Scalable Data Profiling

Distributed Discovery and Analysis of Structural Metadata

**Dissertation
zur Erlangung des akademischen Grades
"Doktor der Naturwissenschaften"
(Dr. rer. nat.)
in der Wissenschaftsdisziplin "Informationssysteme"**

**eingereicht an der
Digital Engineering Fakultät
der Universität Potsdam**

**von
Sebastian Kruse**

Potsdam, den 4. Juni 2018

Reviewers

Prof. Dr. Felix Naumann
Hasso-Plattner-Institut, Universität Potsdam
Primary thesis advisor

Prof. Dr. Ulf Leser
Humboldt-Universität zu Berlin
Secondary thesis advisor

Prof. Dr. Erhard Rahm
Universität Leipzig

Published online at the
Institutional Repository of the University of Potsdam:
URN urn:nbn:de:kobv:517-opus4-412521
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-412521

Abstract

Data profiling is the act of extracting *structural metadata* from datasets. Structural metadata, such as data dependencies and statistics, can support data management operations, such as data integration and data cleaning. Data management often is the most time-consuming activity in any data-related project. Its support is extremely valuable in our data-driven world, so that more time can be spent on the actual utilization of the data, e. g., building analytical models. In most scenarios, however, structural metadata is not given and must be extracted first. Therefore, efficient data profiling methods are highly desirable.

Data profiling is a computationally expensive problem; in fact, most dependency discovery problems entail search spaces that grow exponentially in the number of attributes. To this end, this thesis introduces novel discovery algorithms for various types of data dependencies – namely *inclusion dependencies*, *conditional inclusion dependencies*, *partial functional dependencies*, and *partial unique column combinations* – that considerably improve over state-of-the-art algorithms in terms of efficiency and that scale to datasets that cannot be processed by existing algorithms. The key to those improvements are not only algorithmic innovations, such as novel pruning rules or traversal strategies, but also algorithm designs tailored for distributed execution. While distributed data profiling has been mostly neglected by previous works, it is a logical consequence on the face of recent hardware trends and the computational hardness of dependency discovery.

To demonstrate the utility of data profiling for data management, this thesis furthermore presents Metacrate, a database for structural metadata. Its salient features are its flexible data model, the capability to integrate various kinds of structural metadata, and its rich metadata analytics library. We show how to perform a *data anamnesis* of unknown, complex datasets based on this technology. In particular, we describe in detail how to reconstruct the schemata and assess their quality as part of the data anamnesis.

The data profiling algorithms and Metacrate have been carefully implemented, integrated with the Metanome data profiling tool, and are available as free software. In that way, we intend to allow for easy repeatability of our research results and also provide them for actual usage in real-world data-related projects.

Zusammenfassung

Data Profiling bezeichnet das Extrahieren *struktureller Metadaten* aus Datensätzen. Stukturelle Metadaten, z.B. Datenabhängigkeiten und Statistiken, können bei der Datenverwaltung unterstützen. Tatsächlich beansprucht das Verwalten von Daten, z.B. Datenreinigung und -integration, in vielen datenbezogenen Projekten einen Großteil der Zeit. Die Unterstützung solcher verwaltenden Aktivitäten ist in unserer datengetriebenen Welt insbesondere deswegen sehr wertvoll, weil so mehr Zeit auf die eigentlich wertschöpfende Arbeit mit den Daten verwendet werden kann, z.B. auf das Erstellen analytischer Modelle. Allerdings sind strukturelle Metadaten in den meisten Fällen nicht oder nur unvollständig vorhanden und müssen zunächst extahiert werden. Somit sind effiziente Data-Profiling-Methoden erstrebenswert.

Probleme des Data Profiling sind in der Regel sehr berechnungsintensiv: Viele Datenabhängigkeitstypen spannen einen exponentiell in der Anzahl der Attribute wachsenden Suchraum auf. Aus diesem Grund beschreibt die vorliegende Arbeit neue Algorithmen zum Auffinden verschiedener Arten von Datenabhängigkeiten – nämlich *Inklusionsabhängigkeiten*, *bedingter Inklusionsabhängigkeiten*, *partieller funktionaler Abhängigkeiten* sowie *partieller eindeutiger Spaltenkombinationen* – die bekannte Algorithmen in Effizienz und Skalierbarkeit deutlich übertreffen und somit Datensätze verarbeiten können, an denen bisherige Algorithmen gescheitert sind.

Um die Nützlichkeit struktureller Metadaten für die Datenverwaltung zu demonstrieren, stellt diese Arbeit des Weiteren das System METACRATE vor, eine Datenbank für strukturelle Metadaten. Deren besondere Merkmale sind ein flexibles Datenmodell; die Fähigkeit, verschiedene Arten struktureller Metadaten zu integrieren; und eine umfangreiche Bibliothek an Metadatenanalysen. Mithilfe dieser Technologien führen wir eine *Datenanamnese* unbekannter, komplexer Datensätze durch. Insbesondere beschreiben wir dabei ausführlicher, wie Schemata rekonstruiert und deren Qualität abgeschätzt werden können.

Wir haben oben erwähnte Data-Profiling-Algorithmen sowie METACRATE sorgfältig implementiert, mit dem Data-Profiling-Programm METANOME integriert und stellen beide als freie Software zur Verfügung. Dadurch wollen wir nicht nur die Nachvollziehbarkeit unserer Forschungsergebnisse möglichst einfach gestalten, sondern auch deren Einsatz in der Praxis ermöglichen.

Acknowledgements

This thesis would not have been possible if it had not been for a lot of very kind people. First of all, I want to thank my PhD advisor, Prof. Felix Naumann. Throughout my whole time at his chair, he has always been very engaged in supporting me, giving me guidance, and helping me to stay on track. I really learned a lot from him – about research in specific and about pursuing one's ambitions in general. I also want to thank my second advisor Prof. Ulf Leser and my advisor throughout my internship at the Qatar Computing Research Institute, Jorge Arnulfo Quiané-Ruiz. They gave me new, highly appreciated perspectives on the world of research and their advises were both inspiration and motivation to me.

Furthermore, I am truly grateful for having had such nice colleagues at the chair. I did not only enjoy working with them very much and am happy about the papers that we published together, but they were always there to have a fun chat with. I was happy to have them around and from now on I will certainly miss our bike trips to the HPI and our lunch breaks at the canteen.

Last but not least, I want to thank my wife, my family, and my friends. Even though most of them are not involved with IT, I might not have made it to this point in my life without them. They were always there for me and gave me the support and confidence that helped me to follow my dreams. Without them, I would not be the same, for everyone of them is a part of me.

Contents

Chapter 1
Big Data: Amid Opportunity and Challenge

Over the past years, data has been commonly referred to as “the new oil” [Palmer, 2006; Rotella, 2012; Toonders, 2014]. Indeed, data fuels industries: Not only does data help to streamline business processes and personalize products in classical industries, but also has the data become an industry itself [Pham, 2015]. What is more, data even bears the potential to substantially change some scientific fields, replacing classical theoretical models with statistically inferred ones [Anderson, 2008]. This “Big Data era” has been made possible by the ever-increasing technological capabilities to collect, store, and process ever-growing amounts of data. In fact, it has been reported that 16.1 zettabytes (1.61×10^{23} bytes) of data have been produced in 2016 and this number is expected to increase by an order of magnitude by 2025 [Reinsel et al., 2017].

On the face of this enthusiasm, one may wonder, though: If data is the new oil, where is its refinery? Data comes from all kinds of sources and in a variety of different formats, it can be faulty, incomplete, and outdated. That is, data is initially raw; and it needs to be refined and made fit for its intended use. Such data refinery is not only mission-critical, it is also a very complex task. For instance, a recent survey reports that data scientists spend 60 % of their work time on cleaning and organizing data (and also perceive this as their least enjoyable activity) and another 19 % on selecting datasets [Press, 2016].

In the following, we refer to such and similar refinery tasks as *data management*. Typical data management scenarios encompass *data reverse engineering* [Davis and Alken, 2000; Hainaut et al., 2009], *data cleaning* [Ganti and Sarma, 2013; Rahm and Do, 2000], *data integration* [Leser and Naumann, 2007; Rahm and Bernstein, 2001], and many more. While those activities are complex in nature, *structural metadata* can help to (partially) automate them. In a few words, structural metadata summarizes technical properties of a dataset, e. g., the number of distinct values in a database column or the key candidates of a database table. Such summaries support several concrete data management tasks, such as reconstructing database schemata [Kruse et al., 2016b] or formulating data cleaning rules [Thirumuruganathan et al., 2017], hence allowing data practitioners to focus more on their main tasks. Unfortunately, in the majority of cases datasets are annotated with only little or no structural metadata at all.

The research area of *data profiling* [Abedjan et al., 2015] investigates methods for the automatic extraction of structural metadata (and we henceforth refer to structural metadata also with the simpler term *data profiles*). This is a notoriously difficult task, not only because of the amounts of data to be analyzed but also because of its computational complexity: Many data profiling problems are of exponential complexity [e. g., Gunopulos et al., 2003; Liu et al., 2012]. As a result, state-of-the-art data profiling algorithms are often applicable only to small datasets, so that the benefits of structural metadata for data management are precluded in practical scenarios. Indeed, in many discussions with industry representatives, we learned that only those types of data profiles are being employed for data management operations that can be easily calculated. This circumstance is reflected in the limited capability of industry data profiling suites [Eschrig, 2016]. However, our dialog partners also expressed great interest in more advanced metadata types. A sign of this development is IBM's recently open-sourced Open Discovery Framework [ODF 2017].

Given this context, we set out in this thesis to solve existing and well-known data profiling problems in a more efficient and scalable manner than state-of-the-art methods. In particular, we address this challenge with novel algorithmic approaches, but also by designing our algorithms for distributed execution. Existing data profiling methods rarely address this dimension, which is surprising given the hardness of the problem and the shift towards computer clusters. In addition, we introduce the system METACRATE to store and analyze the structural metadata produced by our data profiling algorithms. We particularly show how to admit an unknown dataset into an organization and refer to this data management scenario as *data anamnesis*. Ultimately, the work presented in this thesis aims to contribute to the "Big Data era" by improving data refinery and, thus, allowing data to be utilized more effectively and efficiently.

In the remainder of this introductory chapter, we give a more concrete motivation for data profiling by explaining in more detail the steps and stumbling blocks of a data anamnesis in Section 1.1. Then, we characterize the term data profiling in Section 1.2 and highlight its subarea of dependency discovery in Section 1.3. We then proceed to give a more thorough explanation in Section 1.4 as to why this thesis focuses on distributed data profiling. Finally, we give an overview of the structure of this thesis and highlight our contributions in Section 1.5.

1.1 Data Anamnesis: A Motivating Example

Oil is not an end in itself. Its value is drawn from the products that can be created from it, such as energy and synthetic materials. The same holds for data: Data is valuable only as far as its owner can put it to its intended or to a novel use. However, while oil is a fixed composition of matter, data is far more individual on a multitude of levels, ranging from the syntactic over semantic to quality aspects. As we exemplify below, it is that individuality that makes data a crop hard to harvest.

Imagine a video streaming service that wants to enrich its videos with discographic information on the music that is being played in its videos. Having obtained a discographical dataset for that purpose is only a very first step; the main challenges are yet to come. For instance, it must be determined what kind of information is actually contained in the dataset (genres? labels? artist biographies?), how to query the dataset (references? implicit assumptions?), and how reliable the data is (completeness? correctness? duplicates?). This situation is very different from that of a synthetic material manufacturer who gets the same kind of oil from all sources.

We refer to above scenario as *data anamnesis* [Kruse et al., 2016b]: Some organization needs to understand a new dataset and assess its quality, so as to determine if, how, and to what extent the data can be leveraged for its intended use. Data anamnesis is a prime example for data management: It involves aspects of data exploration (what information is stored in the database?), database reverse engineering (how is the data modeled? how to query it?), and data cleaning (is the dataset subject to quality issues?).

In an ideal case, the dataset would be accompanied by a documentation to support these tasks. Unfortunately, a documentation is often missing. And even if it existed, it would unlikely address all raised questions. What is more, the documentation might be outdated, imprecise or incorrect and is, hence, not a fully reliable source.

In consequence, a data anamnesis should generally employ structural metadata extracted with data profiling algorithms. Freshly extracted metadata accurately describes the data, it is up-to-date, and one can extract exactly those metadata types that are needed. To get an idea of how metadata is helpful for data anamnesis, consider the following example: The dataset in question has neither primary keys (PKs) nor foreign keys (FKs) annotated. In order to rediscover the PKs, we need to identify column combinations that do not contain duplicate values and are free of `NULL`s, both of which are classical data profiling tasks. For the FKs, we need to find containment relationships among columns, yet another classical data profiling task. Note, however, that data profiling is only a preparatory step. After all, not every key in a relation can be the primary key and not every containment relationship corresponds to a foreign key.

Near the end of this thesis, in Chapter 5, we demonstrate a metadata-driven data anamnesis in much greater detail using our tool Metacrate, thereby reenacting the workflow depicted in Figure 1.1: We profile a dataset to extract its structural metadata. Then, based on those extracted data profiles, we extract its schema (schema discovery), develop an understanding of the schema (conceptualization), and finally assess how suitable the schema is for the data it contains (bottom-up assessment).

However, a precondition to a metadata-driven data anamnesis is the availability of efficient and scalable data profiling algorithms. That is, it must be possible to extract structural metadata of real-world datasets on a short-term basis. Achieving this goal is a main concern of this thesis through the Chapters 2–4. Let us therefore have a closer look on the actual problems and challenges of data profiling.

1.2 Data Profiling

The research area of data profiling deals with the question how structural metadata can be extracted from datasets [Abedjan et al., 2015; Johnson, 2009; Naumann, 2014]. On a

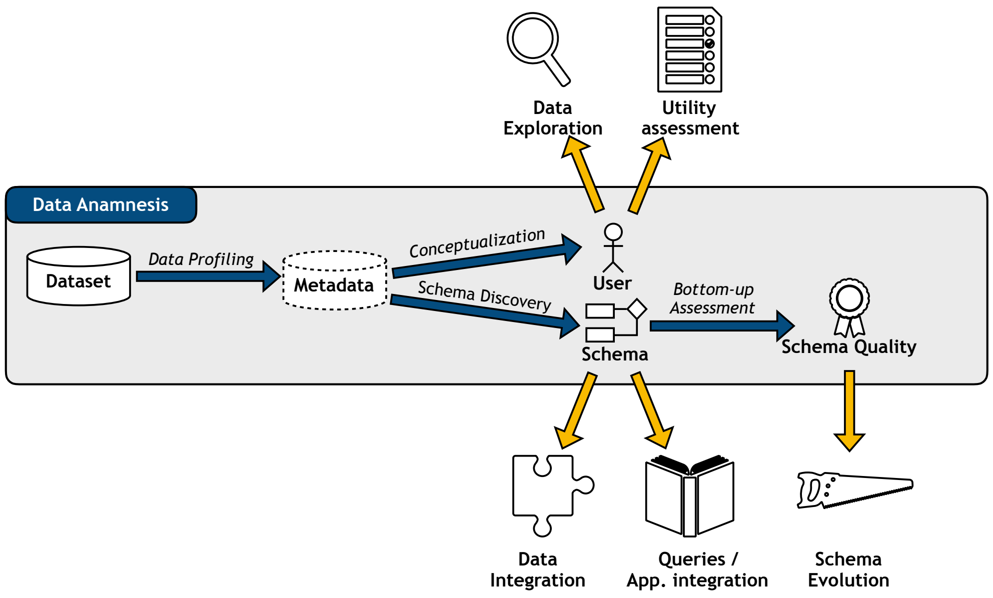


Figure 1.1: Possible workflow of a data anamnesis including follow-up use cases.

high level, structural metadata can be divided into *data synopses* and *data dependencies*, both of which are subject to intensive ongoing research. While the former is associated to the rather "traditional" profiling tasks, such as counting the number of distinct values in database columns, inferring patterns to describe textual attributes, or calculating the correlation of attributes, the latter deals with identifying related columns and also bears many open problems, such as incremental dependency discovery and scalability issues [Naumann, 2014].

Data profiling is furthermore closely related to the field of data mining. Literature mentions following distinguishing characteristics:

- *"**Data profiling** focusses on the instance analysis of individual attributes. It derives information such as the data type, length, [...], etc., providing an exact view of various quality aspects of the attribute. [...] **Data mining** helps discover specific data patterns in large data sets, e. g., relationships holding between several attributes. This is the focus of so-called descriptive data mining models including clustering, summarization, association discovery and sequence discovery."* [Rahm and Do, 2000]

- *"**Data profiling** gathers technical metadata to support data management, while **data mining** and data analytics discovers non-obvious results to support business management. In this way, data profiling results are information about columns and column sets, while data mining results are information about rows or row sets (clustering, summarization, association rules, etc.)."* [Naumann, 2014]

As a matter of fact, the line between data profiling and data mining is blurry. While this thesis aims at advancing the state-of-the-art in data profiling, we find that some of our results are also valuable from a data mining perspective. For instance, *conditional inclusion dependencies* on RDF data (Chapter 3) and *partial functional dependencies* on relational data (Chapter 4) can help to establish new domain knowledge. Furthermore, all dependency discovery problems investigated in this thesis are in some way related to association rule mining, a fundamental data mining problem. In that sense, our data profiling results might well be applicable to data mining by some means or other – and vice versa, of course.

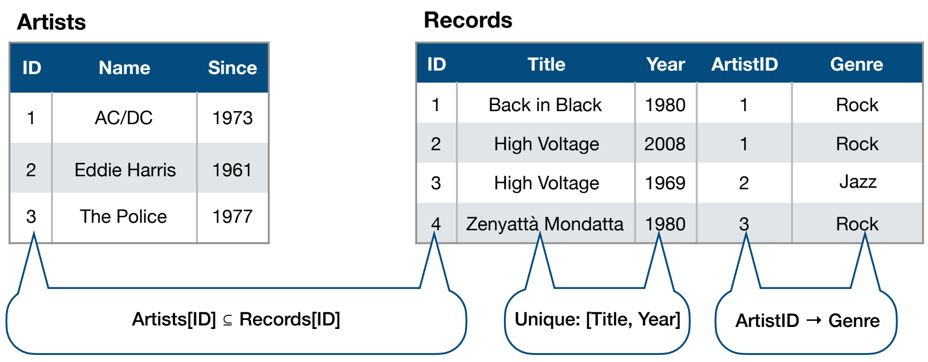
1.3 Data Dependencies

Let us now describe the data profiling subarea of dependency discovery, which is a focus of this thesis, in more detail. We aim to give a brief overview over some of the most relevant types of dependencies. In the field of database theory, a vast number of basic dependencies, specializations of the same, and generalizing frameworks have been conceived [e. g., Abiteboul et al., 1995; Kanellakis, 1989; Thalheim, 2013]. While many theoretical problems, such as dependency implication and relation decomposition under dependencies, have been extensively studied, automatic dependency discovery has been dealt with for only few dependency types. Being aware of this large body of work, we narrow down the following discussion to the most relevant dependency types of *relational* databases [Gyssens, 2009].

Generally speaking, a data dependency is some logical statement on a database. If the statement is true, then the dependency is said to hold on or to be satisfied by the database. That is, we use the term *dependency* to *describe* some property of a specific database instance. This in contrast to *constraints* that *prescribe* some intentional property of databases. As such, constraints can be understood as part of a database schema that restrict the set of *all* valid database instances. Note, however, that in literature these two terms are often used synonymously.

Example 1.1 (Dependency vs. constraint)**.** An *inclusion dependency* (IND, introduced below) states that all values of one database column are also found in some other column: In Figure 1.2, all values of the column Artists[ID] are also found in Records[ID]. A *foreign key (FK) constraint*, in contrast, prescribes that all values of the referencing column must be found in the referenced column. Furthermore, the FK constraint expresses a design intention and usually some meaningful rule of the domain of the data. Thus, from a FK perspective our example IND is not useful, while the IND Records[ArtistID] $\subseteq$ Artists[ID] corresponds to an actual FK constraint.

At this point, the reader might wonder, why different dependency types are needed in the first place when first order logic (FOL) could be used instead. The answer is twofold: The implication among two arbitrary statements is only semidecidable in FOL [Trakhtenbrot, 1950]. Such theoretical properties, and implication in particular, are extremely important, though, when discovering or reasoning about database properties. The different dependency types, in contrast, are usually defined in a way that they have desirable theoretical properties. The second point is that there is an infinite amount of true statements one could conceive for a given database. However, only few are useful. The here



Artists

ID	Name	Since
1	AC/DC	1973
2	Eddie Harris	1961
3	The Police	1977

Records

ID	Title	Year	ArtistID	Genre
1	Back in Black	1980	1	Rock
2	High Voltage	2008	1	Rock
3	High Voltage	1969	2	Jazz
4	Zenyattà Mondatta	1980	3	Rock

Figure 1.2: Example dataset with some data dependencies highlighted.

described dependencies all have known applications, e. g., INDs can be used to discover FKs. Hence, focusing on certain dependency types is already a valuable preselection of interesting statements among all the statements that are fulfilled by a database.

Among the most important dependency types are *unique column combinations (UCCs)*, *functional dependencies (FDs)*, and *inclusion dependencies (INDs)* [Gyssens, 2009]. Those dependencies are in the focus of the thesis. Let us therefore briefly introduce each of them, including their various use cases.

Unique column combinations (UCCs)

Algorithms in this thesis: PYRO *(Chapter 4)*

UCCs, also known as *uniques*, *uniqueness constraints*, and *candidate keys*, are an integral part of the relational model [Codd, 1970]. Codd introduced them as *keys*, which are those sets of columns in a relation, that contain only unique values. Formally, for a relational schema R and a corresponding instance r, $U \subseteq R$ is said to be a UCC, if and only if

$$\forall t_1, t_2 \in r, t_1 \neq t_2 \colon t_1[U] \neq t_2[U]$$

For instance, in Figure 1.2 Records[ID] and Records[Title, Year] form UCCs. What is more, from neither of them we could remove a column without breaking the UCC property. Hence, we call them *minimal* UCCs and in general, we are interested only in such minimal UCCs.

While, obviously, UCCs are helpful to designate the primary key of a relation and ensure entity integrity [Codd, 1979], they have further applications, e. g., in query optimization [Paulley and Larson, 1993; Simmen et al., 1996] and instance matching [Rahm and Do, 2000].

The discovery of only a single minimal UCC is already NP-complete [Lucchesi and Osborn, 1978] and the discovery of all minimal UCCs in a relation is NP-hard in the number of attributes [Gunopulos et al., 2003]. For that matter, it follows from Sperner's theorem [Sperner, 1928] that a relation with n attributes can have $\binom{n}{\frac{n}{2}}$ minimal UCCs. In consequence, we have to acknowledge that the discovery of even this seemingly simple dependency type can already become extremely hard for relations with a large number of columns.

Functional dependencies (FDs)

Algorithm in this thesis: Pyro *(Chapter 4)*

Among all dependencies, FDs have probably attracted the greatest attention: Not only have their theoretical properties been studied extensively, but also practical aspects, such as discovery and use cases, have been actively researched for decades now. FDs were implicitly introduced by Codd in the 1970s to define normalized relational databases [Codd, 1971a]. In few words, an FD $X \to Y$ for some attribute sets X and Y is said to hold, if and only if the values in X functionally determine the values for all attributes in Y. More formally, for a relational schema R with an instance r, and attribute sets $X, Y \subseteq R$, the following condition must hold:

$$\forall t_1, t_2 \in r\colon t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$$

In the Records table from Figure 1.2, ArtistID $\to$ Genre is a valid FD, because every artist ID is associated to exactly one genre. Furthermore, every UCC functionally determines all attributes in a relation.

The uses of FDs are manifold and range from schema normalization [Papenbrock and Naumann, 2017a], over data cleaning [Thirumuruganathan et al., 2017], to query optimization [Gryz, 1998a, 1999; Hong et al., 2005]. It is also worth mentioning that a considerable number of generalizations of FDs have been conceived, e. g., multivalued dependencies [Fagin, 1977; Zaniolo, 1976], join dependencies [Fagin, 1979; Rissanen, 1977], order dependencies [Ginsburg and Hull, 1983], and matching dependencies [Fan, 2008].

Inclusion dependencies (INDs)

Algorithms in this thesis: Sindy *(Chapter 2),* RDFind *(Chapter 3)*

Although Codd was the first to introduce the closely related notion of foreign keys [Codd, 1970], the term "inclusion dependencies" was introduced by Fagin [1981]. INDs are rather different from UCCs and FDs in both theoretical and practical regards. While the latter (UCCs and FDs) belong to the class of equality-generating dependencies, INDs are tuple-generating dependencies. Furthermore, a single IND can span two relations. This has a major impact on discovery algorithms, which must not treat relations independently of each other as is the case for UCCs and FDs.

Concretely, an IND is given between two column combinations of one or two relations, if and only if all value combinations from the one column combination are also found in the other. Formally, we have two relation schemata R and S with instances r and s and two (ordered) column combinations $A = (R_{i_1}, \ldots, R_{i_n})$ and $B = (S_{j_1}, \ldots, S_{j_n})$. Then, $R[A]{\subseteq}S[B]$ is an IND if

$$\forall t_1 \in r\colon \exists t_2 \in s\colon t_1[A] = t_2[B]$$

The most prominent use case for INDs is the detection of foreign keys (FKs) [Rostin et al., 2009; Zhang et al., 2010], but they are also valuable for data integration [Bauckmann, 2013], data cleaning [Bohannon et al., 2005], schema design [Levene and Vincent, 1999], and query optimization [Gryz, 1998b, 1999].

* * *

There a many more dependency types than the above – also for data models other than the relational one. However, those traditional, *exact* dependencies are often too strict for practical use. For instance, data is often erroneous so that actually meaningful dependencies are violated. Therefore, various *generalizations* of dependencies have been proposed that account for such circumstances. In the following, we present those generalizations that are a focus of this thesis.

Partial dependencies

Algorithms in this thesis: SINDY *(Chapter 2),* PYRO *(Chapter 4)*

Generally speaking, a dependency is said to be partial or approximate[1], if it does not hold for the entirety of the data. Partial dependencies are applicable in use cases where dependencies may or must not be exact, e. g., for data cleaning [Kolahi and Lakshmanan, 2009; Thirumuruganathan et al., 2017], query optimization [Ilyas et al., 2004], and data integration [Bauckmann, 2013].

The degree to which a dependency is partial is usually quantified in terms of an *error measure*. In general, there are various alternative error measures for each dependency type. For instance, for FDs well-known error measures are g_1, g_2, and g_3, that focus on either the number of tuples or tuple pairs violating a certain FD [Kivinen and Mannila, 1995].

As an example, consider the dataset in Figure 1.3 with the partial FD $\{\mathsf{ArtistID}, \mathsf{Title}\} \rightarrow \mathsf{Album}$ on the Songs relation. This FD seems reasonable at first glance, because when an artist releases a new album, it should contain new songs. However, there are exceptions to that, e. g., greatest hits compilations and live albums. Concretely, *"Alabama Song"* by the *The Doors* appears on two different albums, causing the second and third tuple of the Songs relation to violate our example FD. That being said, it is important to observe the ambiguity of partial dependencies: Whether the example FD is considered a partial FD depends on (i) the error measure (should we count the number of violating tuple pairs? or the number of tuples involved in a violation?) and (ii) a user-defined error threshold that has to be satisfied (is one violating tuple pair out of ten acceptable?).

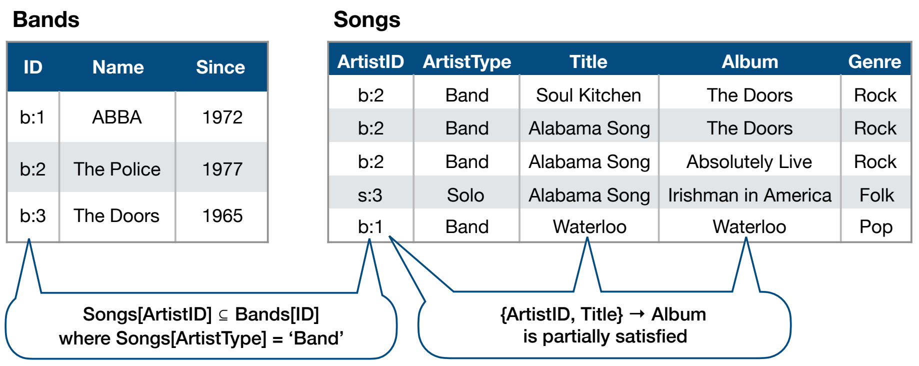
Conditional dependencies

Algorithms in this thesis: RDFIND *(Chapter 3)*

Whenever a dependency is satisfied only partially, it is interesting to characterize that satisfying part of the data. *Conditional dependencies* do exactly that by amending partial dependencies with conditions – usually expressed in terms of *pattern tableaux* – that select those tuples that satisfy the partial dependency [e. g., Bravo et al., 2007]. As a matter of fact, many discovery algorithms for conditional dependencies directly amend partial dependencies with conditions [Bauckmann et al., 2012; Golab et al., 2008].

Arguably, the most popular application of conditional dependencies is data cleaning [Bohannon et al., 2007; Fan et al., 2008], but also data integration [Bauckmann

[1]We do not use the term "approximate dependencies" to avoid confusion with approximate dependency discovery, which – as explained below – approximates the set of dependencies.


Bands

ID	Name	Since
b:1	ABBA	1972
b:2	The Police	1977
b:3	The Doors	1965

Songs

ArtistID	ArtistType	Title	Album	Genre
b:2	Band	Soul Kitchen	The Doors	Rock
b:2	Band	Alabama Song	The Doors	Rock
b:2	Band	Alabama Song	Absolutely Live	Rock
s:3	Solo	Alabama Song	Irishman in America	Folk
b:1	Band	Waterloo	Waterloo	Pop

Songs[ArtistID] ⊆ Bands[ID]
where Songs[ArtistType] = 'Band'
{ArtistID, Title} → Album
is partially satisfied

Figure 1.3: Example dataset with some generalized data dependencies highlighted.

et al., 2012] and query optimization [Kruse et al., 2016a] benefit from conditional dependencies. Furthermore, in Chapter 3 we show that conditional INDs are particularly useful on RDF data as a substitute for traditional INDs.

As an example for a conditional IND, observe that in the dataset in Figure 1.3 the column Songs[ArtistID] is partially included in Bands[ID]. We can refine this insight by noting that this partial IND holds exactly for those tuples t in the Songs relation where $t[\text{ArtistType}] = $ *"Band"*. The combination of this condition and the partial IND forms a conditional IND.

Approximate dependency discovery

Algorithms (not in this thesis): FAIDA *[Kruse et al., 2017b],* AID-FD *[Bleifuß et al., 2016]*

Because dependency discovery is computationally very expensive, one might want to trade correctness guarantees for performance gains – or put otherwise – one might want to only *approximate* the dependencies in a dataset. Depending on the approximation strategy, some of the discovered dependencies might then not be correct or the set of discovered dependencies as a whole might not be complete. Although we have investigated approximation strategies for (exact) INDs and FDs (see above), we do not elaborate on them in this thesis. Still, we would like to make the reader explicitly aware of the distinction between *partial/approximate dependencies* and *approximate dependency discovery*, as the ambiguity of *approximation* has, in our experience, often caused confusion.

1.4 Distributed Dependency Discovery

As stated already several times, a major problem of data profiling (and dependency discovery in particular) is its computational hardness that is facing ever-growing amounts of data. Most of the previous research has tackled this challenge with algorithmic innovations, such as specific data structures and pruning rules. While efficient algorithmic strategies are of paramount importance, it is also crucial to elaborate how those strategies can be efficiently carried out on today's computing hardware.

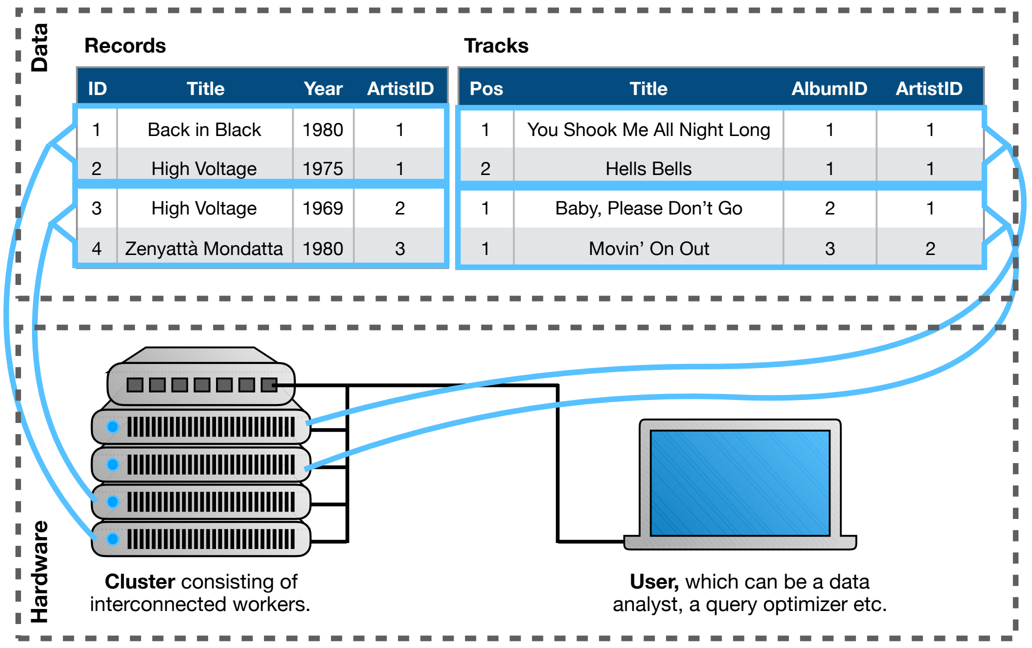
As an example, an early IND discovery algorithm by De Marchi et al. does not thoroughly specify an implementation strategy [de Marchi et al., 2002]. In consequence, straight-forward implementations of this algorithm run out of main memory on many datasets and are not capable of profiling larger datasets. The more recent IND discovery algorithm SPIDER employs the same algorithmic foundations as its predecessor. However, SPIDER explicitly addresses the question how to efficiently incorporate the hard disk so as to cope with main memory limitations [Bauckmann et al., 2006]. As a result, it scales to much larger datasets.

With the importance of aligning algorithms and hardware, it is worthwhile to have a look at recent hardware trends. As far as CPUs are concerned, the performance of individual cores has improved over the last year (e. g., completing CPU instructions in fewer CPU cycles, larger caches, novel SIMD instructions), but not as remarkable as in the time before. Instead, multi-core CPUs have become the standard. However, the number of cores that fit in a single machine is limited: Moore's law, which predicted a two-fold increase of transistors per area on a chip every two years, is now said to cease [Simonite, 2016]. Furthermore, the amount of other resources (main memory, persistent storage, bus bandwidth etc.) that can fit in a single machine is limited, too.

To overcome this limitation, instead of replicating resources within a single machine, the machines themselves can be replicated, forming *computer clusters*. A computer cluster consists of multiple traditional computers, usually called *workers*[2]. Each worker has its very own, dedicated resources, but interacts with the other workers through a network, so as to accomplish some common task. Over the last years, computer clusters have gained massive popularity. One reason is the sheer need for more processing resources due to the ever growing numbers of users to be served and data to be processed. A second reason might be found in the software support for distributed computing. Lately, there has been a "Cambrian explosion" of distributed data processing tools, with Map/Reduce [Dean and Ghemawat, 2008], Apache Spark [Zaharia et al., 2012], and Apache Flink [Alexandrov et al., 2014] as notable representatives.

Given this situation, it is consequential to profile data in a distributed fashion on computer clusters. This is not a trivial task, though: Not only is the implementation of distributed applications much more challenging than it is for their single-machine counterparts, but also the algorithmic engineering is much more complicated. For example, powerful pruning and traversal strategies for data profiling search spaces are key to many data profiling algorithms. If multiple workers simultaneously process a dataset, then they need to exchange pruning information and also avoid traversing overlapping parts of the search space. In a distributed processing environment, such coordination has to be done via the network, which is often a scarce resource with relatively high latency. This issue has to be actively addressed in the algorithm design, such as by Heise et al. [2013]. In other words, there is no straight-forward procedure to let existing single-machine data profiling algorithms exploit the computation resources of computer clusters; the problems are not "embarrassingly parallelizable".

[2]Note that many computer cluster frameworks designate one computer as a *master* that orchestrates the workers. In terms of hardware, there is no necessity to have distinguished master computer, though.


