. e., baa of D_1 and D_2. As b is a prefix of baa, LINKFINDER adds R_1 to the matchedCurrent lists of D_1 and D_2. Further, it stores the matching prefix length 1 for R_1. As we found a match we move the cursor in the referenced attribute objects with current value *minRef*, i. e., R_1.

2. In the second step we compare value ba of R_2 with baa of D_1, D_2 and find again a match. LINKFINDER adds R_2 to the matchedCurrent lists of D_1 and D_2 together with the matching prefix length 2. Again we move the cursor in the currently used referenced object, i. e., R_2.

	min-heap with dependent attribute objects		min-heap with referenced attribute objects						
	D1	**D2**	**R1**	**R2**		**D1.refs**	**D1.matchedCurrent**	**D2.refs**	**D2.matchedCurrent**
					initialize	R1, R2		R1, R2	
Step 1:	baa	baa	b		*found match*	R1, R2	R1(1)	R1, R2	R1(1)
Step 2:	baa	baa		ba	*found match*	R1, R2	R1(1), R2(2)	R1, R2	R1(1), R2(2)
Step 3:	baa	baa	baa		*found match*	R1, R2	R1(1,3), R2(2)	R1, R2	R1(1,3), R2(2)
Step 4:	baa	baa	bba	bba	*baa < bba -> move cursor in D1, D2; manage lists in D1, D2*				
						R1, R2	|prefix(baa, bbaa)| = 1	R1, R2	|prefix(baa,baab)| = 3
							R1(1)		R1(1,3), R2(2)
Step 5:		baab	bba	bba	*baab < bba -> move cursor in D2; manage lists*				
						R1, R2	R1(1)	R1, R2	|prefix(baab,bab)| = 2
									R1(1), R2(2)
Step 6:		bab	bba	bba	*bab < bba -> move cursor in D2 (but end of values); manage lists in D2*				
								R1, R2	
Step 7:	bbaa		bba	bba	*found match*	R1, R2	R1(1,3), R2(3)		
Step 8:	bbaa		bbb		*bbaa < bbb -> move cursor in D1; manage lists in D1*				
						R1, R2	|prefix(bbaa, bbac)| = 3		
							R1(1,3), R2(3)		
Step 9:	bbac		bbb		*bbac < bbb -> move cursor in D1; manage lists in D1*				
						R1, R2			

Figure 5.1: Example run of LINKFINDER. $D_1 = \{baa, bbaa, bbac\}$, $D_2 = \{baa, baab, bab\}$, $R_1 = \{b, baa, bba, bbb\}$, $R_2 = \{ba, bba\}$; Test $D_1 \subseteq_s^? R_1$, $D_1 \subseteq_s^? R_2$, $D_2 \subseteq_s^? R_1$, $D_2 \subseteq_s^? R_2$ and confirm $D_1 \subseteq_s R_1$, $D_1 \subseteq_s R_2$, $D_2 \subseteq_s R_1$, $D_2 \subseteq_s R_2$. Maintaining and deducing key and suffix length is shown in Figure 5.2 using the same example.

3. The third step compares value *baa* of R_1 with value *baa* in D_1, D_2 and finds the match. As R_1 is already included in lists matchedCurrent of D_1 and D_2 LinkFinder only adds the matching prefix length 3. Afterwards we move the cursor in R_1.

4. The next step compares value *bba* of R_1 and R_2 with *baa* of D_1, D_2. As $baa < bba$ LinkFinder moves the cursors in D_1 and D_2 and manages their lists refs and matchedCurrent as follows: We intersect lists refs and matchedCurrent to get the new list refs, i. e., these referenced attributes are a suffix Ind for all values in the dependent attribute that have been compared yet. In our case R_1 and R_2 include values that match value *baa* in D_1 and D_2.

   To update list matchedCurrent we need the length of the common prefix of the current dependent value and the next value for each dependent attribute object:

   - For D_1 we compare *baa* and *bbaa* identifying a common prefix of length one. Thus, we carry over the matching referenced attributes with a length of at most one. In case of D_1 we carry over R_1 with prefix length 1 (i. e., we carry over the match to *b*), but dismiss matches of length 3 in R_1 (i. e., *baa*) and of length two to R_2 (i. e., *ba*).
   - For D_2 we compare *baa* and *baab* and find the common prefix length three. Thus, we carry over the entire list matchedCurrent, which means *baab* also matches *b* and *baa* in R_1 and *ba* in R_2.

5. The fifth step compares still value *bba* of R_1 and R_2 with *baab* of D_2. As $baab < bba$, we move the cursor in D_2 and update its lists refs and matchedCurrent. We update refs to R_1, R_2 and matchedCurrent to R_1 with matching prefix length 1 and R_2 with matching prefix length 2. We discard R_1 with matching prefix length 3, because of the common prefix length 2 between *baab* and *bab*.

6. We now compare *bab* of D_2 with *bba* of R_1 and R_2. As $bab < bba$ we try to move the cursor in D_2 and find no next value. Nevertheless we need to update D_2's list refs before we derive the satisfied suffix Inds: The last value could exclude suffix Ind candidates that were not excluded so far. For D_2 we confirm the suffix Inds $D_2 \subseteq_s R_1$ and $D_2 \subseteq_s R_2$.

7. We compare value *bbaa* of D_1 with *bba* of R_1 and R_2 in the seventh step. We find the match of length 3 and add this information to list D_1.matchedCurrent. Further, we move the cursors in R_1 and R_2. Note that there is no next value for R_2. This means we will not compare any further value of R_2 with upcoming values in dependent attributes (here D_1), but we could deduce matches to previous values in R_2 using the dependent attribute object's lists matchedCurrent.

8. The next comparison of value *bbaa* of D_1 with *bbb* of R_1. We move the cursor in D_1 and update its lists refs to R_1, R_2 and matchedCurrent to R_1 with matching prefix length 1 and 3 and to R_2 with matching prefix length 3 (because of the common prefix length three between *bbaa* and *bbac*).

9. The last comparison results in $bbac < bbb$ and therefore the tried cursor movement in D_1. After updating refs we confirm the suffix INDs $D_1 \subseteq_s R_1$ and $D_1 \subseteq_s R_2$.

The shown run results in four confirmed suffix INDs – but so far we do not have any information on matching key length or suffix length. We need one last extension in the algorithm to gather this important information: We track the matching key and suffix length during the LINKFINDER run. This means we must maintain for any suffix IND candidate the information on common matching key and suffix length of already seen comparisons. This information is maintained in the list refs of dependent attribute objects.

We denote each identified pair of matching key and suffix lengths as pair (keyLength, suffixLength) for each tracked referenced attribute. In the initial step we denote the arbitrary length as pair $(*, *)$. Varying key or suffix length are denoted by a dot. For example a previous match between a dependent attribute D and a referenced attribute R of key length 1 with suffix length 1 $R(1, 1)$ and a further match of again key length 1 but suffix length 2 $R(1, 2)$ results in $R(1, .)$ – indicating that we found so far matches between D and R of constant key length 1 but with varying suffix lengths.

Figure 5.2 shows the previous example extended by the denoted matching key and suffix length. We initialize attributes D_1 and D_2 with $R_1(*, *), R_2(*, *)$ indicating the four suffix IND candidates with arbitrary key and suffix length.

	min-heap with dependent attribute objects		min-heap with referenced attribute objects					
	D1	**D2**	**R1**	**R2**		**D1.refs**	**D1.matchedCurrent**	**D2.refs**
					initialize	R1(*,*), R2(*,*)		R1(*,*), R2(*,*)
Step 1:	baa	baa	b		*found match*	R1(*,*), R2(*,*)	R1(1,2)	R1(*,*), R2(*,*)
Step 2:	baa	baa		ba	*found match*	R1(*,*), R2(*,*)	R1(1,2), R2(2,1)	R1(*,*), R2(*,*)
Step 3:	baa	baa	baa		*found match*	R1(*,*), R2(*,*)	R1(1,2), (3,0), R2(2,1)	R1(*,*), R2(*,*)
Step 4:	baa	baa	bba	bba	*baa < bba -> move cursor in D1, D2; manage lists in D1, D2*			
						R1(1,2), (3,0), R2(2,1)	|prefix(baa, bbaa)| = 1 R1(1,3)	R1(1,2), (3,0), R2(2,1)
Step 5:		baab	bba	bba	*baab < bba -> move cursor in D2; manage lists*			
						R1(1,2), (3,0), R2(2,1)	R1(1,3)	R1(1,.), (3,.), R2(2,.)
Step 6:		bab	bba	bba	*bab < bba -> move cursor in D2 (but end of values); manage lists in D2*			
								R1(1,.), R2(2,.)
Step 7:	bbaa		bba	bba	*found match*	R1(1,2), (3,0), R2(2,1)	R1(1,3), (3,1), R2(3,1)	
Step 8:	bbaa		bbb		*bbaa < bbb -> move cursor in D1; manage lists in D1*			
						R1(1,.), (3,.), R2(.,1)	|prefix(bbaa, bbac)| = 3 R1(1,3), (3,1), R2(3,1)	
Step 9:	bbac		bbb		*bbac < bbb -> move cursor in D1; manage lists in D1*			
						R1(1,.), (3,.), R2(.,1)		

Figure 5.2: Example run of LINKFINDER with maintaining and deducing key and suffix length. $D_1 = \{baa, bbaa, bbac\}$, $D_2 = \{baa, baab, bab\}$, $R_1 = \{b, baa, bba, bbb\}$, $R_2 = \{ba, bba\}$; Test $D_1 \subseteq_s^? R_1$, $D_1 \subseteq_s^? R_2$, $D_2 \subseteq_s^? R_1$, $D_2 \subseteq_s^? R_2$ and confirm $D_1 \subseteq_s^{1,\cdot} R_1$, $D_1 \subseteq_s^{3,\cdot} R_1$, $D_1 \subseteq_s^{\cdot,1} R_2$, $D_2 \subseteq_s^{1,\cdot} R_1$, $D_2 \subseteq_s^{2,\cdot} R_2$. A reduced run without maintaining and deducing key and suffix length is shown in Figure 5.1 using the same example.

1-3. The first three steps run as before, but track matching key and suffix lengths in lists matchedCurrent. For example in the third step we denote in D_1. matchedCurrent matches between D_1 and R_1 of key length one with suffix length two (i. e., *baa* to *b*) and key length three with suffix length zero (i. e., *baa* to *baa*) and matches between between D_1 and R_2 of key length two with suffix length one (i. e., *baa* to *ba*).

4. The fourth step compares again *baa* of D_1 and D_2 with *bba* of R_1 and R_2. As $baa < bba$, we move as before the cursor in D_1 and D_2 but manage their lists refs and matchedCurrent in an extended manner: We intersect the lists refs and matchedCurrent to get the new list refs – this time preserving the information on matching key and suffix lengths. To update list matchedCurrent of D_1 we carry over the match of key length 1 with R_1 and dismiss the matches of key length 3 to R_1 and 2 to R_2. Additionally, we adjust the suffix length for the match to R_1 with key length 1: The current value of D_1 *baa* is shorter than its next value *bbaa*. Thus, with the same matching key length 1 (to value *b* in R_1) we have now suffix length 3. Analogously, we update list matchedCurrent of D_2: We carry over all three matches as the common prefix length of the current value *baa* and the next value *baab* is three and add one to the suffix lengths as *baab* is one character longer than *baa*.

5. The fifth step compares *baab* of D_2 with *bba* of R_1 and R_2. Thus, we move the cursor in D_2 and update its lists refs and matchedCurrent. To update list refs we intersect refs and matchedCurrent: R_1 is contained in both lists with key length one and three, but with different suffix lengths. The resulting intersection is $R_1(1,.)(2,.)$. R_2 is also contained in both lists with key length 2 and different suffix length resulting in $R_2(2,.)$. The update of list matchedCurrent carries over the matches to R_1 with key length 1 and to R_2 with key length 2 as the common prefix length between D_2's current and next value is two. As the next value is one character shorter than the current value, we adjust the suffix lengths by subtracting one.