Data
Hardware
Cluster consisting of interconnected workers.
User, which can be a data analyst, a query optimizer etc.

Records

ID	Title	Year	ArtistID
1	Back in Black	1980	1
2	High Voltage	1975	1
3	High Voltage	1969	2
4	Zenyattà Mondatta	1980	3

Tracks

Pos	Title	AlbumID	ArtistID
1	You Shook Me All Night Long	1	1
2	Hells Bells	1	1
1	Baby, Please Don't Go	2	1
1	Movin' On Out	3	2

Figure 1.4: Distributed setting in which our profiling algorithms operate. The relations of the datasets are dispersed among a computer cluster and can be horizontally partitioned. The workers in the cluster, which do the actual data profiling, can pairwise communicate over a network.

For this reason, all here proposed algorithms are designed with parallel and distributed execution as a first-class objective. We employ different distribution strategies that fit well with the discovery problem at hand. This is crucial, because a good algorithmic strategy is (in our experience) more important than massive exploitation of resources. That is, we propose novel algorithmic strategies for various dependency discovery problems aligned with execution strategies that effectively leverage resources in computer clusters. The cluster environment is the same for all presented algorithms and comprises a common hardware and software stack. With the help of Figure 1.4, let us briefly outline the components of a cluster, where the data to profile is stored, and how our algorithms can operate in this environment.

As mentioned above, a cluster consists of multiple computers, called workers. We assume the relations that should potentially undergo data profiling to be dispersed among the workers of the cluster. Furthermore, relations may be horizontally partitioned and the individual partitions may be replicated within the cluster to provide fault tolerance. For instance, in Figure 1.4 the two relations Records and Tracks are distributed in four partitions over four workers. Note that each worker can in principle access all data, although accesses to a non-local partition involve network traffic to a remote worker that hosts that partition. In our experiments, we use the Apache Hadoop Filesystem (HDFS) [Hadoop] to store datasets in the cluster in a distributed and redundant fashion.

As can be seen in Figure 1.4, the machine that initiates a data profiling task (*"the user"*) can be physically separated from the cluster that does all the heavy work, i. e., storing and profiling the data. How the cluster is utilized to profile the above described data depends completely on the particular profiling algorithm. In general, each algorithm employs some form of *task parallelism* (each worker performs a distinct task) or *data parallelism* (all workers perform the same tasks but on a different share of the dataset). In any case, the workers will have to communicate via network to exchange tasks and/or data. The workers might further subdivide their tasks or data locally to exploit *thread parallelism*, i. e., to distribute the work among their CPU cores. Eventually, the profiling results are either stored within the cluster, e. g., with METACRATE, or directly shipped to the user.

The research presented in this thesis has been carried out in the context of the Stratosphere project, which was financed by the German Research Foundation (DFG), grant no. FOR 1306 [FOR 1306]. The Stratosphere project has particularly borne the distributed processing framework Apache Flink, which is the basis for some of the proposed and distributed data profiling algorithms.

1.5 Thesis Structure and Contributions

Having laid out the motivation and goals of this thesis, let us give a brief overview of the following chapters. In fact, this thesis is structured along four main contributions, as we describe in the following.

Chapter 2 – IND discovery

We present a family of distributed IND discovery algorithms, SINDY [Kruse et al., 2015a], SANDY, and ANDY. Those algorithms are very easy to implement on Map/Reduce-based platforms, such as Apache Flink, and are highly efficient when executed on single machines or on computer clusters. In detail, SINDY discovers unary INDs. Its capability for parallel execution allows it to outperform existing IND algorithms on a single machine already when profiling large datasets. In addition, in our experiments SINDY exhibits a linear speed-up w. r. t. the number of workers when executed in a distributed setup. The algorithm SANDY is a modification of SINDY to discover partial INDs. Our experiments show that this generalization comes with virtually no performance overhead. Last but not least, ANDY is a further extension of SINDY to discover n-ary INDs and adopts a novel strategy to refine INDs into *core INDs* and *augmentation rules*. This unprecedented and yet unpublished approach can describe the discovered INDs more concisely and at the same time improve the efficiency and scalability of the discovery process: In our evaluation, we find ANDY to occasionally outperform existing IND discovery algorithms by more than an order of magnitude – in addition to the improvements gained through distributed execution. SINDY, SANDY, and ANDY have been developed by Kruse, while Papenbrock and Naumann contributed valuable discussions.

Chapter 3 – CIND discovery on RDF data

In this chapter, we propose a novel combination of dependency type and data model, namely conditional INDs (CINDs) on RDF data, and show that it can serve multiple different use cases, such as query optimization and ontology reverse engineering. We develop the algorithm RDFIND to discover CINDs on RDF data [Kruse et al., 2016a]. RDFIND builds upon algorithmic foundations of SINDY, but is tailored to the specific challenges of CIND discovery on RDF data. Our experimental evaluation shows that RDFIND outperforms state-of-the-art CIND discovery algorithms by a factor of up to 419 and scales to much larger datasets. As a matter of fact, RDFIND can process datasets of hundreds of gigabytes and is, thus, well applicable in real-word scenarios. Kaoudi and Quiané-Ruiz initiated the idea to discover CINDs on RDF data, Kruse developed and evaluated RDFIND, and Jentzsch, Papenbrock, and Naumann contributed valuable discussions.

Chapter 4 – Partial FD and UCC discovery
In this chapter, we develop the algorithm PYRO, which is capable of discovering both partial FDs and UCCs [Kruse and Naumann, 2018]. The discovery of FDs and UCCs is very different from IND discovery. However, FDs and UCCs are so closely related that we can discover both using common algorithmic principles. PYRO exploits those commonalities and proposes a unified discovery strategy that interleaves sampling-based error estimations and column-based error calculations to determine the desired partial dependencies. In contrast to SINDY and RDFIND, PYRO does not rely on Apache Flink for distributed execution but defines a custom distribution strategy. This is necessary because Flink's execution model is too rigid and abstract to express PYRO's algorithmic intricacies. In our experimental comparison to several state-of-the-art discovery algorithms, we observe PYRO to outperform all of them. In fact, even when using only a single machine and a single core, PYRO is up to 33 times faster than the best competitor (and up to 123 times faster with parallel execution enabled). PYRO was developed by Kruse, while Naumann contributed valuable discussions.

Chapter 5 – Data anamnesis with METACRATE
Having presented algorithms for the most important dependency discovery problems, we proceed to showcase the value of the discovered dependencies. More concretely, we introduce METACRATE, a system to store and analyze dependencies in a scalable, distributed fashion [Kruse et al., 2017a]. In particular, METACRATE ships with a library of analytics algorithms implemented on top of the execution engine and a choice of visualizations. We use our profiling algorithms and METACRATE to demonstrate several steps of a data anamnesis on a real-world dataset [Kruse et al., 2016b], including the technical reconstruction of the dataset schema and an assessment of its quality. METACRATE has been conceived and designed by Kruse and developed by Kruse and several student assistants, namely Fabian Tschirschnitz, Susanne Buelow, Lawrence Benson, Marius Walter, and David Hahn. The first data anamnesis was a joint effort of Papenbrock, Harmouch, and Kruse. Papenbrock contributed profiling algorithms for FDs and UCCs, proposed a detection of primary keys [Papenbrock and Naumann, 2017b], and analyzed potentials for schema normalization; Harmouch profiled individual columns to propose data types and `NOT NULL` constraints; Naumann contrasted the novel idea of data-driven schema assessment to traditional, application-driven assessment techniques; Kruse developed the general idea of data anamneses, contributed SINDY for IND discovery, proposed an algo-

rithm for FK detection, proposed several metrics to assess schemata based on metadata, and took care of the integration of the proposed techniques with Metacrate.

Finally, in Chapter 6, we draw conclusions from our work. In a more general setting, we reason about potential implications of our research results on real-world scenarios. Along these lines, we propose future directions for data profiling in terms of both research and practical aspects.

Chapter 2

Unary, N-ary, and Partial Inclusion Dependency Discovery

Inclusion dependencies (INDs) are one of the fundamental dependency types in relational databases, in particular, because they generalize the fundamental concept of *foreign keys* [Codd, 1970]. In few words, an IND states that every combination of values that appears in a certain combination of columns of one table also exists in a certain other combination of columns of a potentially distinct table. Let us formalize this notion.

Definition 2.1 (Inclusion dependency)**.** Let r and s be two relational instances with relation schemata $R = (A_1, ..., A_i)$ and $S = (B_1, ..., B_j)$, respectively. Then we call $R[A_{i_1}, ..., A_{i_n}] \subseteq S[B_{j_1}, ..., B_{j_n}]$ an IND if and only if:[1]

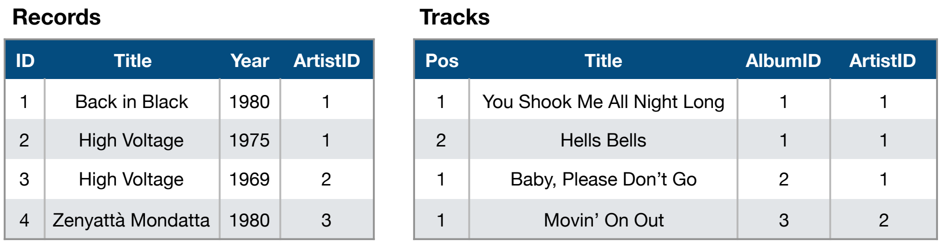
$$\forall t_r \in r\colon \exists t_s \in s\colon t_r[A_{i_1}, ..., A_{i_n}] = S[B_{j_1}, ..., B_{j_n}]$$

$R[A_{i_1}, ..., A_{i_n}]$ is called the *dependent* or *left-hand side* (*LHS*) of the IND and $S[B_{j_1}, ..., B_{j_n}]$ is called the *referenced* or *right-hand side* (*RHS*). Furthermore, we refer to n as the *arity* of the IND. As a convention, we may omit the relation names from the IND wherever those names are not relevant to the context.

Example 2.1. Consider the dataset in Figure 2.1. It contains the *binary* IND (of arity 2) Tracks[ArtistID, AlbumID] $\subseteq$ Records[ArtistID, ID]. INDs can also span only a single relation, such as the (spurious) *unary* IND Tracks[Pos] $\subseteq$ Tracks[AlbumID].

Arguably, the most prominent application of INDs is database reverse engineering. There, INDs help to determine appropriate foreign keys in relational databases [Rostin et al., 2009; Zhang et al., 2010]: In general, every value in a foreign key column should also exist in the referenced column. Hence, INDs can be interpreted as foreign key candidates. However, INDs can also be used for query rewriting [Cheng et al., 1999; Gryz, 1998b, 1999; Johnson and Klug, 1984] and improved schema design with less redundancy and fewer update anomalies [Levene and Vincent, 1999; Ling and Goh, 1992; Mannila and Raiha, 1986].

[1]In literature, one often encounters an additional restriction that the attributes in an IND be pairwise independent. We discuss such pragmatic constraints in Section 2.4.1.



Records

ID	Title	Year	ArtistID
1	Back in Black	1980	1
2	High Voltage	1975	1
3	High Voltage	1969	2
4	Zenyattà Mondatta	1980	3

Tracks

Pos	Title	AlbumID	ArtistID
1	You Shook Me All Night Long	1	1
2	Hells Bells	1	1
1	Baby, Please Don't Go	2	1
1	Movin' On Out	3	2

Figure 2.1: An example dataset with discographical data.

Furthermore, partial INDs (INDs that are only partially fulfilled, cf. Section 1.3) are of special interest. In the field of data integration, partial INDs can help to determine overlapping parts of distinct databases [Bauckmann, 2013]. These overlapping parts can then be used as so-called *value correspondences* for data integration tasks [Miller et al., 2001]. What is more, in data cleaning scenarios partial INDs point to data errors. Automatic repair algorithms can incorporate those when determining how to clean the dataset [Bohannon et al., 2005].

On the face of these applications, it is desirable to automatically discover all INDs in a given (relational) dataset if they are not known in the first place. A peculiarity of INDs is that they can span two relations. In consequence, IND discovery algorithms have to consider all relations in a database at the same time, thereby incurring a high data volume to cope with. Therefore, it seems appropriate to tackle the discovery problem with a distributed algorithm, so as to employ as many computing resources as possible. In this chapter, we present a family of distributed IND discovery algorithms for the discovery of unary, n-ary, and partial INDs that outperform state-of-the-art discovery algorithms when provided sufficient resources.

We start by discussing related work in Section 2.1. In particular, we show that existing IND discovery algorithms follow a common algorithmic approach that is highly amenable for distributed processing. Based on this insight, we develop the distributed algorithm SINDY for unary INDs in Section 2.2. Due to the particular importance of partial INDs, we furthermore propose a corresponding adaptation of SINDY, called SANDY, to relax the discovery in Section 2.3. In Section 2.4, we proceed to extend SINDY for the discovery of n-ary INDs. The resulting algorithm, ANDY, employs a novel perspective on INDs that emerges from the interaction of INDs and functional dependencies (FDs). Eventually, we evaluate all proposed algorithms with a particular focus on efficiency and scalability in Section 2.5 and describe prospect research questions in Section 2.6.

2.1 Related Work

In 1970, Codd introduced the concept of *foreign keys* [Codd, 1970] as a means to preserve data integrity. It was only later, that this concept was generalized to INDs [e. g., Fagin, 1981] and *referential integrity* [Date, 1981]. While the theoretical properties of INDs, in particular the implication problem and the interaction of INDs and FDs, have been studied thoroughly since the 1980s [e. g., Casanova et al., 1982; Chandra and Vardi, 1985; Kanellakis et al., 1983], automatic discovery has been researched only a decade later. In the following, we focus on data-driven discovery methods that determine *all* valid inclusion dependencies in a given dataset. An orthogonal approach would be to discover INDs from SQL queries [e. g., Tan and Zhao, 2003], but this approach will not discover all INDs of a dataset and works only if a query log or application source code is present.

Unary IND discovery

First foundations for the discovery of INDs have been laid out by Kantola et al. [1992], who acknowledge that the discovery of unary INDs is of quadratic complexity in the number of attributes and that a subproblem of n-ary IND discovery, namely finding only a single high-arity IND, is already NP-complete. Furthermore, the authors recognize the potential for pruning IND candidates by considering the datatypes of the involved columns and via the *projection rule* [Casanova et al., 1982]: If $R[A_{i_1}, ..., A_{i_n}] \subseteq S[B_{j_1}, ..., B_{j_n}]$ is an IND, so are $R[A_{i_1}, ..., A_{i_n-1}] \subseteq S[B_{j_1}, ..., B_{j_{n-1}}]$, $R[A_{i_1}, ..., A_{i_n-2}, A_{i_n}] \subseteq S[B_{j_1}, ..., B_{j_{n-2}}, B_{j_n}]$, ..., and $R[A_{i_2}, ..., A_{i_n}] \subseteq S[B_{j_2}, ..., B_{j_n}]$.

A first concrete algorithm for the disovery of unary INDs has been proposed by Bell and Brockhausen [1995]. In their work, IND candidates are tested via SQL queries. These checks are expensive and to reduce their number, the authors propose to exploit the *transitivity rule* [Casanova et al., 1982], that is, if $R[A] \subseteq S[B]$ and $S[B] \subseteq T[C]$ are INDs, then $R[A] \subseteq T[C]$ is an IND, too. Akin to this work, the `NavLog'` algorithm uses SQL statements to determine interesting *partial* INDs, (INDs whose LHS is almost, but not completely included in the RHS) among a given set of columns [Lopes et al., 2002b].

Still, SQL-based approaches are rather inefficient as they require numerous passes over the profiled dataset and repeat data processing operations among IND candidates that share columns. To this end, de Marchi et al. [2002] proposed the IND discovery algorithm MIND that passes over the dataset once, constructs an inverted index from it, and then determines INDs as exact association rules in this inverted index. The entries in the inverted index are of the form $v \rightarrow A_v$, where v is a value from the profiled dataset and A_v is the set of all columns that contain v. While this algorithmic approach is also used by many other algorithms, including SINDY, the original algorithm does not specify an efficient and scalable execution strategy. In fact, this algorithm requires the inverted index to fit in main memory.

The SPIDER algorithm proposes to sort all columns of the profiled dataset and then synchronously iterate all the sorted columns, like in a sort-merge join [Bauckmann et al., 2006]. Doing so, the algorithm implicitly creates above mentioned inverted index, as for every value v of the original dataset it can determine all columns A_v that contain v by means of the synchronous iteration. Because SPIDER uses out-of-core (i. e., disk-based) sorting and need not materialize the inverted index, it is more scalable than MIND. However, for datasets with several thousand columns it is prone to crash, because it has to maintain an open file for each column. The rather similar algorithm S-INDD [Shaabani and Meinel, 2015] avoids this issue by merging sorted columns repeatedly, thereby maintaining only k open files for some user-defined k. Neither SPIDER nor S-INDD are capable to scale across cores or computers for further performance and scalability improvements, though.

The algorithm BINDER improves upon SPIDER by hash-partitioning the profiled dataset rather than sorting it [Papenbrock et al., 2015c]. The partitions are dynamically refined such that they fit into main memory. Once a partition fits into main memory, BINDER applies an improved version of MIND's IND extraction procedure. The partitions are processed one after another and in case the already processed partitions could falsify all IND candidates for a certain attribute, that attribute need not be considered in further partitions anymore. As we see in our evaluation, such pruning, which is not applied in SINDY, does not improve performance substantially. Nonetheless, BINDER is scalable because of its out-of-core partitioning and because it materializes only slices of the inverted index in main memory at once. In contrast to SINDY, BINDER is also not capable to scale across cores or computers, though. Still, it is to our knowledge the most efficient single-machine IND discovery algorithm and thus serves us as baseline in our experimental evaluation of SINDY.

Unlike the above algorithms, the recent algorithm MANY has not been devised for the discovery of INDs in relational databases but among web tables [Tschirschnitz et al., 2017]. Those are usually much smaller than database tables; however, they come in much greater quantities, thereby spanning a huge search space. MANY tackles this specific problem by approximating the INDs with a Bloom filter first, which has also been considered in the context of SPIDER [Bauckmann et al., 2010]. In addition, MANY defines domain-specific filters to prune IND candidates that do not seem interesting. For instance, a filter discards columns that contain numbers only. While MANY defines a parallelization strategy, it is not capable to scale across computer clusters.

Let us finally note that also algorithms for the *approximate discovery* of INDs have been developed that are based on data summaries rather than inverted indices. Concretely, Dasu et al. [2002] propose to discover join paths using signatures of columns. Those precomputed signatures comprise the minimum values of k independent hash functions applied to the respective columns. In an orthogonal approach, Brown and Hass [2003] seek to determine all columns A that reference a certain key column B by means of an IND by sampling A and testing every sampled value for inclusion in B. Finally, Zhang et al. [2010] propose to discover potential INDs using bottom-k sketches. However, we have shown that a combination of sampled inverted indices and HyperLogLog structures yields more accurate results, in particular when the profiled database contains columns with very few *and* very many distinct values [Kruse et al., 2017b]. Nevertheless, unlike above described algorithms and SINDY, those algorithms cannot guarantee to find all correct INDs in the profiled dataset.

Partial IND discovery

Partial INDs constitute a relaxation of (exact) INDs that require the values of two columns to overlap to some extent. A proper inclusion among the columns is not necessary, though. A possible definition for the "partiality" of INDs has been proposed by Lopes et al. [2002b], namely the fraction of distinct values of the LHS of a partial IND that also exist in the RHS. This fraction amounts to 1 if the partial IND is exact, and to 0 if the columns are disjoint. Our partial IND discovery algorithm SANDY is also based on this measure.

Technically, the discovery of partial INDs is closely related to the discovery of exact INDs. Partial INDs can be determined by SQL statements [Koeller and Rundensteiner, 2006; Lopes et al., 2002b; Petit et al., 1996]. Furthermore, SPIDER proposes a simple adaptation to also discover partial INDs [Bauckmann et al., 2010]. This proceeding is, again, more efficient than its SQL-based counterparts. SANDY adopts a scheme similar to that of SPIDER. Additionally, we briefly reason on the applicability of partiality measures to discover n-ary partial INDs.

In a different line of work, Memari [2016, Chapter 8] deals with the specific issue of discovering foreign keys on incomplete data, that is, data with `NULL` values. Particularly, the author discerns three different interpretations of `NULL`s, namely *simple*, *partial*, and *full semantics*. In this chapter, we propose to ignore `NULL`s as most relational database systems do when enforcing foreign keys (FKs); this interpretation coincides with the "simple" semantics. Furthermore, the author describes various algorithms to discover (partial) foreign keys (FKs) – one for each interpretation of `NULL`s. Those algorithms are very different from SANDY: At first, they check only such FK candidates (or IND candidates, respectively) that reference a unique column combination. And second, the algorithms only *estimate* whether a pair of columns or column combinations, respectively, satisfy the IND property. Hence, the proposed algorithms are *approximate* according to our classification scheme from Section 1.3. For that matter, the algorithms extend the above mentioned work of Zhang et al. [2010].

n-ary IND discovery

The most common approach for n-ary IND discovery has been first proposed by de Marchi et al. [2002] as part of the MIND algorithm. The general idea is to first determine the unary INDs with the above described inverted index. Then, all binary IND candidates are generated that are still admissible by means of the unary INDs and the projection rule. The candidate generation is a modification of the broadly known *AprioriGen* algorithm for frequent item set mining [Agrawal and Srikant, 1994]. Those binary IND candidates are then checked with SQL statements, and the verified INDs are used to create ternary IND candidates. This procedure repeats until no IND candidates are generated anymore.

As stated above, testing IND candidates with SQL statements can be rather inefficient. For that reason, SPIDER [Bauckmann et al., 2010] and BINDER [Papenbrock et al., 2015c] can apply their unary IND discovery approach also to verify n-ary IND candidates, just by treating column combinations like single columns. Their candidate generation is also AprioriGen-based.[2] The here presented algorithm ANDY follows this path, too. However, it incorporates a further pruning rule based on functional dependencies that can spare the generation of many IND candidates, let alone their discovery. To evaluate

[2]BINDER's candidate generation does not embrace the full Apriori idea and might create futile candidates.

the effectiveness of Andy's pruning, we compare it to the basic candidate generation procedures of Mind and Binder in our evaluation.

Another line of work specializes on the discovery of INDs with a high arity. Concretely, the algorithms Find_2 [Koeller and Rundensteiner, 2002] and ZigZag [de Marchi and Petit, 2003] initially discover INDs until some user-defined arity k in an apriori-based fashion, but then determine and check the *maximum IND candidates*, thereby skipping potentially very many IND candidates. This approach has two major drawbacks, though: At first, this approach works well only in the presence of INDs of very high arity, which are found rather rarely in our experience. Second, these optimistic search strategies are formulated only for two tables and it is not clear how to generalize them to an arbitrary number of tables. Andy, in contrast, operates on an arbitrary number of tables and its pruning takes effect already after unary INDs have been discovered.

A third approach for n-ary IND discovery is rather different from the two above approaches: Instead of generating IND candidates and then testing them, unary INDs are *merged* to INDs of higher arity. Concretely, de Marchi [2011] laid the foundations of how INDs can be merged without giving a concrete algorithm. The basic idea is that every IND $R[X] \subseteq S[Y]$ connects tuples across R and S that agree in their values of attributes X or Y, respectively. Two INDs can be merged, if they pertain to the same tables and connect more or less the same tuples across those tables. Shaabani and Meinel [2016] then proposed Mind_2, which fleshes out this basic idea with a concrete algorithm. As is the case for ZigZag and Find_2, Mind_2's search strategy applies to only two tables, though. Furthermore, none of the mentioned algorithms is scalable across CPU cores or computer clusters.

2.2 Validating Inclusion Dependency Candidates

Let us now describe how to discover all *unary* INDs of a given dataset in a distributed manner. As we show in Section 2.4, with only minor modifications the algorithm presented here can be reused to validate n-ary IND candidates as well – hence, the section title. Concretely, this section presents the Sindy algorithm (Scalable Ind Discovery) as it was described in [Kruse et al., 2015a]. To preserve the integrity of the name Sindy, we will refer to the modified algorithms for partial and n-ary IND discovery as Sandy and Andy, respectively.

2.2.1 Algorithmic foundations

As described in Section 2.1, Sindy builds upon the same algorithmic foundation as Mind [de Marchi et al., 2002], Spider [Bauckmann et al., 2006], Binder [Papenbrock et al., 2015c], and S-INDD [Shaabani and Meinel, 2015]: Given a relational database r with columns A_i, we build its inverted index that associates all values v in r with the set of columns v appears in, denoted as $\mathcal{A}_v$. Hence, the inverted index consists of tuples $(v, \mathcal{A}_v)$. As we show in the following, any *unary, exact* association rule of attributes in the $\mathcal{A}_v$ sets corresponds to a unary IND. In other words, $A_m \subseteq A_n$ is a valid unary IND if and only if every $\mathcal{A}_v$ set that contains A_m also contains A_n.

Theorem 2.1. *Let r be a relational database with columns A_i. Further, let $i_r = \{(v_1, \mathcal{A}_{v_1}), \dots\}$ be its inverted index that associates every value v_i in r with the set of columns $\mathcal{A}_{v_i}$ that v_i appears in. Then $A_m \subseteq A_n$ is a unary* IND *in r if and only if*

$$\forall (v, \mathcal{A}_v) \in i_r \colon (A_m \in \mathcal{A}_v \Rightarrow A_n \in \mathcal{A}_v)$$

Proof. By definition, we have $A_m \subseteq A_n$ if and only if any value v from column A_m also exists in column A_n. Hence, the corresponding $\mathcal{A}_v$ contain A_m and A_n. For any other value v' not contained in A_m, the corresponding $\mathcal{A}_{v'}$ does not contain A_m. □

Note that Theorem 2.1 generalizes the original formulation by de Marchi et al. [2002, Property 1]:

$$A_m \subseteq A_n \quad \Leftrightarrow \quad A_n \in \bigcap_{(v, \mathcal{A}_v) \in i_r | A_m \in \mathcal{A}_v} \mathcal{A}_v \tag{2.1}$$

Both formulations are equivalent unless A_m is not present at all in the inverted index. This can occur (i) when the relation containing A_m does not comprise any tuples or (ii) when certain values should explicitly be excluded from the inclusion dependency. The latter case is particularly sensible when one wants to treat `NULL` values as the absence of a value. This view is consistent with the foreign key semantics of SQL and can be easily achieved by disallowing an entry with the `NULL` value as key in the inverted index. It is easy to see that, if A_m comprises no or only excluded values, the intersection condition in Equation 2.1 ($A_m \in \mathcal{A}_v$) is not fulfilled by any entry in the inverted index: The n-ary intersection becomes empty (i. e., n=0), which is usually defined to be the empty set. Hence, for any other column A_n, Theorem 2.1 admits $A_m \subseteq A_n$, while Equation 2.1 does not. In the following, we denote such INDs as *void*, as the involved columns do not share any values.

Having settled the algorithmic foundations of unary IND discovery, we now proceed to show how SINDY applies them in a distributed, scalable fashion. In particular, we show how to build the inverted index (Section 2.2.2) and how to extract the INDs from it (Section 2.2.3). It is worth noting that these steps naturally fit with the Map/Reduce paradigm and can thus be easily implemented several distributed data flow systems, such as Apache Flink.

2.2.2 Convert tables into attribute sets

As described above, the first step of unary IND discovery is to convert a given relational database into an inverted index. We refer to the $\mathcal{A}_v$ in the inverted index as *attribute sets* and, because we do not need the actual values v for the subsequent IND extraction anymore, we are actually interested in those attribute sets only.

Recall from Section 1.4 that the relations to be profiled are dispersed among the workers of the cluster and can be horizontally partitioned. As an example, Figure 2.2 displays a distribution of the Records relation from Figure 2.1 in a cluster with two workers. Here, we consider only a single relation to not overload the example. The proceeding with multiple relations is exactly the same, though. For that matter, SINDY does not even need a notion of relations and operates only on columns.

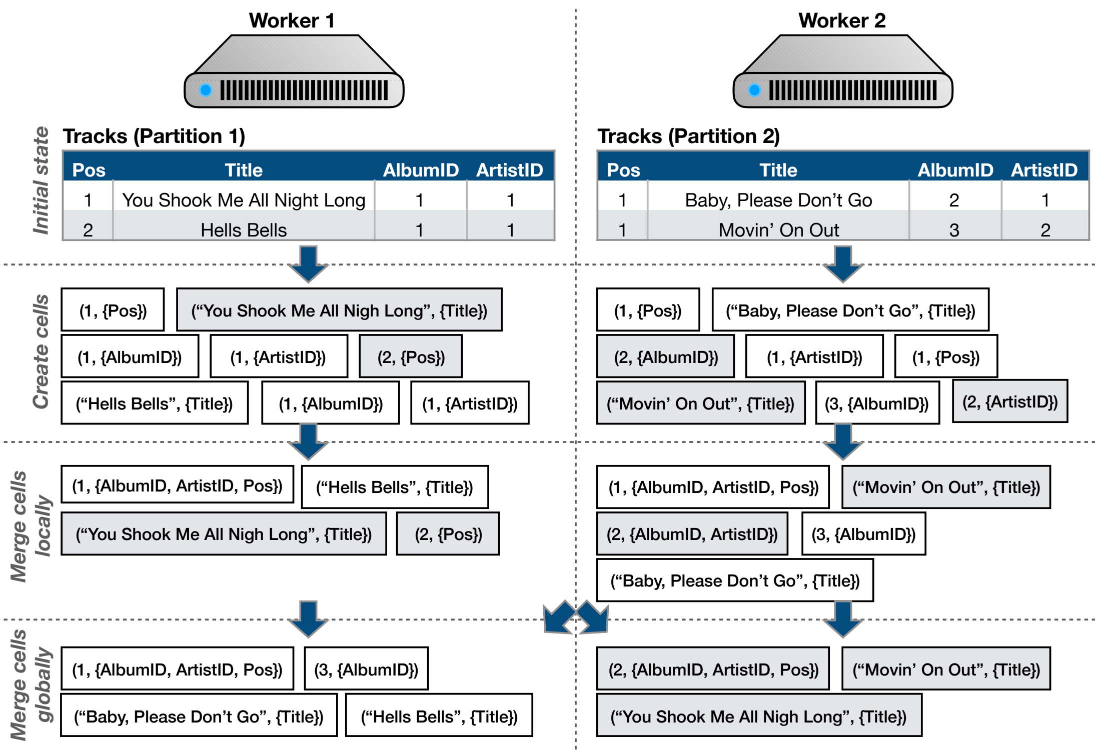

Worker 1
Worker 2
Initial state
Tracks (Partition 1)
Pos | Title | AlbumID | ArtistID
1 | You Shook Me All Night Long | 1 | 1
2 | Hells Bells | 1 | 1
Tracks (Partition 2)
Pos | Title | AlbumID | ArtistID
1 | Baby, Please Don't Go | 2 | 1
1 | Movin' On Out | 3 | 2
Create cells
(1, {Pos})
("You Shook Me All Nigh Long", {Title})
(1, {AlbumID})
(1, {ArtistID})
(2, {Pos})
("Hells Bells", {Title})
(1, {AlbumID})
(1, {ArtistID})
(1, {Pos})
("Baby, Please Don't Go", {Title})
(2, {AlbumID})
(1, {ArtistID})
(1, {Pos})
("Movin' On Out", {Title})
(3, {AlbumID})
(2, {ArtistID})
Merge cells locally
(1, {AlbumID, ArtistID, Pos})
("Hells Bells", {Title})
("You Shook Me All Nigh Long", {Title})
(2, {Pos})
(1, {AlbumID, ArtistID, Pos})
("Movin' On Out", {Title})
(2, {AlbumID, ArtistID})
(3, {AlbumID})
("Baby, Please Don't Go", {Title})
Merge cells globally
(1, {AlbumID, ArtistID, Pos})
(3, {AlbumID})
("Baby, Please Don't Go", {Title})
("Hells Bells", {Title})
(2, {AlbumID, ArtistID, Pos})
("Movin' On Out", {Title})
("You Shook Me All Nigh Long", {Title})

Figure 2.2: Creation of the inverted index for the data from Figure 2.1 with two workers. For simplicity, we depict only the Tracks relation. The color of the boxes indicates on which worker the contained value will be handled eventually.

In the first step towards the attribute sets, every worker creates *cells* from its local share of the input relation. In fact, for each tuple with n fields a worker creates n cells – one for each field. A cell itself comprises (i) the value of the respective field and (ii) a singleton set of the column to which that field belongs. For instance in Figure 2.2, Worker 1 creates the cell $(1, \{\mathsf{Pos}\})$ for the Pos field of the first tuple in its local partition.

Next, cells with the same value that reside on the same worker are merged by calculating the union of their attributes. For instance, Worker 2 can merge the cells $(2, \{\mathsf{AlbumID}\})$ and $(2, \{\mathsf{ArtistID}\})$ to $(2, \{\mathsf{AlbumID}, \mathsf{ArtistID}\})$. Merging cells *locally* reduces the network load in the final step: Cells with the same value are eventually merged *globally*. For n workers, SINDY applies a partitioning function $p\colon \mathbb{V} \to \{1, ..., n\}$ to the value of each cell to determine to which worker that cell should be sent. Assume that in our example p determines that cells with the value 1 should be sent to Worker 1. This proceeding guarantees that all cells with the same value end up on the same worker, which can then perform the global merging to create the final inverted index entries.

It is worth noting that the three above described steps (create cells, merge cells locally, and merge cells globally) need not be executed one after another, but can be pipelined instead. As soon as cells are created for an input tuple, they can immediately be merged with locally existing cells. Furthermore, it is not necessary to merge local cells exhaustively. Rather, it is sufficient to merge as many cells locally as is possible in the light of available main memory on each worker. As soon as the main memory is fully occupied, the pre-merged cells can be forwarded to the workers that do the global merging. Thus, SINDY avoids expensive out-of-core execution with disk accesses in this step. However, the global merging of cells might involve out-of-core execution to accommodate datasets that exceed the main memory capacity of the worker.

Eventually, each worker can strip the values from its final, merged cells to yield the attribute sets for the input relations. Note that the attribute sets are again distributed in the cluster and, hence, can be further processed in parallel by all workers.

2.2.3 Extract INDs from attribute sets

As per Theorem 2.1, the attribute sets are sufficient to determine the unary INDs of a dataset. SINDY performs the IND extraction in two phases. At first, it extracts the non-void INDs using Equation 2.1, which is particularly amenable to distributed computation. Then, it determines all void INDs.

Figure 2.3 continues the example from Section 2.2.2 and shows how to extract the non-void INDs from the attribute sets on two workers. At first, each attribute set $\mathcal{A}$ is converted into a set of *IND candidate sets* $A \subseteq \mathcal{A} \backslash A$ for all $A \in \mathcal{A}$. For instance, the attribute set $\{\mathsf{AlbumID}, \mathsf{ArtistID}, \mathsf{Pos}\}$ on Worker 1 yields, amongst others, the IND candidate set $\mathsf{AlbumID} \subseteq \{\mathsf{ArtistID}, \mathsf{Pos}\}$. Intuitively, it states that $\mathsf{AlbumID} \subseteq \mathsf{ArtistID}$ and $\mathsf{AlbumID} \subseteq \mathsf{Pos}$ might be valid INDs, but there cannot be any other valid INDs with AlbumID as LHS. The rationale for this interpretation is that there exists some value v in the profiled database that yielded said attribute set (in this example, that value is $v = 1$; cf. Figure 2.2). This implies that v exists *only* in the columns AlbumID, ArtistID, and Pos. As a result, AlbumID (and also ArtistID and Pos, for that matter) cannot be included in any other column.

To consolidate the IND candidate sets with the same LHS, we merely need to intersect their RHSs. SINDY achieves this with the same two-phase approach as for the creation of the attribute sets: IND candidate sets are at first merged locally. Then, IND candidate sets with the same LHS are brought to a single workers by means of a partition function. Finally, that worker concludes the consolidation of those candidate sets and obtains the final *IND sets*, which represent the actual non-void INDs of the input dataset. As an example, the IND set $\mathsf{Pos} \subseteq \{\mathsf{ArtistID}, \mathsf{AlbumID}\}$ represents the INDs $\mathsf{Pos} \subseteq \mathsf{ArtistID}$ and $\mathsf{Pos} \subseteq \mathsf{AlbumID}$. As for the creation of the inverted index, the above described IND extraction steps can be pipelined.

At this point, it remains to be shown how SINDY can determine the void INDs. For that purpose, it keeps track of which columns appeared as a LHS of any of the IND sets, regardless of whether this IND set does not describe any IND, such as $\mathsf{Title} \subseteq \{\}$. If a column A does not appear as an LHS, that means that A did not cause the creation of a cell as described in Section 2.2.2, which in turn means that A is empty or consists of ignored values only (cf. Section 2.2.1). In that case, A is included in any other column of the profiled dataset, so SINDY declares the IND $A \subseteq B$ for all columns $B \neq A$.

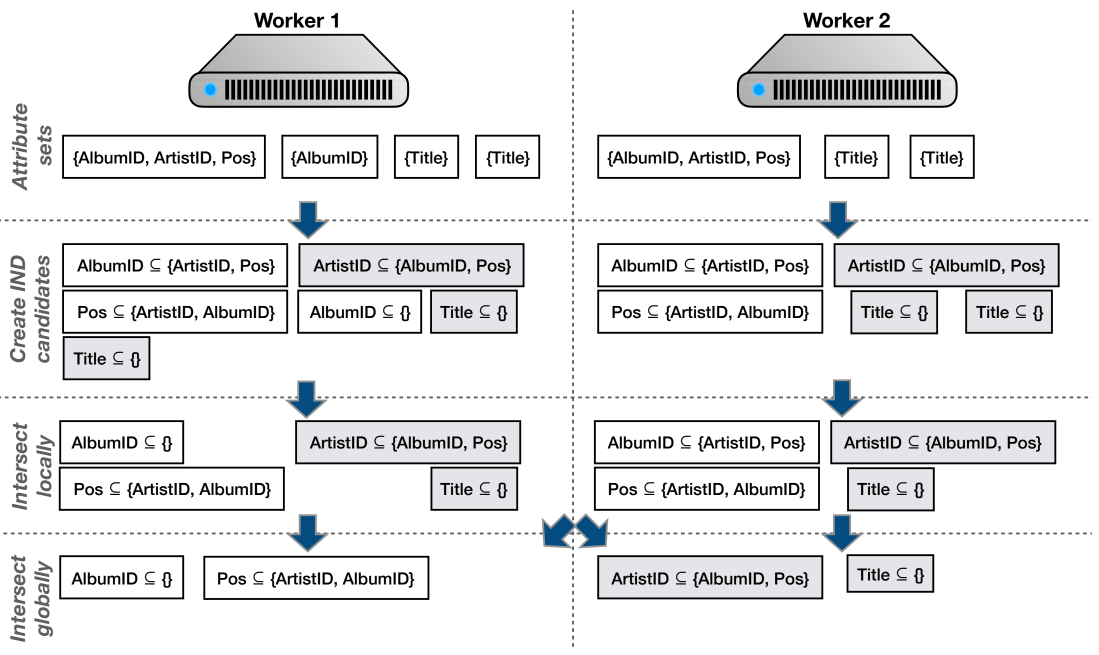

Worker 1
Worker 2
Attribute sets
{AlbumID, ArtistID, Pos}
{AlbumID}
{Title}
{Title}
{AlbumID, ArtistID, Pos}
{Title}
{Title}
Create IND candidates
AlbumID ⊆ {ArtistID, Pos}
ArtistID ⊆ {AlbumID, Pos}
Pos ⊆ {ArtistID, AlbumID}
AlbumID ⊆ {}
Title ⊆ {}
Title ⊆ {}
AlbumID ⊆ {ArtistID, Pos}
ArtistID ⊆ {AlbumID, Pos}
Pos ⊆ {ArtistID, AlbumID}
Title ⊆ {}
Title ⊆ {}
Intersect locally
AlbumID ⊆ {}
ArtistID ⊆ {AlbumID, Pos}
Pos ⊆ {ArtistID, AlbumID}
Title ⊆ {}
AlbumID ⊆ {ArtistID, Pos}
ArtistID ⊆ {AlbumID, Pos}
Pos ⊆ {ArtistID, AlbumID}
Title ⊆ {}
Intersect globally
AlbumID ⊆ {}
Pos ⊆ {ArtistID, AlbumID}
ArtistID ⊆ {AlbumID, Pos}
Title ⊆ {}

Figure 2.3: Extraction of the unary INDs from attributes sets on two workers. The color of the boxes indicates on which worker the contained IND candidates will be handled eventually.

2.2.4 Implementation on Distributed data flow Systems

A practical advantage of SINDY is that it can be easily implemented on distributed data flow systems, such as Apache Flink [Alexandrov et al., 2014; Flink], Apache Spark [Spark; Zaharia et al., 2012], RHEEM [Agrawal et al., 2016], and Apache Hadoop [Hadoop]. This not only renders the implementation of SINDY per se very easy, but it also transparently provides efficient data processing techniques to the IND discovery, such as distributed computing, out-of-core execution, and cache-aware data structures.

Figure 2.4 outlines how SINDY can be expressed as a data flow that can be deployed on the above mentioned systems. The creation of cells and IND candidate sets is done using Flatmap operators, which can produce multiple output elements for every input element. The merging of cells and IND candidate sets, respectively, can be expressed as a chain of a Combine operator for the local merging and a Reduce operator for the global merging. Note that the Combine operator can be omitted if it is not supported by the underlying platform, although this might impair SINDY's efficiency.

While the data flow defines the general outline of the algorithm, it is also important to carefully model the data and the *user-defined functions* (*UDFs*), which run inside of the operators. As far as data is concerned, it is important to keep the memory footprint low whenever data is moved between workers. For that purpose, we encode attributes as integers and represent attribute sets as sorted integer arrays. They are not only small in size but also allow for efficient set union (to merge cells) and set intersection (to consolidate IND candidate sets) operations: To union or intersect two arrays within a

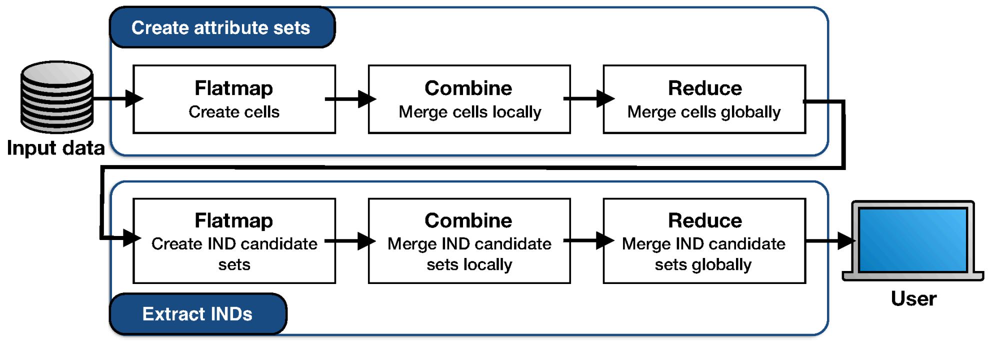


Figure 2.4: Implementation scheme of SINDY as a parallelizable data flow.

UDF, we iterate them synchronously as in a sort-merge join, thereby producing a new result array.

2.3 Partial Inclusion Dependencies

Having laid out how to discover unary INDs in a distributed fashion, let us now show the necessary modifications to discover also partial unary INDs. At first, however, we need to define an error measure to quantify the "partiality" of INDs (cf. Section 1.3). For that purpose, we employ the g_3' error as defined by Lopes et al. [2002b]:

Definition 2.2 (IND error g_3')**.** Let r and s be two (possibly identical) relational instances and let $R[A] \subseteq S[B]$ be an IND candidate among them. Then its error is

$$g_3'(R[A] \subseteq S[B]) := \frac{|\{v \mid \exists t_r\colon t_r[A] = v \land \nexists t_s \in s\colon t_s[B] = v\}|}{|\{v \mid \exists t_r\colon t_r[A] = v\}|}$$

Example 2.2. Consider the IND candidate Records[ID] $\subseteq$ Tracks[AlbumID] for the dataset in Figure 2.1. Intuitively, the g_3' error describes the portion of values in Records[ID] that are not included in Tracks[AlbumID]. Hence, the error of the candidate is $\frac{1}{4}$.

On the basis of Definition 2.2, we can now precisely describe the problem statement for partial IND discovery: Given a relational dataset and user-defined error threshold $e_{\max}$, we want to determine all partial INDs in the dataset whose g_3' error is not greater than $e_{\max}$. The definition above also hints on how to solve this discovery task: We need to determine the number of distinct values of each column and the number of overlapping distinct values of all column pairs. As we show in the following, this can be done with very little overhead over exact IND discovery by amending SINDY. We refer to the resulting algorithm as SANDY.

Just like SINDY, SANDY starts by creating sets of attributes for each value that appears in the input dataset (cf. Section 2.2.2). On this data structure, the number of distinct values of a column is simply the number of attribute sets containing that column; and the number of overlapping distinct values of two columns is merely the number of attribute sets containing both columns simultaneously.

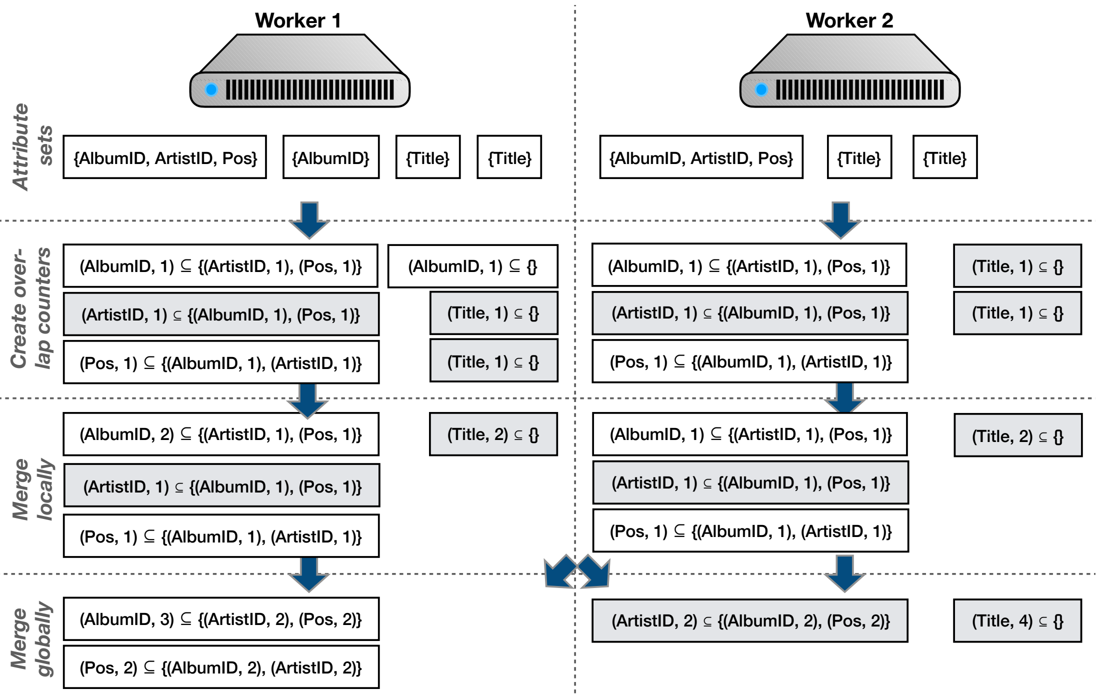

Worker 1
Worker 2
Attribute sets
Create overlap counters
Merge locally
Merge globally
{AlbumID, ArtistID, Pos}
{AlbumID}
{Title}
{Title}
{AlbumID, ArtistID, Pos}
{Title}
{Title}
(AlbumID, 1) ⊆ {(ArtistID, 1), (Pos, 1)}
(ArtistID, 1) ⊆ {(AlbumID, 1), (Pos, 1)}
(Pos, 1) ⊆ {(AlbumID, 1), (ArtistID, 1)}
(AlbumID, 1) ⊆ {}
(Title, 1) ⊆ {}
(Title, 1) ⊆ {}
(AlbumID, 1) ⊆ {(ArtistID, 1), (Pos, 1)}
(ArtistID, 1) ⊆ {(AlbumID, 1), (Pos, 1)}
(Pos, 1) ⊆ {(AlbumID, 1), (ArtistID, 1)}
(Title, 1) ⊆ {}
(Title, 1) ⊆ {}
(AlbumID, 2) ⊆ {(ArtistID, 1), (Pos, 1)}
(ArtistID, 1) ⊆ {(AlbumID, 1), (Pos, 1)}
(Pos, 1) ⊆ {(AlbumID, 1), (ArtistID, 1)}
(Title, 2) ⊆ {}
(AlbumID, 1) ⊆ {(ArtistID, 1), (Pos, 1)}
(ArtistID, 1) ⊆ {(AlbumID, 1), (Pos, 1)}
(Pos, 1) ⊆ {(AlbumID, 1), (ArtistID, 1)}
(Title, 2) ⊆ {}
(AlbumID, 3) ⊆ {(ArtistID, 2), (Pos, 2)}
(Pos, 2) ⊆ {(AlbumID, 2), (ArtistID, 2)}
(ArtistID, 2) ⊆ {(AlbumID, 2), (Pos, 2)}
(Title, 4) ⊆ {}

Figure 2.5: Determining the distinct value of columns and the overlapping distinct values of column pairs from attributes sets on two workers. The color of the boxes (gray/white) indicates on which worker the contained overlap sets will be handled eventually.

As exemplified in Figure 2.5, the distinct and overlapping values can be calculated in a single pass over the attribute sets. The basic idea is to create for every attribute set $\mathcal{A}$ several *overlap counters* $(A, 1) \subseteq \{(A'_1, 1), (A'_2, 1), ...\}$ $(A'_i \in \mathcal{A}\backslash\{A\})$, one for each attribute $A \in \mathcal{A}$. Concretely, the "1" in $(A, 1)$ is meant to count the distinct values of A while the "1"s in the $(A'_i, 1)$ count the distinct values of A'_i that overlap with A.

In the subsequent merge phase, Sandy merges the overlap counters with the same LHS attribute, thereby adding the distinct value counts of matching columns in both the LHS and RHS. Eventually, the overlap counters contain all information relevant to determine the partial INDs. For instance, the final overlap counter $(\mathsf{AlbumID}, 3) \subseteq \{(\mathsf{ArtistID}, 2), (\mathsf{Pos}, 2)\}$ in Figure 2.5 states that AlbumID has three distinct values of which two also exist in ArtistID and two (potentially different) values exist in Pos. That is, we get $g'_3(\mathsf{AlbumID} \subseteq \mathsf{ArtistID}) = g'_3(\mathsf{AlbumID} \subseteq \mathsf{Pos}) = \frac{1}{3}$ and report them as partial INDs unless $e_{\max} < \frac{1}{3}$.

At this point, the reader might wonder why Sandy performs duplicate work: After all, the number of overlapping values of AlbumID and ArtistID has also been calculated in the overlap counter $(\mathsf{ArtistID}, 2) \subseteq \{(\mathsf{AlbumID}, 2), (\mathsf{Pos}, 2)\}$, so we could have avoided one of the two overlap calculations. However, the merging of overlap counters can usually be done very quickly in comparison to the creation of the attribute sets. Furthermore, if the overlap count between any two attributes were only present in one overlap counter, Sandy would need to transfer that overlap count to the other overlap counter in an

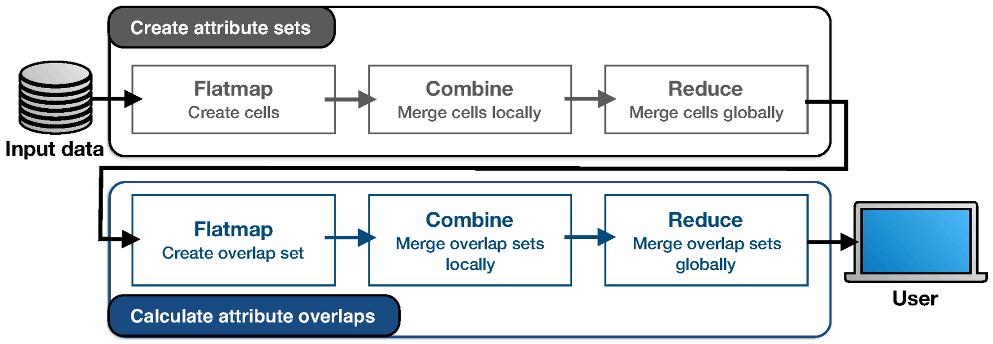


Figure 2.6: Implementation scheme of SANDY as a parallelizable data flow. Differences to SINDY are highlighted in blue.

additional network-based exchange. Because the expected efficiency gains of calculating each overlap only once are so small, we waived this optimization for the sake of simplicity.

Finally, let us show how SANDY can be implemented on distributed data flow systems. One may think of SANDY as a modification of SINDY that merges the IND candidate sets with a multi-set union rather than an intersection. Therefore, their implementations are quite similar, as can be seen in Figure 2.6. In fact, only the second half of SANDY's data flow plan is slightly different. A particular interesting point is the representation of the overlap counters. As for SINDY, we use a sorted array to store the attributes of the counters. In addition, we use a second array of the same length to store the counts of each attribute. When SANDY merges two overlap counters, it can synchronously iterate the attribute arrays (as in SINDY) and additionally add the counts in the counting arrays. Hence, merging two overlap counters is of linear complexity in the sum of their sizes.

2.4 A hybrid cover for n-ary INDs

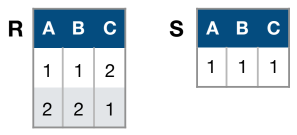
In the above sections, we limited the discussion to unary INDs, i. e., INDs among individual columns. Nonetheless, n-ary INDs among combinations of columns are of interest, too. For instance, they can reveal composite foreign keys and uncover materialized views that may, for instance, be found in a CSV export of a database. That being said, their discovery is significantly harder: While a database with k columns entails $k^2 - k$ unary IND candidates that need be verified, it can have up to $\binom{k}{\frac{k}{2}} \cdot \frac{k}{2}!$ non-redundant k-ary IND candidates. In fact, Bläsius et al. [2017] have shown n-ary IND discovery to be a W[3]-hard problem, which distinguishes it even among the NP-hard problems.

Generally speaking, a set of n-ary IND candidates can be checked with the proceeding described in Section 2.2 by substituting columns (and their values) with column combinations (and their value combinations). However, besides the need to check potentially many more n-ary than unary IND candidates, checking individual n-ary INDs is also computationally more expensive than is the case for their unary counterparts. At first, checking n-ary IND candidates involves n-ary value combinations, which require more memory and more CPU cycles to compare than individual values. Second, there are in general more n-ary value combinations for a column combination than there are individual values in any of the involved columns. And third, each n-ary IND candidate may involve column combinations distinct from those in the other n-ary IND candidates. That is, for m n-ary IND candidates, we might need to create an inverted index for up to $2m$ column combinations. In contrast, in a database with k columns, each of those columns appears in $2k-1$ unary IND candidates, so that for the $m' = k^2 - k$ unary IND candidates, we need to create the inverted index for $k \approx \sqrt{m'}$ columns only.

Facing these challenges, we propose the novel algorithm ANDY in this section, which represents the set of n-ary INDs in a concise *cover* consisting of *core* INDs and *augmentation rules*. Not only is this cover more informative than plain INDs but it can also be discovered more efficiently. In the following, we explain the search space for n-ary INDs in more detail, then describe our new IND cover, and finally explain how ANDY efficiently determines this cover.

2.4.1 The n-ary IND search space

As per Definition 2.1, an n-ary IND candidate spans arbitrary column combinations of two tables. Because of the projection rule, which states that projecting corresponding LHS and RHS attributes from an IND always yields a valid IND, the *maximum* INDs form a *cover* of all INDs in a dataset, i. e., they constitute a subset of INDs that logically implies all remainder INDs [Casanova et al., 1982]. For instance, the binary IND Tracks[ArtistID, AlbumID] $\subseteq$ Records[ArtistID, ID] from Figure 2.1 can be projected to the two unary INDs Tracks[ArtistID] $\subseteq$ Records[ArtistID] and Tracks[AlbumID] $\subseteq$ Records[ID]. This binary IND is also a maximum IND, because it is not a projection of any other IND in the dataset. To discover the maximum INDs in a dataset, virtually every IND discovery algorithm reverses the projection rule to prune the candidate space: An n-ary IND candidate needs be checked only if its projections are known INDs. Which projections are considered before testing an IND candidate differs between the algorithms, though.



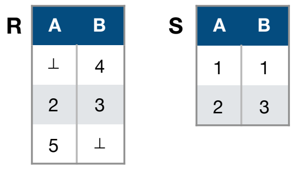
A	B	C
1	1	2
2	2	1

A	B	C
1	1	1

Figure 2.7: Two relations of which S contains edge case INDs.

Among all n-ary IND candidates, those with column repetitions constitute edge cases. In principle, they are an interesting subject to discovery: They are neither trivial (i. e., there are relation instances that violate them) nor can they be inferred from INDs of lower arity. More concretely, attributes can be repeated within the LHS, within the RHS, and across LHS and RHS (even on corresponding positions) without rendering the IND candidate uninteresting from a logical point of view. Consider the tables R and S in Figure 2.7. Within each table, all columns are mutually included in each other. However, S further satisfies $S[AA] \subseteq S[BC]$, $S[BC] \subseteq S[AA]$, and $S[BB] \subseteq S[BC]$, while their counterparts in R do not hold. Nevertheless, due to pragmatic reasons such IND candidates are neglected in related work: While considering those candidates may cause considerable computational effort, the value of corresponding verified INDs for practical applications is questionable. Therefore, we also deliberately exclude IND candidates with repeated columns.

Let us conclude our theoretical considerations of the n-ary search space with a discussion of NULL values. In Section 2.2.1, we propose to ignore NULL values for unary IND discovery in accordance with the foreign key semantics as defined by the SQL standard. Transferring this argument to n-ary IND discovery means that we should ignore any value combination that contains at least one NULL value. However, this extended IND definition disables the projection rule as exemplified in Figure 2.8: This definition admits $R[AB] \subseteq S[AB]$ as an IND, although neither $R[A] \subseteq S[A]$ nor $R[B] \subseteq S[B]$ are INDs. As a result, all algorithms that use the projection rule (including ANDY) will not discover $R[AB] \subseteq S[AB]$. On the other hand, waiving the projection rule for search space pruning would have serious impact on the algorithm performance and most likely render n-ary IND discovery completely infeasible. We are not aware of any work that acknowledges this problem (let alone solving it), but believe that it might be addressed best by a dedicated foreign key discovery algorithm with specific pruning rules. In the context of this work, we admit the above proposed treatment of column combinations with NULL values with the caveat that only those INDs will be discovered whose projections are also valid INDs. In particular, regular INDs without NULL values are not affected. This approach is consistent with the above described exclusion of repeated attributes within a single IND: To stave off tremendous performance penalties, we exclude edge case IND candidates with a rather questionable utility.



R

A	B
⊥	4
2	3
5	⊥

S

A	B
1	1
2	3

Figure 2.8: Two relations of which R contains NULL values.

2.4.2 A hybrid IND cover

As stated above, the maximal INDs of a dataset form a cover of all INDs. That is, all (non-trivial) INDs can be deduced from the maximal INDs by means of the projection rule. Therefore, existing IND discovery algorithms search for all maximal INDs in a dataset. Let us show that there is a different way to characterize all INDs in a dataset.

Casanova et al. [1982] noted that INDs interact with functional dependencies (FDs)[3]: Assume we know the INDs $R[XY] \subseteq S[TU]$ and $R[XZ] \subseteq S[TV]$ along with the FD $T \rightarrow U$, where R and S are tables, X and T are column lists of the same length (i. e., $|X| = |T|$), and Y, U, V, and Z are single columns. Then we can conclude that the IND $R[XYZ] \subseteq S[TUV]$ is also valid. In few words, the reason is that any tuples t_r from R and t_s from S with $t_r[X] = t_s[T]$ also satisfy $t_r[Y] = t_s[U]$ because of $R[XY] \subseteq S[TU]$ in conjunction with $T \rightarrow U$; therefore, $R[XZ] \subseteq S[TV]$ can be augmented to $R[XYZ] \subseteq S[TUV]$. This interaction is particularly interesting for two reasons. First, it allows to deduce INDs of higher arity from INDs of lower arity. Hence, if a bottom-up IND discovery algorithm knew the FDs in the profiled tables, it could spare work by logically deducing IND candidates rather than performing expensive IND validations. Second, we note that the deduced INDs are not "refining" the already known INDs.

[3]Recall from Section 1.3 that for two column combinations X and Y from the same relation r an FD $X \rightarrow Y$ states that whenever two tuple from r agree in X, they also agree in Y.

Corollary 2.2 (IND refinement)**.** *Assume two relation schemata R and S with instances r and s, respectively, along with the* IND $R[XYZ] \subseteq S[TUV]$ *($|X| = |T|$, $|U| = |V|$) and the* FD $T \to U$*. Further, let* $\mathrm{match}(R[A_1, \ldots, A_n] \subseteq S[B_1, \ldots, B_n]) := \{(t_r, t_s) \in r \times s \mid t_r[A_1, \ldots, A_n] = t_s[B_1, \ldots, B_n]\}$ *denote the tuple pairs across r and s that have the same values for their respective* IND *projection attributes. Then we have* $\mathrm{match}(R[XYZ] \subseteq S[TUV]) = \mathrm{match}(R[XZ] \subseteq S[TV])$*.*

Proof. Let $t_r \in r$ and $t_s \in s$ be two tuples, such that $t_r[XYZ] = t_s[TUV]$. It follows that $t_r[XZ] = t_s[TV]$ and, thus, $\mathrm{match}(R[XYZ] \subseteq S[TUV]) \subseteq \mathrm{match}(R[XZ] \subseteq S[TV])$. We show $\mathrm{match}(R[XZ] \subseteq S[TV]) \subseteq \mathrm{match}(R[XYZ] \subseteq S[TUV])$ by contradiction. Let $t'_r \in r$ and $t'_s \in s$ be two tuples, such that $t'_r[XYZ] \neq t'_s[TUV]$. Further assume $t'_r[XZ] = t'_s[TV]$. Because of $R[XYZ] \subseteq S[TUV]$, there must be a further tuple $t''_s \in s$, such that $t'_r[XYZ] = t''_s[TUV]$. It follows that $t'_s[T] = t''_s[T]$ but $t'_s[U] \neq t''_s[U]$, which contradicts the FD $T \to U$. □

In simple terms, an IND "connects" tuples across two relations whenever they share the same value in the IND's projection attributes. Corollary 2.2 states that, if we augment an IND by means of an FD, both the original and the augmented IND connect exactly the same tuples. This can be indeed of practical relevance. Consider the IND Track[AlbumID, ArtistID] ⊆ Records[ID, ArtistID] from Figure 2.1 along with the FD ID → ArtistID on the Records relation. Then the unary IND Track[AlbumID] ⊆ Records[ID] connects the same tuples between the two relations as the binary IND. For instance when reconstructing foreign keys from INDs, one might therefore consider to promote the unary IND to a foreign key and omit the ArtistID attributes because (i) they do not refine the mapping from tracks to records and (ii) a unary foreign key is easier to enforce in a production system.

We conclude that considering FDs and INDs simultaneously cannot only make the IND discovery more efficient but also gives additional insights on the INDs themselves. We therefore introduce a hybrid cover formalism for INDs that is based on *IND augmentation rules.*

Definition 2.3 (Hybrid IND cover)**.** A hybrid IND cover $C_I = (\mathcal{I}, \mathcal{A})$ consists of a set of INDs $\mathcal{I}$ and a set of augmentation rules (ARs) $\mathcal{A}$. An AR $A \in \mathcal{A}$ is of the form $R[X]{\subseteq}S[Y] \Rightarrow R[A]{\subseteq}S[B]$ and states that any IND $R[X_+] \subseteq S[Y_+]$ that can be projected to $R[X] \subseteq S[Y]$ can also be augmented to $R[X_+A] \subseteq S[Y_+B]$. The hybrid IND cover C_I describes all INDs that can be inferred from $\mathcal{I}$ with traditional IND inference rules (see [Casanova et al., 1982]) in combination with the augmentation rules in $\mathcal{A}$.

Example 2.3. In a hybrid IND cover for the data from Figure 2.1 we can replace the maximal IND Track[AlbumID, ArtistID] ⊆ Records[ID, ArtistID] with the IND Track[AlbumID] ⊆ Records[ID] and the AR Track[AlbumID]⊆Records[ID] ⇒ Track[ArtistID]⊆Records[ArtistID]. Now assume that the example database would also model the specific countries, in which a record has been released, thereby yielding the additional IND Track[AlbumID, Country] ⊆ Records[ID, Country]. Then the combination of this IND and the AR implies the ternary IND Track[AlbumID, Country, ArtistID] ⊆ Records[ID, Country, ArtistID]

The reader might wonder, why we introduce the notion of ARs for hybrid IND covers rather than using FDs. The first reason is that FDs are sufficient but not necessary to formulate ARs. The second reason is that ARs allow to remove INDs of higher arity from the cover and are therefore more concise to read. For instance, if we chose a hybrid IND cover representation with FDs rather than ARs, we would have to keep the binary IND Track[AlbumID, ArtistID] $\subseteq$ Records[ID, ArtistID] in the above example.

2.4.3 Augmentation rule discovery

Having motivated and defined the hybrid IND cover, it remains to be shown how our n-ary IND discovery algorithm ANDY can discover ARs efficiently and use them to prune the IND search space. Its main idea is as follows: We need to discover any FD $X \rightarrow A$ in the profiled dataset whose attributes X and A all appear in the RHS of some n-ary IND. The FD $X \rightarrow A$ holds on some relation instance r if and only if the projection of r on X has as many distinct values as the projection of r on XA [Huhtala et al., 1999]. As explained in Section 2.3, the calculation of distinct values can easily be piggy-backed on SINDY's IND validation procedure. Finally, if we traverse the IND search space in a strictly apriori-like manner, i. e., if we check an IND $R[XA] \subseteq S[YB]$ only after having verified the IND $R[X] \subseteq S[Y]$ (recall the projection rule), we can immediately discover any FD within an IND's RHS by keeping track of the number of distinct values and applying above FD criterion.

To better explain the details of ANDY, let us introduce a slightly altered example dataset in Figure 2.9, which embeds the AR from Example 2.3 along with two additional ARs. Those latter ARs are a consequence of missing data and can considerably increase the number of n-ary INDs in a dataset, thereby impairing the performance of discovery algorithms. While those phenomena could be simply detected and treated by handcrafted rules, they can instead be elegantly modeled as ARs.

At first, we observe that the Year column in Records contains only `NULL` values. If we ignore value combinations containing `NULL` values as proposed in Section 2.4.1, then any combination of the column Year with other columns yields an empty column combination, which in turn forms a void IND[4] with any other column combination of the same arity. Second, there are two Note columns that contain only the empty string. Real-world schemata often comprise columns (or even tables) that are not filled with any actual data. Such columns then contain only a default value. In our example, this phenomenon causes the two INDs Records[Note] $\subseteq$ Tracks[Note] and, vice versa, Tracks[Note] $\subseteq$ Records[Note]. In particular, these INDs can augment any other IND across Records and Tracks, thereby increasing the number of n-ary INDs and, hence, the number of IND candidates to check.

Let us now explain ANDY's overall workflow, which is outlined in Algorithm 2.1, alongside our modified example data in Figure 2.9. ANDY starts by initializing an empty hybrid cover (Line 1). Then, it runs a modified version of SINDY (Line 2): Besides discovering all unary INDs in the input dataset, it also counts the distinct values of every column, as explained in Section 2.3. These counts are used to discover the first ARs

[4]Recall from Section 2.2.1 that we call an IND void if its LHS does not contain any values.

Records

ID	Title	Year	ArtistID	Note
1	Back in Black	⊥	1	
2	High Voltage	⊥	1	
3	High Voltage	⊥	2	
4	Zenyattà Mondatta	⊥	3	

Tracks

Pos	Title	AlbumID	ArtistID	Note
1	You Shook Me All Night Long	1	1	
2	Hells Bells	1	1	
1	Baby, Please Don't Go	2	1	
1	Movin' On Out	3	2	

Figure 2.9: Another example dataset with discographic data. The $\perp$ represents `NULL` values and blank cells represent empty strings.

(Lines 3–6). As described in the above paragraph, INDs whose RHS comprises only a single distinct value can be used to augment any other IND that spans the same two tables. In our example we discover the IND Records[Note] $\subseteq$ Tracks[Note], which can augment any other IND between Records and Tracks, such as Records[ArtistID] $\subseteq$ Tracks[AlbumID]. The reason is that the column Note in the relation Tracks functionally depends on any other column of the same relation. Therefore, Andy exchanges said IND in the hybrid cover with the "zero-ary" AR $\emptyset \Rightarrow$ Records[Note]$\subseteq$Tracks[Note]. The same is done when a discovered IND is void, i. e., its LHS bears no values at all, as is the case for Tracks[Year] $\subseteq$ Records[AlbumID]. Here, the rationale is that any augmentation of that IND will form another void IND. Note that ARs for void INDs are not covered by an FD as explained in Section 2.4.2 but nevertheless are compatible with Definition 2.3 of hybrid covers.

This concludes the unary IND discovery phase of Andy, which then starts iteratively discovering INDs of higher arities (Line 7). At first, IND candidates of the next higher arity are generated from the already discovered INDs (Line 9). This candidate generation is handled by the `GenNext` algorithm as described by de Marchi et al. [2002]: An IND candidate is generated if and only if all its projections are known INDs. At this point, it is important to note that Andy generates fewer candidates than other IND discovery algorithms because some of the discovered INDs have been converted into ARs and do therefore not participate in the candidate generation. In fact, all those omitted IND candidates are actual INDs and can be deduced from the cover. As an example, consider the INDs Tracks[ArtistID] $\subseteq$ Records[ArtistID] and Tracks[Note] $\subseteq$ Records[Note]. Together, they yield the binary IND candidate Tracks[ArtistID, Note] $\subseteq$ Records[ArtistID, Note], which obviously holds on the dataset in Figure 2.9. However, because Andy removed the IND Tracks[Note] $\subseteq$ Records[Note] and replaced it with the AR $\emptyset \Rightarrow$ Tracks[Note]$\subseteq$Records[Note], it will not generate a corresponding binary IND candidate. Instead the binary IND is already included in the hybrid IND cover by means of the IND Tracks[ArtistID] $\subseteq$ Records[ArtistID] and the AR $\emptyset \Rightarrow$ Tracks[Note]$\subseteq$Records[Note].

Andy completes as soon as no more IND candidates are generated (Line 10). If there are candidates, though, they are checked with the already mentioned modified Sindy algorithm (Line 11). There are only two modifications necessary to check n-ary IND candidates. First, rather than creating cells for each field in each tuple (cf. Section 2.2.2), Andy creates cells for value *combinations* that appear in any of the IND candidates. For instance, for the IND candidate Tracks[Pos, ArtistID] $\subseteq$ Records[ID, ArtistID] Andy needs to create cells for the column combinations Tracks[Pos, ArtistID] and Tracks[Pos, ArtistID].

Algorithm 2.1: Overall workflow of ANDY.

```
Data: database schema $\mathcal{S}$ with database instance $\mathcal{D}$
Result: hybrid IND cover $C_I$
1  $\mathcal{I} \leftarrow \emptyset; \mathcal{A} \leftarrow \emptyset$;
2  $\mathcal{I}_1, D_1 \leftarrow$ run SINDY with distinct value counting on $\mathcal{D}$;
3  foreach $\langle A \subseteq B \rangle \in \mathcal{I}_1$ do
4      if $D_1[B] = 1 \vee D_1[A] = 0$ then
5          $\mathcal{I}_1 \leftarrow \mathcal{I}_1 \setminus \{A \subseteq B\}$;
6          $\mathcal{A} \leftarrow \mathcal{A} \cup \{\emptyset \Rightarrow A \subseteq B\}$;
7  for $n \leftarrow 2$ to $\infty$ do
8      $\mathcal{I} \leftarrow \mathcal{I} \cup \mathcal{I}_{n-1}$;
9      $I_n^c \leftarrow$ GenNext($\mathcal{I}_{n-1}$);
10     if $I_n^c = \emptyset$ then break;
11     $\mathcal{I}_n, D_n \leftarrow$ run SINDY-like IND test for $n$-ary INDs with distinct value counting
        for $\mathcal{I}_n^c$ on $\mathcal{D}$;
12     foreach $\langle A_1...A_n \subseteq B_1...B_n \rangle \in \mathcal{I}_n$ do
13         if $D_n[A_1...A_n] = 0$ then
14             $\mathcal{I}_n \leftarrow \mathcal{I}_n \setminus \{A_1...A_n \subseteq B_1...B_n\}$;
15             for $i \leftarrow 1$ to $n$ do
16                 $\mathcal{A} \leftarrow \mathcal{A} \cup \{A_1...A_{i-1}A_{i+1}...A_n \subseteq B_1...B_{i-1}B_{i+1}...B_n \Rightarrow A_i \subseteq B_i\}$;
17         else
18             for $i \leftarrow 1$ to $n$ do
19                 if $D_n[B_1...B_n] = D_{n-1}[B_1...B_{i-1}B_{i+1}...B_n]$ then
20                     $\mathcal{I}_n \leftarrow \mathcal{I}_n \setminus \{A_1...A_n \subseteq B_1...B_n\}$;
21                     $\mathcal{A} \leftarrow \mathcal{A} \cup \{A_1...A_{i-1}A_{i+1}...A_n \subseteq B_1...B_{i-1}B_{i+1}...B_n \Rightarrow A_i \subseteq B_i\}$;
22 return $C_I = (\mathcal{I}, \mathcal{A})$;
```

Second, for any discovered n-ary IND, ANDY checks whether it is actually an IND candidate. When ignoring value combinations that comprise a `NULL` value, the IND test can turn out positive for non-IND candidates because the downward closure property does not hold then (cf. Section 2.4.1). To better control the IND discovery, we sieve out such "pseudo-INDs".