6. We compare *bab* of D_2 with *bba* of R_1 and R_2 in the sixth step, move the cursor in D_2 and manage list refs. As there is no matching key length 3 for R_1 in list matchedCurrent, we exclude this key length and remain with matches $R_1(1,.)$ and $R_2(2,.)$. This means, we confirm two suffix INDs: $D_2 \subseteq_s^{1,\cdot} R_1$ and

$D_2 \subseteq_s^{2,\cdot} R_2$ with key length 1 and 2 respectively and varying suffix lengths.

7. The seventh step compares *bbaa* of D_2 with *bba* of R_1 and R_2, adds the identified matching key length 3 with suffix length 1 to R_1 and R_2 in list matchedCurrent of D_2, and moves the cursors in R_1 and R_2.

8. The next step compares *bbaa* of D_1 with *bbb* of R_1, which results in moving the cursor in D_1 and managing its lists: The new list refs results in matching key lengths of 1 and 3 with varying suffix lengths for R_1. For R_2 we observe a varying key length, but a common suffix length of 1. The new list matchedCurrent is the same as the old list, because the common prefix length between D_1's current and next value is three and both values share the same length.

9. In the last step we compare *bbac* of D_1 with *bbb* of R_1, move the cursor in D_1, and update list refs. We confirm three suffix INDs $D_1 \subseteq_s^{1,\cdot} R_1$ and $D_1 \subseteq_s^{3,\cdot} R_1$, and $D_1 \subseteq_s^{\cdot,1} R_2$.

5.3.3 The LinkFinder Algorithm

The data structure of LinkFinder represents the attributes and their values as attribute objects – similarly to Spider. Each attribute object stores the attribute's distinct sorted values. The values can be iterated by a cursor, which is set initially to the smallest value.

For LinkFinder, we distinguish dependent attribute objects from referenced attribute objects. While referenced attribute objects provide only the iterable values, dependent attribute objects further hold information on suffix Ind candidates using two sets refs and matchedCurrent. A suffix Ind candidate $A \subseteq_s^? B$ is represented by adding the referenced attribute object of B to the dependent attribute object's set refs. Set refs represents all suffix Ind candidates that have not been excluded as unsatisfied so far. Set matchedCurrent represents all suffix Ind candidates that have been confirmed so far for the current dependent attribute's value.

Additionally, each linked referenced attribute object in a dependent attribute object's sets refs and matchedCurrent is supplemented by the matching key lengths and the suffix lengths that have been discovered yet.

The LinkFinder algorithm is given in Algorithm 3. The input to LinkFinder are two min-heaps: depHeap containing the dependent attribute objects with their

initialized sets refs and matchedCurrent, and refHeap containing the referenced attribute objects. LINKFINDER iterates the attribute objects by processing those attributes with the current minimal dependent and referenced values. Lines $1, 2, 10, 14$, and 29 determine these current minimal attributes *minDep* and *minRef* depending on the applied cursor movements. Lines $4 - 9$ process identified matches between *minDep* and *minRef*, i. e., add all referenced attributes in *minRef* and the corresponding key and suffix lengths to all *minDep*'s set matchedCurrent and move the cursors of all attribute objects in *minRef*. Lines $11 - 13$ handle the case *minDep* $>$ *minRef*, i. e., just move the cursors in all attribute objects in *minRef* to enable discovering (further) matches for the current value in all *minDep* attributes. Lines $15 - 28$ handle the remaining case *minDep* $<$ *minRef*, i. e., move all cursors in the dependent attribute objects in *minDep* and update their lists refs and matchedCurrent to exploit the collected information on matches to the current value and prepare collecting information on matches to the next value. Therefore, set refs is intersected with set matchedCurrent to represent all suffix IND candidates that have been not excluded as unsatisfied after testing all value including the current value. Set matchedCurrent carries over all matches with a maximum key length of the common prefix length between the current and the next value in the dependent attribute object. The suffix lengths are adjusted to consider differing lengths of the current and next value. If there is no next value in the dependent attribute object all referenced attributes in set refs represent satisfied suffix INDs. The suffix INDs are added to the result set.

Complexity analysis for LINKFINDER We denote the number of dependent and referenced attributes as n_d and n_r respectively, and the maximum number of values in the dependent and referenced attributes as t_d and t_r respectively. The number of all attributes is denoted as n, the maximum number of values as t. Further, we denote the maximum length of all referenced values as l_r.

For sorting the data we need $O(nt \log t)$ comparisons. Adding and removing attribute objects to and from the heaps need – as for SPIDER– $O(n_d t_d \log n_d + n_r t_r \log n_r)$.

We need to compare the current minimum dependent and referenced attribute object's values to identify the next step. After each step either the dependent attribute object's cursor is moved on or the referenced attribute object's cursor.

Algorithm 3: LINKFINDER.

```
Input: depHeap: Min-Heap with dependent attribute objects;
       refHeap: Min-Heap with referenced attribute objects
1  minDep := depHeap.removeMinAttributes() ;
2  minRef := refHeap.removeMinAttributes() || ∞ ; /*  minRef = ∞ if
                refHeap is empty; aimed result:  minDep < minRef  */
3  while depHeap != ∅ do      /* get & process attr. obj. with min. value */
4      if prefix(minDep) == minRef then
           /* update sets matchedCurrent and move minRef cursors      */
5          foreach ref ∈ minRef do
6              foreach dep ∈ minDep do
7                  dep.matchedCurrent.add(
8                      ref(ref.value.length, dep.value.length − ref.value.length)) ;
9              if ref has next value then  ref.moveCursor() ; refHeap.add(ref) ;
10         minRef := refHeap.removeMinAttributes() || ∞ ;
11     else if minDep > minRef then              /* move minRef cursors */
12         foreach ref ∈ minRef do
13             if ref has next value then  ref.moveCursor() ; refHeap.add(ref) ;
14         minRef := refHeap.removeMinAttributes() || ∞ ;
15     else          /* update sets refs, matchedCurrent; move minDep cursors */
16         foreach dep ∈ minDep do
17             dep.refs := dep.refs ∩ dep.matchedCurrent ;
18             if dep.refs != ∅ ∧ dep has next value then
               /* move cursor, upd. matchedCurr.  */
19                 currentValue := dep.value ;
20                 dep.moveCursor() ; depHeap.add(dep) ; nextValue := dep.value ;
21                 dep.matchedCurrent.remainPrefixLengthRefs(
22                     commonPrefixLength(currentValue, nextValue)) ;
23                 foreach ref ∈ dep.matchedCurrent do
24                     ref.setSuffixLength(nextValue.length − currentValue.length) ;
25             else                                  /* save suffix INDs  */
26                 foreach ref ∈ dep.refs do
27                     INDs:= INDs ∪ {dep ⊆_s^{ref.keyLength,ref.suffixLength} ref}
28         minDep := depHeap.removeMinAttributes()
29 return INDs
```

Thus, we need at most $max(n_d t_d, n_r t_r)$ comparisons. Assuming $n_d t_d >> n_r t_r$, we need $O(n_d t_d)$ comparisons.

We assume the representation of sets matchedCurrent and refs as follows: For storing the included referenced attribute objects we use a bit array with one bit per referenced attribute (i. e., of length n_r). Additionally we store the key and suffix length in a "bucket" per referenced attribute: Each bucket is stored as two corresponding arrays – one bit array for key length of length l_r and one array storing the corresponding suffix length.

Adding a referenced attribute object to set matchedCurrent (after a found match) is then a simple bit-wise OR operation ($O(1)$). Adding the key and suffix length costs also $O(1)$. As these update needs to be done at most in each step of LinkFinder we need $O(n_d t_d)$ bit-wise operations.

We need to intersect sets matchedCurrent and refs to update set refs and remove references from matchedCurrent when we move the cursors in dependent attribute objects. Intersecting both lists is a bit-wise AND on the bit arrays representing the referenced attribute objects ($O(1)$). Updating each key lengths array needs also a bit-wise AND, as there are n_r key lengths arrays we need $O(n_r)$. Updating the suffix lengths arrays needs $O(l_r n_r)$ comparisons and value updates. For removing references from matchedCurrent we need the common prefix length, i. e., one comparison. We update the key lengths arrays using a key lengths array with bits set to 1 for length lower or equal the identified common prefix length or set to 0 otherwise. Intersecting the key length buckets needs $O(r_n)$ bit-wise AND operations, updating the suffix lengths again $O(l_r n_r)$ value updates. As these updates also need to be done at most in each step of LinkFinder we need overall $O(n_d t_d l_r n_r)$ operations for this task.

As we assume the effort for comparisons much larger than the effort of bit operations, and as the comparisons for the testing part are lower than the comparisons to sort the data we expect the time to sort the data to dominate the overall runtime of LinkFinder.

5.3.4 Extending LinkFinder

So far LinkFinder enables discovering exact suffix INDs. Now we want to extend the algorithm to also discover prefix INDs as well as approximate suffix and prefix INDs.

Discovering prefix INDs. LINKFINDER uses the order on dependent and referenced attribute's values. Obviously, it is easier to compare if two values share a common prefix, which is necessary for suffix IND discovery. Deciding if two values share a common suffix is considerably more difficult – at least if we use the same order of values. But we can easily reduce the problem of discovering prefix INDs to discovering suffix INDs. We need to use the reverse values as input for LINKFINDER: Then all values are ordered starting from their last character, and deciding if one value is a suffix of another reduces to deciding if a value is a prefix of this other value. This means, we can apply LINKFINDER without any changes to discover prefix INDs– just by using reverse values.

Discovering approximate suffix and prefix INDs. We already extended SPIDER to discover approximate INDs as in real world their are dirty data, which necessitate to weaken the IND requirements. For LINKFINDER we face a similar situation: Remember we aim to discover links between data sources. Due to different versions of the data sources and due to data sources referencing several data sources from only one attribute we need to enable discovering approximate suffix and prefix INDs. Remember that the amount of allowed violating values, i.e., the error rate, is defined for approximate suffix and prefix INDs analogously to approximate INDs (see Sec. 2.2).

We can achieve this goal by extending LINKFINDER analogously to extending SPIDER for approximate IND discovery: We hold a counter for each reference stored in set refs. When intersecting refs with matchedCurrent to union all information on matches for all previous values with the current value of a dependent attribute (Alg. 3, line 17) we only refuse references with an error counter larger than the threshold. This way, we enable discovery of approximate suffix and prefix INDs.

5.3.5 Evaluating LINKFINDER

We evaluate LINKFINDER using our life sciences data sources CATH, SCOP, UniProt, and PDB. As we are interested in cross-references between data sources we used all attributes as potential dependent attributes and all accession numbers (identified in Sec. 4.2.3) as potential referenced attributes.

Identified Suffix and Prefix INDs. First, we look at identified cross-references to PDB. In Sec. 4.2.3 we identified three potential accession numbers, namely `struct_keywords.entry_id`, `exptl.entry_id`, and `struct.entry_id`. As all three contain the same data, we list only identified suffix and prefix INDs to `struct.entry_id`. We discovered approximate suffix INDs using LINKFINDER. We observed that all identified suffix INDs with an error rate below 25 % are real cross-references.

We identified three suffix INDs from CATH to PDB:

- `domain_list.domain_name` $\subseteq_s^{4,2}$ `struct.entry_id` with an error rate of 7.2 %,
- `chain_list.domain_name` $\subseteq_s^{4,2}$ `struct.entry_id` with an error rate of 4.5 %, and
- `names.repr_protein_domain` $\subseteq_s^{4,2}$ `struct.entry_id` with an error rate of 3.9 %.

All three are correct: Attribute `domain_name` is built using a PDB accession number concatenated with a suffix representing the domain of this protein (assigned by CATH). The first two relations, i. e., `domain_list` and `chain_list`, represent different classes of classified proteins. The third used relation `names` represents all these classifications with an example protein, namely `repr_protein_domain`. In summary, we found the expected cross-reference from CATH to PDB.

From SCOP we actually found an unexpected, yet correct cross-reference with an error rate 14.4 %: `description.description` $\subseteq_s^{4,\cdot}$ `struct.entry_id`. Indeed, the major part of values in `description.description` reference an accession number of PDB with a short comment. We also found the expected cross-reference from SCOP to PDB: `classification.pdb_id` $\subseteq_s^{4,0}$ `struct.entry_id` with error rate 0.4 %. As the suffix length is zero we found an actual approximate IND, which are of course a special case of approximate suffix INDs.

In UniProt (parsed into schema BIOSQL) we identified accession number `sg_bioentry.accession`. We found three to us previously unknown, but plausible suffix INDs from PDB to UniProt:

- `struct_ref.pdbx_db_accession` $\subseteq_s^{6,\cdot}$ `sg_bioentry.accession` with an error rate of 39.6 %,
- `struct_ref_seq.pdbx_db_accession` $\subseteq_s^{6,\cdot}$ `sg_bioentry.accession` with an error rate of 45.6 %, and
- `struct_ref_seq_dif.pdbx_db_accession_code` $\subseteq_s^{6,\cdot}$ `sg_bioentry.accession` with an error rate of 40.9 %.

These three relations are used in PDB to reference external databases. The high error rate stems from many differently structured values, which are cross-references

to yet other data sources.

Efficiency of LinkFinder We use all attributes of our life sciences data sources CATH, SCOP, UniProt, and PDB (except the extremely large relation `ATOM_SITE`) as potentially dependent attributes, and the identified accession numbers as referenced attributes. Thus we test 6,656 suffix and prefix IND candidates over 1,445 potentially dependent attributes and 5 potentially referenced attributes with a dataset size of 1.9 GB.

We vary the error threshold for discovering approximate suffix and prefix INDs to vary the number of identified INDs. In each run we sort, read, ship each attribute's data from the database and write them to disk, as for SPIDER. Additionally, we reverse each value, sort the attribute's values, and write them to disk. In this way, we are able to discover suffix and prefix INDs in one run.

Figure 5.3 shows runtime and number of discovered suffix and prefix INDs for this set-up. The runtime increases slowly as expected from our complexity analysis, because more values must be compared to confirm or (in most cases) dismiss a suffix or prefix IND candidate. Note, that even for an error threshold of 99 % and 100 % (i. e., the last two data points in the diagram) the runtime increases only slowly – as opposed to the number of discovered suffix and prefix INDs.

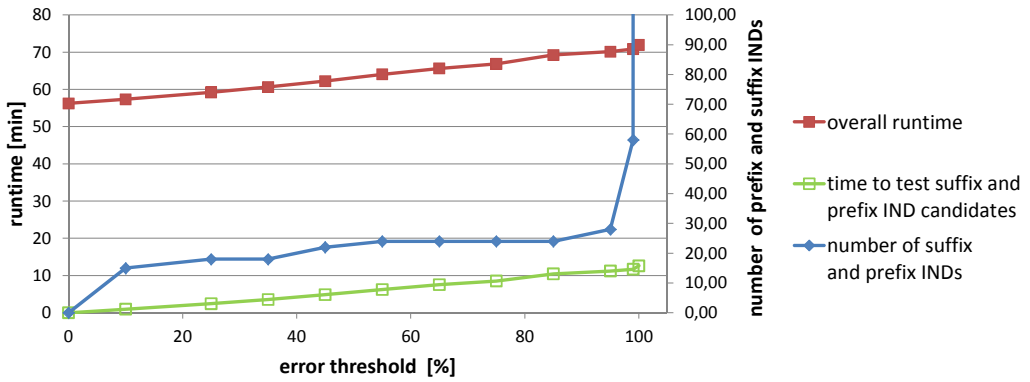


Figure 5.3: Runtime and number of identified approximate suffix and prefix INDs of LinkFinder over varying error threshold.

The overall runtime splits up as follows: Sorting, reading, shipping each attribute's data from the database and writing them to disk took in average 40 min for each run. Reversing each value, sorting the attribute's values (in main memory),

and writing them to disk to enable prefix IND discovery took additional 15.6 min. Thus, we confirm the decision to test suffix and prefix INDs in one run. The time to test the suffix and prefix INDs dominates the increase of the overall runtime, as can be seen in Fig. 5.3 (red and green line, left axis).

Interestingly, the number of identified suffix and prefix INDs does not increase largely with increasing error thresholds: For an error threshold of 99 % we only identify 58 suffix and prefix INDs, only for an error threshold of 100 % we confirm all 6656 suffix and prefix IND candidates. The reason are the restrictive characteristics of accession numbers as combination of numbers and values (mostly with character or number for a fixed position). For example, PDB accession numbers are 4 characters long enabling at most $36^4 = 1,679,616$ different values. In our data set we found only 32,485 different PDB accession numbers, i. e., only 2 % of all possible accession number values were used. Thus, the probability to find a match by accident is very low. This feature of accession numbers is very useful for suffix and prefix IND discovery, because it allows to set higher error thresholds without losing confidence in the identified results.

In an additional experiment we used the same data set as above, but enlarged the number of potential dependent attributes, which in turn enlarges the number of suffix and prefix IND candidates. We used all 26 identified accession number *candidates* from Sec. 4.2.3 and an error threshold of 25 %. Thus, the time to prepare the data structure (i. e., read, sort, reverse the data) remains the same. The time to test the suffix and prefix IND candidates increases from 3 min to 4.7 min, the number of identified suffix and prefix INDs from 15 to 43. These results show that the runtime increases only slightly compared to the increased number of suffix and prefix IND candidates – which again confirms our complexity analysis.

5.3.6 Related Work

We are not aware of related work close to our (indeed very special) problem of suffix and prefix IND discovery.

Warren and Tompa [66] propose an approach to discover composite matches between two schemas: Several attributes of relation A map to one attribute of relation B. The algorithm chooses attributes of A that should be used and determines their order. Concatenating the values of attributes in A provides the value in B. But we do not have two columns - one for the referenced accession number and an-

other for the suffix (or prefix) in the cross-referenced data source – which could be concatenated to match the referencing attribute.

There are several approaches of learning and leveraging regular expressions from data, e. g., [18, 48]. We also followed the idea of learning a regular expression for accession numbers of a given data source. But we found there are too many patterns for each of our data sources with overlaps to other data source's "accession number patterns". Thus, we did not follow this idea further, but instead proposed our approach of suffix and prefix INDs.

If we expected only a small set of fixed prefixes (or suffixes) that must be removed from the dependent attribute's values such that the IND holds, we also could use keyword trees [35]. Reading the potentially dependent values in a keyword tree we would find the prefixes as several branches starting from root node with broader splits in branches for the actual accession numbers. After removing the prefixes we could test the prefix IND candidates using SPIDER. But following this approach would exclude discovering suffix INDs, such as CATH referencing PDB with PDB accession numbers concatenated with an assigned domain, or SCOPs "cross-reference" to PDB using its description attribute.

Chapter 6

Discovering Conditional Inclusion Dependencies

In this chapter we provide two algorithms to discover conditional inclusion dependencies (CINDs). We focus on the definition and discovery of "good" conditions. We first regard the requirements of CIND discovery in Section 6.1: We identify and motivate features of conditions and reveal the need to measure the quality of these features. We further show the challenges of condition discovery and formulate our problem statement. In Section 6.2 we formally define the condition features and and their measures from Sec. 6.1. We provide an SQL approach for condition discovery in Section 6.3, and our efficient algorithms CINDERELLA and PLI in Sections 6.4 and 6.5.

We show the relevance of our approach using the application of discovering candidates for missing links in linked open data as described in Sec. 1: We use the German and the English DBpedia data for persons [15]. The idea is to look for characteristics of persons in the German DBpedia that are also included in the English DBpedia, and vice versa. The discovered conditions will probably also match some persons that are not (yet) included in the other data set. We suggest these persons as good candidates for missing links.

We use the following relational schema to represent information about persons:

- `person(pid, cent, description)`,
- `birthplace(pid, bplace)`,
- `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 6.1 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. 6.1 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 those conditions within Person_EN that summarize properties of 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.

6.1 Requirements of CIND discovery

We approach the problem of CIND discovery in three steps: (i) detecting an approximate IND, (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 approximate SPIDER (see Sec. 5.2), or it could be manually performed by an expert user. The problem of finding an optimal pattern tableau has been addressed for CFDs in [32]. Here we assume approximate INDs to be given and focus on the second step, namely on efficiently detecting "good" conditions that turn given approximate INDs into CINDs. We outline in related work how the third step can be realized by applying our algorithms to the ideas of [32].

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`

Figure 6.1: 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.

6.1.1 Features of Conditions

To achieve the goal of identifying good conditions, we need to formulate desired features of conditions. In the following, we reason over conditions and their features. Given an approximate inclusion dependency $R_1[X] \sqsubseteq R_2[Y]$ between attributes X in relation R_1 and attributes Y in relation R_2: A condition over the dependent relation R_1 should distinguish those tuples of R_1 that are 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. 6.1 shows that simply relying on counting the number of tuples that match a condition may not give the desired results. In our 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).

6.1.2 Quality of Conditions

Our use case requires to find valid and covering conditions only with a certain quality; we search not only 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. 7). 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. 6.2).

6.1.3 Challenges of Condition Discovery

Discovering valid and covering, or valid and complete 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 thesis, 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 manually specify the set of attributes over which the condition is generated.

Recall from the definition of CIND (including the definition of pattern tableau T_P) in Section 2.2 that a CIND 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.

Thus, we propose our problem statement for condition discovery as follows:

Problem Statement 5: Condition Discovery Given an approximate IND $R_1[A] \sqsubseteq R_2[B]$, two instances I_1, I_2 of R_1, R_2 respectively, and quality thresholds for validity and covering (or completeness); find all selecting conditions over an arbitrary subset of attributes in R_1 (of arbitrary size) that satisfy the given quality thresholds. □

6.2 Classifying CINDs

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.

6.2.1 Defining Condition Features

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 in g_x 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.

Def. 12: Valid Condition A condition is *valid* if all tuples of I_1 that match t_p also satisfy the embedded IND, i. e., $I_1[t_p] \subseteq I_\varphi$.

□

Def. 13: Completeness Condition A condition is *complete* if it matches all included tuples, i. e., $I_\varphi \subseteq I_1[t_p]$.

□

Def. 14: Covering Condition A condition is *covering* if it matches all included groups, i. e., $G_\varphi \subseteq G_1[t_p]$.

□

6.2.2 Measuring Condition Features

One will rarely find 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 quality 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 in relation to the number of all matching tuples:

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

A validity of 1 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 quality measure of this feature for groups: Consider a condition `cent = 18` for persons in `Person_DE` to be included in `Person_EN` (see Fig. 6.1). Counted over tuples, this condition would be 0.8-valid, but over groups (or persons) it is just 0.67-valid. That is, the three 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 in relation to the number of all included tuples:

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

A completeness of 1 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 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 in relation to the number of all included groups:

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

A covering of 1 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 confidence for the rule "If a group is an included group, then the condition matches at least one tuple in this group".

6.3 Discovering Restricted Conditions with SQL

For a given approximate IND and a set of condition attributes, we can use SQL to detect conditions and their quality measures. 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. To discover *all* conditions these 3 steps can be repeated for each subset of attributes (used as condition attributes).

lhs inclusion attribute	condition attributes		rhs inclusion attribute
Person_DE.personID	**Person_DE.**		**Person_EN.personID**
	cent	**deathplace**	
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

Figure 6.2: Left outer join over relations `Person_DE` and `Person_EN` as given in Fig. 6.1, projected on inclusion and condition attributes and grouped by condition attributes. URL-specific parts of values are omitted for readability.