Eventually, when the n-ary IND candidates have been verified and the distinct values of their column combinations have been counted, ANDY determines ARs for void INDs (Lines 13–16). Additionally, ANDY also checks for FDs in the RHS of INDs and, if such FDs exist, converts the IND to one or more corresponding ARs (Lines 17–21). As an example, the binary IND Tracks[AlbumID, ArtistID] $\subseteq$ Records[ID, ArtistID] comprises the FD ID $\rightarrow$ ArtistID in its RHS, so ANDY exchanges it with the AR Tracks[AlbumID]$\subseteq$Records[ID] $\Rightarrow$ Tracks[ArtistID]$\subseteq$Records[ArtistID]. When all candidates and ARs have been discovered, they form the hybrid IND cover of the input datset (Line 22).

2.5 Evaluation

Having explained the algorithms SINDY, SANDY, and ANDY, we now proceed to experimentally evaluate them. In particular, we seek to answer following questions: (i) *How do the algorithms compare to existing* IND *discovery algorithms?* (ii) *How do our algorithms scale as the profiled datasets grow in size?* (iii) *And how well do our algorithms scale out across computer clusters?* Before answering these questions, we outline our experimental setup in Section 2.5.1. Then, in Section 2.5.2 we evaluate our algorithms for (partial) unary IND discovery, SINDY and SANDY, and proceed to evaluate the n-ary IND discovery algorithm ANDY in Section 2.5.3.

2.5.1 Experimental setup

We have implemented SINDY, SANDY, and ANDY in Java 8 and chose Apache Flink 1.3.1 as distributed data flow system to back up our algorithms. Our profiled datasets resided in an Apache Hadoop HDFS 2.7.3. For our experiments with BINDER, we used its original implementation, which is also written in Java [Papenbrock et al., 2015c]. In terms of hardware, we used a cluster consisting of a master node (IBM xServer x3500 with 2× Xeon X5355 Quadcore (2.66 GHz, 8 MB L2 cache), 32 GB RAM, CentOS 6.9) and four workers (IBM PowerEdge R430 with Intel Xeon E5-2630 v4 (2.2 GHz, 10 cores, 25 MB L2 cache), 32 GB RAM, Ubuntu 16.04.3). For the distributed experiments, we granted 24 GB of main memory to the driver, 1 GB to the Flink job manager, and 24 GB to the Flink workers. For the non-distributed experiments, which were executed on only one of the workers and required only a single operating system process each, we granted 24 GB of main memory to those processes. An overview of all used datasets can be found in Table 2.1. In addition, the algorithm source code and pointers to the sources of the datasets are given on our repeatability page: `https://hpi.de/naumann/projects/repeatability/data-profiling/metanome-ind-algorithms.html`

2.5.2 Unary discovery

Comparison of BINDER and SINDY

Let us begin with a comparison of SINDY with the state-of-the-art discovery algorithm BINDER. We limit our comparison to BINDER because it has been shown to be most efficient among the exact unary IND discovery algorithm [Papenbrock et al., 2015c] (except for SINDY, to which it has not been compared). Although being even more efficient than BINDER, we do not include FAIDA in this comparison, because it is an IND approximation algorithm that cannot guarantee the correctness of the discovered INDs [Kruse et al., 2017b].

BINDER does not define a scale-out strategy, so to allow for a fair comparison with SINDY both algorithms are executed on a single of our workers. Nevertheless, we consider two configurations of SINDY. The first configuration limits it to a single CPU core and is supposed to answer the question how BINDER and SINDY compare when granted exactly

Table 2.1: Overview of our evaluation datasets.

Name	Size [MB]	Tables [#]	Columns [#]	Tuples [#]
EMDE	2	11	161	3,379
SCOP	15	4	22	342,195
CENSUS	112	2	48	299,285
Spots	177	3	45	1,439,733
WIKIPEDIA	540	2	14	14,802,104
BIOSQLSP	567	29	148	8,306,268
WIKIRANK	696	7	29	5,994,972
LOD2	825	13	164	4,792,549
ENSEMBL	836	2	26	3,991,313
CATH	908	5	115	452,652
TESMA	1,112	2	114	600,000
RFAM	15,220	29	202	112,359,686
MusicBrainz	33,437	206	1,053	222,019,536
PDB	45,869	117	1,791	266,352,038
Plista	62,431	4	131	123,397,711

the same resources. In the second configuration, we allow SINDY to use all 10 cores of the worker computer, thereby assuming a more pragmatic viewpoint: Both BINDER and SINDY use virtually all the worker's main memory and disk I/O bandwidth. Therefore, it is reasonable to assume that, when using only one CPU core, the other cores cannot be effectively utilized by any other application due to the lack of main memory and I/O bandwidth. Hence, the second configuration seeks to answer the question how BINDER and SINDY compare when granted one worker.

Figure 2.10 displays the results of the comparison. Let us first focus on the single-core configuration of SINDY. As can be seen, BINDER could profile all datasets more efficiently than SINDY in this configuration. We can determine three reasons for that. First, SINDY has start-up costs of approximately 12 seconds; these are due to Apache Flink pre-allocating main memory. Second, BINDER's dynamic hash partitioning, which is specifically tailored to IND discovery, has a higher efficiency than using Apache Flink's generic `GroupReduce` operator. In fact, an inspection of Flink's physical execution strategy for that operator reveals that it always performs a sorting-based aggregation of the cells – however, Papenbrock et al. [2015c] have already shown that the hash-partitioning of BINDER works more efficiently than the sort-merge approach of the IND discovery algorithm SPIDER. As of writing this thesis, Flink does not support hash-based aggregation, but such a feature is planned[5] and would make a repetition of this experiment very interesting. Finally, BINDER employs pruning rules when evaluating INDs on the hash-partitioned dataset, while SINDY does not define any equivalent functionality due to the limited flexibility of the control flow on distributed data flow systems. However, we suspect the lack of such pruning rules to exert only a small influence on the overall algorithm runtimes, because the hash-partitioning in BINDER and the attribute set creation in SINDY dominate the runtimes (usually around 75 %–90 % of the runtime).

[5]See `https://issues.apache.org/jira/browse/FLINK-2237` (accessed September 12, 2017).

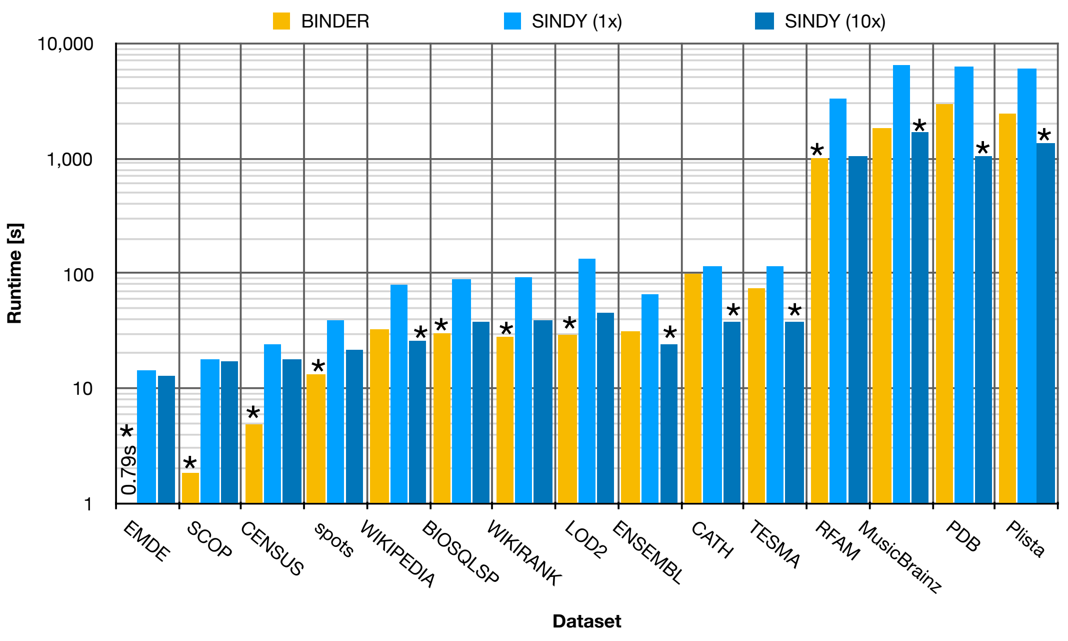


Figure 2.10: Runtime comparison of BINDER and SINDY on a single worker. While "SINDY (1×)" utilized a single CPU core only, "SINDY (10×)" utilized all 10 cores. The asterisk marks the fastest algorithm for each dataset.

When granting SINDY all 10 cores, we obtain a more differentiated comparison, though. We observe a speed-up over the single-core configuration on all datasets, although for the smallest datasets the efficiency gains are minor. The reason is that the single-core and parallel configuration incur the same start-up costs. However, for larger datasets we observe up to six-fold speed-ups, so that the parallel SINDY is indeed more efficient than BINDER on some datasets. In summary, we conclude that in non-distributed settings SINDY is not suited to profile small datasets, while it can be indeed the most efficient algorithm if the profiled dataset is several gigabytes in size and the profiling computer has a sufficient number of CPU cores.

Parallelization of SINDY

The above comparison regards SINDY in a non-distributed deployment only. However, SINDY is actually designated to run on computer clusters. Therefore, we proceed to analyze SINDY's scale-out behavior. In other words, we determine how the provision of additional CPU cores and computers influences the efficiency of the algorithm. Figure 2.11 depicts SINDY's runtime on several datasets in several hardware configurations: This experiments grants the algorithm between 1 and 4 workers (and all their cores). Obviously, for smaller datasets scaling out increases the efficiency only slightly and can even be counter-productive for very small datasets. The main reasons are the worker-independent start-up costs and that distributed computing incurs additional overhead for coordination and exchanging data across machines. For large datasets, in contrast, we measured an even super-linear speed-up factor of up to 4.7× (for MusicBrainz) when

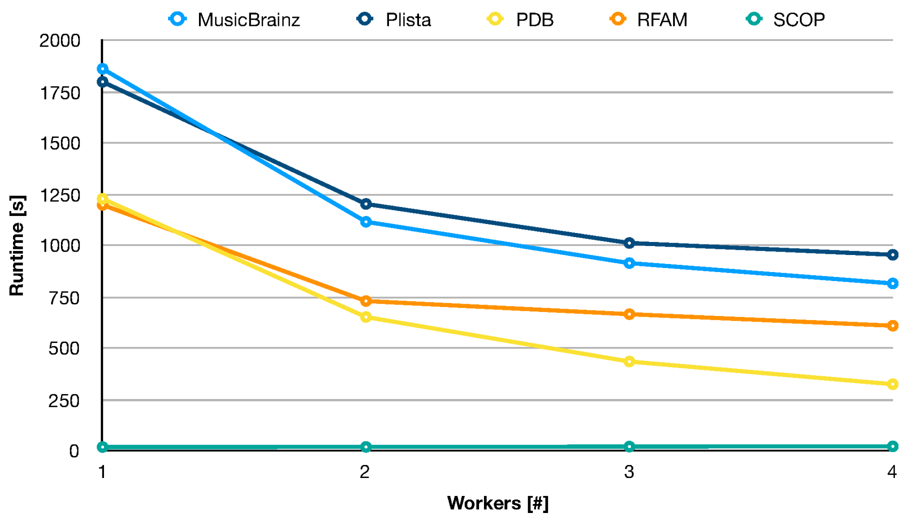


Figure 2.11: Runtime behavior of SINDY as it is using an increasing number of workers.

utilizing all 4 workers. A potential explanation for this observation are the increased main memory capacities of 4 workers in comparison to a single one: The global main memory capacities of 96 GB might suffice to host all intermediate working data, thereby avoiding costly out-of-core (i. e., disk-based) data processing.

Along these lines, it is an intriguing question how providing SINDY with more CPU cores (scale-in) compares to providing SINDY with more worker computers (scale-out). While the latter adds more resources, such as main memory and disk bandwidth, it also adds network overhead. To answer this question, we executed SINDY with 10 cores on a single worker as a baseline. The resulting runtime was compared to running SINDY with 10 cores across n workers ($n > 1$). The measured speed-up over the baseline across various datasets is presented in Figure 2.12. The results confirm our intuition w. r. t. the scale-out experiment: For small datasets, a scale-out configuration only causes overhead. For larger datasets, however, scale-out is effective because SINDY indeed profits from the additional computing resources other than CPU cores. Again, these experiments show that SINDY is particularly advancing the efficiency of IND discovery for large datasets and sufficient hardware provisioning. In the following experiments, we therefore focus on such scenarios.

Scalability to larger datasets

We seek to determine SINDY's scalability behavior w. r. t. dataset size and compare it to BINDER. Concretely, we determine the behavior of both algorithms as the profiled dataset grows in the number of columns or in the number of tuples. As mentioned in the above paragraph, we run SINDY at full scale now, i. e., with all 4 workers and all 40 cores, while BINDER is running on a single worker. After all, we are interested in the scaling behavior of the two algorithms in their intended hardware setups.

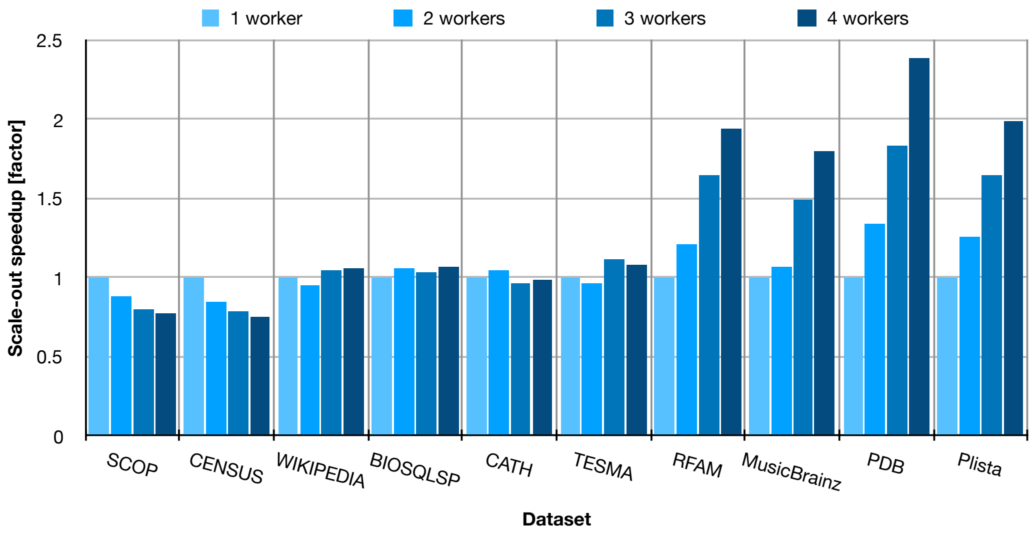


Figure 2.12: Speed up of SINDY when using four instead of only one worker, while maintaining a parallelism of 10.

To determine the column scalability, we instrumented the two algorithms in such a way that they would only consider the first k columns of each relation in the profiled dataset for some user-specified value k. As an alternative, we could just consider the "first" k tables of the profiled dataset. However, in most datasets the distribution of tuples over the tables is strongly skewed. By considering a subset of columns from *all* tables simultaneously, we attempt to mitigate the effects of the different table sizes in our experiments.

As can be seen in Figure 2.13, both BINDER and SINDY appear to scale linearly in the number of considered columns. This observation is truly interesting, because unary IND discovery has a worst case complexity of at least $\mathcal{O}(c^2)$ (c being the number of columns): After all, there are up to $c^2 - c$ non-trivial INDs in a dataset and profiling algorithms are bound to have a complexity at least linear in the output size. That being said, the most expensive step of all IND algorithms, namely building the inverted index, is not sensitive to the number of attributes but rather to the amount of input data. Furthermore, typical relational datasets usually comprise only so many columns and INDs. For instance, PDB consists of 1,791 columns that entail 800,636 INDs. Specific datasets, such as web tables, can have many more columns and INDs, though, and call for specialized algorithms [Tschirschnitz et al., 2017]. In such scenarios, the suspected quadratic complexity can actually be observed.

The other important scaling dimension, besides the number of columns, is the number of tuples in a dataset. To experimentally determine the scaling behavior of BINDER and SINDY along this dimension, we had both algorithms profile systematic samples of the input tables. For instance, for a sampling ratio of 75 % the algorithms would discard every fourth tuple of every table, but continue normal operation for the remainder tuples. Again, we find both algorithms to exhibit a linear scaling behavior as shown in Figure 2.14. For BINDER, which creates the inverted index using hash-partitioning, this

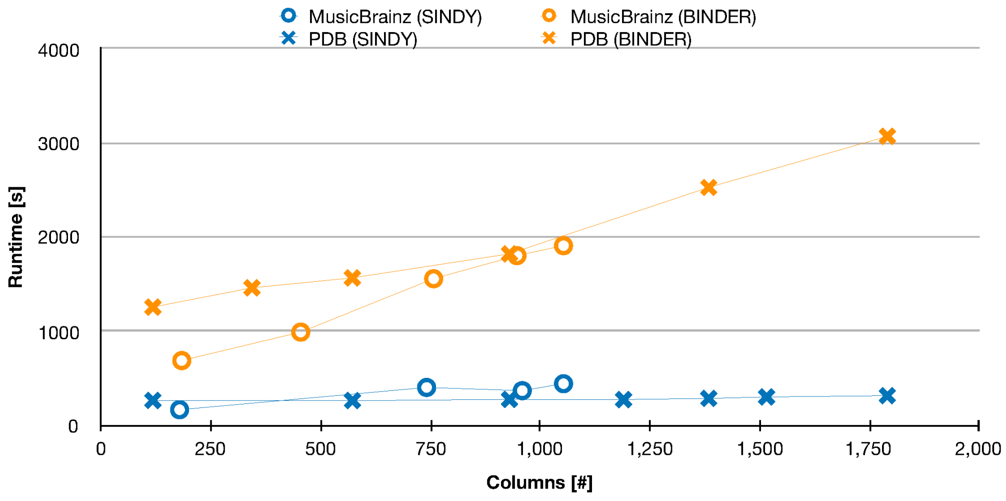


Figure 2.13: Scalability of BINDER and SINDY as profiled datasets grow in the number of columns.

is the expected behavior. SINDY, in contrast, builds upon Apache Flink's sorting-based aggregation and might therefore expose a complexity of at least $\mathcal{O}(t \cdot \log t)$ (t being the number of tuples). Hence, we conclude that SINDY's runtime is not dominated by the sorting per se: Disk-based sorting techniques, such as the two-phase multiway merge-sort, usually have linear I/O complexity by writing every input element to disk only once. It is indeed plausible, that the I/O time dominates the sorting; in particular, because Flink's sorting mechanisms are highly tuned and even employ cache-aware data structures.[6] In summary, in both experiments SINDY exhibits the same scaling behavior as BINDER, although its implementation on Apache Flink prohibits some of BINDER's optimizations. At the same time, SINDY, which utilizes more computational resources than BINDER, achieves shorter absolute runtimes.

Partial IND discovery

The above experiments focus on SINDY but do not consider SANDY, the modification of SINDY to discover partial INDs. Indeed, we waive a repetition of above experiments for SANDY– not only for the sake of space: SINDY and SANDY are technically so similar that they entail basically identical runtimes on all the datasets. This observation is also in accordance with our intuition that the attribute set creation, which is part of both SINDY and SANDY, dominates the algorithms' runtimes.

Nonetheless, it is intriguing to compare the results of both algorithms. For this purpose, Figure 2.15 depicts the g_3' error of *all* partial IND candidates of several datasets. Obviously, the vast majority of partial IND candidates across all datasets have a rather high g_3' error and their columns mostly do not share any value at all. In consequence,

[6]See, e. g., `https://flink.apache.org/news/2015/05/11/Juggling-with-Bits-and-Bytes.html` (accessed September 15, 2017).

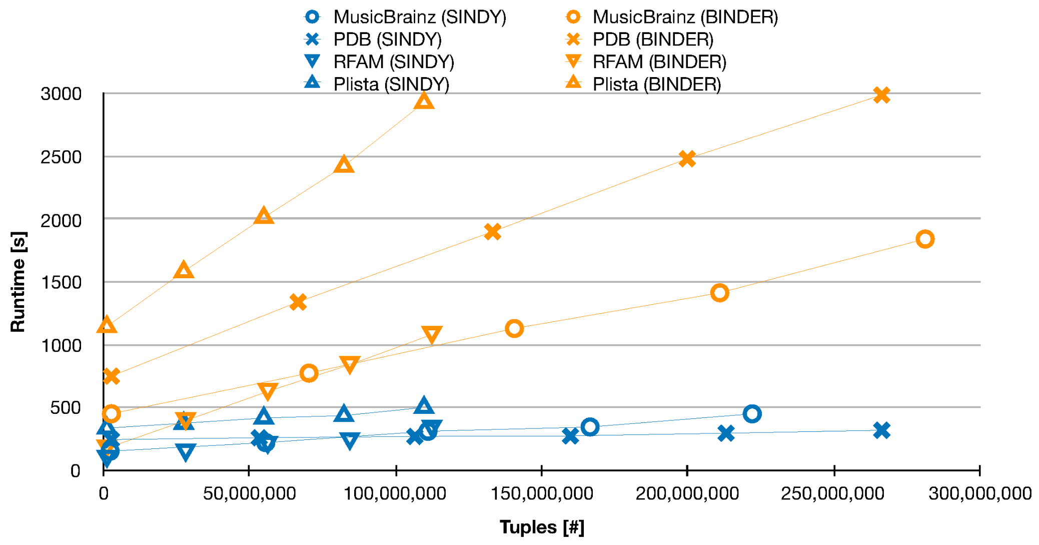


Figure 2.14: Scalability of BINDER and SINDY as profiled datasets grow in the number of tuples.

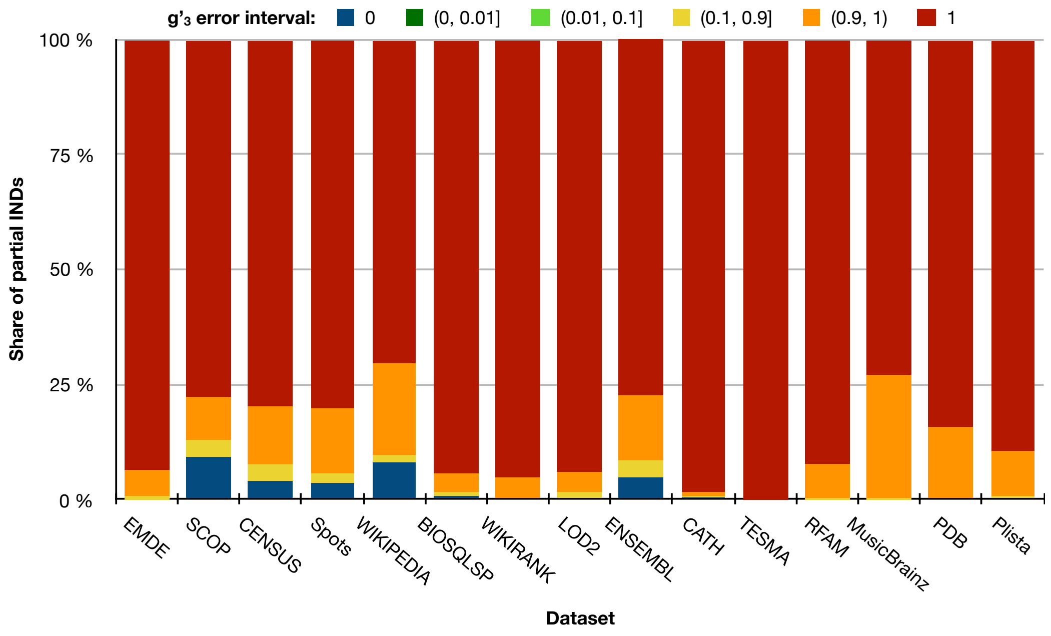


Figure 2.15: The g_3' error for all partial IND candidates among several datasets.

regarding partial instead of exact INDs does significantly increase the number of discoverable dependencies – at least in the unary case.

We further observe a “gap” between exact and partial INDs. In fact, there is not a single partial, non-exact IND in all profiled datasets that has a g_3' error less than or equal to 0.01; and only few with a g_3' error less than or equal to 0.1. An explanation for this issue might be that our experiment datasets are not subject to foreign key violations.

Hence, it would be interesting to investigate partial INDs on other datasets that lack foreign keys, e. g., on web tables or for integrating datasets.

Nevertheless, we could determine meaningful partial INDs even in our rather "clean" datasets. A specific example is the partial IND SG_TERM[Start_Pos] $\subseteq$ SG_TERM[End_Pos] with a g_3' error of less than 0.1, telling that most term start positions are also term end positions. A user can conclude that these two fields are likely related and carry out further investigations (for that matter, those attributes are used to form a hierarchy among technical terms). Furthermore, we regularly observed partial INDs (i) among columns that contain the same kind of data, such as dates and auto-incremented IDs; (ii) among columns that reference the same primary key column; and (iii) as the inverse of foreign keys, i. e., if $A \subseteq B$ is a foreign key, then $B \subseteq A$ often is a partial IND. We suppose that the former observation can be used for data discovery [cf. Fernandez et al., 2016] and column classification while the latter two might support foreign key detection [cf. Rostin et al., 2009; Zhang et al., 2010].

2.5.3 N-ary discovery

Comparison of MIND, BINDER, and ANDY

As for SINDY, let us begin our evaluation of ANDY with a comparison to state-of-the-art discovery techniques. As explained in Section 2.1, n-ary IND discovery algorithms mostly differ with regard to which IND candidates they generate in which order, while the checking mechanisms are either the same as for unary INDs or based on SQL queries. Hence, to focus the comparison on the candidate generation strategies, we integrated several candidate generation strategies with SINDY's (and ANDY's, for that matter) checking procedure. Concretely, we consider MIND's basic apriori-based generation `GenNext`, upon which ANDY builds (SINDY+MIND); we consider BINDER's candidate generation, also an apriori-based procedure that admits some futile IND candidates, though, and that discards unary void INDs (SINDY+BINDER); and we consider an enhanced version of MIND's candidate generation that also discards void INDs (SINDY+MIND-VOID). For pragmatic reasons, we stopped any algorithm that took more than 16 hours to complete.

The results of our comparison are displayed in Figure 2.16. At a first glance, SINDY+MIND seems to be the least efficient strategy in almost all datasets. In fact, it is the only strategy that includes void INDs in the candidate generation. In consequence, it has to cope with many more IND candidates in datasets with many empty columns, such as BIOSQLSP and WIKIRANK. Hence, we focus on the other strategies below.

At a second glance, SINDY+BINDER and SINDY+MIND-VOID perform almost identical on most datasets. There are two particularly notable exceptions, though. Among the two strategies, only SINDY+MIND-VOID is capable to profile the very small EMDE dataset and the larger MusicBrainz dataset within the given time and resource constraints. Indeed, we observe SINDY+BINDER to generate significantly more IND candidates. As an example, while SINDY+MIND-VOID generates 131,502 5-ary IND candidates from 96,475 4-ary INDs, SINDY+BINDER hypothesizes 528,817 candidates based on the same INDs. At this point, SINDY+BINDER fails because Flink's job manager is overloaded by that

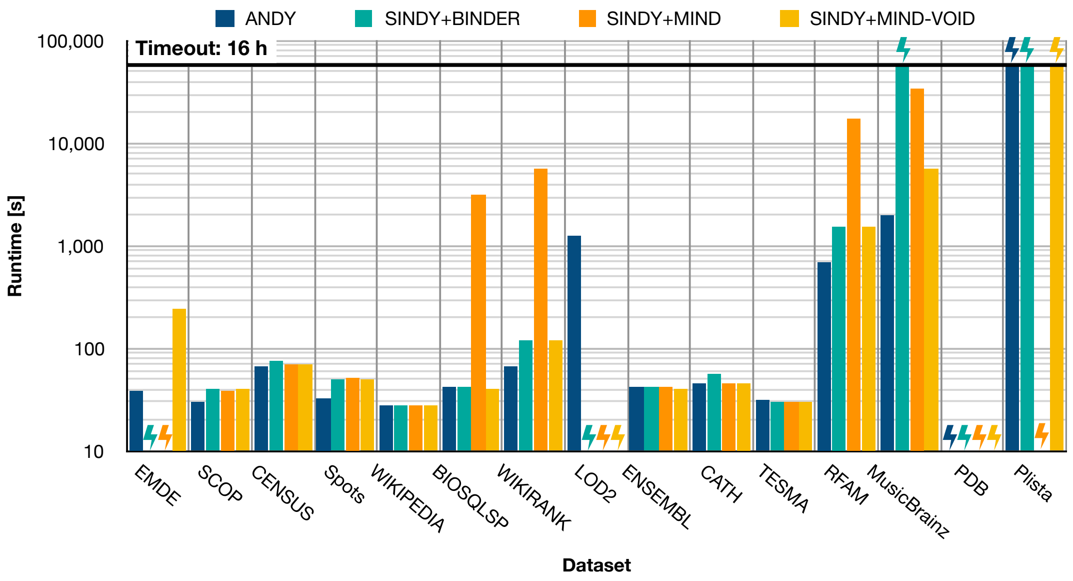


Figure 2.16: Comparison of Sindy in combination with several candidate generators and Andy. The flash symbols mark algorithm failures or timeouts.

number of IND candidates. The reason for the greater number of IND candidates is that Binder's candidate generator does not fully leverage the projection rule, thereby creating futile IND candidates that cannot be actual INDs. So, although Binder's candidate generation is slightly more efficient, this does not outweigh the effort for checking those additional futile IND candidates.

Let us now consider Andy. Apparently, it is as efficient as Sindy+Mind-Void on 6 out of 15 datasets and outperforms all other candidate generation strategies on 7 datasets (and the remaining 2 datasets, PDB and Plista, could not be processed by any strategy). Furthermore, Andy's speed-up differs among the datasets, depending on how many ARs it could identify and employ for pruning. For instance, Andy discovered no ARs on CENSUS and only 2 ARs on CATH, but 3,427 ARs on EMDE, 2,587 ARs on LOD2, and even 75,596 ARs on MusicBrainz. In fact, on MusicBrainz the number of discovered INDs was reduced from 120,569 (for Sindy+Mind-Void) to 52,441 and the number of checked n-ary candidates was reduced from 5,740,344 to 79,730. When surveying a sample of the discovered ARs, we found most of them to be spurious, that is, they did not seem to correspond to a semantic rule. However, that does not necessarily impair the utility of the hybrid IND cover: After all, it still staves off the checking and materialization of many uninteresting INDs and IND candidates.

Scalability to larger datasets

As can be seen in Figure 2.16, Andy (and the other algorithms, for that matter) could not fully profile the large datasets PDB and Plista. For the former, Andy ran out of disk space while checking 1,654,313 binary IND candidates, even though it could reduce the number of candidates by a factor of 6 compared to the other candidate generation

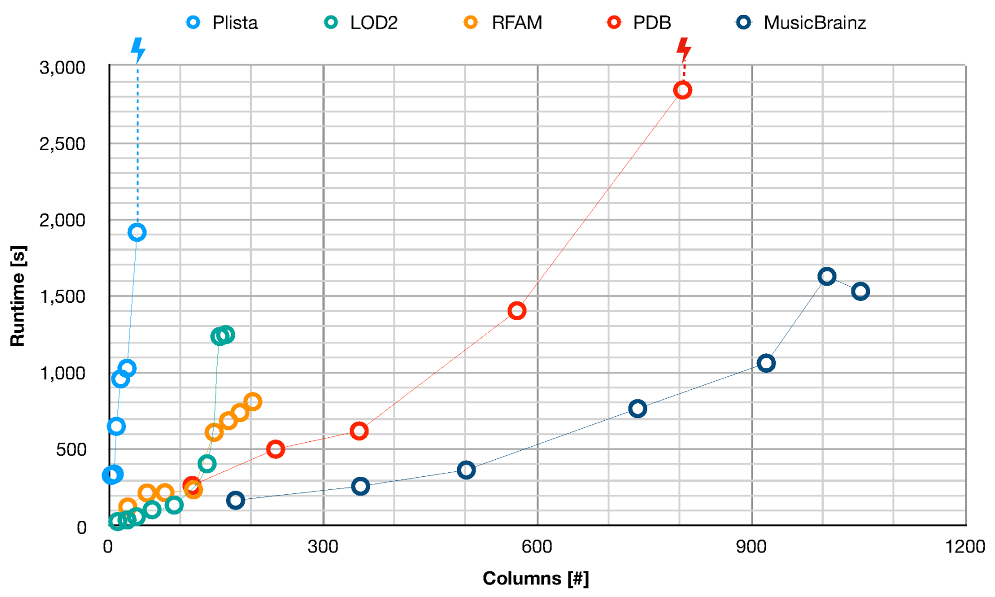


Figure 2.17: Runtime behavior of Andy for different datasets and numbers of considered columns.

strategies. On the latter Plista dataset, Andy created as many binary IND candidates as Sindy+Mind-Void, namely 15,586, and ran out of time while checking them. Having shown that Andy is at least as efficient as state-of-the-art discovery strategies for n-ary INDs (and often more efficient), it is now useful to analyze Andy's scaling behavior, so as to understand its limitations.

The scalability of Andy w. r. t. the amounts of provided hardware resources and the number of database tuples can be concluded from Section 2.5.2: The runtime of Andy is clearly dominated by the checking of IND candidates; this checking procedure is simply a slightly modified version of Sindy and therefore has the same scaling properties. That being said, we present a dedicated evaluation of Andy's scaling behavior in the number of database columns. Generally speaking, the reason is that more columns entail more INDs. In consequence, Andy needs to check more IND candidates of higher arities, as it considers more columns. That is, the influence of the number of columns has to be regarded in conjunction with the candidate generation. To vary the number of considered columns, we consider the first k columns of every table in a dataset, as done for Sindy's column scaling experiment. The maximum allowed runtime is limited to 2 hours. The results are depicted in Figure 2.17.

Apparently, Andy's runtime grows in a super-linear manner as the number of considered columns increases. Such behavior is expected, because the number of n-ary INDs potentially grows exponentially in the number of columns (cf. Section 2.4). Nevertheless, the inclines of the scaling curves seem to correlate neither with the number of columns, nor with the absolute dataset size, nor with the number of IND candidates to be checked: While Andy does not manage to check 443 4-ary IND candidates of the Plista dataset within 86 minutes, it can check all 79,730 candidates of the MusicBrainz dataset in about 25 minutes. As a matter of fact, these factors interact and must not be regarded in isolation. For instance, it is not sufficient to consider only the number of IND candidates, as different IND candidates entail different computational effort depending on the number of distinct values in their columns, the sizes of the individual values, and the numbers of tuples in the columns' tables. As a result, whether or not a dataset can be profiled by ANDY within reasonable resource constraints can hardly be predicted.