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 `Person_DE` (see Fig. 6.2). `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. 6.3a 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 `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 6.3b 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` and `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 completeness_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

Figure 6.3: SQL statements to find conditions and their quality measures for embedded IND `Person_DE` $\sqsubseteq$ `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. In our DBpedia 3.6 persons data set (see Sec. 2.3) 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 on a commercial DBMS. Assuming this as the average runtime, the estimated runtime for all combinations is about 80 min and 6 h40 min, respectively.

In the next two sections we describe efficient 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. 6.6. 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). 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 6.4 shows this representation for the embedded IND `Person_DE.pid` $\subseteq$ `Person_EN.pid`.

6.4 Discovering General CINDs With CINDERELLA

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)". We first apply this idea to detect γ-valid$_g$ and λ-covering conditions, and modify it in Section 6.4.2 to also detect γ-valid and δ-complete conditions.

There are two challenges: (i) mapping the problem of condition discovery to association rule mining and (ii) improving efficiency based on characteristics of condition discovery. 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. 6.4. Note that we only need the right-hand side inclusion indicator and the

	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

Figure 6.4: Left outer join over `Person_DE` and `Person_EN` in Fig. 6.1; Modified representation handles multiple occurrences of one attribute for a single person. URL-specific parts of values are omitted for readability.

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 attribute identifier. Using prefixes A to D for our example yields the following basket for the first group of Fig. 6.4: { 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 [2] 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, namely 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 deLimited Apriori (CINDERELLA) in the next section.

6.4.1 The 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 [62]. 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 4 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. 5) to itemset candidates C_k and testing them afterwards. The algorithm stops if L_{k-1} is empty.

Algorithm 4: 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 `*/`
1 $L_1 = \{\text{frequent 1-itemsets}\}$;
2 $L_2 = \{(\text{INCLUDED}, l_1) \mid l_1 \in L_1\}$;
3 **for** *k=3;* $L_{k-1} \neq \emptyset$*; k++* **do**
4 $C_k = \text{aprioriGen-Constrained}(L_{k-1})$;
5 **foreach** *basket* $b \in$ *baskets* **do**
6 $C_t = \text{subset}(C_k, b)$;
7 **foreach** $c \in C_t$ **do**
8 c.count++;
9 $L_k = \{c \in C_k \mid c.\text{count} \geq \lambda * |\text{baskets}|\}$;
10 **return** $(\cup_k L_k) \bigcup L_2$;

Method `aprioriGen-Constrained` (Alg. 5) 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.

The rule generation step uses the identified frequent itemsets and computes the validity of conditions: The number of included tuples matching the condition is the number of occurrences (support) of a frequent itemset; the number of all tuples

Algorithm 5: 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} ∧
5                 p.item_{k-1} < q.item_{k-1} ;
6  foreach candidate c ∈ C_k do
7     foreach (k-1)-subsets s of c containing item INCLUDED do
8        if s ∉ L_{k-1} then
9           delete c from C_k ;
10 return C_k ;
```

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.

6.4.2 Discovering 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.

6.5 Discovering General CINDs with PLI

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 for lower numbers of attributes more memory efficient than CINDERELLA because of its depth-first approach. The PLI algorithm is a modification of the algorithm in Ziawasch Abedjans master's thesis [1]. We adapted the algorithm to discover conditions [6, 7].

The position list representation of values has also been applied by the algorithm TANE for discovering functional dependencies [41]. 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.

6.5.1 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 [41]) 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. 6.4) 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 δ-valid$_g$ conditions.

Table 6.5 illustrates the position lists for the attributes `cent` and `deathplace` from the example in Fig. 6.4. The frequency of each value is implicitly 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 an additional 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 6.5.

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

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	$\{3, 4\}$

Figure 6.5: Position lists of attributes `cent`, `deathplace`, `en.pid` for the example in Fig. 6.4 (i. e., left outer join over relations `Person_DE` and `Person_EN` of our example).

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 $\{1, 2\}$. Altogether the cross-intersection forms eight intersections of which four are empty.

6.5.2 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 idea of PLI is twofold: (i) We use a special position list for included groups, i. e., *includedPositions*. (ii) We cross-intersect position lists of attributes to test value combinations (i. e., conditions) for the intersected attributes, e. g., intersect each position list of attribute A with each position list of attribute B. The covering of a condition then corresponds to the ratio of the cardinality of its position list P intersected with *includedPositions* to the cardinality of *includedPositions* ($|P \cap \text{includedPositions}|/|\text{includedPositions}|$). The validity of a condition corre-

sponds to $(|P \cap \text{includedPositions}|/|P|)$.

The algorithm is based on two 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. This way it is assured that the maximum number of concurrent required sets of position lists in memory is bounded to twice the number of the size of X_p.

Position list retrieval. The algorithm needs to scan the table once for retrieving position lists of each potential condition attribute. In addition, the 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 long 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 6) 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.

Algorithm 6: 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, getPositionLists (i), ∅, ∅) ;
3 return conditions ;
```

Function *analyze* is shown in Algorithm 7. The method traverses the power-set lattice of the condition attributes depth-first. Its parameters are the currently added attribute (*attrNo*), the position lists of the attribute (*attrNoPLs*), the 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 6-11). 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 long 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

Algorithm 7: 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 (attrNoPLs, combPLs);
4  else
5      newCombPLs ←attrNoPLs;
   /* Check each position list.                                         */
6  foreach pl ∈ newCombPLs do
7      plIncluded ← (pl ∩ includedPositions) ;
       /* Compute measures of covering, valid_g.                        */
8      if |plIncluded| / |includedPositions| > λ then
9          coveringCombPLs.add (pl) ;
10         if |plIncluded| / |pl| > γ then
11             conditions.add( newCombAttrNos, pl.values (), covering, valid) ;
12 if ¬ coveringCombPLs.isEmpty () then
13     for i = attrNo + 1 to |attributes| do
14         conditions.addAll (analyze (i, getPLs (i), newColCombs,
           coveringCombPLs ))) ;
15 return conditions ;
```

attribute was the last attribute. In both cases, the set of generated conditions is returned.

6.5.3 Discovering 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 trivially be adapted by using tuple-IDs of the relational representation instead of group-IDs as positions.

6.6 Complexity of CINDERELLA and 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 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 complete 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 for lower numbers of attributes, 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 $\binom{n}{l}$ generated candidates of each level l in memory. In the worst case, the number of candidates corresponds to $\binom{n}{\frac{n}{2}}$ 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 $O(\binom{n}{\frac{n}{2}} \cdot v^{\frac{n}{2}} \cdot \frac{n}{2} + v \cdot n)$. As a position list requires (in most cases) more memory than an itemset, PLI should outperform CINDERELLA with regard

to memory usage for lower numbers of attributes, because of its quadratic complexity compared to CINDERELLA's exponential complexity. For larger numbers of attributes CINDERELLA outperforms PLI. In Section 7.1 we confirm our complexity analysis using actual experimental results.

6.7 Related Work

Conditional inclusion dependencies (CINDs) were proposed by Bravo et al. [19] for data cleaning and and contextual schema matching. In [19], 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, however, is not addressed.

Different aspects of CIND discovery have been addressed in [23, 33, 53]: De Marchi et al. propose data mining algorithms to discover approximate INDs, i. e., INDs that are satisfied by part of a given database [53]. 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 (see Sec. 5.2). Approximate INDs are input to our CIND discovery algorithms. The work on discovering approximate INDs is therefore orthogonal to our work on CIND discovery.

Algorithms for generating pattern tableaux for given INDs are proposed in [23, 33]. The algorithm in [23], 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 [33]. Given an approximate IND, the system generates a pattern tableau that is 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 pattern tableau 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 [33] is an extension of the algorithm proposed in [32] for generating pattern tableaux for conditional functional dependencies (CFDs). CFDs were

introduced in [27] 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 [32] 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 [32]. To regard marginal local support and confidence defined in [32], 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 [22, 28]. In contrast to other approaches, the work in [22] 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 [22] can be applied to CIND discovery. Fan et al. propose an algorithm for discovering constant CFDs based on closed itemset mining [28]. 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 [28] 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 [49, 62]. These algorithms were the motivation for our's in Sections 6.4 and 6.5.

Contradiction pattern are also a form of association rules with fixed antecedent [55]. 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 [55].

Chapter 7

Evaluating and Leveraging Conditional Inclusion Dependencies

In this chapter we evaluate the discovery of conditional inclusion dependencies. We evaluate the efficiency of our algorithms in Section 7.1 using our DBpedia persons data sets and using generated data sets. In Section 7.2 we evaluate the effectiveness of CIND discovery using three applications: Our DBpedia persons use case, a Wikipedia Images use case used in [33], and a life science use case.

7.1 Efficiency of CIND Discovery

We set up two experiments on the DBpedia 3.6 person data sets (see Sec. 2.3) to evaluate (i) the effect of the number of conditions to be identified, and (ii) the effect of the number of tuples in the dependent data set. Further, we set up experiments on generated data, which allowed us to vary the number and distribution of condition attributes over all attributes. We evaluated (i) the effect of the number of attributes (regardless if these are used as conditions or not) and (ii) the effect of the distribution of the condition attributes over all attributes. We measure runtime and memory consumption in all 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.

7.1.1 Varying the Number of Conditions

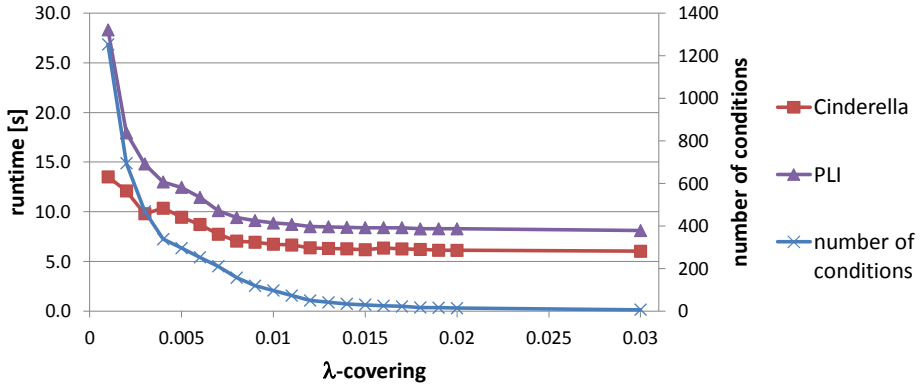
In this experiment we want to test the effect of the number of conditions to be identified. We use the German DBpedia person data set and its included persons in the English DBpedia. There are 296,454 persons in the English DBpedia 3.6 and 175,457 persons in the German DBpedia 3.6; 74,496 persons are included in both data sets. We mapped these data sets into relations (as described in Sec. 2.3) containing 474,630 tuples for the English DBpedia and 280,913 tuples for the German DBpedia with an intersection of 133,208 tuples.

We vary the number of identified conditions by varying λ or δ for detecting covering or completeness conditions, respectively.

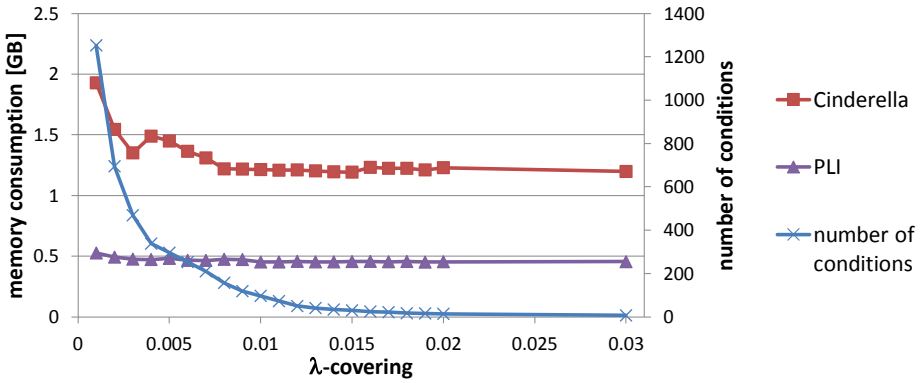
Figures 7.1(a) and 7.1(b) show the runtime and memory consumption for varying λ, i. e., for detecting covering conditions. In both diagrams we also show the number of identified conditions using a secondary y-axis on the right. The runtime of both algorithms correlates with the number of identified conditions, 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 analysis in Sec. 6.6.

Experiments on completeness conditions reveal equivalent results as can be seen in Fig. 7.2. In comparison to discovering covering conditions, the number of identified conditions increases 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 groupIDs. Again Cinderella increases less in runtime while PLI increases less in memory consumption, as expected by our complexity analysis.

Due to Pli's 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



(a) runtime



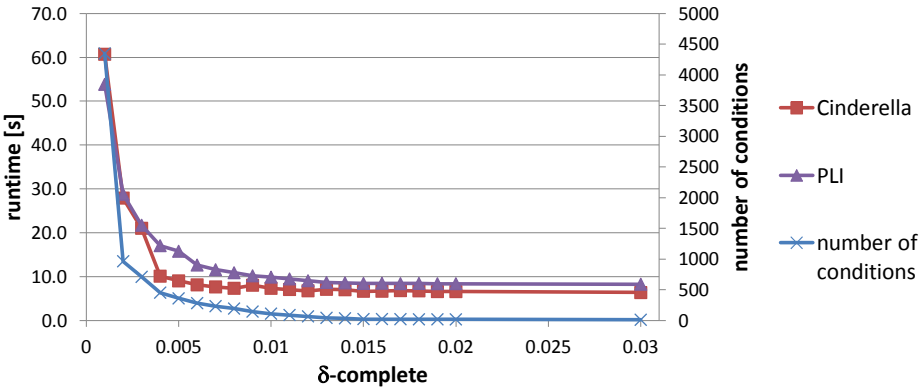
(b) memory consumption

Figure 7.1: Results for varying covering thresholds.

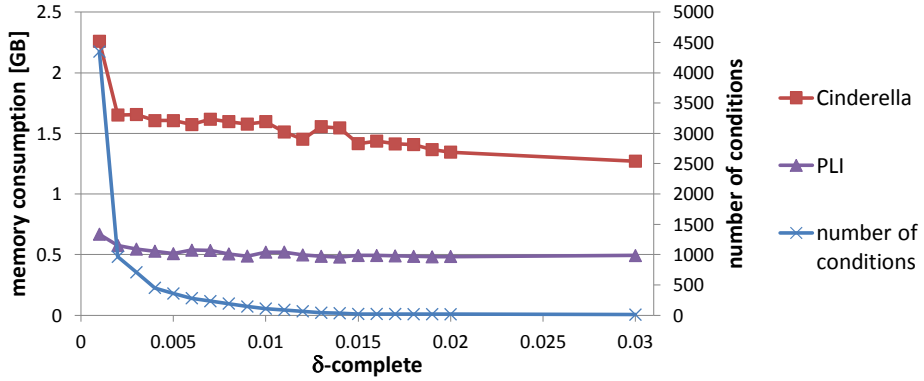
using 5.8 GB memory and delivering 9,214,406 conditions. With CINDERELLAs higher memory requirements, we could perform the same experiment only up to at least 10 covered persons. The run took 3 min20 s using 7.8 GB RAM and returned 12,587 conditions. Clearly, this entire set of conditions is not useful to find individual interesting conditions. But we can use it for a scatter plot as given in Fig. 7.7 in Sec. 7.2 (evaluating the effectiveness of CIND discovery): The scatter plot gives an intuition about the distribution of conditions and helps to set profitable thresholds.

7.1.2 Varying the 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 the ratio of included and non-included groups (or tuples). Therefore, we concatenate multiple instances of the German



(a) runtime



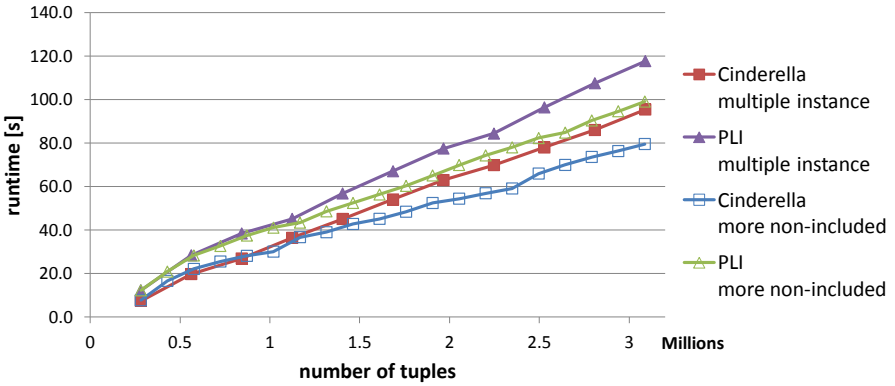
(b) memory consumption

Figure 7.2: Results for varying completeness thresholds.

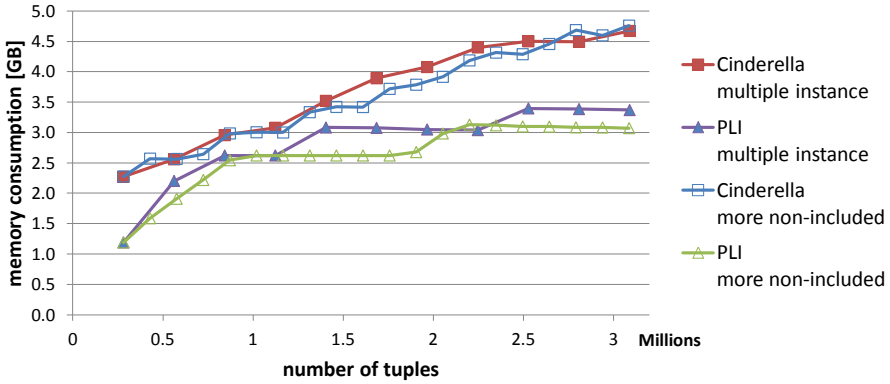
DBpedia data set. To increase the absolute size of the data set with a constant ratio of included-to-non-included tuples we consider multiples of the entire data set. A second class of data sets is created by adding multiples of non-included persons to decrease the ratio of included-to-non-included persons. 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 threshold, because it relates to the constant number of included persons.

Figures 7.3(a) and 7.3(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

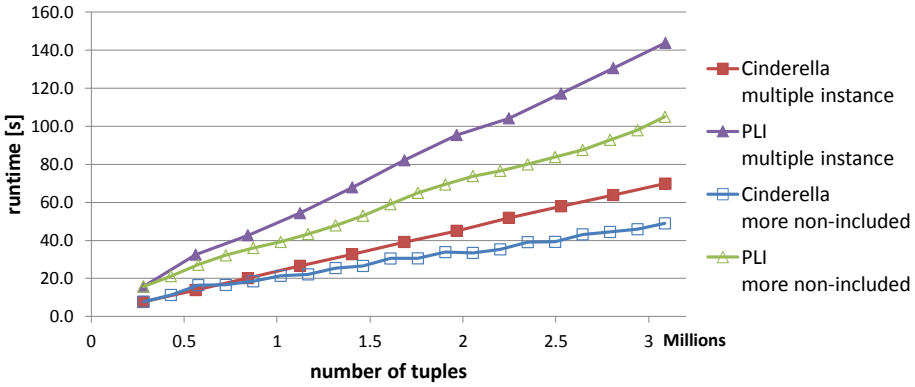


(b) memory consumption

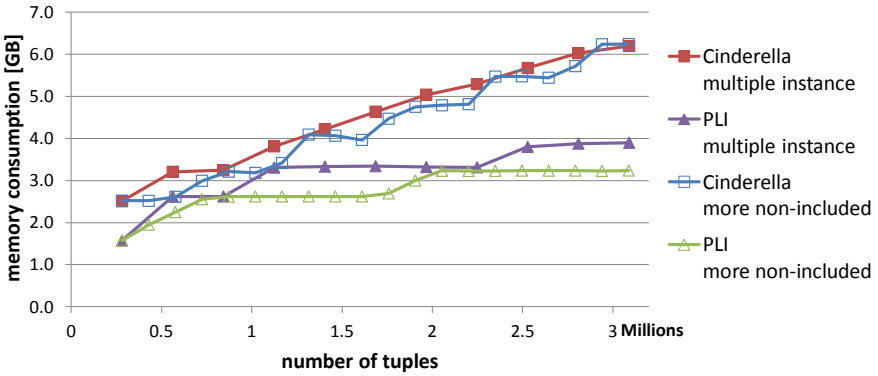
Figure 7.3: Results for discovering covering conditions over varying number of tuples.

Experiments for detecting completeness conditions using equivalent setups show comparable results (Fig. 7.4). All observations appear even stronger pronounced, which results again from the increased amount of work for discovering completeness conditions instead of covering conditions on the same data set.

Our experimental results confirm our comparison of the algorithms in Sec. 6.6, 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.



(a) runtime



(b) memory consumption

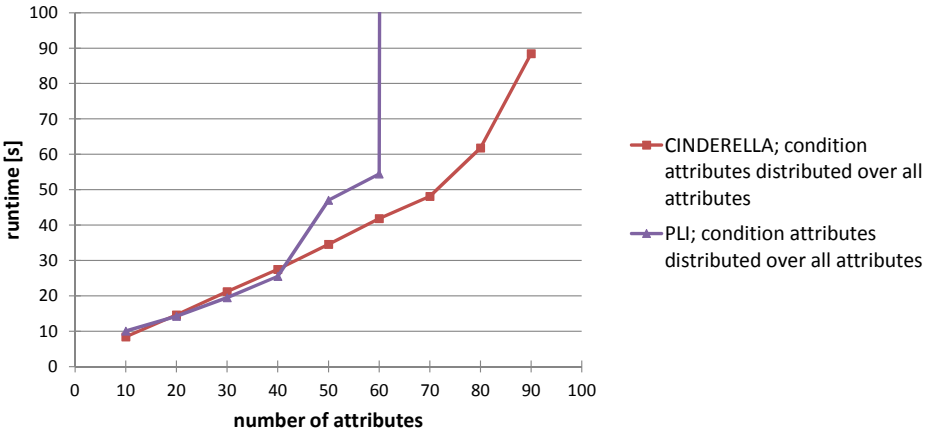
Figure 7.4: Results for discovering completeness conditions over varying number of tuples.

7.1.3 Varying the Number Of Attributes

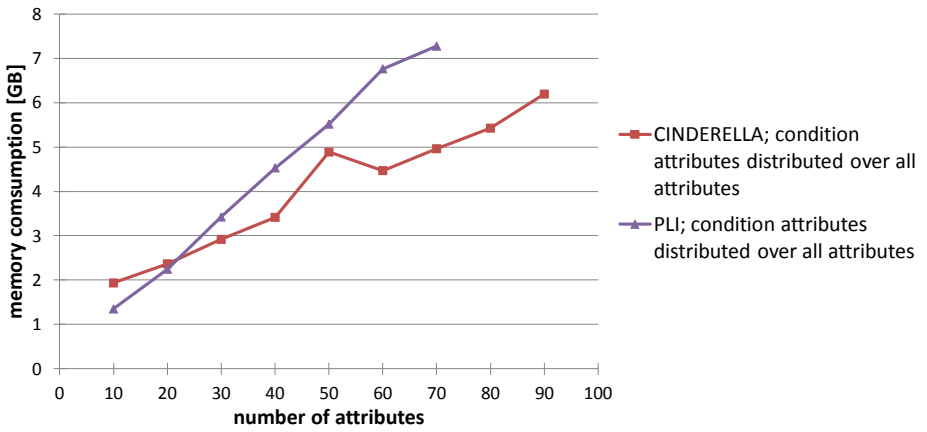
We evaluate the effect of the number of considered attributes. We generated data sets of 300,000 tuples with 150,000 included tuples over 20 attributes providing 5 conditions of size 3, 10 conditions of size 2, and 20 conditions of size one. Each 300 tuples build a group. We varied the number of attributes for each data set from 10 to 90 and distributed the conditions over all attributes of these data sets. Note that each condition of size 3 implies three further conditions of size 2 and three conditions of size 1 with the same or higher validity and covering measures. Thus, our set-up defines overall 85 conditions.

Figure 7.5 shows runtime and memory consumption for CINDERELLA and PLI. The runtime increases – as expected – with the number of attributes in a data set.