2.6 Summary

This chapter presented algorithms for various IND discovery problems on relational data: SINDY discovers all unary IND candidates in a distributed and scalable fashion. As shown with the algorithm SANDY, it can also be easily adapted to discover partial INDs. Furthermore, we showed that the n-ary INDs of a dataset can be represented by a hybrid cover consisting of "core" INDs and IND augmentation rules. Not only is this hybrid cover more informative than plain INDs, but its discovery is also much more efficient on many datasets, as we have shown with the algorithm ANDY. All in all, the presented algorithms improve over state-of-the-art methods in terms of efficiency and scalability, especially when they are executed on computer clusters and applied to large datasets.

Let us describe potential future work in the area of IND discovery. An obvious blank spot is the discovery of n-ary partial INDs, which bears two particular issues. The first problem is that the established g'_3 measure is not monotonous, i. e., $g'_3(X \subseteq Y) \leq g'_3(XA \subseteq YB)$ does not necessarily hold for some column combinations X, Y and columns A and B. As a result, apriori-like algorithms for partial IND discovery might miss actual n-ary partial INDs. This problem could be avoided by using a monotonous error measure instead: For instance, g'_3 could be based on tuples rather than distinct values to establish the monotonicity. This leads to the second problem, though: There are more partial than exact INDs in a dataset. Even though we observed only a slight increase of unary INDs when including partial INDs, the numbers of n-ary partial INDs still might increase drastically. Their discovery would then cause significantly more computational effort. For that matter, it is not clear in how far augmentation rules could be applied for pruning partial IND candidates. Hence, novel algorithmic strategies for n-ary partial IND discovery might be necessary.

A second line of future work could deal with further improvements for the discovery of exact n-ary INDs; after all, our experiments comprised datasets that ANDY could not profile due to the lack of time or disk space. To cope with this problem, one might consider using an approximate algorithm first, such as FAIDA [Kruse et al., 2017b], and then verify those approximated INDs using SINDY. This approach has two drawbacks, though. At first, the number of (approximate) n-ary INDs might be huge, so that the verification step still fails; and second, FAIDA is not capable of detecting augmentation rules, so that a combination of FAIDA and SINDY might turn out less scalable than ANDY. To this end, we are working on an algorithm that is centered around *position list indices (PLIs)* [Heise et al., 2013; Huhtala et al., 1999]. PLIs are highly compact representations of columns of a relation and are usually used for the discovery of functional dependencies and unique column combinations. However, we discovered that they can also be used to verify IND candidates and offer two particular benefits. At first, PLIs do not contain actual values, so they do not require as much disk space as SINDY's verification mechanism. The second benefit is that PLIs can uncover more types of augmentation rules than is possible for ANDY, so that a more effective pruning of the search space is possible. However, a downside of this approach is that it cannot be implemented on distributed data flow systems, such as Apache Flink. Therefore, custom memory management and distribution strategies are required.

Chapter 3

Conditional Inclusion Dependency Discovery on RDF data

In the above chapter, we presented discovery algorithms for inclusion dependencies (INDs), which form a fundamental data dependency on relational databases and support data management tasks, such as foreign key discovery and query optimization. Given their importance and utility, it is an intriguing question if and how the notion of INDs can be transferred to other data models. Concretely, in this chapter we investigate this question for RDF data and show that for this data model INDs should be amended with conditions. So called conditional inclusion dependencies (CINDs), which restrict their including and included parts of the data in terms of conditions, support various data management tasks on RDF data, in particular *ontology reverse engineering*, *knowledge discovery*, and *query optimization*, as we demonstrate in this chapter. However, the addendum of conditions drastically adds to the computational complexity of CIND discovery.

To this end, this chapter describes the novel algorithm RDFIND to discover the CINDs in RDF data, based on [Kruse et al., 2016a]. We start with a brief introduction of the RDF data model in Section 3.1 along with a formalization of CINDs and concrete examples of how they can support RDF data management. Then, in Section 3.2, we characterize *pertinent* CINDs, a subset of potentially interesting CINDs among all CINDs, and introduce RDFIND's general approach to discover all pertinent CINDs in an RDF dataset. The following Sections 3.3–3.5 explain RDFIND's three principal components in detail. Then, we proceed to thoroughly evaluate our system in Section 3.6. Eventually, we discuss related work in Section 3.7 and summarize our research results in Section 3.8.

3.1 Inclusions within RDF Datasets

The *Resource Description Framework* (RDF) is a flexible data model that shapes data as a set of subject-predicate-object triples [Hayes and Patel-Schneider, 2014]. RDF was initially introduced for the Semantic Web. Due to its flexibility and simplicity, it is currently used in a much broader spectrum of applications ranging from databases [Bornea

	Subject	Predicate	Object
t_1	patrick	rdf:type	gradStudent
t_2	mike	rdf:type	gradStudent
t_3	john	rdf:type	professor
t_4	patrick	memberOf	csDepartment
t_5	mike	memberOf	bioDepartment
t_6	patrick	undergradDegreeFrom	hpi
t_7	tim	undergradDegreeFrom	hpi
t_8	mike	undergradDegreeFrom	cmu

Figure 3.1: Example RDF dataset about university affiliates.

et al., 2013; Chong et al., 2005] and data integration systems [Calvanese et al., 2011] to scientific applications [Marshall et al., 2012; Redaschi and UniProt Consortium, 2009]. As a result, very large volumes of RDF data are made available, in particular in the context of the Linked (Open) Data movement [Bizer et al., 2009]. It is expected that this expansion of RDF data will perpetuate, leading to enormous amounts of large heterogeneous datasets [Kaoudi and Manolescu, 2015].

A major particularity of RDF is that, in contrast to relational databases, its schema (ontology) is not always available – in fact, Schmachtenberg et al. [2014] reported that only 19.25 % of the proprietary RDF vocabulary in the Linked Open Data cloud was dereferencable. And even if a schema is available, the data may violate the schema constraints. This impedes the use of RDF datasets. For instance, one might make incorrect assumptions about the data and it gets harder to formulate sound queries. To cope with such issues, we recognize a particular necessity to profile RDF data.

3.1.1 Refining inclusion dependencies with conditions

In the above chapter, we describe inclusion dependencies (INDs)[1], which are among the most fundamental data dependencies on relational data, and they have proven to be useful in various data management scenarios, such as foreign key and join path discovery [Zhang et al., 2010], query optimization [Gryz, 1998b], and schema (re-)design [Levene and Vincent, 1999].

Clearly, these data management operations are relevant to RDF data, too. However, in contrast to the relational model, RDF datasets do not reflect the schema of their data in terms of data structures. For that matter, RDF distinguishes only *subjects*, *predicates*, and *objects* on the data structure level. These three sets are too coarse-grained to find meaningful INDs, as can be seen in the example dataset in Figure 3.1: No pair of columns in the table constitutes an IND – and even if it would, a statement such as *"The subjects are included in the objects."* is rather generic and not informative.

That being said, *conditional inclusion dependencies* (CINDs) can be seen as refined INDs. Informally, a CIND filters the including and included data of an IND with conditions

[1]Recall that a (unary) IND describes that the set of all values from one column is included in the set of all values from a further column.

and requires only these filtered sets to satisfy the IND. As a result, CINDs allow to describe meaningful inclusions within RDF data:

Example 3.1. Assume a CIND stating that the set of subjects occurring in triples with predicate rdf:type and object gradStudent is a subset of all subjects occurring in triples with predicate undergradDegreeFrom. Figure 3.1 satisfies this CIND, because the graduate students patrick and mike form a subset of people with an undergraduate degree, namely patrick, tim, and mike.

Before having a closer look on the various applications of CINDs on RDF data, let us formalize the notion of RDF data and their CINDs. The RDF dataset in Figure 3.1 resembles a relation with three columns. However, the rows in that table do not constitute (relational) tuples, which usually correspond to some entity, but *triples*, which *relate* either two entities or an entity and a value. Formally, an RDF triple is a statement of the form (s, p, o), where s is the subject, p is the predicate, and o is the object. The subject and predicate are RDF resources (URIs), while the object can also be an RDF literal, i. e., a typed value.[2] A set of triples is an RDF *dataset*. For a triple t, we denote with $t.s$, $t.p$, and $t.o$ the projection of the triple on the subject, predicate, and object, respectively. Note that we can interchangeably use the elements of triples, s, p, and o, in most definitions and algorithms. Therefore, we use the symbols α, β, and γ to denote any of these three elements.

Having formalized RDF data, we proceed to CINDs. In the relational data model, CINDs are defined as a combination of an *embedded* IND and a *pattern tableau* [Ma et al., 2014]. While the embedded IND is partially violated, the pattern tableau selects only such tuples that do not violate the embedded IND. However, the notion of embedded INDs is not appropriate for RDF data, as INDs without conditions are not meaningful here. Instead, we introduce a novel and compact CIND formalism that abandons embedded INDs and instead lifts conditions as first-class selectors for the including and included sets of CINDs. As the main advantage, our definition treats CINDs similarly to INDs, which allows us to exploit algorithmic foundations of IND discovery from Chapter 2 for the CIND case. We define a CIND based on a simple concept called *capture*. Intuitively, a capture defines a projection of a triple element α over a set of triples that satisfy a *unary* or *binary condition* on some of the other two elements β and/or γ.

Definition 3.1 (Condition)**.** A unary condition is a predicate $\phi(t) :\equiv t.\beta{=}v_1$ and a binary condition is a predicate $\phi(t) :\equiv t.\beta{=}v_1 \wedge t.\gamma{=}v_2$ where t is an RDF triple and v_1 and v_2 are constants, i. e., either an RDF resource, or literal. For simplicity, we may omit t and simply write $\phi \equiv \beta{=}v_1$ whenever it is clear.

Definition 3.2 (Capture)**.** A capture $c := (\alpha, \phi)$ combines the unary or binary condition ϕ with a projection attribute α, which is not used in ϕ. The interpretation of c on an RDF dataset T is $\mathcal{I}(T, c) := \{t.\alpha \mid t \in T \wedge \phi(t)\}$.

[2]The RDF standard further admits so called *blank nodes* as subjects and objects of triples. Essentially, blank nodes represent anonymous resources. Because the identifier of a blank node should be unique within a single dataset, we treat them as URIs in this chapter.

Example 3.2. The binary condition $\phi \equiv p{=}\mathsf{rdf{:}type} \wedge o{=}\mathsf{gradStudent}$ holds for the triples t_1 and t_2 from the example dataset in Figure 3.1. From it, we can define the capture (s, ϕ) with the interpretation $\{\mathsf{patrick}, \mathsf{mike}\}$.

Having defined a capture, we can now define a CIND in a similar fashion to a relational IND. The only difference is that an IND describes the inclusion of relational attributes, while a CIND describes the inclusion of captures.

Definition 3.3 (CIND)**.** A CIND ψ is a statement $c \subseteq c'$, where c and c' are captures. Analogous to INDs, we call c the dependent capture and c' the referenced capture. An RDF dataset T *satisfies* the CIND if and only if $\mathcal{I}(T, c) \subseteq \mathcal{I}(T, c')$.

Example 3.3. The CIND $(s, p{=}\mathsf{rdf{:}type} \wedge o{=}\mathsf{gradStudent}) \subseteq (s, p{=}\mathsf{undergradDegreeFrom})$ is satisfied by the dataset in Figure 3.1, because the interpretation of the dependent capture $\{\mathsf{patrick}, \mathsf{mike}\}$ is a subset of the referenced capture's interpretation $\{\mathsf{patrick}, \mathsf{mike}, \mathsf{tim}\}$.

3.1.2 Applications of conditional inclusion dependencies

As stated in the beginning of this chapter, INDs enable various data management tasks on relational databases, e. g., foreign key discovery and query optimization. We intend to position CINDs as their counter-part on RDF data. However, being almost 30 years younger, there is by far not that much research regarding RDF data as there is for relational databases; let alone the utilization of INDs or CINDs. Therefore, let us elaborate on three interesting opportunities to utilize CINDs to manage RDF datasets, namely *ontology reverse engineering*, *knowledge discovery*, and *query optimization.*

Ontology reverse engineering

RDF data is not always accompanied by an ontology and even if it is, it does not always adhere to that ontology's constraints. CINDs can provide general insights and reveal statements not reflected by the ontology (if there is one), as we demonstrate with actual insights on the diverse real-world datasets. Note that explaining ontologies and the RDF schema formalism is out of scope of this section; hence, we would like to refer the reader to the RDF specification for reference [Hayes and Patel-Schneider, 2014].

Ontology engineers can utilize CINDs to determine class and predicate relationships, such as: (i) class hierarchies, (ii) predicate hierarchies, and (iii) the domain and range of predicates. For instance, the CINDs $(s, p{=}\mathsf{associatedBand}) \subseteq (s, p{=}\mathsf{associatedMusicalArtist})$ and $(o, p{=}\mathsf{associatedBand}) \subseteq (o, p{=}\mathsf{associatedMusicalArtist})$ suggest that the associatedBand predicate is a subproperty of associatedMusicalArtist. This is because, according to the RDF semantics, if the predicate associatedBand is a subproperty of associatedMusicalArtist, then the set of subjects and the set of objects of the predicate associatedBand are a subset of the set of subjects and set of objects of associatedMusicalArtist, respectively.

Furthermore, the CIND $(s, p{=}\mathsf{rdf{:}type} \wedge o{=}\mathsf{Leptodactylidae}) \subseteq (s, p{=}\mathsf{rdf{:}type} \wedge o{=}\mathsf{Frog})$ reveals that the class Leptodactylidae could be a subclass of the class Frog. Similarly, the CIND

$(o, p{=}\text{movieEditor}) \subseteq (s, p{=}\text{rdf:type}{\wedge}o{=}\text{foaf:Person})$ reveals that the range of the predicate movieEditor can be the class foaf:Person.

Another interesting CIND, namely $(s, p{=}\text{classificationFunction}{\wedge}o{=}\text{“hydrolase activity”}) \subseteq (s, p{=}\text{classificationFunction} \wedge o{=}\text{“catalytic activity”})$, suggests that everything that classifies as hydrolase activity also classifies as catalytic activity. This gives hints to the ontology engineer to promote these two activities from string literals to RDF classes that could further be placed into a class hierarchy.

Our final example gives a peek ahead on the following contents of this chapter. As we show in Section 3.2.1, association rules form a special case of CINDs. As an example, the association rule $o{=}\text{lmdb:performance} \rightarrow p{=}\text{rdf:type}$ states that every triple in the profiled dataset, that has the object lmdb:performance, has the predicate rdf:type. This reveals that the entity lmdb:performance is an RDF class. All the above examples show that CINDs can help to reconstruct a missing ontology for a given RDF dataset or to revise and amend existing ontologies in a data-driven manner.

Knowledge discovery

CINDs can reveal unknown facts about individual data instances that cannot be inferred from the ontology. As an example, the two CINDs $(s, p{=}\text{writer} \wedge o{=}\text{Angus\_Young}) \subseteq (s, p{=}\text{writer} \wedge o{=}\text{Malcolm\_Young})$ along with its inversion $(s, p{=}\text{writer} \wedge o{=}\text{Malcolm\_Young}) \subseteq (s, p{=}\text{writer} \wedge o{=}\text{Angus\_Young})$ reveal that the AC/DC members, Angus and Malcolm Young, have written all their songs together. This fact is not *explicitly* stored in the original RDF dataset. A second example is the CIND $(s, p{=}\text{areaCode}{\wedge}o{=}\text{559}) \subseteq (s, p{=}\text{partOf}{\wedge}o{=}\text{California})$ meaning that cities with the area code 559 are located in California. A third example comes from a dataset about drugs: There, we discovered many similar CINDs, e. g., $(o, s{=}\text{drug00030}{\wedge}p{=}\text{target}) \subseteq (o, s{=}\text{drug00047}{\wedge}p{=}\text{target})$. This CIND in particular discloses that anything cured by drug00030 is also cured by drug00047.

Query optimization

Furthermore, CINDs can be employed for *SPARQL query optimization*. SPARQL stands for “SPARQL Protocol and RDF Query Language” and is the default language to query RDF datasets [SPARQL 1.1]. SPARQL queries usually comprise a considerable number of joins [Gallego et al., 2011], which negatively impacts performance. Knowing CINDs allows to remove useless query triples (i. e., query predicates) and hence to reduce the number of joins that need to be evaluated (query minimization). For example, consider the following 2-join SPARQL query on the data of Figure 3.1:

```
SELECT ?d ?u
WHERE {
  ?s rdf:type gradStudent .
  ?s memberOf ?d .
  ?s undergradDegreeFrom ?u .
}
```

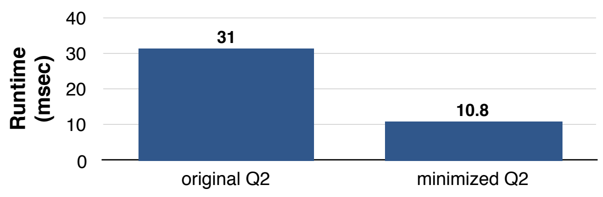


Figure 3.2: Effect of query minimization using CINDs.

Note that the data in Figure 3.1 satisfies the CIND $(s, p{=}\mathsf{memberOf}) \subseteq (s, p{=}\mathsf{rdf{:}type} \wedge o{=}\mathsf{gradStudent})$. This CIND tells us that all department members are graduate students and thus allows to remove the Line 3 from above query, thereby reducing its number of joins from two to one without affecting the final results.

As a proof of concept for this query rewriting, we used the LUBM benchmark [Guo et al., 2005] to generate RDF data and store it in an RDF-3X triple store [Neumann and Weikum, 2010]. Then, we executed the benchmark Query 2 once in its original form and once in its CIND-optimized form. Query 2 originally contains six query triples; exploiting CINDs, we could reduce the query to three triples. This rewriting results in a speed up of the query execution by a factor of 3 as depicted in Figure 3.2.

3.2 RDFind's approach to CIND discovery

Despite the various interesting applications of CINDs in RDF datasets, we are the first to consider the problem of CIND discovery for RDF data. A reason why this problem has not been tackled yet might be its complexity: Because each CIND involves two conditions, the search space for CINDs is (at least) quadratic in the number of possible conditions. As per Definition 3.1, we consider conditions that compare one or two fields of an RDF triple to one or two constants. As a result, the number of conditions that are satisfied by at least one triple of a given dataset grows with the number of distinct values and binary value combinations in said dataset.

Let us stress this point with two examples. In the tiny example data from Figure 3.1 with only 8 triples, we can already identify 35 conditions, from which we can form 48 captures, which in turn yield 2,256 CIND candidates. And looking at our second smallest evaluation dataset Diseasome with 72,445 triples, we face even over 50 billion CIND candidates – of which 1.3 billion (!) are actual CINDs.

Arguably, such large sets of CINDs do not form "small but informative summaries", as Johnson [2009] characterized data profiling results. Furthermore, such huge search spaces pose two major technical problems. First, it is difficult to maintain and explore such a large number of CIND candidates. Second, the validation of a single CIND candidate can already be very costly, let alone billions or trillions of candidates. We acknowledge that a few algorithms have been proposed for the discovery of CINDs on relational data [e. g., Bauckmann et al., 2012] and we discuss those as related work in Section 3.7. However, none of them is suited or could be easily adapted to efficiently discover CINDs within RDF datasets. Likewise, scalable IND algorithms, including SINDY, are not directly applicable to our problem: While they scale well to great numbers of tuples, they do not address the problem of handling humongous numbers of IND candidates.

The here presented algorithm RDFind tackles these issues on two levels. At first, we steer the discovery towards presumably informative CINDs, thereby reducing the result size considerably. Second, RDFind incorporates a lazy-pruning approach, which successively prunes the CIND search space throughout its different phases. We explain both measures in the following.

3.2.1 Taming the CIND search space

To cope with the utterly large CIND search spaces and result sizes entailed by RDF datasets, we narrow the search space to *pertinent* CIND*s* only. In few words, we define pertinent CINDs in a way that excludes a great number of CINDs (or CIND candidates) that are most likely not interesting or useful. Furthermore, we show that association rules are a special class of CINDs and help to further prune the CIND search space.

Pertinent CINDs

Focusing on pertinent CINDs is crucial to significantly reduce the search space and, hence, make the CIND discovery efficient. We consider a CIND as pertinent if it is both *minimal* and *broad*. Intuitively, minimal CINDs form a non-redundant cover of all valid CINDs and broad CINDs comprise a sufficiently large number of included elements. In the following, we describe both types of CINDs in more detail.

Let us begin with definining minimal CINDs. As for many other integrity constraints, we can infer certain CINDs from other ones. Given a set of valid CINDs, we call those CINDs minimal that cannot be inferred from any other CIND. For example, the CIND $(s, p{=}\mathsf{memberOf}) \subseteq (s, p{=}\mathsf{rdf{:}type} \wedge o{=}\mathsf{gradStudent})$ from Figure 3.1 is minimal and implies various non-minimal CINDs. Specifically, we consider two inference rules: the *dependent* and *referenced* implications [Ma et al., 2014]. Intuitively, tightening the dependent condition of a valid CIND (by making a unary condition binary) or relaxing the referenced condition (by making a binary condition unary) yields a new valid CIND. For instance, $(s, p{=}\mathsf{memberOf} \wedge o{=}\mathsf{csDepartment}) \subseteq (s, p{=}\mathsf{rdf{:}type} \wedge o{=}\mathsf{gradStudent})$ and $(s, p{=}\mathsf{memberOf}) \subseteq (s, p{=}\mathsf{rdf{:}type})$ are also valid, but not minimal, because they can be inferred from the above CIND.

Formally, we denote with $\phi \Rightarrow \phi'$ the fact that a binary condition ϕ implies a unary condition ϕ', i. e., the predicate of ϕ' is one of the two predicates of ϕ. For instance, $\phi \equiv p{=}\mathsf{memberOf} \wedge o{=}\mathsf{csDepartment}$ implies $\phi' \equiv p{=}\mathsf{memberOf}$. Consequently, $(\alpha, \phi) \subseteq (\alpha, \phi')$ is a valid CIND if $\phi \Rightarrow \phi'$. Therefore, if a CIND $(\alpha, \phi_1) \subseteq (\beta, \phi_2)$ holds in a dataset T, then: (i) a dependent implication states that any CIND $(\alpha, \phi'_1) \subseteq (\beta, \phi_2)$ with $\phi'_1 \Rightarrow \phi_1$ also holds in T, because $(\alpha, \phi'_1) \subseteq (\alpha, \phi_1) \subseteq (\beta, \phi_2)$; and similarly (ii) a referenced implication states that any CIND $(\alpha, \phi_1) \subseteq (\beta, \phi'_2)$ with $\phi_2 \Rightarrow \phi'_2$ also holds in T, because $(\alpha, \phi_1) \subseteq (\beta, \phi_2) \subseteq (\beta, \phi'_2)$. This allows us to define minimal CINDs as follows:

Definition 3.4 (Minimal CIND)**.** A CIND $(\alpha, \phi_1) \subseteq (\beta, \phi_2)$ is minimal if and only if (i) $(\alpha, \phi_1') \subseteq (\beta, \phi_2)$ is not a valid CIND for any condition ϕ_1' that implies ϕ_1; and (ii) $(\alpha, \phi_1) \subseteq (\beta, \phi_2')$ is not a valid CIND for any condition ϕ_2' that is implied by ϕ_2.

Example 3.4. Figure 3.3 depicts four CINDs for the dataset from Figure 3.1. The CIND ψ_1 implies ψ_2 and ψ_3, which in turn imply ψ_4, respectively. Hence, only ψ_1 is minimal.

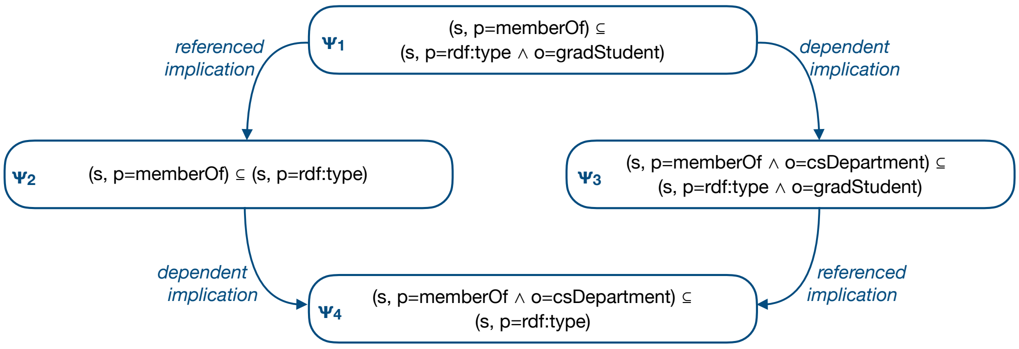

Ψ1
(s, p=memberOf) ⊆
(s, p=rdf:type ∧ o=gradStudent)
referenced
implication
dependent
implication
Ψ2
(s, p=memberOf) ⊆ (s, p=rdf:type)
Ψ3
(s, p=memberOf ∧ o=csDepartment) ⊆
(s, p=rdf:type ∧ o=gradStudent)
dependent
implication
referenced
implication
Ψ4
(s, p=memberOf ∧ o=csDepartment) ⊆
(s, p=rdf:type)

Figure 3.3: Extract from the CIND search space for Figure 3.1. The nodes are CIND candidates and the arrows implications.

Having settled minimality, let us now proceed to explain broad CINDs. Intuitively, a broad CIND describes the inclusion of a sufficient or large number of distinct values. For instance, if we require the inclusion of at least two values, then $(s, p{=}\mathsf{rdf{:}type} \wedge o{=}\mathsf{gradStudent}) \subseteq (s, p{=}\mathsf{undergradDegreeFrom})$ is broad. Focusing on broad CINDs avoids CINDs that (i) embed void conditions, which do not match a single triple in a given RDF dataset and which are infinite in number, (e. g., $(s, p{=}\mathsf{memberOf} \wedge o{=}\mathsf{geoDepartment}) \subseteq (s, p{=}\mathsf{rdf{:}type} \wedge o{=}\mathsf{professor})$) and that (ii) pertain to very few distinct values, which are neither useful to summarize nor to state the general properties of a given RDF dataset (e. g., $(o, s{=}\mathsf{patrick} \wedge p{=}\mathsf{rdf{:}type}) \subseteq (o, s{=}\mathsf{mike} \wedge p{=}\mathsf{rdf{:}type})$). Formally, we define this number of distinct values as *support*, inspired from the widely adopted measure for association rules [Agrawal and Srikant, 1994]:

Definition 3.5 (Support)**.** Given an RDF dataset T, the support of a CIND $\psi = c \subseteq c'$ is defined as $\mathrm{supp}(\psi) := |\mathcal{I}(T, c)|$.

Example 3.5. The CIND $(s, p{=}\mathsf{memberOf} \wedge o{=}\mathsf{csDepartment}) \subseteq (s, p{=}\mathsf{undergradDegreeFrom} \wedge o{=}\mathsf{hpi})$ has a support of 1 in the dataset from Figure 3.1. This is because its dependent capture $(s, p{=}\mathsf{memberOf} \wedge o{=}\mathsf{csDepartment})$ selects a single value, namely patrick. Thus, this CIND describes a rather specific insight that pertains only to a single person.

Definition 3.6 (Broad CIND)**.** Given an RDF dataset T and a user-defined support threshold $s_{\min}$, the CIND ψ is considered broad if its support satisfies the support threshold, i. e., $\mathrm{supp}(\psi) \geq s_{\min}$.

The choice of the support threshold depends on the use case and its dataset. In our experience, h=1,000 is a reasonable choice for the query optimization and ontology reverse engineering use cases, while h=25 works well for the knowledge discovery use case. That being said, we evaluate the impact of the support threshold in detail in the evaluation in Section 3.6.4. Without any loss of generality, we assume this threshold to be given. Usually, even small support thresholds bear great pruning power. In the aforementioned Diseasome dataset, over 84 % of its 219 million minimal CINDs have a support of 1 and from the other 34.9 million CINDs, another 97.4 % have a support below 10.

CINDs as Association Rules

CINDs are natural extensions of regular INDs, but they also share some properties with exact *association rules*[3] (ARs[4]), i. e., those with confidence 1. By interpreting RDF triples (e. g., (patrick, rdf:type, gradStudent)) as transactions ($\{s{=}$patrick, $p{=}$rdf:type, $o{=}$gradStudent$\}$), we can find ARs in RDF datasets, such as $o{=}$gradStudent $\rightarrow$ $p{=}$rdf:type in Figure 3.1.

Every AR $\alpha{=}v_1 \rightarrow \beta{=}v_2$ implies the CIND $(\gamma, \alpha{=}v_1) \subseteq (\gamma, \alpha{=}v_1 \wedge \beta{=}v_2)$, e. g., the above example AR implies the CIND $(s, o{=}\text{gradStudent}) \subseteq (s, p{=}\text{rdf:type} \wedge o{=}\text{gradStudent})$. The reason is as follows: Every triple satisfying the condition $o{=}$gradStudent also satisfies $p{=}$rdf:type. In other words, the triples that satisfy $o{=}$gradStudent are a subset of those triples that satisfy $p{=}$rdf:type (and $o{=}$gradStudent, for that matter). It follows that the projections of those triple sets to the subject s are also in a subset relationship.

Nevertheless, the inverse implication is not necessarily correct: For example, adding the triple (patrick, status, gradStudent) to the RDF dataset in Figure 3.1 would invalidate the AR but not the CIND. Also, in our experience the vast majority of CINDs is not implied by any AR. In particular, all example CINDs in Section 3.1 are not implied by ARs unless explicitly mentioned otherwise.

That being said, when profiling an RDF dataset for CINDs, ARs can replace some those CINDs, thereby enhancing the result's understandability and enabling further applications, such as selectivity estimation [Ilyas et al., 2004]. Moreover, AR discovery is less complex than CIND discovery. We leverage this circumstance: We quickly discover the ARs in a dataset and use them to prune the CIND search space, thereby improving the efficiency of our CIND discovery algorithm.

* * *

The above theoretical insights allow us to avoid a naïve, hardly tractable problem statement ("Discover all CINDs in a given dataset.") and formulate a more elaborate one instead: For a given dataset T and a user-defined CIND support threshold $s_{\min}$, efficiently discover all *pertinent* CINDs that hold on T, that is, all CINDs that are both minimal and broad. Moreover, if a pertinent CIND ψ is implied by an AR r, provide r instead of ψ due to its stronger semantics.

[3]The association rules considered in this chapter are different from the ones used by Abedjan and Naumann [2013]. See Section 3.7 for details.

[4]Note that association rules and augmentation rules (the latter being introduced in the context of IND discovery in Definition 2.3) happen to share the abbreviation "AR", but are unrelated concepts.

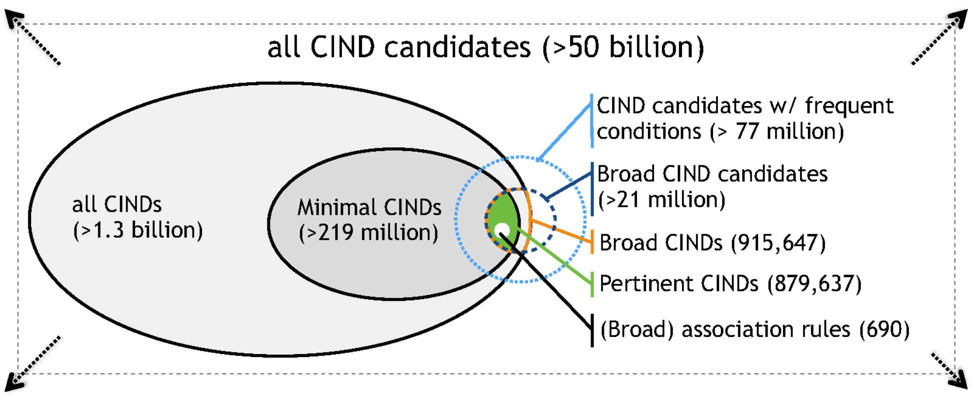

all CIND candidates (>50 billion)
all CINDs (>1.3 billion)
Minimal CINDs (>219 million)
CIND candidates w/ frequent conditions (> 77 million)
Broad CIND candidates (>21 million)
Broad CINDs (915,647)
Pertinent CINDs (879,637)
(Broad) association rules (690)

Figure 3.4: The CIND search space for the Diseasome dataset (72,445 triples) and a support threshold of 10.

3.2.2 Algorithm Overview

Having defined a concrete CIND discovery problem, we proceed to outline how our algorithm RDFIND solves it. Similar to SINDY, we formulate RDFIND as a distributable data flow (cf. Section 2.2.4). Furthermore, RDFIND operates in the same environment presented in Section 1.4, with one minor modification, though: RDFIND operates on RDF data instead of relational data. The triples of this dataset, however, may also be distributed among the workers, e. g., by means of an HDFS or a triple store.

In order to efficiently discover all pertinent CINDs in such a scenario, RDFIND proceeds as explained in the following. In practice, the set of minimal CINDs is often considerably larger than the set of broad CINDs. For example, the Diseasome dataset, which we mentioned already in the above section, comprises approximately 219 million minimal CINDs but fewer than 1 million broad CINDs for a support threshold of 10 (see Figure 3.4). On the face of these numbers, it seems highly reasonable to reduce the search space to broad CIND candidates first and only then proceed to consider the minimality criterion.

To this end, we devise a *lazy pruning* strategy that reduces the search space in two phases. Figure 3.5 illustrates the overall architecture of RDFIND, which comprises three main components: the *Frequent Condition Detector* (FCDetector), the *Capture Groups Creator* (CGCreator), and the CIND *Extractor* (CINDExtractor). The first and third component are responsible for specific steps in the lazy pruning employed by RDFIND. The second component reorganizes the input data in a manner that allows for efficient CIND extraction. We briefly discuss these components next and give the algorithmic details in the following sections.

Before initiating the actual search for CINDs, the **FCDetector** first narrows the search space to a set of CIND candidates having *frequent conditions* only, i. e., conditions on the input dataset that are satisfied by a certain minimum number of triples. This represents the first phase of the lazy pruning. This pruning works, because all broad CINDs embed only frequent conditions, as we prove in Section 3.3). RDFIND also exploits frequent conditions to easily derive association rules and, consequently, to further prune the search space.

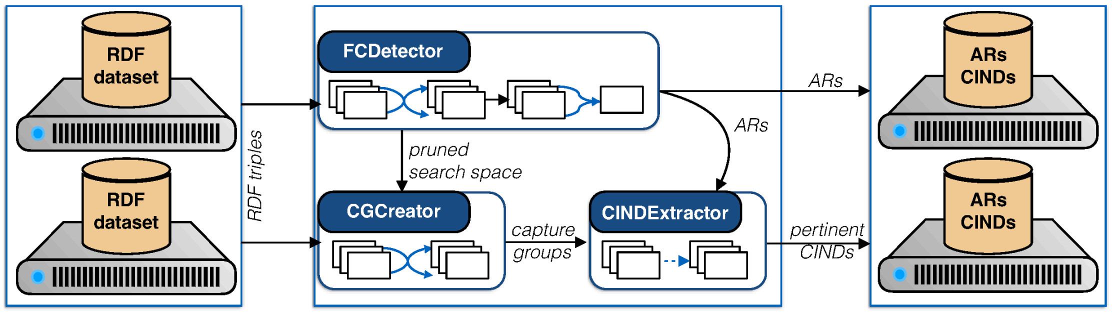


Figure 3.5: Overview of the RDFind system.

Next, the **CGCreator** transforms all RDF triples in the previously pruned search space into compact representations, called *capture groups*, from which it can efficiently extract CINDs. A capture group is a set of captures whose interpretations have a certain value in common. For example, in the data from Figure 3.1 the value patrick spawns a capture group containing, amongst others, the capture $(s, p{=}\text{rdf:type} \wedge o{=}\text{gradStudent})$. We explain these steps in detail in Section 3.4.

Given the capture groups, the **CINDExtractor** reduces the search space of CIND candidates with frequent conditions to the set of broad CIND candidates. This is the second phase of our lazy pruning strategy. Subsequently, this third component extracts the broad CINDs and their support from the capture groups. As CIND extraction is usually the most expensive step, the CINDExtractor is equipped with several techniques, such as load balancing, to perform this step efficiently. Finally, it mines for pertinent CINDs, i. e., it searches for minimal CINDs among all discovered broad CINDs. The details of this component are given in Section 3.5.

3.3 Frequent Condition Discovery

As a start, RDFind executes the first phase of our lazy pruning strategy and reduces the search space to the set of CIND candidates whose conditions (cf. Definition 3.1) are frequent. Knowing frequent conditions is crucial for two main reasons: First, they allow RDFind to significantly reduce the search space and, thus, to achieve low execution times and memory footprints. Second, they yield ARs (cf. Section 3.2.1) at little cost, which improve the output usefulness. In the following, we further explain *why* frequent conditions (as well as ARs) help us to reduce the search space towards finding broad CINDs. Then, we detail *how* we discover frequent conditions and ARs.