But CINDERELLA is much less sensitive to increasing numbers of attributes than PLI; actually the run of PLI over the data set of 70 attributes took 1h 45min compared to CINDERELLA with 48s. This observation confirms our complexity comparison of both approaches: PLI cannot prune the search space as drastically as CINDERELLA, which results in the longer runtime of PLI.



(a) runtime



(b) memory consumption

Figure 7.5: Results for discovering covering conditions over varying number of attributes.

The memory consumption also increases with the increasing number of attributes. PLI needs less memory than CINDERELLA for the first two data sets, i. e., for 10 and 20 attributes. For larger data sets PLI consumes more memory than CINDERELLA. This observation also confirms our complexity estimation: The increasing number of attributes requires more position lists, which are the important factor for PLI.

On the other hand, CINDERELLA stores all data once and additionally the frequent itemsets, which are constant over all generated data sets for itemset size larger than one.

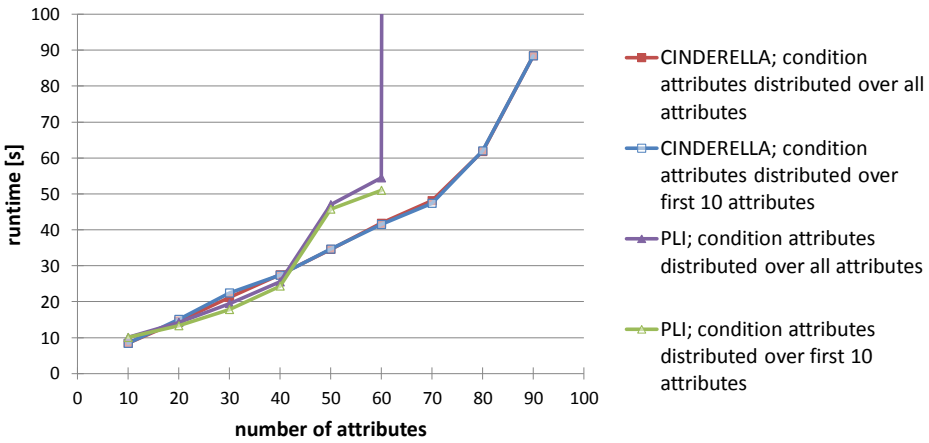
7.1.4 Varying Distribution of Condition Attributes

This experiment evaluates the distribution of the actual condition attributes over all considered attributes: Is the amount of work smaller if a constant number of conditions covers less condition attributes? We reused the generated data sets from the previous experiment. Additionally we generated data sets of the same numbers of tuples, groups, and conditions, but used only the first 10 attributes as condition attributes. That is, the algorithms must consider more attributes – without finding any conditions on these attributes.

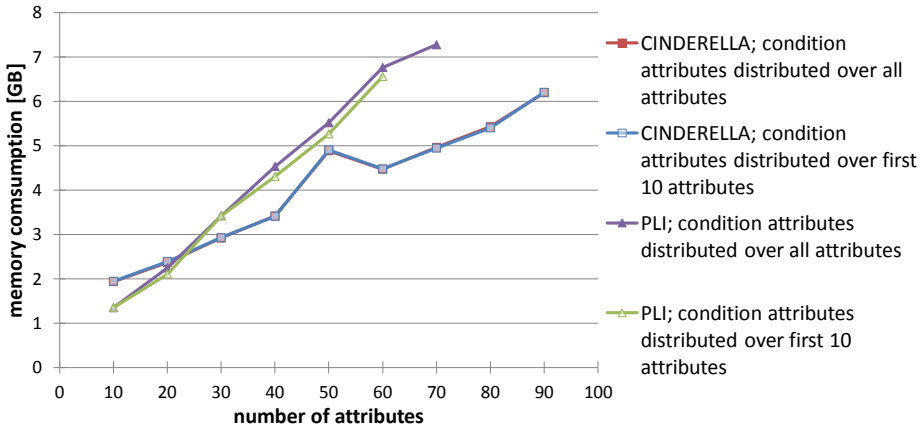
Figure 7.6(a) shows the runtime and Figure 7.6(b) the memory consumption for CINDERELLA and PLI for both set-ups. Surprisingly – at least on first sight – the runtime and memory consumption do not differ for both set-ups.

On second sight, the decisive factor for both measures is the number of candidates that must be checked – not the number of attributes. The number of candidates are the combinations of (sets of) attributes and their values. For CINDERELLA these candidates are the built and tested itemsets, for PLI the position list intersections. This number of candidates does not vary for both set-ups, which in turn results in the same runtime and memory consumption.

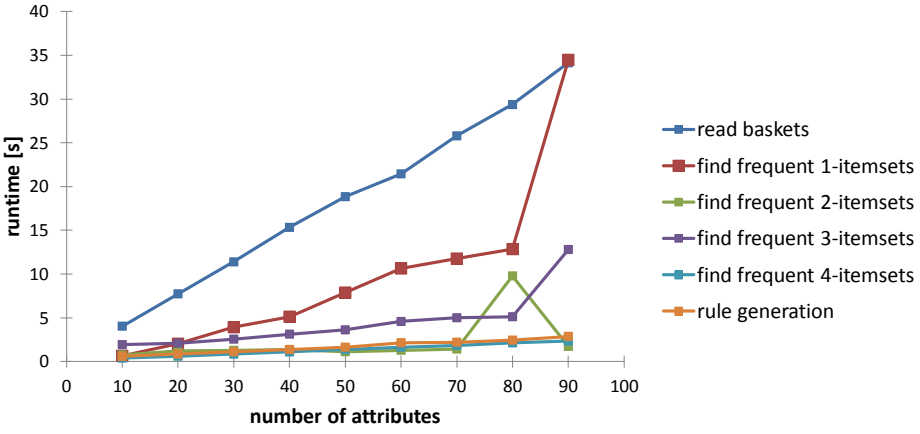
Figure 7.6(c) confirms this explanation using the example of CINDERELLA's runtime: We split up the runtime in its individual parts. The larger amount of attributes and values, i. e., the larger baskets, increase the time for reading the data and building the baskets. The time to create the one-itemsets also increases recognizably as we cannot prune in this step. The runtime for the following steps does not increase considerably, because of the pruned search space. This observation exactly fits the expectation from our explanation.



(a) runtime



(b) memory consumption



(c) runtime of CINDERELLA split up into individual parts

Figure 7.6: Results for discovering covering conditions over varying number of attributes with varying distribution of condition attributes.

7.2 Effectiveness of CIND Discovery

In this section we evaluate the effectiveness of our approach using three applications: Our DBpedia persons use case, a Wikipedia Images use case, and a life science use case.

7.2.1 Evaluating the DBpedia Persons Use Case

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 7.7 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.

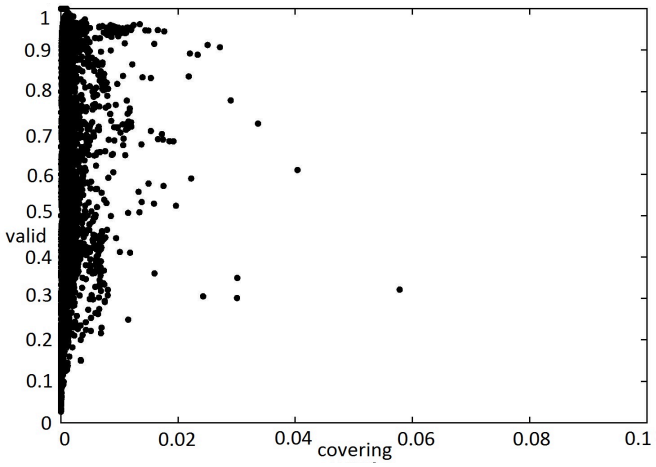


Figure 7.7: Identified conditions.

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 among the 85 conditions with $\gamma\text{-valid}_g > 0.84$ with the largest covering measure are `description = American actor`[1] ($\gamma\text{-valid}_g = 0.91$, λ-covering $= 0.029$, i. e., 2173 persons) or `description = American actress` ($\gamma\text{-valid}_g = 0.89$, λ-covering $= 0.024$, i. e., 1791 persons). We also found both conditions in conjunction with the condition `birthcentury = 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. Note that these conditions are non-trivial: Similar conditions that might be expectable were not found (i. e., confirmed). We found the following conditions:

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

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. 7.8: 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.

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

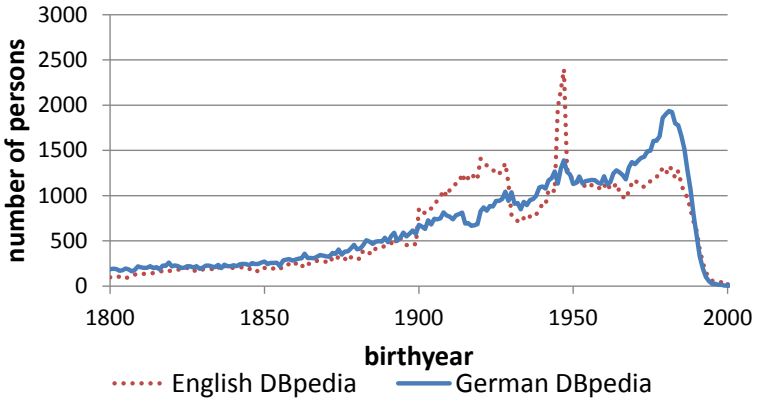


Figure 7.8: Persons per year of birth.

English DBpedia persons included in the German DBpedia. The validity of the empty condition is 0.25, i. e., 25% of persons in the English DBpedia are also included in the German 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, i. e., 1,820 persons) or `deathplace=Germany` (γ-valid$_g$ = 0.55, λ-covering = 0.011, i. e., 832 persons), 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 following surprising conditions: We already introduced in Chapter 6 the condition `deathplace = United States` $\wedge$ `birthcentury = 18` (γ-valid$_g$ = 0.51, λ-covering = 0.008, i. e., 888 persons), i. e., persons in the English DBpedia who were born in the 19th century and died in the United States are also included in the German DBpedia. This condition could be explained by the large number of European emigrants to the United States in the 19th century. We also found the unexpected condition `description = road bicycle racer` with γ-valid$_g$ = 0.66 and λ-covering = 0.012, i. e., 597 persons.

7.2.2 Evaluating a Wikipedia Use Case

A work closely related to ours is [33], 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. [33] 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 [33], except the one on `user_text = ProteinBoxBot`, which shows a lower validity in our experiments (γ-valid$=0.622$ in our experiments vs. 0.971 in [33]). 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 [33] (on presumably different hardware).

However, our algorithms also discover more detailed conditions, which cannot be found by [33]: 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 [33] 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 ran 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 completeness conditions satisfying the same validity requirements, which in turn provide a base to build better pattern tableaux. The authors of [33] 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).

7.2.3 Evaluating a Life Sciences Application: UniProt

As last use case we consider our test data source UniProt, which references 67 databases – including our test data source PDB. We already mentioned this use case in Section 1.1.

The BIOSQL schema, into which we parsed the UniProt data, defines relation `dbxref` with attributes `dbxref.dbname` and `dbxref.accessionnr` to store links to other data sources. Thus, `dbxref.accessionnr` is the dependent attribute of the approximate IND to PDB (which we are able to discover with approximate SPIDER).

We set up two experiments: First, we used only relation `dbxref` to discover conditions for the CIND. As attribute `dbxref.accessionnr` is a key in this relation we discover completeness conditions. Second, we joined relations of BIOSQL to `dbxref` – using the INDs discovered by SPIDER. We used this join results as input for condition discovery. In this case we must discover covering conditions, because `dbxref.accessionnr` is not a key on the join result.

In the first set-up we were able to discover two conditions: `dbname = PDB` and `dbname = PDB` $\wedge$ `dbxref_version = 0`. Both conditions are correct and provide the same measures for validity (0.976) and completeness (1.0).

The second set-up provides much more interesting insights: We identified 28 conditions within 74 s. We found condition `dbname = PDB` $\wedge$ `division = HUMAN` (and the more explicit condition `dbname = PDB` $\wedge$ `division = HUMAN` $\wedge$ `dbxref_version = 0` $\wedge$ `bioentry_version = 0` $\wedge$ `tax_oid = 2326040`) with γ-valid$_g = 0.96$, λ-covering $= 0.24$. Further we found condition `dbname = PDB` $\wedge$ `division = ECOLI` (and again the more explicit condition `dbname = PDB` $\wedge$ `division = ECOLI` $\wedge$ `dbxref_version = 0` $\wedge$ `bioentry_version = 0` $\wedge$ `tax_oid = 2336138`) with γ-valid$_g = 0.989$, λ-covering $= 0.12$. This observation means, about 35 % of all proteins in UniProt that reference PDB are proteins of Human or Escherichia coli: We are able to explain semantically

which proteins reference PDB and can therefore identify proteins that probably should cross-reference PDB.

Chapter 8

Conclusions

In this thesis we propose several approaches for dependency discovery, which aim to support data integration. We focus on discovering inclusion dependencies – in several variations – and conditional inclusion dependencies. We use several applications from data integration in the life sciences and from link discovery for linked open data to show the relevance of our approaches and to show their effectiveness and efficiency.

In particular, we propose SPIDER – an efficient algorithm for unary inclusion dependency discovery. SPIDER leverages the sorting capabilities of RDBMS and tests afterwards all IND candidates in parallel while saving computation. We compare SPIDER to several SQL approaches and a brute force approach providing a complexity estimation and an extensive experimental evaluation. The complexity of SPIDER depends only in the number of attributes, while other approaches depend on the number of IND candidates, i. e., the square of the number of attributes. For large schemas SPIDER outperforms previous approaches by an order of magnitude. This feasibility enables profiling entirely unknown data sources. It led to a cooperation with FUZZY! Informatik AG, a German company providing the data profiling tool FUZZY! DIME. The tool's IND discovery is based on an implementation of SPIDER.

We leverage discovered INDs for deriving foreign keys. We propose two approaches for filtering foreign keys from discovered INDs based on heuristics and machine learning. Both approaches reach F-measures of in average 93 %. Further we use discovered INDs to derive the primary relation of data sources – a domain specific property in life sciences data sources, which is used later on for discovering cross-references between data sources.

We extend SPIDER in three ways: (i) Composite SPIDER discovers composite INDs in a given schema by generating and testing composite IND candidates level-wise. It is especially suited for discovering composite INDs of lower levels, which makes it a good extension for related approaches targeting discovery of composite INDs of higher levels. (ii) Approximate SPIDER discovers approximate INDs, which are necessary for IND discovery on dirty data. We confirm SPIDER's efficiency also for approximate IND discovery. (iii) LINKFINDER discovers suffix and prefix INDs, which aim for discovering cross-references between data sources. LINKFINDER efficiently discovered previously known and even unknown, yet correct cross-references between our life sciences test data sources.

All these approaches support gathering information for data integration in the life sciences. We united the algorithms in our tool Aladin to enable discovering and leveraging data dependencies for data integration. In a first step unary, composite, approximate INDs, and accession number candidates can be discovered. Afterwards, foreign keys and the primary relation of a data source can be derived. In the third step, cross-references between data sources can be found using LINKFINDER or approximate SPIDER.

Future work on inclusion dependency discovery could investigate data sources on typical pattern of INDs to improve data profiling at all: Are there typical pattern of INDs in data sources? Are there domain dependent pattern? The algorithms proposed in this thesis for discovering INDs in a given data source provide the basis for this application. Another valuable direction for future work regards refining SPIDER for parallel, distributed algorithms to support IND discovery in further increasing amounts of data, up to Big Data. We propose two alternatives: (i) We split up the attributes' sorted values into blocks of data with minimum and maximum values and check the IND candidates over all attributes for each block in parallel. Only those INDs are satisfied that are confirmed in all blocks. The challenge is to define the thresholds to split the set of all attributes' values into equally sized blocks. (ii) We split up the set of attributes while assigning each attribute to several blocks. After checking all IND candidates in each block in parallel, the transitivity of INDs is used to confirm or refuse unchecked IND candidates. Remaining IND candidates must be checked afterwards. The challenge is to define the overlap between chunks of attributes that allows an overall speed-up.

For conditional inclusion dependency discovery we extend the definition of CINDs

to distinguish covering from completeness conditions. In this way, we enable new applications showing the value of CIND discovery. In the domain of linked open data we propose candidates for missing links, in the domain of life sciences we describe which instances reference other data sources providing a semantical explanation of the cross-references. We propose two algorithms for condition discovery: CINDERELLA is based on association rule mining; PLI is based on position list intersections. Both algorithms discover conditions conforming given quality thresholds without the need to pre-select condition attributes – as opposed to previous approaches. This feasibility enables discovering unexpected, yet useful conditions as we showed with our test data sources and with an application from related work. Our experimental evaluation showed that CINDERELLA is faster than PLI, while PLI needs less memory for smaller numbers of attributes (< 20 in our experiments).

Future work on CIND discovery should adapt the distinction of covering and complete conditions to demanding conditions, i. e., the right-hand side of the pattern tableau, to enable further benefits for link discovery in linked open data: Note that the known CIND definition matches only completeness demanding conditions. For instance, if there is a person in English DBpedia matching the selecting condition `birthplace = California`, we could require the referenced person in the German DBpedia to match `birthplace = Kalifornien`. But according to the CIND definition this person is not allowed to have also a triple stating `birthplace = United States`. Thus, we would need to modify the CIND definition for this adaptation. This modification promises valuable possibilities to leverage CINDs: The obvious use is to improve data quality, which would be enough justification in itself. But there is again another application in linked open data. Spoken in our DBpedia person's use case, we can so far identify persons that should probably have a link without saying which corresponding person should be linked at. Demanding conditions can provide a set of probably corresponding persons: Consider a selecting condition selecting included persons; these included persons reference their corresponding persons in the referenced relation. We can discover demanding conditions over these corresponding persons, e. g., using our algorithms. Persons that match these demanding conditions but are not (yet) corresponding persons should probably be linked at from non-included persons matching the selecting condition.

In summary, dependency discovery is a powerful tool for gathering necessary information for data integration – as we showed using our two motivating application

domains: Data integration in life sciences data sources and link discovery for linked open data. In today's steeply increasing amounts of data and data sources, effective and efficient data profiling techniques become even more important. This thesis contributes to this challenge providing several algorithms for the field of dependency discovery.

Bibliography

[1] Abedjan, Ziawasch: *Discovering unique column combinations within a database*, Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam, Master's thesis, 2010

[2] Agrawal, Rakesh ; Srikant, Ramakrishnan: Fast Algorithms for Mining Association Rules in Large Databases. In: *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*. San Francisco, CA, USA, 1994, S. 487–499

[3] Albrecht, Oliver: *Filtern von Fremdschlüsseln aus Inklusionsbeziehungen*. term paper at Humboldt-Universität zu Berlin (Studienarbeit), 2007

[4] Bairoch, A. ; Apweiler, R. ; Wu, C. H. ; Barker, W. C. ; Boeckmann, B. ; Ferro, S. ; Gasteiger, E. ; Huang, H. ; Lopez, R. ; Magrane, M. ; Martin, M.J. ; Natale, D.A. ; O'Donovan, C. ; Redaschi, N. ; Yeh, L.S.: The Universal Protein Resource (UniProt). In: *Nucleic Acids Research* 33(Database issue) (2005), S. D154–9

[5] Baker, P. ; Brass, A. ; Bechhofer, S. ; Goble, C. ; Paton, N. ; Stevens: TAMBIS: Transparent Access to Multiple Bioinformatics Information Systems. In: *Proceedings of the 6th International Conference on Intelligent Systems for Molecular Biology*, 1998, S. 25–34

[6] Bauckmann, Jana ; Abedjan, Ziawasch ; Leser, Ulf ; Müller, Heiko ; Naumann, Felix: Covering or Complete? Discovering Conditional Inclusion Dependencies / Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam. 2012 (62). – Technical Report

[7] Bauckmann, Jana ; Abedjan, Ziawasch ; Leser, Ulf ; Müller, Heiko ; Naumann, Felix: Discovering conditional inclusion dependencies. In: *CIKM*

'12: 21st ACM International Conference on Information and Knowledge Management, 2012, S. 2094–2098

[8] Bauckmann, Jana ; Leser, Ulf ; Naumann, Felix: Efficient and Exact Computation of Inclusion Dependencies for Data Integration / Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam. 2010 (34). – Technical Report

[9] Bauckmann, Jana ; Leser, Ulf ; Naumann, Felix ; Tietz, Véronique: Efficiently Detecting Inclusion Dependencies. In: *ICDE '07: Proceedings of the 23rd International Conference on Data Engineering*. Istanbul, Turkey, 2007, S. 1448–1450

[10] Bell, Siegfried ; Brockhausen, Peter: Discovery of Constraints and Data Dependencies in Databases / Universität Dortmund, Fachbereich Informatik, Lehrstuhl VIII, Künstliche Intelligenz. 1995. – Research Report

[11] Bell, Siegfried ; Brockhausen, Peter: Discovery of Data Dependencies in Relational Databases. In: *Statistics, Machine Learning and Knowledge Discovery in Databases, ML–Net Familiarization Workshop*, 1995, S. 53–58

[12] Belleau, François ; Nolin, Marc-Alexandre ; Tourigny, Nicole ; Rigault, Philippe ; Morissette, Jean: Bio2RDF: Towards a mashup to build bioinformatics knowledge systems. In: *Journal of Biomedical Informatics* 41 (2008), Nr. 5, S. 706 – 716

[13] Berman, H.M. ; Westbrook, J. ; Feng, Z. ; Gilliland, G. ; Bhat, T.N. ; Weissig, H. ; Shindyalov, I.N. ; Bourne, P.E.: The Protein Data Bank. In: *Nucleic Acids Research* 28 (2000), Nr. 1, S. 235–242

[14] Berti-Equille, Laure ; Moussouni, Fouzia: Quality Aware Integration and Warehousing of Genomic Data. In: *ICIQ '05: Proceedings of the 10th International Conference on Information Quality*, 2005

[15] Bizer, Christian ; Lehmann, Jens ; Kobilarov, Georgi ; Auer, Sören ; Becker, Christian ; Cyganiak, Richard ; Hellmann, Sebastian: DBpedia - A crystallization point for the Web of Data. In: *Journal of Web Semantics* 7 (2009), Nr. 3, S. 154–165

[16] Bloom, Burton H.: Space/time trade-offs in hash coding with allowable errors. In: *Communications of the ACM* 13 (1970), Nr. 7, S. 422–426

[17] Boulakia, Sarah C. ; Leser, Ulf: Next Generation Data Integration for Life Sciences. In: *ICDE '11: Proceedings of the 27th International Conference on Data Engineering*, 2011, S. 1366–1369

[18] Brauer, Falk ; Rieger, Robert ; Mocan, Adrian ; Barczynski, Wojciech M.: Enabling information extraction by inference of regular expressions from sample entities. In: *CIKM '11: Proceedings of the 20th ACM International Conference on Information and Knowledge Management*. New York, NY, USA : ACM, 2011, 1285–1294

[19] Bravo, Loreto ; Fan, Wenfei ; Ma, Shuai: Extending dependencies with conditions. In: *VLDB '07: Proceedings of the 33rd International Vonference on Very Large Data Bases*, 2007, S. 243–254