3.3.1 Why Frequent Conditions?

A frequent condition is that condition whose number of satisfying triples (*condition frequency*) is not below the user-defined support threshold (cf. Definition 3.5). The support of a CIND is tightly connected with the condition frequency of its dependent and referenced captures, as we assert in the following, simple lemma.

Lemma 3.1. *Given a* CIND $\psi := (\alpha, \phi) \subseteq (\beta, \phi')$ *with support* $\text{supp}(\psi)$*, the condition frequencies of* ϕ *and* ϕ' *are equal to or greater than* $\text{supp}(\psi)$*.*

Proof. From the support definition, we have that the interpretation of the dependent capture (α, ϕ) contains $\text{supp}(\psi)$ values. The referenced capture (β, ϕ'), which is a superset, therefore contains at least $\text{supp}(\psi)$ values. As each value in the interpretation of a capture must be found in at least one triple and each triple yields exactly one value, the capture's embedded condition must satisfy at least as many triples as this number of values. Thus, both ϕ and ϕ' must have a condition frequency $\geq \text{supp}(\psi)$. □

With Lemma 3.1 we do not need to validate CIND candidates having conditions with a frequency below a user-specified support. Indeed, finding frequent conditions drastically reduces the CIND search space. Figure 3.6 shows that, for real world-datasets, the vast majority of the conditions are satisfied by only very few triples. For instance, in the DB14-MPCE dataset, 86% of the conditions have a frequency of 1, i. e., they hold for a single triple only, and 99% of the conditions have a frequency of less than 16. In practice, however, most CIND use cases require the conditions to have a high frequency.

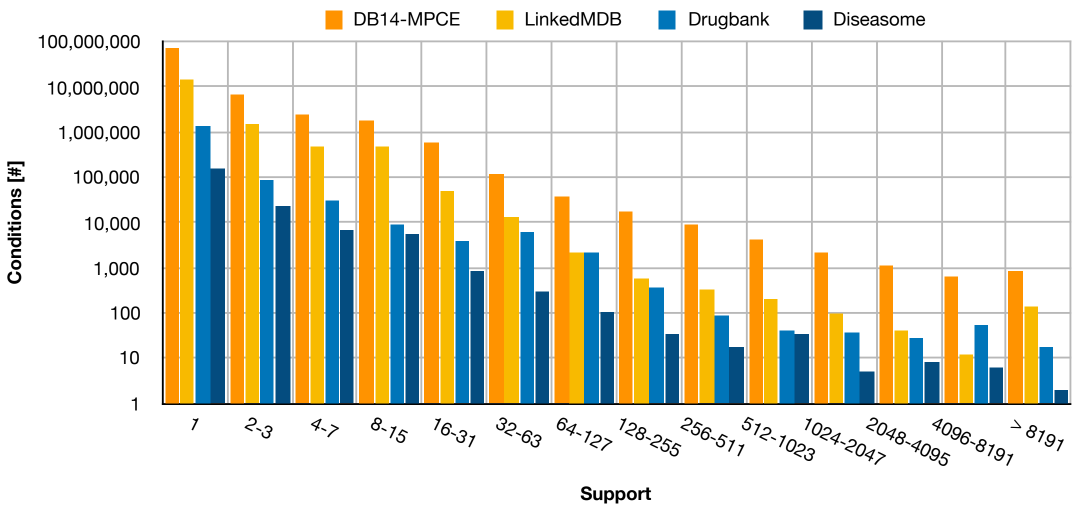


Figure 3.6: Number of conditions by frequency in real-world datasets of varying size from $\sim 72k$ (Diseasome) to $\sim 33M$ (DBP14-MPCE) triples.

In addition, frequent conditions allow RDFIND to easily derive ARs. As discussed in Section 3.2.1, the system can use ARs to further prune the CIND search space. Recall that the AR $\theta := \beta{=}v_1 \rightarrow \gamma{=}v_2$ implies the CIND $\psi := (\alpha, \beta{=}v_1) \subseteq (\alpha, \beta{=}v_1 \wedge \gamma{=}v_2)$. For instance, the data in Figure 3.1 contains o=gradStudent $\rightarrow$ p=rdf:type, which implies $(s, p{=}\text{rdf:type}) \subseteq (s, p{=}\text{rdf:type} \wedge o{=}\text{gradStudent})$. Therefore, RDFIND can simply keep ARs and exclude all its implied CINDs from the CIND search.

Besides replacing CINDs with ARs, the latter further allow for *equivalence pruning*: The reverse CIND of ψ, i. e., $(\alpha, \beta{=}v_1 \wedge \gamma{=}v_2) \subseteq (\alpha, \beta{=}v_1)$, trivially holds, because its dependent condition logically implies its referenced condition. In consequence, the AR θ

implies that (the interpretations of) $(\alpha, \beta{=}v_1)$ and $(\alpha, \beta{=}v_1 \land \gamma{=}v_2)$ are equal. Accordingly, our above example captures $(s, o{=}\text{gradStudent})$ and $(s, p{=}\text{rdf:type}\land o{=}\text{gradStudent})$ contain the exact same values, namely patrick and mike. Therefore, an AR $\beta{=}v_1 \rightarrow \gamma{=}v_2$ allows RDFind to prune all CIND candidates involving the capture $(\alpha, \beta{=}v_1 \land \gamma{=}v_2)$; they are equivalent to the candidates involving the capture $(\alpha, \beta{=}v_1)$.

3.3.2 Finding Frequent Conditions

We formulate the problem of finding frequent unary and binary conditions as a frequent itemset discovery problem. As such, we interpret each triple (s_1, p_1, o_1) as a transaction $\{\langle s{=}s_1\rangle, \langle p{=}p_1\rangle, \langle o{=}o_1\rangle\}$[5]. Directly applying the state-of-the-art Apriori algorithm [Agrawal and Srikant, 1994] to find all frequent unary and binary conditions, however, would be inefficient as it was designed for single-node settings and itemsets with arbitrarily many items. Furthermore, it does not scale to large amounts of frequent itemset candidates, because it needs to keep all candidates in memory. Therefore, we tailor the Apriori algorithm to our specific problem. The result is a fully distributed algorithm that scales to arbitrary amounts of candidates by *checking candidates on-demand* using space-efficient indices. Figure 3.7 depicts the data flow of our algorithm to discover frequent conditions. Generally speaking, it consists of two passes over the data and an AR detection phase that takes place on the fly. In more detail, it operates in the following four main steps:

In the first step, we discover frequent *unary* conditions. For that purpose, we assume that all triples are distributed among the RDFind workers of a cluster, e. g., by means of a distributed triple store (cf. Figure 3.5). Each worker can then process an independent horizontal partition of the input dataset. A worker reads each input RDF triple in its data partition and creates three unary conditions with a *condition counter* set to 1 (Step (1) in Figure 3.7). For instance, a worker creates the three unary condition counters $(\langle s{=}\text{patrick}\rangle, 1)$, $(\langle p{=}\text{rdf:type}\rangle, 1)$, and $(\langle o{=}\text{gradStudent}\rangle, 1)$ for the triple (patrick, rdf:type, gradStudent) from Figure 3.1. All workers then run a global `GroupReduce`[6] on the conditions of the unary condition counters and add up the counters for each resulting group (Step (2)). As this global `GroupReduce` requires shuffling data through the network, RDFind runs early-aggregations on the counters before shuffling the data, which significantly reduces the network load. After globally aggregating counters, the workers discard all non-frequent conditions with a frequency less than the user-specified support threshold.

In the second step, we compact the frequent unary conditions: Once all those conditions are found, RDFind needs to index them for efficient containment tests both in the frequent binary conditions discovery and the CINDs pruning phase. RDFind tolerates false positives in this index, so we use a Bloom filter to attain constant look-up time and a small memory footprint (tens of MB for the largest datasets). Additionally, RDFind can create this Bloom filter in a fully distributed manner: Each worker encodes all of its

[5]The $\langle ... \rangle$ notation introduces its enclosed formulas as syntactical elements rather than their results.

[6]As in Chapter 2, `GroupReduce` refers to the grouping aggregation operation of distributed data flow systems, such as Apache Flink [Flink].

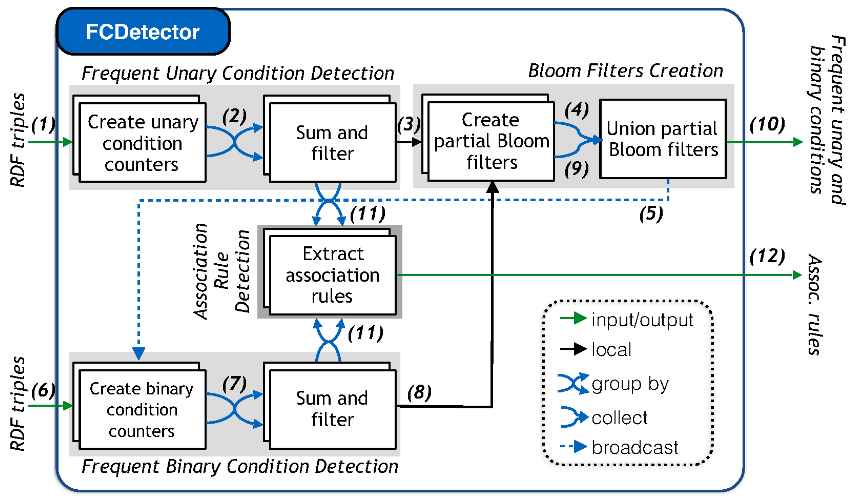

FCDetector
Frequent Unary Condition Detection
Bloom Filters Creation
RDF triples
(1)
Create unary condition counters
(2)
Sum and filter
(3)
Create partial Bloom filters
(4)
(9)
Union partial Bloom filters
(10)
Frequent unary and binary conditions
(5)
(11)
Association Rule Detection
Extract association rules
(12)
Assoc. rules
(11)
RDF triples
(6)
Create binary condition counters
(7)
Sum and filter
(8)
Frequent Binary Condition Detection
input/output
local
group by
collect
broadcast

Figure 3.7: Data flow of the FCDetector.

locally residing frequent conditions in a Bloom filter (Step (3)). The workers then send their local Bloom filter to a single worker that unions them by calculating a bit-wise `OR` (Step (4)).

Finding frequent *binary* conditions constitutes the third step of the frequent condition discovery. A binary condition can be frequent only if its two embedded unary conditions are frequent [Agrawal and Srikant, 1994]. At this point, the original Apriori algorithm would generate all possible frequent binary condition candidates and organize them in a tree structure for subsequent counting. In RDF scenarios, however, this tree structure easily becomes too large to fit into main memory. To overcome this challenge, RDFIND never materializes the candidates and, instead, introduces on-demand candidate checking. For this, it broadcasts the previously created Bloom filter to all workers (Step (5)). Each worker runs Algorithm 3.1 to find the frequent binary condition candidates. In detail, each worker reads the triples from its data partition (Line 1 & Step (6)) and performs the candidate check on demand: First, it probes each unary condition embedded in the input triple against the Bloom filter (Line 2). Then, it generates all possible frequent binary condition candidates using the frequent unary conditions of the triple (Lines 3-5). For example, consider the triple (patrick, memberOf, csDepartment) from Figure 3.1. Knowing by means of the Bloom filter that only the two embedded unary conditions s=patrick and p=memberOf are frequent, the only candidate for a frequent binary condition is s=patrick $\wedge$ p=memberOf. It then creates a binary condition counter for each frequent binary condition candidate (Line 6). As for frequent unary condition discovery, RDFIND globally aggregates the binary condition counters (Step (7)) and keeps only the frequent binary conditions.

In the fourth and final step of the frequent condition discovery, RDFIND encodes all frequent binary conditions in a Bloom filter in order to speed up the CIND pruning phase (Steps (8) and (9)). This is basically the same procedure as the compaction of the frequent unary conditions. As a result of this process, RDFIND outputs the set of

Algorithm 3.1: Create counters for binary conditions

```
Data: RDF triples $T$, unary condition Bloom filter $B_u$
1 foreach $t \in T$ do
2     probe $\langle s{=}t.s\rangle$, $\langle p{=}t.p\rangle$, and $\langle o{=}t.o\rangle$ in $B_u$;
3     foreach $(\alpha, \beta) \in \{(s,p),(s,o),(p,o)\}$ do
4         $v_\alpha \leftarrow t.\alpha$; $v_\beta \leftarrow t.\beta$;
5         if $\langle \alpha{=}v_\alpha\rangle$ and $\langle \beta{=}v_\beta\rangle$ are frequent then
6             forward $(\langle \alpha{=}v_\alpha \wedge \beta{=}v_\beta\rangle, 1)$;
```

frequent unary and binary conditions, which implicitly represent the pruned CIND search space (Step (10)).

3.3.3 Extracting Association Rules

As in [Agrawal and Srikant, 1994], our frequent conditions discovery algorithm also allows RDFind to extract association rules at little extra cost. It simply performs a distributed join of the frequent unary condition counters with the frequent binary condition counters on their embedded unary conditions (Step (11)). For instance, $(\langle p{=}\text{rdf:type}\rangle, 3)$ and $(\langle o{=}\text{gradStudent}\rangle, 2)$ both join with $(\langle p{=}\text{rdf:type} \wedge o{=}\text{gradStudent}\rangle, 2)$. Then each worker checks for each pair in its partition of the join result, if the unary and binary condition counters have the same counter value. In our example, this is true for $(\langle o{=}\text{gradStudent}\rangle, 2)$ and $(\langle p{=}\text{rdf:type}\wedge o{=}\text{gradStudent}\rangle, 2)$, hence, the responsible worker derives the association rule $o{=}\text{gradStudent} \rightarrow p{=}\text{rdf:type}$ with a support of 2. RDFind uses the association rules to further prune the CIND search, as described in Section 3.3, and additionally includes them in the final result for users (Step (12)), because they are a special class of CINDs. In particular, the association rule support is equal to the support of its implied CINDs according to the following lemma.

Lemma 3.2. *The support s of the association rule $\alpha{=}v \rightarrow \beta{=}v'$ is equal to the support of its implied* CIND $(\gamma, \alpha{=}v) \subseteq (\gamma, \alpha{=}v \wedge \beta{=}v')$.

Proof. Let s be equal to the support of $\alpha{=}v \rightarrow \beta{=}v'$. By definition, the frequencies of the conditions $\phi_1 := \alpha{=}v \wedge \beta{=}v'$ and $\phi_2 := \alpha{=}v$ are also s. Because all triples in an RDF dataset are distinct, the s triples selected by ϕ_1 (and hence by ϕ_2) must have pairwise distinct values in γ. Thus, the interpretation of capture $(\gamma, \alpha{=}v)$ contains s elements. □

3.4 Compact RDF Representation

After discovering the frequent unary and binary conditions, RDFind transforms the RDF triples into a compact representation that allows it to efficiently create CIND candidates. We call this compact data structure *capture group*. A capture group is a set of captures

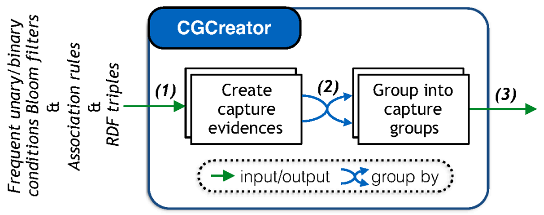

Frequent unary/binary conditions Bloom filters & Association rules & RDF triples
CGCreator
(1)
Create capture evidences
(2)
Group into capture groups
(3)
input/output
group by

Figure 3.8: Data flow of the CGExtractor.

(Definition 3.2) whose interpretations have a value in common. Captures having n values in common co-occur in n capture groups. We first explain how the system creates captures groups and then demonstrate that we can obtain all broad CIND candidates from capture groups only. Note that the component responsible for the creation of capture groups, the CGExtractor, is algorithmically similar to (but more complex than) SINDY's attribute set creation described in Section 2.2.2.

3.4.1 Capture Groups

RDFIND creates capture groups in two steps that are depicted in Figure 3.8. It first outputs the evidence that a certain value belongs to a capture (*capture evidence*). For this, it takes into consideration the previously found frequent conditions and ARs. Then, it groups and aggregates capture evidences with the same value, thereby creating the capture groups. We detail these two steps in the following.

A capture evidence is a statement that a certain value exists in a capture (interpretation). There are nine possible capture evidences per triple: For each of the three values of a triple, we could create three capture evidences, of which one has a binary condition and the other two have unary conditions (cf. Definitions 3.1 and 3.2). For example, the triple (patrick, memberOf, csDepartment) entails, amongst others, the capture evidences patrick $\in (s, p{=}\text{memberOf})$ and patrick $\in (s, p{=}\text{memberOf} \wedge o{=}\text{csDeparment})$. One might think that this is an expensive task as it would increase the input data volume by a factor of nine. However, remember that at this point RDFIND works on a highly pruned search space containing only frequent conditions (see Section 3.3). In addition, our system further reduces the number of capture evidences via implications between binary and unary conditions. For instance, consider again the triple (patrick, memberOf, csDepartment) and its capture $(s, p{=}\text{memberOf} \wedge o{=}\text{csDepartment})$. The resulting capture evidence patrick $\in (s, p{=}\text{memberOf} \wedge o{=}\text{csDepartment})$ subsumes both patrick $\in (s, p{=}\text{memberOf})$ and patrick $\in (s, o{=}\text{csDepartment})$. Hence, it suffices to keep the first binary capture evidence and discard the two unary ones.

Algorithm 3.2 shows in detail the capture evidence creation process. Prior to its execution, the frequent condition Bloom filters and the ARs discovered by the FCDetector are broadcast, so that they are available to every worker (cf. Figure 3.7). As for the frequent condition discovery, each worker then processes its partition of the input triples. For each triple, it first picks a projection attribute α (Line 3), e. g., $\alpha = s$, and two condition attributes β and γ (Line 4), e. g., $\beta = p$ and $\gamma = o$. Then, for the two emerging unary conditions (p=memberOf and o=csDepartment), it checks whether they might be frequent using the Bloom filter for frequent unary conditions (Lines 5–7). If so, it also checks whether the emerging binary condition (p=memberOf$\wedge o$=csDepartment) is frequent, too[7] (Line 8) and does not embed a known AR (Lines 9–10). In this case, it creates a capture evidence with the binary condition only (Line 11; (patrick $\in$ (s, p=memberOf$\wedge o$=csDepartment)). Otherwise, it creates the capture evidences for those of the two unary conditions that are frequent (Lines 12–14). Finally, these steps are repeated for the remaining projection attributes (p and o).

Algorithm 3.2: Creating capture evidences

```
Data: triples $T$, Bloom filter $B_u$ for frequent unary conditions, Bloom filter $B_b$
      for frequent binary conditions, ARs $AR$
Result: Evidences of relevant captures $C$
1  foreach $t \in T$ do
2      $C \leftarrow \emptyset$;
3      foreach $\alpha \in \{s, p, o\}$ do
4          $\{\beta, \gamma\} \leftarrow \{s, p, o\} \setminus \{\alpha\}$;
5          $v_\alpha \leftarrow t.\alpha$; $v_\beta \leftarrow t.\beta$; $v_\gamma \leftarrow t.\gamma$;
6          if $\langle \beta = v_\beta \rangle \in B_u$ then
7              if $\langle \gamma = v_\gamma \rangle \in B_u$ then
8                  if $\langle \beta = v_\beta \wedge \gamma = v_\gamma \rangle \in B_b$
9                  $\wedge \langle \beta{=}v_\beta \rightarrow \gamma{=}v_\gamma \rangle \notin AR$
10                 $\wedge \langle \gamma{=}v_\gamma \rightarrow \beta{=}v_\beta \rangle \notin AR$ then
11                     $C \leftarrow C \cup \{\langle v_\alpha \in (\alpha, \beta{=}v_\beta \wedge \gamma{=}v_\gamma] \rangle\}$;
12                 else $C \leftarrow C \cup \{\langle v_\alpha \in (\alpha, \beta{=}v_\beta) \rangle, \langle v_\alpha \in (\alpha, \gamma{=}v_\gamma) \rangle\}$;
13             else $C \leftarrow C \cup \{\langle v_\alpha \in (\alpha, \beta{=}v_\beta) \rangle\}$;
14         else if $\langle \gamma{=}v_\gamma \rangle \in B_u$ then  $C \leftarrow C \cup \{\langle v_\alpha \in (\alpha, \gamma{=}v_\gamma) \rangle\}$ ;
```

RDFind aggregates all capture evidences with the same value using a global `Group-Reduce` and calculates the union of their captures in order to create *capture groups.* Again, early aggregates are calculated whenever possible to reduce network and memory pressure. Although each capture group corresponds to a certain value from the input RDF dataset, the system discards the values as they are no longer needed. For instance, for the dataset in Figure 3.1, a support threshold of 3 and the value patrick, we have the capture evidences patrick $\in$ (s, p=rdf:type) and patrick $\in$ (s, p=undergradDegreeFrom). The aggregation combines them into the capture group $\{(s, o$=rdf:type$), (s, p$=undergradDegreeFrom$)\}$. Note that the capture groups are distributed among the workers after this step and can therefore be processed in a distributed manner by the following component, the CINDExtractor.

[7]Notice that testing the unary conditions before the binary ones avoids some false positives from the binary Bloom filter.

3.4.2 From Capture Groups to Broad CINDs

Let us now show that it is possible to extract all broad CINDs from a given RDF dataset using capture groups. To this end, we exploit commonalities of the capture-based CIND definition (i. e., Definition 3.3) with our attribute set-based IND characterization in Theorem 2.1. Intuitively, a CIND is satisfied if each capture group that contains the referenced capture also contains the dependent capture. Formally:

Lemma 3.3. *Let T be an RDF dataset and $\mathcal{G}$ its capture groups. Then, a CIND $\psi := c \subseteq c'$ is valid on T iff $\forall G \in \mathcal{G}\colon c \in G \Rightarrow c' \in G$, with $supp(\psi) = |\{G \in \mathcal{G} \mid c \in G\}|$.*

Proof. By construction of the capture groups, for each value v in the interpretation of a certain capture c, $\mathcal{I}(T, c)$, c is contained in a dedicated capture group G_v – and vice versa. In other words, values and capture groups are in a one-to-one relationship. In consequence, (the interpretation of) c' contains all values of c if and only if c' is member of all capture groups in which c is a member. Moreover, the number of capture group memberships of c is exactly the number of values in c and, hence, the support of ψ. □

Because at this step RDFind operates in the search space of CIND candidates having frequent conditions, we infer from the above lemma that all broad CINDs can be extracted from capture groups.

Theorem 3.4. *Given an RDF dataset T with its capture groups $\mathcal{G}$ and a support threshold s_{min}, any valid CIND ψ with support $supp(\psi) \geq s_{min}$ can be extracted from $\mathcal{G}$.*

Proof. This trivially holds from Lemmata 3.1 and 3.3. □

3.5 Fast CIND Extraction

As a final step, RDFind's CINDExtractor component (see Figure 3.9) extracts pertinent CINDs from the previously created capture groups. It proceeds in two main steps: It first extracts broad CINDs from capture groups and then finds the minimal CINDs among the broad ones. In the following, we first show that directly extracting broad CINDs is inadequate for RDF datasets (Section 3.5.1). We then show how RDFind extracts broad CINDs from capture groups efficiently (Section 3.5.2). We finally show how our system extracts the pertinent CINDs from broad CINDs (Section 3.5.3).

3.5.1 Inadequacy of a Direct Extraction

RDFind's broad CIND extraction mechanism could be based on existing IND discovery techniques, such as Sindy's IND extraction as described in Section 2.2.3. It would work as follows: Because a valid CIND's dependent capture is in the same capture groups as its referenced capture (Lemma 3.3), all workers enumerate all CIND candidates within their capture groups. They also add a support counter (initially set to 1) to each CIND candidate as shown in Example 3.6.

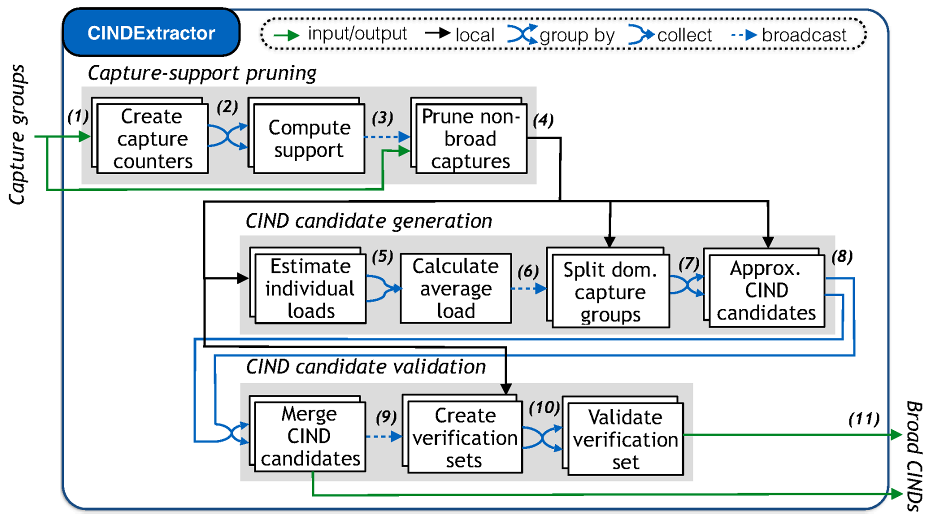


Figure 3.9: Data flow of the CINDExtractor.

Example 3.6. Consider a scenario with three capture groups: $G_1 = \{c_a, c_b, c_c, c_d, c_e\}$, $G_2 = \{c_a, c_b\}$, and $G_3 = \{c_c, c_d\}$. In this case, the naïve approach generates five CIND *candidate sets* for G_1, e. g., $(c_a \sqsubseteq \{c_b, c_c, c_d, c_e\}, 1)$, and two for G_2 and G_3.

The system then performs a global GroupReduce on the dependent capture of the CIND candidate sets and aggregates them by intersecting all their referenced captures and summing up all its supports. The aggregated CIND candidate sets represent all valid CINDs and the support count is used to retain only the broad CINDs.

The performance of this approach suffers from capture groups with a large number of captures (*dominant capture groups*). Processing a capture group with n captures yields n CIND candidate sets with up to n contained captures each, i. e., the overall number of captures in CIND candidate sets is quadratic in the size of the corresponding capture group. Therefore, dominant capture groups entail enormous memory consumption and CPU load as well as a highly skewed load distribution among workers, which severely impacts performance. Formally, inspired by Kolb and Rahm [2012], we consider a capture group G as dominant if its processing load, estimated by $|G|^2$, is larger than the average processing load among all w workers, $\frac{\sum_{G_i \in \mathcal{G}} |G_i|^2}{w}$. For example, assuming two workers (i. e., $w = 2$), we consider capture group G_1 in Example 3.6 as dominant, because $|G_1|^2 > \frac{|G_1|^2+|G_2|^2+|G_3|^2}{2}$. In practice, RDF datasets lead to several very large capture groups that emerge from frequently occurring values, such as rdf:type. This renders the above naïve solution inadequate for RDF.

3.5.2 Cracking Dominant Capture Groups

To efficiently cope with dominant capture groups, we enhance in the following the above simple extraction mechanism by (i) pruning the capture groups as much as possible, (ii) load balancing, and (iii) a two-phase CIND extraction strategy. Figure 3.9 depicts the complete process.

As the first step to deal with dominant capture groups, RDFIND applies a novel *capture-support pruning* technique, which is also the second phase of our lazy pruning technique. The capture-support pruning reduces the size of all capture groups by removing some of their captures. The first phase of our lazy pruning technique (see Section 3.3) retains CIND candidates whose captures embed frequent conditions only. In general, however, these CIND candidates are a proper superset of the broad CIND candidates, because the support of a capture can be smaller than the frequency of its embedded condition. The capture-support pruning identifies and removes these captures from all capture groups. Basically, our system first computes all capture supports by distributedly counting their occurrences in the capture groups (Steps (1) & (2) in Figure 3.9), then broadcasts the prunable captures with a support less than the user-defined support threshold $s_{\min}$ to each worker, and eventually each worker removes these captures from its capture groups (Step (3)). For example, assuming that $s_{\min}$ is set to 2, we can then identify in Example 3.6 that c_e appears only in G_1, so its support is 1. We therefore remove it from G_1, resulting in the following smaller capture groups: $G'_1 = \{c_a, c_b, c_c, c_d\}$, $G_2 = \{c_a, c_b\}$, and $G_3 = \{c_c, c_d\}$.

While the capture-support pruning significantly reduces the size of capture groups, some dominant capture groups remain and still severely impact performance. RDFIND now clears the skewed work distribution caused by them. For this purpose, each worker estimates its current load by summing the squares of its capture groups' sizes. These loads are then summed up on a single worker (Step (5)), divided by the number of workers, and the resulting average load is broadcast back to all workers (Step (6)). Then, each worker identifies its dominant capture groups and divides them into w work units, and uniformly redistributes them among all workers (Step (7)). Concretely, it divides each dominant capture group G evenly into w subsets and constructs for each such subset $\hat{G}_i$ the work unit $(\hat{G}_i, G)$, whereby $\hat{G}_i$ assigns dependent captures to consider during the upcoming CIND candidate generation. For instance, the work units for G'_1 are $(\{c_a, c_b\}, \{c_a, c_b, c_c, c_d\})$ and $(\{c_c, c_d\}, \{c_a, c_b, c_c, c_d\})$.

However, most of the CIND candidates enumerated by dominant capture groups are rather incidental and do not yield valid CINDs. As stated above, dominant capture groups emerge from frequent RDF-specific values, such as rdf:type. If two entities e_1 and e_2 occur with rdf:type and, thus, $(s, p{=}e_1)$ and $(s, p{=}e_2)$ are both in the capture group that corresponds to rdf:type, it is still unlikely that $(p, s{=}e_1) \subseteq (p, s{=}e_2)$ is a valid CIND. Our system exploits this observation in an approximate-validate CIND extraction approach that avoids creating a large number of these unnecessary CIND candidates. Each worker creates all CIND candidate sets for each of its capture groups and work units as discussed earlier (see Example 3.6). For the work units, which emerge from dominant capture groups, however, it encodes the referenced captures in a Bloom filter of constant size k instead (Step (7)). This encoding reduces the space complexity of the CIND candidate sets from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \cdot k) = \mathcal{O}(n)$, where n is the number of captures in a work unit. We experimentally observed that $k = 64$ bytes yields the best performance.

In Example 3.6, our system creates CIND candidate sets as discussed before for G_2 and G_3. In contrast, for the work units derived from G'_1, it creates CIND candidate sets as follows: $(c_a \sqsubseteq \mathit{Bloom}(c_b, c_c, c_d), 1)^*$, where the mark $*$ indicates that this candidate comes from a dominant capture group and is *approximate*. This mark allows our system to trace back all Bloom-filter-based CIND candidates for further validation as Bloom filters may generate false positives.

As in the basic extraction, our system then aggregates all the CIND candidate sets with the same dependent capture for validation (Step (8)). Algorithm 3.3 shows this aggregation process, where Boolean functions f_1, f_2, and f_3 mark whether a candidate is approximate or not. Conceptually, the workers intersect the referenced captures, but distinguish three cases: (i) If none of the two CIND candidate sets are approximate, they are intersected as in the basic extraction (Lines 1 & 2 in Algorithm 3.3). As a result, the system obtains *certain* CINDs (i. e., that do not require further validation). (ii) If both candidates are approximate, we calculate the bitwise `AND` of the Bloom filters to approximate the intersection of their elements (Lines 3 & 4). (iii) If only one of the candidates is approximate, we probe the other candidate against the Bloom filter and retain them on probing success (Lines 5–9). For instance, if we need to merge the above mentioned CIND candidate sets $(c_a \sqsubseteq \mathit{Bloom}(c_b, c_c, c_d), 1)^*$ and $(c_a \sqsubseteq \{c_b\}, 1)$, the result will be $(c_a \sqsubseteq \{c_b\}, 2)^*$.[8] Such CIND sets that have an approximate CIND candidate set lineage are *uncertain* and require further validation (unless the set of referenced captures is empty (Line 10)).

Algorithm 3.3: CIND candidates validation

```
   Data: CIND candidate set $(c \sqsubseteq C_1, count_1)^{f_1}$, $(c \sqsubseteq C_2, count_2)^{f_2}$
   Result: merged CIND candidate set $(c \sqsubseteq C_3, count_3)^{f_3}$
1  if $\neg\, hasBloomFilter(C_1) \wedge \neg\, hasBloomFilter(C_2)$ then
2      $C_3 \leftarrow C_1 \cap C_2$;
3  else if $hasBloomFilter(C_1) \wedge hasBloomFilter(C_2)$ then
4      $C_3 \leftarrow C_1$ AND $C_2$;
5  else
6      $C'_1 \leftarrow C_i$ where $\neg isBloomFilter(C_i)$;
7      $C'_2 \leftarrow C_j$ where $isBloomFilter(C_j)$;
8      $C_3 \leftarrow \{c \in C'_1 \mid c \in C'_2\}$;
9  $count_3 \leftarrow count_1 + count_2$;
10 $f_3 \leftarrow (f_1 \wedge f_2) \vee isEmpty(C_3)$;
```

To validate these uncertain CINDs, RDFind broadcasts the uncertain CIND sets to each worker (Step (9)) that organize them in a map m with the dependent capture as key and referenced captures as value. Then, the workers iterate through their local work units. If a dependent capture c in a work unit is a key in m, the worker intersects the captures of the work unit with the referenced captures in $m[c]$ and issues the result as a new CIND validation set. For instance for the work unit $(\{c_a, c_b\}, G'_1)$, the respective worker finds that c_a is a key in m and creates the validation set $c_a \sqsubseteq (G'_1 \cap m[c_a])$, i. e., $c_a \sqsubseteq \{c_b\}$. Notice that the validation sets are in general much smaller in number and extent than the above explained CIND candidate sets. Finally, these validation sets are intersected as

[8]Note that in general, the result might be further aggregated with CIND candidate sets, e. g., due to early aggregation.

in the basic extraction (Step (10)) and the resulting CINDs complement the previously found certain CINDs, denoting then the complete set of broad CINDs (Step (11)).

3.5.3 From Broad to Pertinent CINDs

Once all broad CINDs are extracted, RDFIND obtains the pertinent CINDs by retaining only the minimal ones. Recall that a minimal CIND must not be implied by a further CIND: neither by dependent nor referenced implication (see Definition 3.5). RDFIND basically detects all implications among broad CINDs in two steps. First, it removes non-minimal CINDs with a binary dependent and a unary referenced condition ($\Psi_{2:1}$) by *consolidating* them with CINDs that either have only unary or only binary conditions ($\Psi_{1:1}$ and $\Psi_{2:2}$). Then, in a second step, it removes non-minimal CINDs from the latter two by consolidating them with CINDs that have a unary dependent condition and a binary referenced condition ($\Psi_{1:2}$).

Let us clarify the above described consolidation process by example for the CIND $\psi = (s, p{=}\mathsf{memberOf}) \subseteq (s, p{=}\mathsf{rdf{:}type}) \in \Psi_{1:1}$ from Figure 3.1. Because CINDs in $\Psi_{1:1}$ might be subject to dependent implication with CINDs in $\Psi_{1:2}$, RDFIND joins $\Psi_{1:1}$ and $\Psi_{1:2}$ on their dependent captures and referenced project attribute. In our example, this join matches ψ with $\psi' = (s, p{=}\mathsf{memberOf}) \subseteq (s, p{=}\mathsf{rdf{:}type} \wedge o{=}\mathsf{gradStudent})$. RDFIND then finds that ψ' implies ψ and discards ψ. Altogether, RDFIND needs to process four such join operations; one for each type of implication, i. e., one for each edge in Figure 3.3 on page 54.

3.6 Experiments

We implemented RDFIND on top of Apache Flink 0.9.0 using Scala 2.10 and exhaustively evaluate it using both real-world and synthetic datasets. We conducted our experiments with five questions in mind: (i) *How good is* RDFIND *compared to the state-of-the-art?* (ii) *How well does* RDFIND *scale?* (iii) *How well does* RDFIND *behave under different support thresholds?* (iv) *What efficiency do our pruning techniques have?* (v) *Can we start from broad* CINDs *to generate pertinent* CIND *candidates?* We provide the implementation, pointers to the datasets, and the exact measurements for repeatability purposes at `https://hpi.de//naumann/projects/repeatability/data-profiling/cind-discovery-on-rdf-data.html`.

3.6.1 Experimental Setup

To gain comprehensive insights into RDFIND and its results, we gathered a broad range of datasets from different domains and of different sizes: seven real-world datasets and a synthetic one, summarized in Table 3.1.

We compare RDFIND to CINDERELLA [Bauckmann et al., 2012], the state-of-the-art CIND discovery algorithm for relational data. CINDERELLA assumes that partial INDs were previously discovered. It basically performs left-outer joins on these partial

Table 3.1: Evaluation RDF datasets.

Name	Size [MB]	Triples
Countries	0.8	5,563
Diseasome	13	72,445
LUBM-1	17	103,104
DrugBank	102	517,023
LinkedMDB	870	6,148,121
DB14-MPCE	4,334	33,329,233
DB14-PLE	21,770	152,913,360
Freebase	398,100	3,000,673,968

INDs using a database to generate conditions that match only the included tuples of the partial IND. We used both MySQL 5.6 and PostgreSQL 9.3 with default settings as underlying database. Additionally, we devised an optimized version CINDERELLA* that performs more memory-efficient joins and avoids self-joins, allowing it to significantly reduce its memory footprint. Notice that we do not compare RDFIND to the PLI-variant [Bauckmann et al., 2012], because CINDERELLA is shown to be faster. We also do not compare RDFIND to *Data Auditor* [Golab et al., 2011], because it discovers only the broadest CIND for a partial IND, which is not appropriate for the RDF case. However, PLI and *Data Auditor* apply the same overall strategy as CINDERELLA and differ only in the generation of conditions.

We conducted all experiments on a commodity hardware cluster consisting of a master node (Dell PowerEdge R310 (Intel Xeon X3450 (4× 2.66 GHz, 8 MB Smart Cache), 8 GB RAM, Ubuntu 14.04.5) and 10 worker nodes (Dell OptiPlex 780 with an Intel Core 2 Duo (2× 2.6 GHz, 2 MB L2 cache), 8 GB RAM, Ubuntu 14.04.5). All nodes are interconnected via Gigabit Ethernet in a star topology. Furthermore, our prototype reads RDF datasets from NTriple files distributed in HDFS. In our experiments, Flink (and hence RDFIND) was granted 4 GB of RAM on each worker node, leaving the remaining RAM to other components, such as HDFS.

3.6.2 Comparison of Cinderella and RDFind

We first show that RDFIND significantly prevails over the state-of-the-art when discovering CINDs in RDF data. For this purpose, we compare the runtimes of RDFIND with CINDERELLA [Bauckmann et al., 2012] on different datasets. As CINDERELLA is not a distributed algorithm, we ran our experiments on only the master node of our cluster, granting the algorithms 4 GB of main memory. Furthermore, due to CINDERELLA's high main memory requirements, we use only our two smallest datasets, Countries and Diseasome.

Figure 3.10 shows the results of this comparison with CINDERELLA. On the very small Countries dataset, RDFIND consistently outperforms the standard CINDERELLA by a factor from 8 to 39. However, the optimized version on PostgreSQL is up to 20 seconds faster, because RDFIND's runtime is dominated by a fix overhead, particularly Flink's

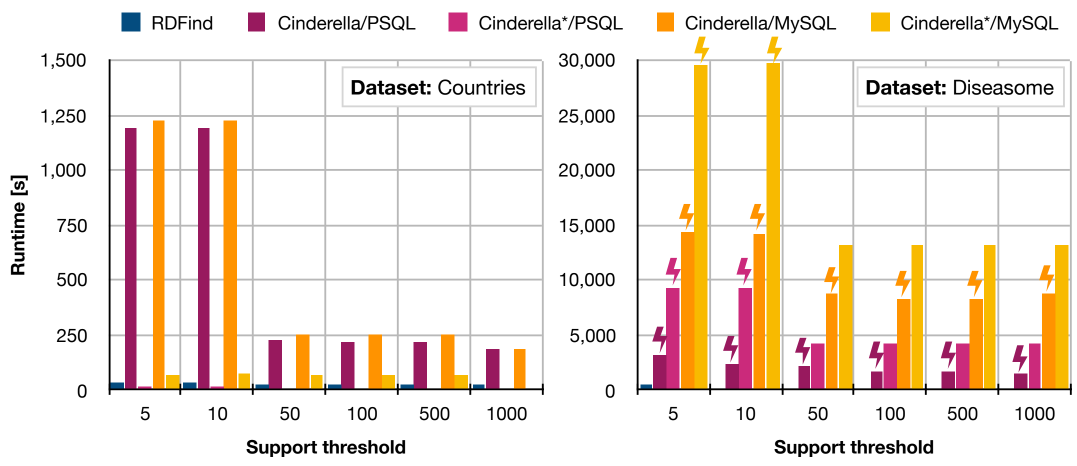


Figure 3.10: RDFind vs. standard and optimized Cinderella on MySQL and PostgreSQL (PSQL). Flash symbols indicate algorithm failures and therefore present lower bounds on the execution time.

start-up costs. For the larger (but still small) Diseasome dataset this overhead is already redeemed, though. We observe that the regular version of Cinderella failed for each execution and the optimized version for the support thresholds 5 and 10 due to their high memory consumption. Our system, in contrast, handles all executions flawlessly and outperforms Cinderella by a factor of up to 419 without considering the failed runs. This is mainly because, in contrast to Cinderella, which performs a join for each designated combination of projection attributes using a database (which also explains the differences among PostgreSQL and MySQL), our system covers the complete cind search space in a single execution using optimized data structures and algorithms. Note that in contrast to RDFind, Cinderella does not consider referenced conditions, which is a strong simplification of the cind discovery problem. This increased generality, the higher efficiency, and the robustness w. r. t. main memory render RDFind superior to Cinderella on rdf data. Therefore, we henceforth focus on evaluating only our system for bigger datasets that Cinderella cannot (efficiently) handle.

3.6.3 Scalability

We proceed to study the scalability of RDFind in terms of both the number of input triples and worker computers. At first, we analyze RDFind's robustness by evaluating its efficiency when varying the number of input triples. For this experiment, we consider the Freebase dataset, which is among the largest rdf datasets: With 3 billion triples and a total size of 400 GB, it exceeds the amount of available memory in our cluster by a factor of 10. We run our system over different sample sizes of the Freebase dataset using a support threshold of 1,000. Furthermore, we consider predicates only in conditions, because the above experiments rarely showed meaningful cinds on predicates.

Figure 3.11 illustrates the runtime of RDFind and the number of cinds and ars discovered for different numbers of input triples. We observe a slightly quadratic runtime behavior of our system. This is because the size of capture groups increases along with the number of input triples, and the CIND extraction runtime grows quadratically with the capture group sizes. Nevertheless, RDFind can process the full dataset, which demonstrates its high scalability, discovering more than 1 million pertinent CINDs. Indeed, including more triples leads to more pertinent CINDs. In contrast, the number of association rules (ARs) grows to a peak at 1 billion triples and declines afterwards. ARs have stricter semantics than CINDs and hence they are more easily violated by adding triples. Although this impairs the effectiveness of our pruning with ARs, overall the system shows to scale well with the number of input triples.

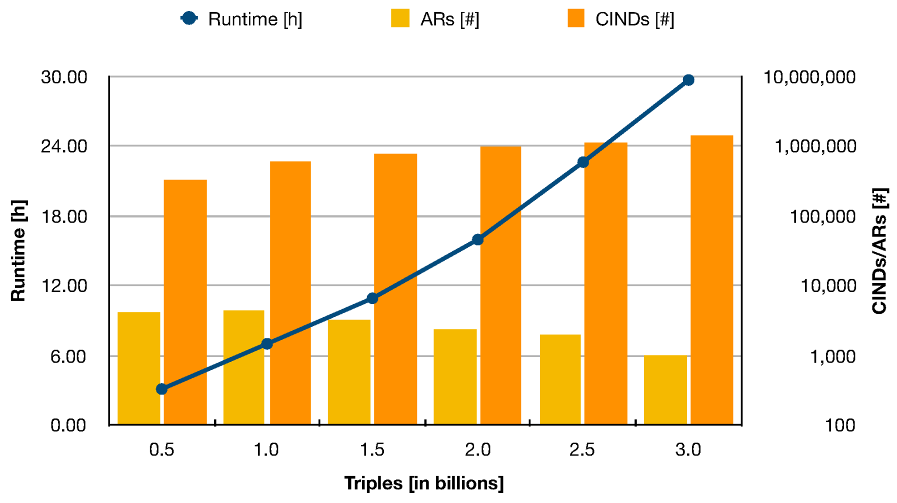


Figure 3.11: RDFind when increasing the number of input triples with a support threshold of 1,000.

Next, we evaluate the scalability of our system when increasing the number of worker computers. We consider the medium-size LinkedMDB dataset with a varying support threshold $s_{\min}$ and number of machines, each running a single thread. As Flink allows for parallelism in a single node through multi-threading, we consider an additional case with 10 workers, each running two threads. Figure 3.12 shows the measured runtimes and the average speed-up w. r. t. a parallelism of 1. We observe that our system scales almost linearly with the number of machines. In particular, when the support threshold is low and very few capture groups dominate the runtime, the load balancing ensures high resource utilization. On average, we measured a speed-up of 8.14 on 10 machines. We also observe that the intra-node parallelism allows RDFind to gain an additional speed-up of 1.38 on average. This shows that our system scales both with the number of machines as well as with the number of cores per machine.

3.6.4 Impact of the support threshold

Having shown RDFind's scalability, we now focus on evaluating the efficiency of our system when the number of pertinent CINDs increases. For that purpose, we run RDFind on all our datasets with varying support thresholds. Concretely, let us first focus on how

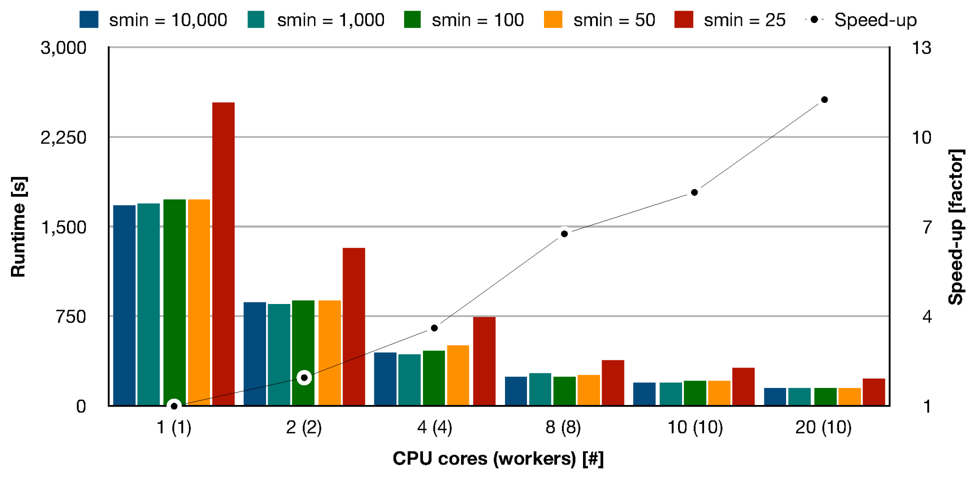


Figure 3.12: RDFind when increasing the number of machines on LinkedMDB with varying support threshold s_{min}.

the support threshold impacts RDFind's runtime. Figure 3.13 shows the runtimes for our system when discovering pertinent CINDs on multiple datasets and different support thresholds s_{min} and reveals a pattern on all datasets: For large support thresholds, RDFind is almost indifferent to the threshold setting and provides almost a constant runtime. For instance, for LUBM1, this is the case for $h \geq 10$. In contrast, the runtime quickly rises when decreasing s_{min} below 10. The reason for this is the distribution of conditions w. r. t. their frequencies: As shown in Figure 3.6, most conditions in datasets hold on only very few triples. Thus, for very small support thresholds our system can prune only few conditions, which leads to more captures and, more importantly, to larger capture groups. This agrees with our observation that the CIND extraction becomes the dominating component for small support thresholds because of its quadratic complexity w. r. t. capture group sizes. However, as we show in the following experiments, low-support CINDs are also useful for specific applications.

Besides the runtime, we also evaluate how the support threshold impacts the result of our system. Figure 3.14 displays the complete number of pertinent CINDs (including ARs) that RDFind discovers for multiple support thresholds. We observe that the number of pertinent CINDs is to some extent inversely proportional to the support threshold. Decreasing the support threshold by two orders of magnitude increases the number of CINDs by three orders of magnitude in the evaluation datasets (the ARs behave similarly and usually account for 10–50% of the CINDs). In consequence, the majority of CINDs in RDF datasets have a small support, while there are only a few broad CINDs, i. e., only few CINDs are supported by a large number of triples.

Still, these very broad CINDs are of high importance as they state general properties of their respective dataset. For example, we found that the DBpedia dataset (DB14-MPCE) embeds the two CINDs $(o, p{=}\text{associatedBand}) \subseteq (o, p{=}\text{associatedMusicalArtist})$ and $(s, p{=}\text{associatedBand}) \subseteq (s, p{=}\text{associatedMusicalArtist})$ with supports of 41,300 and 33,296, respectively. This suggests that the associatedBand predicate is a subproperty of associatedMusicalArtist and hence it is a hint to revise the ontology. On the other side, low-

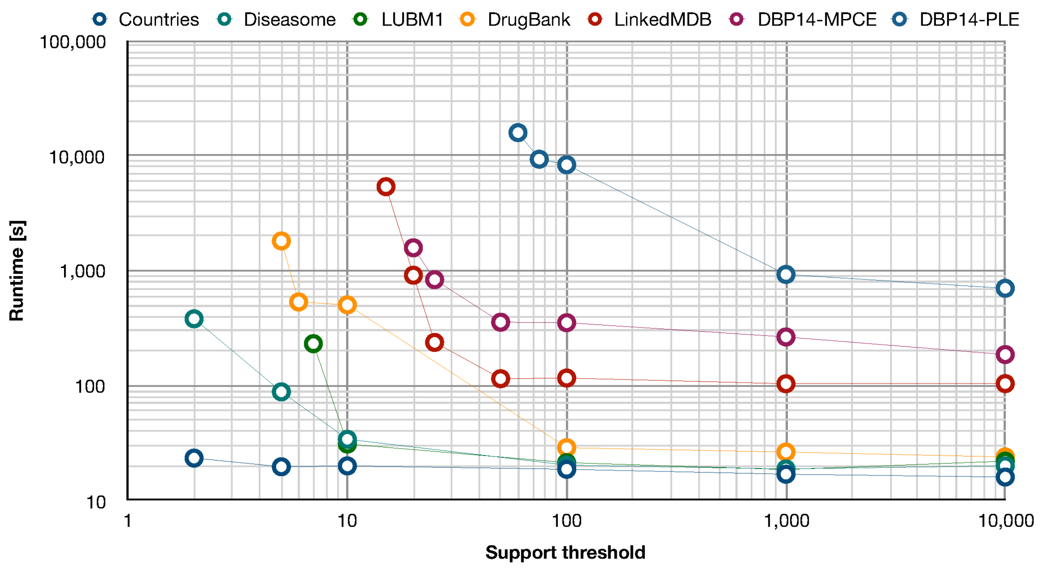


Figure 3.13: RDFind with different datasets and support thresholds.

support CINDs are also useful to some applications. For instance, DBpedia holds the CINDs $(s, p{=}\text{writer} \wedge o{=}\text{Angus\_Young}) \subseteq (s, p{=}\text{writer} \wedge o{=}\text{Malcolm\_Young})$ and, vice versa, $(s, p{=}\text{writer} \wedge o{=}\text{Malcolm\_Young}) \subseteq (s, p{=}\text{writer} \wedge o{=}\text{Angus\_Young})$, both having a support of 26. This reveals that the AC/DC members Angus and Malcolm Young have written all their songs together: a new fact, that is not explicitly stated in DBpedia. These results demonstrate the relation between CIND support and CIND semantics and justify why users should set different support thresholds for different use cases.

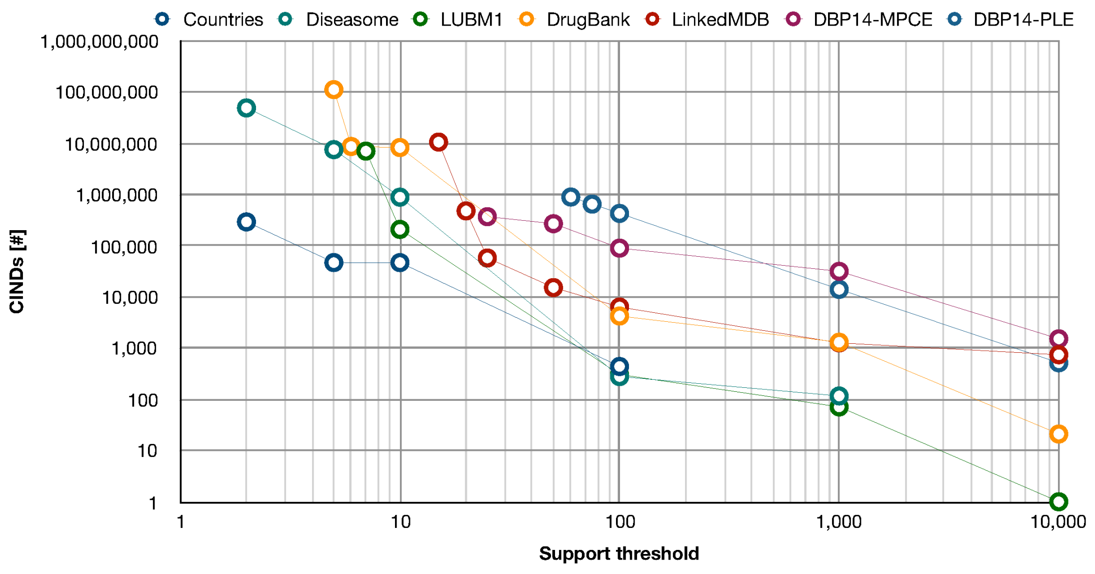


Figure 3.14: Number of pertinent CINDs for different datasets and supports.

3.6.5 Pruning Effectiveness

To investigate the effectiveness of our pruning techniques and algorithmic choices, we compare RDFIND with two simplified versions: RDFIND-*DE* (Direct Extraction) extracts CINDs from capture groups without the capture-support pruning, load balancing, and approximate-validate extraction (see Section 3.5.1). RDFIND-*NF* (No Frequent Conditions) additionally waives any operation related to frequent conditions (see Section 3.3).

Figure 3.15 compares the three variants on our two smallest datasets. While RDFIND and RDFIND-DE are similarly efficient, RDFIND-NF exhibits drastically inferior performance in all measurements. We conclude that the pruning techniques related to frequent conditions are highly effective, even for small datasets and low support thresholds.

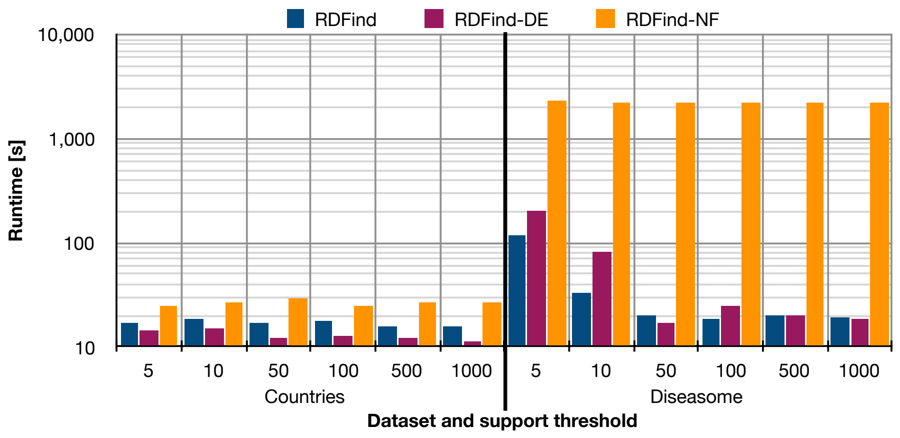


Figure 3.15: Runtimes of RDFIND, RDFIND-DE, and RDFIND-NF for various support thresholds.

We proceed to compare only RDFIND and RDFIND-DE on larger datasets in Figure 3.16. For the large support thresholds, RDFIND-DE is slightly more efficient than RDFIND in four of five cases. In these scenarios, the overhead for coping with dominant capture groups in RDFIND is not redeemed. Still, RDFIND's accumulated runtime is 4.6 minutes shorter than RDFIND-DE's runtime. For the small support thresholds, RDFIND is much more efficient and robust. On the three smaller dataset, it achieves an average speed-up of 5.7 over RDFIND-DE. Because small support thresholds entail long runtimes, this is an absolute speed-up of 50.4 minutes. Furthermore, RDFIND-DE was not able to handle the larger DB14-MPCE and DB14-PLE due to main memory requirements. These results show the improved robustness, overall higher efficiency, and the low overhead of cracking dominant capture groups of RDFIND.

3.6.6 Why not minimal CINDs first?

Recall that RDFIND finds all broad CINDs at first and then removes all non-minimal ones. It is an intriguing idea to discover only minimal CINDs at first by employing referenced and dependent implication as explained in Section 3.2.1. We implemented this idea by

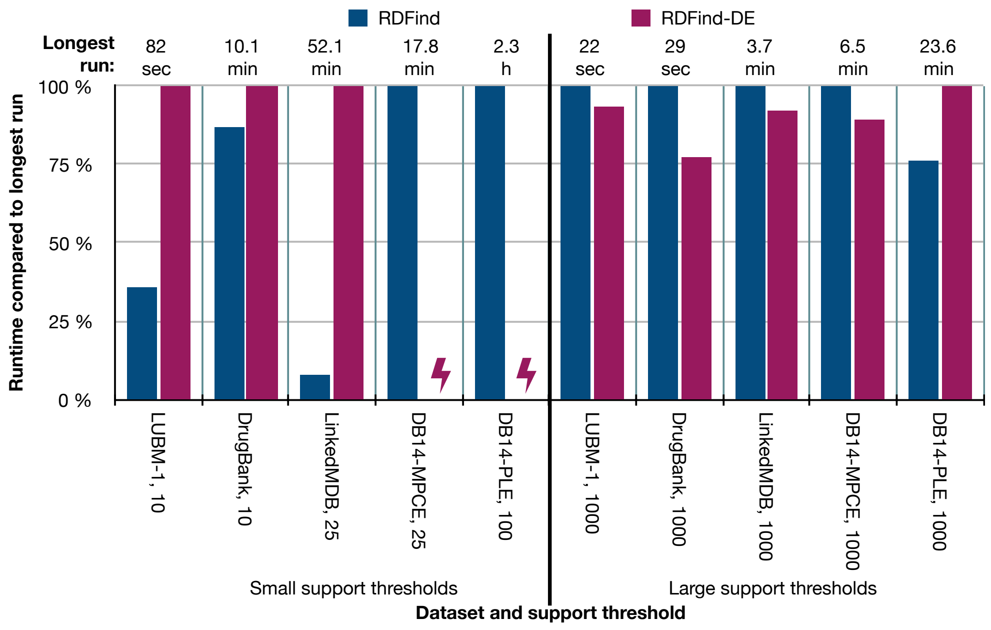


Figure 3.16: Runtimes RDFind and RDFind-DE for small and large supports. The longest run refers to the execution time of the slower algorithm, respectively. The flash symbols mark algorithm failures.

doing multiple passes over the capture groups, extracting only certain kinds of CINDs in each pass, and generating a reduced candidate set for the next pass. This strategy turned out to be up to 3 times slower even than RDFind-DE. Usually, broader CINDs are also minimal CINDs as exemplified in Figure 3.4, so the overhead of this strategy is greater than its savings. This clearly shows the efficiency of the strategy we follow in RDFind.

3.7 Related Work

As we have pointed out throughout this chapter, RDFind incorporates and extends concepts of Sindy, our algorithm for unary IND discovery on relational data. As such, RDFind is also to some extent related to other relational IND discovery algorithms. At this point, we omit a detailed discussion of works in that area, though, as we have thoroughly surveyed those in Section 2.1. Instead, this section focuses particularly on (i) RDF profiling in general, (ii) CIND discovery algorithms, and (iii) on only a small selection of relevant IND algorithms.

RDF profiling

Given the recent popularity of the LOD initiative [Bizer et al., 2009], a variety of systems for analyzing and profiling RDF data have been devised. Most of them focus on either RDFS/OWL schema discovery [Khatchadourian and Consens, 2010; Li, 2012] or on gathering statistics of Linked Data [Auer et al., 2012; Käfer et al., 2013]. ProLOD++, in contrast, comprises a wider variety of data profiling and mining tasks for RDF data, such as schema discovery and key discovery [Abedjan et al., 2014a]. However, none of these systems tackles the problem of discovering CINDs.

In a more closely related line of research, Abedjan and Naumann [2013] propose to profile *association rules* on RDF data, which can be used for schema analysis and clustering, amongst others. Those association rules are different from our ARs[9], though: While our ARs consider the RDF triples as transactions, the association rules are mined on transactions that are formed by (i) grouping the triples on some RDF attribute (subject, predicate, or object) and (ii) projecting each triple group on some other RDF attribute. For example, they can group the data from Figure 3.1 by their subject and project the groups to the predicate; then they obtain the transactions {rdf:type, memberOf, undergradDegreeFrom} (for patrick), {rdf:type, memberOf, undergradDegreeFrom} (for mike), {rdf:type} (for john), and {undergradDegreeFrom} (for tim).

As a matter of fact, the association rules are closely related to CINDs. As an example, the above transactions admit the association rule memberOf $\rightarrow$ undergradDegreeFrom, which corresponds to the CIND $(s, p{=}\text{memberOf}) \subseteq (s, p{=}\text{undergradDegreeFrom})$. Therefore, the use cases for association rules are also applicable to CINDs to a certain extent. However, neither of the two concepts, association rules and CINDs, dominates the other. While association rules allow for n-ary antecedants and admit non-exact rules, CINDs are much more expressive in the choice of their projection and condition attributes. For instance, the CIND $(s, p{=}\text{rdf:type} \wedge o{=}\text{gradStudent}) \subseteq (s, p{=}\text{undergradDegreeFrom})$ from Figure 3.1 cannot be expressed as an association rule.

As far as the discovery is concerned, Abedjan and Naumann [2013] employ the well-known FP-growth algorithm to explore association rules in RDF data [Han et al., 2000]. In contrast, for the discovery of ARs RDFind builds upon a distributed version of the Apriori algorithm [Agrawal and Srikant, 1994], that is made robust through use of Bloom filters. Furthermore, RDFind's overall strategy borrows its algorithmic foundations from IND discovery (especially from the Sindy algorithm) and is therefore different from the work of Abedjan and Naumann [2013].

CIND discovery

While much research has been conducted regarding the discovery of data dependencies, especially on relational data, the subarea of conditional dependency discovery has received relatively little attention [Abedjan et al., 2015]. Initially, Bohannon et al. [2007] proposed to amend functional dependencies with conditions to form data cleaning rules; Bravo et al. [2007] transferred this idea to INDs, thereby formalizing CINDs for relational data and studying their theoretical properties. Subsequently, also the discovery problem for CINDs has been tackled – at least for relational data. We are aware of only two main lines of work in this area, though. A reason for this scarcity of research might be the high computational complexity of CIND discovery.

[9]To avoid confusion, we refer to our notion of association rules exclusively as ARs in this discussion.

Bauckmann et al. [2012] propose two algorithms, namely Cinderella and Pli, both of which adopt the same basic approach: They accept a partial IND[10] that has been discovered by a partial IND discovery algorithm and then determine conditions that exclude such tuples that violate the given partial IND. This proceeding vastly differs from RDFind: At first, RDFind directly discovers all CINDs in a dataset, while Cinderella and Pli depend on a partial IND discovery algorithm and have to be applied to every discovered partial IND individually. Furthermore, these two algorithms specify conditions only on the dependent/LHS table of any given IND. As a result, the discovered CINDs might not be minimal in the sense of Section 3.2.1.

Golab et al. [2011] also pursue this approach of decorating a given partial IND with conditions, but with a twist: their system Data Auditor seeks to determine the *most concise* condition to restrict the partial IND to a CIND. In doing so, Data Auditor also supports disjunctions of conjunctive queries. However, because this system too restricts only the dependent/LHS tables, such CINDs can be split into sets of CINDs with purely conjunctive conditions. In other words, the CINDs that RDFind discovers can be assembled to CINDs with conjunctions. For instance, the dataset in Figure 3.1 satisfies the CINDs $(s, p{=}\mathsf{rdf{:}type} \wedge o{=}\mathsf{gradStudent}) \subseteq (s, p{=}\mathsf{undergradDegreeFrom})$ and $(s, p{=}\mathsf{memberOf}) \subseteq (s, p{=}\mathsf{undergradDegreeFrom})$, which could be assembled to $(s, (p{=}\mathsf{rdf{:}type} \wedge o{=}\mathsf{gradStudent}) \vee p{=}\mathsf{memberOf}) \subseteq (s, p{=}\mathsf{undergradDegreeFrom})$.

IND discovery

RDFind creates capture groups (see Section 3.4) and extracts CINDs from them (see Section 3.5). On a superficial level, this is RDFind's equivalent of Sindy's attribute set creation (see Section 2.2.2) and its subsequent IND extraction (see Section 2.2.3), which in turn is the algorithmic foundation of many other IND discovery algorithms. A particularly notable algorithm among those is the recent Many algorithm [Tschirschnitz et al., 2017].

Many was conceived specifically for the IND discovery among web tables. While "traditional" relational databases usually contain only thousands of columns (cf. Section 2.5) – perhaps sometimes tens of thousands – Tschirschnitz et al. [2017] consider hundreds of thousands of columns. The resulting challenge is the huge number of IND candidates, which grows quadratically in the number of columns. Many makes use of the fact that the vast majority of IND candidates do not form actual INDs. To this end, it creates a Bloom filter-based signature matrix for the input data and then performs bitwise `AND` operations on this matrix to quickly falsify the majority of non-INDs.

RDFind faces more or less the same challenge, especially on low support thresholds, because the number of CIND candidates grows quadratically in the number of unpruned captures. However, RDFind tackles this problem with different means: (i) Its lazy pruning strategy successively prunes the number of captures; (ii) it detects dominant capture groups and load-balances their processing; (iii) and for CIND candidates from dominant captures group, RDFind applies a two-step CIND extraction process that approximates

[10]Recall from Section 2.3 that an IND $A \subseteq B$ is partial whenever the set of values in column A are almost a subset of the set of values in column B.

CIND first and only then verifies them. In contrast to MANY, RDFIND can further be distributed on computer clusters.

3.8 Summary

This chapter introduced the novel concept of pertinent CINDs on RDF datasets and presented the RDFIND system for their discovery. In contrast to existing CIND algorithms, which find partial INDs at first and then generate conditions for each partial IND individually, RDFIND discovers all CINDs in a single run employing efficient pruning techniques. We showed experimentally that our algorithm outperforms the state-of-the-art algorithm CINDERELLA by orders of magnitude (while considering a more general class of CINDs) and is robust and scalable enough to handle large RDF datasets that were not possible to handle before.

For the future, it would be helpful to (inter-)actively aid users in determining an appropriate support threshold to find the *relevant* CINDs for their applications. Also, discerning *meaningful* and *spurious* CINDs, e. g., using the local closed world assumption, is an interesting aspect to investigate [Dong et al., 2014]. In any case, we have enabled new research to incorporate CINDs in many RDF data management scenarios, e. g., data integration, ontology re-engineering, knowledge extraction, and query optimization. Finally, it would be intriguing to propagate RDFIND's novel algorithmic strategies to SINDY to allow it to scale to greater numbers of columns.

Chapter 4

Discovery of Partial Functional Dependencies and Partial Unique Column Combinations

The two previous chapters deal with the discovery of INDs on relational data and CINDs on RDF data. As their names indicate, both dependency types are closely related and the solutions we propose for their discovery share some algorithmic ideas. This chapter returns to the relational terrain, but it deals with profoundly different, yet fundamental, important types of dependencies and their discovery, namely *partial*[1] *unique column combinations* (PUCCs) and *partial functional dependencies* (PFDs). In fact, INDs belong into the class of tuple-generating dependencies, while (exact) UCCs and FDs are instances of the class of equality-generating dependencies [Abiteboul et al., 1995]. In consequence, PUCCs and PFDs have to be discovered with strategies that are completely distinct from those presented in the previous chapters.

In this chapter, we present PYRO [Kruse and Naumann, 2018], a unified algorithm to discover both PUCCs and PFDs. At its core, PYRO uses a separate-and-conquer strategy that uses sampling to quickly discover promising dependency candidates and then applies a strategy that reduces the effort to validate those candidates. However, before explaining our algorithm, we motivate the need for the efficient discovery of PUCCs and PFDs in Section 4.1 and discuss related work in Section 4.2. Then, Section 4.3 gives an overview of how PYRO operates, followed by detailed descriptions of the algorithm's components: Section 4.4 lays out how PYRO estimates and calculates the error of PUCC and PFD candidates; Section 4.5 combines these operations in a separate-and-conquer search strategy; and Section 4.6 shows how PYRO can employ multiple cores and even multiple machines for its computation. Finally, Section 4.7 experimentally evaluates our algorithm and compares it to the state of the art, before we summarize this chapter in Section 4.8.

[1]Recall from Section 1.3 that partial dependencies allow for violations in the data.

4.1 The Exception Proves the Rule

Functional dependencies (FDs) and *unique column combinations* (UCCs) are among the most important dependencies for relational databases. Recall from Section 1.3 that an FD states that some attributes in a relational instance functionally determine the value of a further attribute. A UCC, in contrast, indicates that some columns uniquely identify every tuple in a relational instance. More formally, for a relation r with the schema R with attribute sets $X, Y \subseteq R$, we say that $X \rightarrow Y$ is an FD with *left-hand side (LHS)* X and *right-hand side (RHS)* Y if we have $t_1[X]{=}t_2[X] \Rightarrow t_1[Y]{=}t_2[Y]$ for all pairs of distinct tuples $t_1, t_2 \in r$. Likewise, we say that X is a UCC if $t_1[X] \neq t_2[X]$ for all such tuple pairs. For instance, in a table with address data, the country and ZIP code in an address might determine the city name and every address might be uniquely identified by its ZIP code, street, and house number.

The applications of FDs and UCCs are manifold, ranging from schema discovery [Kruse et al., 2016b] over data integration [Miller et al., 2001] to schema design [Köhler et al., 2015], normalization [Papenbrock and Naumann, 2017b], and query relaxation [Nambiar and Kambhampati, 2004]. However, in most scenarios neither the FDs nor the UCCs are known. To this end, various algorithms have been devised over the last decades to automatically discover these dependencies [Heise et al., 2013; Huhtala et al., 1999; Papenbrock and Naumann, 2017b; Sismanis et al., 2006].

That being said, most existing algorithms discover only *exact* dependencies, which are completely satisfied by the data – without even a single violation. Real-world dependencies are all too often not exact, though. Let us exemplify why that is the case with the help of Figure 4.1:
(1) Data errors: One might be interested to establish a primary key for the given data, and {First name, Last name} seems to form a reasonable candidate. However, it is not a UCC: tuple t_4 is a duplicate of t_1. For that matter, the table does not contain a single exact UCC.
(2) Exceptions: Most English first names determine a person's gender. There are exceptions, though. While Alex in tuple t_1 is male, Alex in t_5 is female. In consequence, the FD First name $\rightarrow$ Gender is violated.
(3) Ambiguities: In contrast to first names and genders, a ZIP code is defined to uniquely determine its city. Still, we find that t_3 violates ZIP $\rightarrow$ Town, because it specifies a district rather than the city.

	First name	Last name	Gender	ZIP	Town
t_1	Alex	Smith	m	55302	Brighton
t_2	John	Kramer	m	55301	Brighton
t_3	Lindsay	Miller	f	55301	Rapid Falls
t_4	Alex	Smith	m	55302	Brighton
t_5	Alex	Miller	f	55301	Brighton

Figure 4.1: Example table with person data.