[20] Brown, Paul ; Haas, Peter J.: BHUNT: Automatic Discovery of Fuzzy Algebraic Constraints in Relational Data. In: *VLDB '03: 29th International Conference on Very Large Data Bases*, 2003, S. 668–679

[21] Casanova, Marco A. ; Fagin, Ronald ; Papadimitriou, Christos H.: Inclusion dependencies and their interaction with functional dependencies. In: *PODS '82: Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*. New York, NY, USA : ACM, 1982, S. 171–176

[22] Chiang, Fei ; Miller, Renée J.: Discovering data quality rules. In: *Proceedings of the VLDB Endowment (PVLDB)* 1 (2008), S. 1166–1177

[23] Curé, Olivier: Conditional Inclusion Dependencies for Data Cleansing: Discovery and Violation Detection Issues. In: *QDB 2009: 7th International Workshop on Quality in Databases*, 2009

[24] Dale, Nell ; Walker, Henry M.: *Abstract Data Types. Specifications, Implementations and Applications*. Jones and Bartlett Publishers, Inc., 1996

[25] Dasu, Tamraparni ; Johnson, Theodore ; Muthukrishnan, S. ; Shkapenyuk, Vladislav: Mining Database Structure; Or, How to Build a

Data Quality Browser. In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, 2002, S. 240–251

[26] Fan, Wenfei: Dependencies revisited for improving data quality. In: *PODS '08: Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. New York, NY, USA : ACM, 2008, S. 159–170

[27] Fan, Wenfei ; Geerts, Floris ; Jia, Xibei ; Kementsietsidis, Anastasios: Conditional functional dependencies for capturing data inconsistencies. In: *ACM Transactions on Database Systems (TODS)* 33 (2008), Nr. 2, S. 1–48

[28] Fan, Wenfei ; Geerts, Floris ; Li, Jianzhong ; Xiong, Ming: Discovering Conditional Functional Dependencies. In: *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 23 (2011), Nr. 4, S. 683–698

[29] Franklin, Michael ; Halevy, Alon ; Maier, David: From databases to dataspaces: a new abstraction for information management. In: *SIGMOD Record* 34 (2005), Dezember, Nr. 4, S. 27–33

[30] Galperin, Michael Y. ; Fernández-Suárez, Xosé M.: The 2012 Nucleic Acids Research Database Issue and the online Molecular Biology Database Collection. In: *Nucleic Acids Research* 40 (2012), Nr. D1, S. D1–D8

[31] Goble, Carole ; Stevens, Robert: State of the nation in data integration for bioinformatics. In: *Journal of Biomedical Informatics* 41 (2008), Nr. 5, S. 687 – 693

[32] Golab, Lukasz ; Karloff, Howard ; Korn, Flip ; Srivastava, Divesh ; Yu, Bei: On generating near-optimal tableaux for conditional functional dependencies. In: *Proceedings of the VLDB Endowment (PVLDB)* 1 (2008), August, S. 376–390

[33] Golab, Lukasz ; Korn, Flip ; Srivastava, Divesh: Efficient and Effective Analysis of Data Quality using Pattern Tableaux. In: *IEEE Data Engineering Bulletin* 34 (2011), Nr. 3, S. 26–33

[34] Guérin, E. ; Marquet, G. ; Burgun, A. ; Loréal, O. ; Berti-Equille, L. ; Leser, U. ; Moussouni, F.: Integrating and warehousing liver gene

expression data and related biomedical resources in GEDAW. In: *DILS '05: Proceedings of the Second international conference on Data Integration in the Life Sciences*, 2005, S. 158–174

[35] Gusfield, Dan: *Algorithms on Strings, Trees, and Sequences.* Cambrige University Press., 1997

[36] Halevy, Alon ; Franklin, Michael ; Maier, David: Principles of dataspace systems. In: *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, ACM Press, 2006, S. 1–9

[37] Halpin, Harry ; Hayes, Pat ; McCusker, James P. ; McGuinness, Deborah ; Thompson, Henry S.: When owl:sameAs isn't the Same: An Analysis of Identity in Linked Data. In: *ISWC2010: 9th International Semantic Web Conference*, 2010

[38] Hegewald, Jan: *Automatisiertes Auffinden von Präfix- und Suffix-Inklusionsabhängigkeiten in relationalen Datenbankmanagementsystemen*, Humboldt-Universität zu Berlin, Diplomarbeit, 2007. – also published in Hegewald, J. Gebauer, D. M. (Ed.) Informationsintegration in Biodatenbanken. Automatisches Finden von Abhängigkeiten zwischen Datenquellen Vieweg+Teubner Research, 2009

[39] Hernandez, Thomas ; Kambhampati, Subbarao: Integration of biological sources: current systems and challenges ahead. In: *SIGMOD Record* 33 (2004), Nr. 3, S. 51–60

[40] Hubbard, T. ; Barker, D. ; Birney, E. ; Cameron, G. ; Chen, Y. ; Clark, L. ; Cox, T. ; Cuff, J. ; Curwen, V. ; Down, T. ; al. et: The Ensembl genome database project. In: *Nucleic Acids Research* 30(1) (2002), S. 38–41

[41] Huhtala, Ykä ; Kärkkäinen, Juha ; Porkka, Pasi ; Toivonen, Hannu: TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. In: *Computer Journal* 42 (1999), Nr. 2, S. 100–111

[42] Kantola, Martti ; Mannila, Heikki ; Räihä, Kari-Jouko ; Siirtola, Harri: Discovering Functional and Inclusion Dependencies in Relational Databases. In: *International Journal of Intelligent Systems* 7 (1992), S. 591–607

[43] Koeller, Andreas: *Integration of Heterogeneous Databases: Discovery of Meta-Information and Maintenance of Schema-Restructuring Views*, Worcester Polytechnic Institute, PhD thesis, 2001

[44] Koeller, Andreas ; Rundensteiner, Elke A.: Discovery of High-Dimensional Inclusion Dependencies. In: *ICDE '03: Proceedings of the 19th International Conference on Data Engineering*, 2003, S. 683–685

[45] Koeller, Andreas ; Rundensteiner, Elke A.: Heuristic Strategies for the Discovery of Inclusion Dependencies and Other Patterns. In: *Journal on Data Semantics* 5 (2006), S. 185–210

[46] Lenzerini, Maurizio: Data integration: a theoretical perspective. In: *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. New York, NY, USA : ACM, 2002, S. 233–246

[47] Leser, Ulf ; Naumann, Felix: (Almost) Hands-Off Information Integration for the Life Sciences. In: *CIDR '05: Proceedings of the Second Biennial Conference on Innovative Data Systems Research*, 2005

[48] Li, Yunyao ; Krishnamurthy, Rajasekar ; Raghavan, Sriram ; Vaithyanathan, Shivakumar ; Jagadish, H. V.: Regular expression learning for information extraction. In: *EMNLP '08: Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Stroudsburg, PA, USA : Association for Computational Linguistics, 2008, S. 21–30

[49] Liu, Bing ; Hsu, Wynne ; Ma, Yiming: Integrating Classification and Association Rule Mining. In: *KDD-98: Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, 1998, S. 80–86

[50] Lo Conte, Loredana ; Brenner, Steven E. ; Hubbard, Tim J. ; Chothia, Cyrus ; Murzin, Alexey G.: SCOP database in 2002: refinements accommodate structural genomics. In: *Nucleic Acids Research* 30(1) (2002), S. 264–267

[51] Lopes, Stéphane ; Petit, Jean-Marc ; Toumani, Farouk: Discovering interesting inclusion dependencies: application to logical database tuning. In: *Information Systems* 27 (2002), Nr. 1, S. 1–19

[52] Marchi, Fabien D. ; Lopes, Stéphane ; Petit, Jean-Marc: Efficient Algorithms for Mining Inclusion Dependencies. In: *EDBT '02: Proceedings of the 8th International Conference on Extending Database Technology*, Springer-Verlag, 2002, S. 464–476

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

[54] Marchi, Fabien D. ; Petit, Jean-Marc: Zigzag: a new algorithm for mining large inclusion dependencies in databases. In: *ICDM '03: Proceedings of the Third IEEE International Conference on Data Mining*, 2003, S. 27–34

[55] Müller, Heiko ; Leser, Ulf ; Freytag, Johann-Christoph: Mining for patterns in contradictory data. In: *IQIS '04: International Workshop on Information Quality in Information Systems*, 2004

[56] Murzin, A. G. ; Brenner, S. E. ; Hubbard, T. ; Chothia, C.: SCOP: a structural classification of proteins database for the investigation of sequences and structures. In: *Journal of Molecular Biology* 247 (1995), Nr. 4, S. 536–40

[57] Orengo, CA ; Michie, AD ; Jones, S ; Jones, DT ; Swindells, MB ; Thornton, JM: CATH–a hierarchic classification of protein domain structures. In: *Structure* 5 (1997), Nr. 8, S. 1093–1108

[58] Petit, Jean-Marc ; Toumani, Farouk ; Boulicaut, Jean-Francois ; Kouloumdjian, Jacques: Towards the Reverse Engineering of Denormalized Relational Databases. In: *ICDE '96: Proceedings of the 12th International Conference on Data Engineering* (1996), S. 218

[59] Rahm, E. ; Bernstein, P. A.: A Survey of Approaches to Automatic Schema Matching. In: *The VLDB Journal* 10 (2001), Nr. 4, S. 334–350

[60] Rostin, Alexandra ; Albrecht, Oliver ; Bauckmann, Jana ; Naumann, Felix ; Leser, Ulf: A Machine Learning Approach to Foreign Key Discovery. In:

12th International Workshop on the Web and Databases (WebDB). Providence, Rhode Island, 2009

[61] Schrag, Roger: *Speeding Up Queries with Semi-Joins and Anti-Joins: How Oracle Evaluates EXISTS, NOT EXISTS, IN, and NOT IN*. White paper @ dbspecialists.com. `http://www.dbspecialists.com/files/presentations/semijoins.html`. Version: 2005

[62] Srikant, Ramakrishnan ; Vu, Quoc ; Agrawal, Rakesh: Mining Association Rules with Item Constraints. In: *KDD-97: Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, 1997, S. 67–73

[63] Trissl, Silke ; Rother, Kristian ; Müller, Heiko ; Steinke, Thomas ; Koch, Ina ; Preissner, Robert ; Frömmel, Cornelius ; Leser, Ulf: Columba: An Integrated Database of Proteins, Structures, and Annotations. In: *BMC Bioinformatics* 6 (2005), S. 81

[64] Villanueva-Rosales, Natalia ; Dumontier, Michel: yOWL: An ontology-driven knowledge base for yeast biologists. In: *Journal of Biomedical Informatics* 41 (2008), Nr. 5, S. 779 – 789

[65] Vitter, Jeffrey S.: Algorithms and Data Structures for External Memory. In: *Foundations and Trends in Theoretical Computer Science* 2 (2006), Nr. 4, S. 305–474

[66] Warren, Robert H. ; Tompa, Frank W.: Multi-column substring matching for database schema translation. In: *VLDB '06: Proceedings of the 32nd International Conference on Very Large Data Bases*, 2006, S. 331–342

[67] Zhang, Meihui ; Hadjieleftheriou, Marios ; Ooi, Beng C. ; Procopiuc, Cecilia M. ; Srivastava, Divesh: On multi-column foreign key discovery. In: *Proceedings of the VLDB Endowment (PVLDB)* 3 (2010), September, S. 805–814

Selbständigkeitserklärung

Hiermit versichere ich, die vorliegende Dissertation eigenständig und ausschließlich unter Verwendung der angegebenen Hilfsmittel, angefertigt zu haben. Alle öffentlichen Quellen sind als solche kenntlich gemacht. Die vorliegende Arbeit ist in dieser oder anderer Form zuvor nicht als Prüfungsarbeit zur Begutachtung vorgelegt worden.

Potsdam, den 2013/02/28