These few examples illustrate why relevant data dependencies in real-world datasets are often not exact, so that most existing discovery algorithms fail to find them. To cope with this problem, the definition of exact dependencies can be relaxed to allow for a certain degree of violation (as described in Section 1.3 and applied for inclusion dependencies in Section 2.3 in the context of the algorithm SANDY). We refer to such relaxed UCCs and FDs as *partial* UCCs (PUCCs) and *partial* FDs (PFDs). PUCCs and PFDs can not only substitute their exact counterparts in many of the above mentioned use cases, but they also reveal data inconsistencies and thus form an essential input to data cleaning systems [Beskales et al., 2010; Geerts et al., 2013; Khayyat et al., 2015; Kolahi and Lakshmanan, 2009; Thirumuruganathan et al., 2017]. Furthermore, they can help to improve poor cardinality estimates of query optimizers by revealing correlating column sets [Ilyas et al., 2004; Leis et al., 2015]; and they can support feature selection for machine learning algorithms (especially for those assuming mutual independence of features, such as Naïve Bayes) by exposing dependent feature sets.

Before giving any algorithmic details, let us formalize the problem of finding the PFDs and PUCCs in a dataset. As we mention in Section 1.3, when referring to partial dependencies, we need to quantify the degree of "partiality". In this work, we use a slight adaptation of the established g_1 error [Kivinen and Mannila, 1992] that ignores reflexive tuple pairs, so that its values span the whole range from 0 (exact dependency) to 1 (completely violated dependency).

Definition 4.1 (PFD/PUCC error)**.** Given a dataset r and a PFD candidate $X \to A$, we define its error as

$$e(X \to A, r) = \frac{|\{(t_1, t_2) \in r^2 \mid t_1[X]{=}t_2[X] \wedge t_1[A]{\neq}t_2[A]\}|}{|r|^2 - |r|}$$

Analogously, the error of a PUCC candidate X is defined as

$$e(X, r) = \frac{|\{(t_1, t_2) \in r^2 \mid t_1 \neq t_2 \wedge t_1[X]{=}t_2[X]\}|}{|r|^2 - |r|}$$

Example 4.1. Intuitively, these errors quantify the ratio of tuple pairs in a dataset that violate the respective dependency. For the example from Figure 4.1, we can calculate $e(\textsf{First name} \to \textsf{Gender}, r) = \frac{4}{5^2-5} = 0.2$ (violated by (t_1, t_5), (t_4, t_5), and their inverses) and $e(\{\textsf{First name}, \textsf{Last name}\}, r) = \frac{2}{5^2-5} = 0.1$ (violated by (t_1, t_4) and its inverse).

These error measures lend themselves for PYRO for two reasons. First, and as we demonstrate in Section 4.4, they can be easily calculated from different data structures. This allows us to efficiently *estimate* and *calculate* errors. Second, and more importantly, the error measures are *monotonous*. That is, for any PFD $X \to Y$ and an additional attribute A we have $e(X \to Y) \geq e(XA \to Y)$. And analogously for a PUCC X, we have $e(X) \geq e(XA)$. In other words, adding an attribute to the LHS of a PFD or to a PUCC can only reduce the number of violating tuple pairs but never increase it. We refer to those $XA \to Y$ and XA, respectively, as *specializations* and to $X \to Y$ and X, respectively, as *generalizations*. With these observations, we can now precisely define our problem statement.

Problem statement 4.1. Given a relation r and error thresholds e_ϕ and e_υ, we want to determine all *minimal* PFDs with a single RHS attribute and all *minimal* PUCCs. A minimal PFD has a PFD error less than or equal to e_ϕ, while all its generalizations have a PFD error greater than e_ϕ. Analogously, a minimal PUCC has a PUCC error of at most e_υ, while all its generalizations have a PUCC error greater than e_υ.

Example 4.2. Assume we want to find all minimal PUCCs in Figure 4.1 with the error threshold of $e_\upsilon = 0.1$. Amongst others, $\upsilon_1 = \{\text{First name}, \text{Last name}\}$ and $\upsilon_2 = \{\text{First name}, \text{Last name}, \text{Gender}\}$ have a PUCC error of $0.1 \leq e_\upsilon$. However, υ_1 is a generalization of υ_2, so υ_2 is not minimal and need not be discovered explicitly.

Note that Problem statement 4.1 explicitly excludes PFDs with *composite* RHSs, i. e., with more than one attribute – let us explain why. For exact dependency discovery, this exclusion is sensible because the FD $X \to AB$ holds if and only if $X \to A$ and $X \to B$ hold [Armstrong, 1974]. For PFDs as defined in Definition 4.1, this is no longer the case. However, considering composite RHSs potentially increases the number of PFDs drastically and might have serious performance implications. Furthermore, it is not clear how the use cases mentioned above would benefit from such additional PFDs, or whether they would even be impaired by their huge number. Hence, we deliberately focus on single RHSs for pragmatic reasons and do so in accordance with related work [Flach and Savnik, 1999; Huhtala et al., 1999; Lopes et al., 2002a]. Nevertheless, it is easy to see that the error of a PFD $X \to AB$ in a relation instance r is at least $\max\{e(X \to A, r), e(X \to B, r)\}$: Every tuple pair that violates $X \to A$ or $X \to B$ also violates $X \to AB$, because its tuples agree in their X values but do not agree in their AB values. Hence, PFDs with a single RHS can be used to prune PFD candidates with composite RHSs.

4.2 Related Work

Dependency discovery has been studied extensively in the field of data profiling [Abedjan et al., 2015]. The efficient discovery of exact FDs has gained particular interest [Papenbrock et al., 2015b]. Further, many extensions and relaxations of FDs have been proposed [Caruccio et al., 2016], e. g., using similarity functions, aggregations, or multiple data sources. PYRO focuses on partial dependencies (also often referred to as *approximate dependencies* in literature; see Section 1.3) that may be violated by a certain portion of tuples or tuple pairs. Note that this is different from *dependency approximation algorithms* [Bleifuß et al., 2016; Kivinen and Mannila, 1995], which trade correctness guarantees of the discovered dependencies for performance improvements. In the following, we focus on those works that share goals or have technical commonalities with PYRO.

Approximate dependency discovery

While there are many works studying the discovery of FDs under various relaxations, only relatively few of them consider PFDs. To cope with the problem complexity, some discovery algorithms operate on samples of the profiled data and therefore cannot guarantee the correctness of their results [Ilyas et al., 2004; Kivinen and Mannila, 1995] (that is, they only approximate the partial FDs). This does not apply to PYRO. In addition, CORDS discovers only unary PFDs, which is a much easier problem, as there are only quadratically many unary but exponentially many n-ary PFD candidates) [Ilyas et al., 2004].

Another approach to harness the complexity is to use heuristics to prune potentially uninteresting PFD candidates [Sánchez et al., 2008]. Because the interestingness of a PFD depends on the designated use case, this can cause the loss of relevant results. Therefore, PYRO instead discovers *all* partial dependencies for some given error threshold and leaves filtering or ranking of the dependencies to use case specific post-processing. This prevents said loss and also frees users from the burden of selecting an appropriate interestingness threshold.

Along these lines, exact approaches for the discovery of PFDs and PUCCs have been devised. Arguably, the most adapted one is TANE [Huhtala et al., 1999], which converts the columns of a profiled relation into *stripped partitions (also: position list indices, PLIs)* and exhaustively combines them until it has discovered the minimal PFDs. Being mainly designed for exact FD and UCC discovery, some of TANE's pruning rules do not work in the partial case, leading to degraded performance. In fact, before discovering a partial dependency involving n columns, TANE tests $2^n - 2$ candidates corresponding to subsets of these columns. Note that many works build upon TANE without changing these foundations [Atoum, 2009; King and Legendre, 2003; Li et al., 2016]. PYRO avoids these problems by estimating the position of minimal partial dependencies and then immediately verifying them.

Further approaches to infer partial dependencies are based on the pairwise comparison of all tuples. The FDEP algorithm proposes (i) to compare all tuple pairs in a database, thereby counting any FD violation; (ii) to apply an error threshold to discard infrequent violations; and (iii) to deduce the PFDs from the residual violations [Flach and Savnik, 1999]. We found this algorithm to yield incorrect results, though: Unlike exact FDs, PFDs can be violated by *combinations* of tuple pair-based violations, which Step (ii) neglects. In consequence, FDEP is not aware of all dependency violations and infers incorrect results. In addition to that, the quadratic load of comparing all tuple pairs does not scale well to large relations [Papenbrock et al., 2015b]. In a related approach, Lopes et al. propose to use tuple pair comparisons to determine the most specific non-FDs in a given dataset whose error should then be calculated subsequently [Lopes et al., 2002a]. This approach is quite different from the aforementioned ones because it discovers only a small subset of all PFDs.

In a different line of work, an SQL-based algorithm for PFD discovery has been proposed [Matos and Grasser, 2004]. As stated by the authors themselves, the focus of that work lies on ease of implementation in practical scenarios rather than performance.

Last but not least, the discovery of dependencies under the presence of `NULL` values has been studied [Köhler et al., 2015]. The proposed algorithm reasons on replacements for the `NULL` values, such that exact dependencies emerge. This problem is very different from that of Pyro, which does not incorporate a special treatment of NULL but considers arbitrary dependency violations.

Exact dependency discovery

Many algorithms for the discovery of exact FDs and UCCs have been devised, e. g., [Heise et al., 2013; Huhtala et al., 1999; Papenbrock and Naumann, 2017b; Sismanis et al., 2006]. These algorithms can generally be divided into (i) those that are based on the pairwise comparisons of tuples and scale well with the number of attributes and (ii) those that are based on PLI intersection and scale well with the number of tuples [Papenbrock et al., 2015b].

The algorithms Ducc [Heise et al., 2013] and the derived Dfd [Abedjan et al., 2014b] belong to the latter and resemble Pyro in that they use a depth-first search space traversal strategy. Still, both exhibit substantial differences: While Ducc and Dfd perform a random walk through the search space, Pyro performs a sampling-based best-first search along with other techniques to reduce the number of dependency candidate tests. Interestingly, the authors of Dfd suggest that this algorithm could be modified to discover PFDs. We will therefore consider a modified version of this algorithm (in combination with the Ducc algorithm) in our evaluation.

The recent HyFD and HyUCC algorithms manage to scale well with growing numbers of tuples and columns by combining tuple comparisons and PLI intersections [Papenbrock and Naumann, 2016, 2017a]. Pyro also combines these two base techniques. However, HyFD and HyUCC aggressively prune FDs and UCCs, respectively, as soon as they discover a violation of the same. While this pruning is key to the algorithms' efficiency, it is not applicable to partial dependencies.[2] Instead, Pyro uses tuple comparisons to hypothesize dependency candidates rather than falsifying them and its search space traversal is adaptive rather than bottom-up. Another optimization in HyFD and HyUCC, that is not applicable in the partial case, is the verification of dependencies: These algorithms stop a PLI-based validation of a dependency as soon as a violation has been detected. In contrast, Pyro would have to detect many violations before stopping a dependency validation. And because Pyro checks only promising dependency candidates, such an optimization is not worthwhile.

Lattice search

In a broader sense, Pyro classifies nodes in a power set lattice, as we explain in the following section. Apart from dependency discovery, several other problems, such as frequent itemset mining [e. g., Agrawal and Srikant, 1994; Han et al., 2000], belong in this category and can be tackled with the same algorithmic foundations [Mannila and Toivonen, 1997]. For instance, PFD discovery can be modeled as an association rule

[2]HyFD's and HyUCC's pruning is based on the above described deduction step of Fdep, which deduces dependencies from non-dependencies. As explained, such deduction does not work for PUCCs and PFDs. This circumstance additionally prohibits the application of HyFD and HyUCC to PFD and PUCC discovery.

mining problem; however, such adaptations require additional tailoring to be practically usable [Sánchez et al., 2008].

4.3 Algorithm Overview

Let us now outline with the help of Algorithm 4.1 and Figure 4.2 how Pyro discovers all minimal PFDs and PUCCs for a given dataset and error thresholds e_ϕ and e_υ as defined in Problem statement 4.1. For simplicity (but without loss of generality), we assume $e_\phi = e_\upsilon = e_{\max}$ for some user-defined $e_{\max}$. Furthermore, we refer to PFD and PUCC candidates with an error $\leq e_{\max}$ as *dependencies* and otherwise as *non-dependencies*.

Now, given a dataset with n attributes, Pyro spawns $n+1$ *search spaces* (Line 1): one search space to discover the minimal PUCCs and one search space for each attribute to discover the minimal PFDs with that very attribute as RHS. A PUCC search space is a power set lattice of all attributes, where each attribute set directly forms a PUCC candidate. Similarly, a PFD search space is a power set lattice with all but the RHS attribute A, where each attribute set X represents the PFD candidate $X \to A$. In other words, each attribute set in the power set lattices forms a unique dependency candidate. As a convention, we can therefore use attribute sets and dependency candidates synonymously.

In a second preparatory step before the actual dependency discovery, Pyro builds up two auxiliary data structures (called *agree set sample (AS) cache* and *position list index (PLI) cache*; Lines 2–3), both of which support the discovery process for *all* search spaces by estimating or calculating the error of dependency candidates. We explain these data structures in Section 4.4.

Eventually, Pyro traverses each search space with a *separate-and-conquer* strategy to discover their minimal dependencies (Lines 4–5). Said strategy employs computationally inexpensive error estimates (via the AS cache) to quickly locate a promising minimal dependency candidate and then efficiently checks it with only few error calculations (via the PLI cache). As indicated in Figure 4.2, the verified (non-)dependencies are then used to prune considerable parts of the search space and as a result Pyro needs to inspect only the residual dependency candidates. Notice that our traversal strategy is sufficiently abstract to accommodate both PFD and PUCC discovery without any changes.

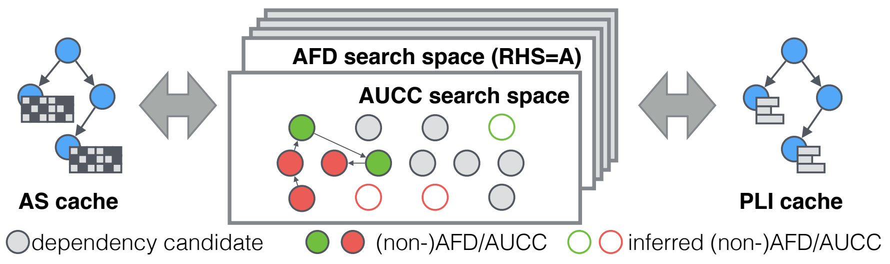


Figure 4.2: Intermediate state of Pyro while profiling a relation with the schema $R = (A, B, C, D)$.

4.4 Error Assessment

As Pyro traverses a search space, it needs to estimate and calculate the error of dependency candidates. This section explains the data structures and algorithms to perform both operations efficiently.

4.4.1 PLI Cache

As we shall see in the following, both the error estimation and calculation involve *position list indices (PLIs)* (also known as *stripped partitions* [Huhtala et al., 1999]):

Definition 4.2 (PLI)**.** Let r be a relation with schema R and let $X \subseteq R$ be a set of attributes. A *cluster* is a set of all tuple indices in r that have the same value for X, i. e.,$c(t) = \{i \mid t_i[X] = t[X]\}$. The PLI of X is the set of all such clusters except for singleton clusters:

$$\bar{\pi}(X) := \{c(t) \mid t \in r \wedge |c(t)| > 1\}$$

We further define the *size* of a PLI as the number of included tuple indices, i. e.,$\|\bar{\pi}(X)\| := \sum_{c \in \bar{\pi}(X)} |c|$.

Example 4.3. Consider the attribute Last name in Figure 4.1. Its associated PLI consists of the clusters $\{1, 4\}$ for the value *Smith* and $\{3, 5\}$ for the value *Miller*. The PLI does not include the singleton cluster for the value *Kramer*, though.

Pyro (and many related works, for that matter) employ PLIs for various reasons. First, and that is specific to Pyro, PLIs allow to create focused samples on the data, thereby enabling precise error estimates of dependency candidates. Second, PLIs have a low memory footprint because they store only tuple indices rather than actual values and omit singleton clusters completely. Third, the g_1 error can be directly calculated on them, as we show in the next section. Finally, $\bar{\pi}(XY)$ can be efficiently calculated from $\bar{\pi}(X)$ and $\bar{\pi}(Y)$ [Huhtala et al., 1999], denoted as *intersecting PLIs*. In consequence, we can represent any combination of attributes as a PLI.

Algorithm 4.1: Pyro's general workflow.

```
  Data: Relation schema $R$ with instance $r$, PFD error threshold $e_\phi$, PUCC error
        threshold $e_\upsilon$
  ▷ Section 4.3
1 search-spaces ← {create-pucc-space($R, e_\upsilon$)}∪
    $\bigcup_{A \in R}$ create-pfd-space($R \setminus \{A\}, A, e_\phi$)
  ▷ Section 4.4
2 pli-cache ← init-pli-cache($r$)
3 as-cache ← init-as-cache($r$, pli-cache)
  ▷ Section 4.5
4 foreach space ∈ search-spaces do
5     traverse(space, pli-cache, as-cache)
```

For clarity, let us briefly describe the PLI intersection. As an example, consider the data from Figure 4.1 and assume we want to intersect the PLIs $\bar{\pi}(\textsf{First name}) = \{\{1,4,5\}\}$ and $\bar{\pi}(\textsf{Last name}) = \{\{1,4\},\{2,5\}\}$. In the first step, we convert $\bar{\pi}(\textsf{Last name})$ into the *attribute vector* $v_{\textsf{Last name}} = (1,0,2,1,2)$, which simply is a dictionary-compressed array of the attribute Last name with one peculiarity: All values that appear only once are encoded as 0. This conversion is straight-forward: For each cluster in $\bar{\pi}(\textsf{Last name})$, we simply devise an arbitrary ID and write this ID into the positions contained in that cluster. In the second step, the *probing*, we group the tuple indices within each cluster of $\bar{\pi}(\textsf{First name})$. Concretely, the grouping key for the tuple index i is the i-th value in $v_{\textsf{Last name}}$ unless that value is 0: In that case the tuple index is dropped. For the cluster $\{1,4,5\}$, we obtain the groups $1 \to \{1,4\}$ and $2 \to \{5\}$. Eventually, all groups with a size greater than 1 form the clusters of the new PLI. In our example, we get $\bar{\pi}(\textsf{First name}, \textsf{Last name}) = \{\{1,4\}\}$. Because t_1 and t_4 are the only tuples in Figure 4.1 that agree in both First name and Last name, our calculated PLI indeed satisfies Definition 4.2.

That being said, intersecting PLIs is computationally expensive. Therefore, Pyro puts calculated PLIs into a *PLI cache* (cf. Figure 4.2) for later reuse. Caching PLIs has been proposed in context of the Ducc algorithm [Heise et al., 2013] (and was adopted by Dfd [Abedjan et al., 2014b]), however, a description of the caching data structure has not been given. It has been shown, however, that the set-trie of the Fdep algorithm [Flach and Savnik, 1999] is suitable to index and look up PLIs [Zwiener, 2015].

As exemplified in Figure 4.3, Pyro's PLI cache adopts a similar strategy: It is essentially a trie (also: prefix tree) that associates attribute sets to their respective cached PLI. Assume we have calculated $\bar{\pi}(\{C,E\})$. Then we convert this attribute set into the list (C,E), which orders the attributes according to their order in the relation schema. Then, we index $\bar{\pi}(\{C,E\})$ in the trie using (C,E) as key.

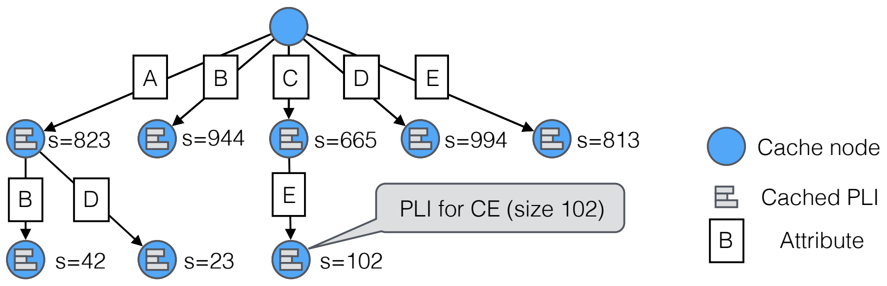


Figure 4.3: Example PLI cache.

However, when Pyro requests some PLI $\bar{\pi}(X)$, it may well not be in the cache. Still, we can leverage the cache by addressing the following criteria:
(1) *We want to obtain $\bar{\pi}(X)$ with only few PLI intersections.*
(2) *In every intersection $\bar{\pi}(Y) \cap \bar{\pi}(Z)$, where we probe $\bar{\pi}(Y)$ against v_Z, we would like $\|\bar{\pi}(Y)\|$ to be small.*

While Criterion 1 addresses the number of PLI intersections, Criterion 2 addresses the efficiency of the individual intersections, because probing few, small PLI clusters is beneficial performance-wise. Algorithm 4.2 considers both criteria to serve PLI re-

Algorithm 4.2: Retrieve a PLI from the PLI cache.

```
Data: PLI cache cache, attribute set $X$
1  $\Pi \leftarrow$ lookup PLIs for subsets of $X$ in cache
2  $\bar{\pi}(Y) \leftarrow$ pick the smallest PLI indices from $\Pi$
3  $\mathcal{Z} \leftarrow$ new list, $C \leftarrow Y$
4  while $C \subset X$ do
5      $\bar{\pi}(Z) \leftarrow$ pick PLI from $\Pi$ that maximizes $|Z \setminus C|$
6      append $\bar{\pi}(Z)$ to $\mathcal{Z}$
7      $C \leftarrow C \cup Z$
8  sort $\mathcal{Z}$ by the PLIs' sizes, $C \leftarrow Y$
9  foreach $\bar{\pi}(Z) \in \mathcal{Z}$ do
10     $\bar{\pi}(C \cup Z) \leftarrow \bar{\pi}(C) \cap \bar{\pi}(Z)$, $C \leftarrow C \cup Z$
11     if coin flip shows head then put $\bar{\pi}(C)$ into cache
12 return $\bar{\pi}(C)$
```

quests utilizing the PLI cache. As an example, assume we want to construct the PLI $\bar{\pi}(ABCDE)$ with the PLI cache from Figure 4.3. At first, we look up all PLIs for subsets of $ABCDE$ in the cache (Line 1). This look-up can be done efficiently in tries. Among the retrieved PLIs, we pick the one $\bar{\pi}(Y)$ with the smallest size (Line 2). In our example, this is the case for $\bar{\pi}(AD)$ with a size of 23. This smallest PLI shall be used for probing in the first PLI intersection. The resulting PLI, which cannot be larger in size, will then be used for the subsequent intersection's probing and so on. This satisfies Criterion 2.

Next, we need to determine the remaining PLIs to probe against. Here, we follow Criterion 1 and repeatedly pick whatever PLI provides the most new attributes to those in the already picked PLIs (Lines 3–7). In our example, we thus pick $\bar{\pi}(CE)$, which provides two new attributes, and then $\bar{\pi}(B)$. Finally, all attributes in $ABCDE$ appear in at least one of the three selected PLIs. Note that Pyro always maintains PLIs for the single attributes in the PLI cache and can therefore serve any PLI request.

Having selected the PLIs, we intersect them using small PLIs as early as possible due to Criterion 2 (Lines 8–10). For our example, this yields the intersection order $(\bar{\pi}(AD) \cap \bar{\pi}(CE)) \cap \bar{\pi}(B)$. Compared to intersecting PLIs of single attributes, we save two out of four intersection operations. Additionally, we can use the PLI $\bar{\pi}(AD)$, which is much smaller than any single-attribute PLI. Hence, the PLI cache is useful to address both Criteria 1 and 2.

Finally, we cache randomly selected PLIs (Line 11). We forego caching *all* calculated PLIs, because it quickly fills the cache with redundant PLIs or those that will not be needed again. Our random approach, in contrast, caches frequently needed PLIs with a higher probability – with virtually no overhead.

4.4.2 Evaluating Dependency Candidates

PLIs are vital to calculate the error of a PFD or PUCC, respectively. Having shown how to efficiently obtain the PLI for some attribute set X, let us show how to calculate the g_1 error (see Definition 4.1) from $\bar{\pi}(X)$ in Algorithm 4.3.

Algorithm 4.3: Error calculation for PFDs and PUCCs.

```
1  Function $e(X, r) =$ calc-pucc-error$(\bar{\pi}(X), r)$
2      return $\sum_{c \in \bar{\pi}(X)} \frac{|c|^2 - |c|}{|r|^2 - |r|}$
3  Function $e(X \to A, r) =$ calc-pfd-error$(\bar{\pi}(X), v_A, r)$
4      $e \leftarrow 0$
5      foreach $c \in \bar{\pi}(X)$ do
6          $counter \leftarrow$ dictionary with default value 0
7          foreach $i \in c$ do
8              if $v_A[i] \neq 0$ then increase $counter[v_A[i]]$
9          $e \leftarrow e + \left(|c|^2 - |c|\right) - \sum_{c_A \in counter} \left(c_A^2 - c_A\right)$
10     return $\frac{e}{|r|^2 - |r|}$
```

For a PUCC candidate X, the error calculation given $\bar{\pi}(X)$ is trivial: We merely count all tuple pairs inside of each cluster because these are exactly the violating tuple pairs (Lines 1–2). In contrast, the error calculation of a PFD candidate $X \to A$ is a bit more complex. According to Definition 4.1, those tuple pairs violate $X \to A$ that agree in X and disagree in A. We do not count these tuple pairs directly. Instead, for each cluster of $\bar{\pi}(X)$ we calculate the number of tuple pairs also agreeing in A (Lines 4–8) and then subtract this number from all tuple pairs in the cluster (Lines 9–10). For this calculation, we need the attribute vector v_A of attribute A (cf. Section 4.4.1), in addition to $\bar{\pi}(X)$. Note that we must not count zeros in v_A, because they represent singleton values. By summing the errors of all clusters in $\bar{\pi}(X)$, we finally obtain $e(X \to A, r)$.

4.4.3 Estimating Dependency Errors

A key idea of PYRO is to avoid costly PLI-based error calculations by estimating the errors of dependency candidates and only then conduct a few targeted error calculations. As a matter of fact, an error *calculation* can be orders of magnitudes slower than an error *estimation*. Generally speaking, we can estimate dependency errors by comparing a subset of tuples – or better: a subset of tuple pairs – and extrapolate the number of encountered violations to the whole relation. Such error estimation is related to (but far more efficient than) algorithms that exhaustively compare *all* tuple pairs to discover dependencies [Flach and Savnik, 1999; Lopes et al., 2002a]. The basis for this approach are *agree set samples (AS samples)*.

Definition 4.3 (AS sample)**.** Given a relational instance r with schema R and two tuples $t_1, t_2 \in r$, their *agree set* [Beeri et al., 1984] is $ag(t_1, t_2) := \{A \in R \mid t_1[A] = t_2[A]\}$.[3] Further, let $s \subseteq r^2$ be a sample of tuple pairs of the relational instance r. Then, s induces the AS sample

$$\mathsf{AS} := \{(a, c(a)) \mid \exists (t_1, t_2) \in s \colon a = ag(t_1, t_2)\}$$

where $c(a) := |\{(t_1, t_2) \in s \mid a = ag(t_1, t_2)\}|$ counts the number of occurrences of each agree set in s.

Example 4.4. Assume that we randomly sample three tuple pairs from Figure 4.1, e. g., (t_1, t_3), (t_1, t_5), and (t_2, t_3). This gives us the AS sample $\mathsf{AS} = \{(\{\mathsf{Gender}, \mathsf{Town}\}, 1)$, $(\{\mathsf{First\ name}, \mathsf{Town}\}, 1)$, $(\{\mathsf{Gender}, \mathsf{ZIP}\}, 1)\}$.

Now to estimate PFD and PUCC errors from an AS sample AS, we define a query that reports the number of agree sets in AS that include some attribute set *inc* and do not contain any attribute of a further attribute set *exc*:

$$\texttt{count}(\mathsf{AS}, inc, exc) := \sum_{(a,c) \in \mathsf{AS}} \begin{cases} c & \text{if } inc \subseteq a \wedge exc \cap a = \emptyset \\ 0 & \text{otherwise} \end{cases}$$

PYRO stores agree sets efficiently as bit masks using one-hot encoding. This allows to keep AS samples in memory and perform `count` efficiently with a full scan over the AS sample. So, to estimate the error of a PFD candidate $X \to A$, we could count its numbers of violating agree sets in AS as $\texttt{count}(\mathsf{AS}, X, \{A\})$ and divide the result by $\|\mathsf{AS}\|$. Likewise, for a PUCC candidate X, we could count the violations $\texttt{count}(\mathsf{AS}, X, \emptyset)$ and, again, divide by $\|\mathsf{AS}\|$:

Lemma 4.1. *Let $\hat{e}$ be the estimated error of a* PFD *or* PUCC *candidate using the AS sample AS. Further, let e denote the actual dependency error. Then $\hat{e}$ is unbiased, i. e., its expectation value is exactly e, and the probability that $|e - \hat{e}| \leq \varepsilon$ for some user-defined ε is given by*

$$P_\varepsilon(\mathsf{AS}, e) := \sum_{i=\lceil (e-\varepsilon)\cdot\|\mathsf{AS}\| \rceil}^{\lfloor (e+\varepsilon)\cdot\|\mathsf{AS}\| \rfloor} \binom{\|\mathsf{AS}\|}{i} e^i (1-e)^{\|\mathsf{AS}\|-i}$$

Proof. Sampling n tuple pairs and testing whether they violate an PUCC or PFD candidate follows a binomial distribution whose probability parameter is exactly the dependency error as defined in Definition 4.1. The mean of this distribution, i. e., the expected number of violations in AS, is $e \cdot \|\mathsf{AS}\|$ $(= \mathbb{E}[\hat{e}] \cdot \|\mathsf{AS}\|)$ and the above error bounds can be immediately derived from the cumulative distribution function of the binomial distribution. □

Interestingly, the accuracy of our error estimates does not depend on the size of the input relation, which makes it highly scalable. Instead, we observe an influence of the

[3]For improved memory and computation efficiency, we calculate agree sets from cached attribute vectors (see Section 4.4.1) rather than the original input dataset.

actual dependency error e. Indeed, the variance of the binomial distribution (and thus the uncertainty of our estimator) is maximized for $e = 0.5$. However, for typical error thresholds we need accurate estimates only when e approaches 0 to tell apart partial dependencies and non-dependencies. For instance, for $e = 0.01$ and $\|AS\| = 1,000$ our error estimates are off by at most 0.006 with a probability of 0.96.

However, if we had only a single AS sample for the whole relation, this sample might need to be huge to achieve high precision estimates when needed: The standard deviation of our bionomially distributed error estimator, $\sqrt{\frac{e(1-e)}{\|\mathsf{AS}\|}}$, is inversely proportional to the square root of the sample size. Intuitively, one might suspect that the above accuracy for $\|AS\| = 1,000$ is sufficient to discover partial PFDs and PUCCs with an error threshold of 0.01, but that is not necessarily the case. As an example, assume a relation with n attributes A_i ($1 \leq i \leq n$), each having a PUCC error of 0.0101, while their combination $A_1 \ldots A_n$ has an error of 0.0999. In this scenario, any set of two or more attributes might be a minimal PUCCs and, for that matter, there are $2^n - (n+1)$ such sets. Obviously, we would need samples with much more than the above 1,000 agree sets to reasonably predict where the minimal PUCCs might be, which would come at a high cost.

To provide high precision error estimates from small AS samples, Pyro uses a *focused sampling* technique. However, the resulting samples must still be random, so as to preserve the above explained unbiasedness of our estimator. We solve this conflict as follows: The main idea is to sample only such agree sets a that are supersets of some given attribute set X, i.e. $a \supseteq X$. Such a sample can be created efficiently: We obtain the PLI $\bar{\pi}(X)$ and then sample only such tuple pairs that co-occur in some cluster $c \in \bar{\pi}(X)$. As an example, consider the PLI $\bar{\pi}(\{\mathsf{Zip}\}) = \{\{1,4\},\{2,3,5\}\}$ for Figure 4.1. This PLI restricts the sampling to tuple pairs from $\{t_1, t_4\}$ or $\{t_2, t_3, t_5\}$.

In detail, to sample a tuple pair that agrees in X, we first select a cluster $c' \in \bar{\pi}(X)$ with a probability of $\frac{|c'|^2-|c'|}{pairs(X)}$ where $pairs(X) := \sum_{c \in \bar{\pi}(X)} |c|^2 - |c|$ denote the number of overall tuple pairs agreeing in X. That is, the probability of picking c' is proportional to the number of its tuple pairs. Then, we randomly sample two distinct tuples from c', so that each tuple pair within c' is sampled with the probability $\frac{1}{|c'|^2-|c'|}$. In consequence, *any* tuple pair with tuples agreeing in X from the input relation has the same probability $\frac{1}{pairs(X)}$ of being sampled. Finally, we calculate the agree sets for the sampled tuple pairs and obtain a focused, yet random, AS sample, denoted AS_X.

Based on AS_X, we can now estimate the error of any PFD candidate $Y \rightarrow A$ and PUCC candidate Y if $Y \supseteq X$. In fact, the error of the PUCC candidate Y in a relation r can be estimated as

$$\hat{e}(Y, r) := \frac{\mathtt{count}(\mathsf{AS}_X, Y, \emptyset)}{\|\mathsf{AS}_X\|} \cdot \frac{pairs(X)}{|r|^2 - |r|}$$

and the error of the PFD candidate $Y \rightarrow A$ as

$$\hat{e}(Y \rightarrow A, r) := \frac{\mathtt{count}(\mathsf{AS}_X, Y, \{A\})}{\|\mathsf{AS}_X\|} \cdot \frac{pairs(X)}{|r|^2 - |r|}$$

where $\|\mathsf{AS}_X\| := \sum_{(a,c) \in \mathsf{AS}_X} c$.

Theorem 4.2. *Given a* PUCC *candidate Y or* PFD *candidate $Y \to A$ $(Y \supseteq X)$, our focused estimators based on sample AS_X are unbiased and the probability that $|e - \hat{e}| \leq \varepsilon$ for an actual dependency error e, error estimate $\hat{e}$, some user-defined ε is given by $P_\varepsilon(\mathsf{AS}_X, e\frac{|r|^2-|r|}{pairs(X)})$.*

Proof. The first terms of the estimators estimate the ratio of the tuple pairs violating the dependency candidate among all tuple pairs agreeing in X; Lemma 4.1 shows their unbiasedness and error bounds. Because all violating tuple pairs must agree in X, the additional terms *exactly* extrapolate this "focused" estimate to the whole relation, thereby preserving the unbiasedness and shrinking the error bounds by a constant factor. □

Theorem 4.2 explains why focused samples are effective. Consider again the ZIP column in Figure 4.1: Out of all 10 tuple pairs, only 4 agree in their ZIP values, so that a ZIP-focused estimator is $\frac{10}{4} = 2.5\times$ more precise than an unfocused estimator with the same sample size and confidence level. This effect is even stronger in larger, real-world data sets. For instance, a focused estimator for an attribute with 1,000 equally distributed values shrinks the error bounds by a factor of 10^6. Hence, it is more efficient to create and use multiple focused samples rather than one highly extensive one. In fact, PYRO operates only on focused samples – initially one for each attribute.

Having explained focused AS samples and how to estimate PFD and PUCC errors with them, it remains to be shown how PYRO serves an actual request for an error estimate of some dependency candidate. Without loss of generality, assume that the dependency candidate in question is the PUCC candidate Y. As for the PLIs, whenever PYRO creates an AS sample, it caches it in a trie (cf. Figure 4.2 and Section 4.4.1). PYRO first determines all AS samples with focus X for some $X \subseteq Y$, which can be done efficiently because the AS cache is a trie. Recall that the focus of the AS sample must be a subset of the dependency candidate to obtain an unbiased estimate. Then, PYRO picks the AS sample with the highest *sampling ratio*, i. e.,the ratio of sampled agree sets to the population; formally $\frac{\|\mathsf{AS}_X\|}{pairs(X)}$. The reason is that larger AS samples and smaller sampling foci yield more precise error estimates (see Theorem 4.2). What is more, when the population for the sample AS_X is very small (because X is almost a UCC), then the sample can even be exhaustive and, hence, the error estimate is known to be exact and need not be calculated with PLIs anymore.

4.5 Search Space Traversal

To discover dependencies, it is not only important to efficiently assess individual dependency candidates as explained in the above section; it is also crucial to develop a search space traversal strategy that combines efficient error estimations and final error calculations. In contrast to TANE, which systematically traverses large parts of the search space, and in contrast to DUCC/DFD, which perform a random walk through the search space, PYRO employs a novel separate-and-conquer strategy: It separates a part of the search space, estimates the position of the minimal dependencies within that subspace,

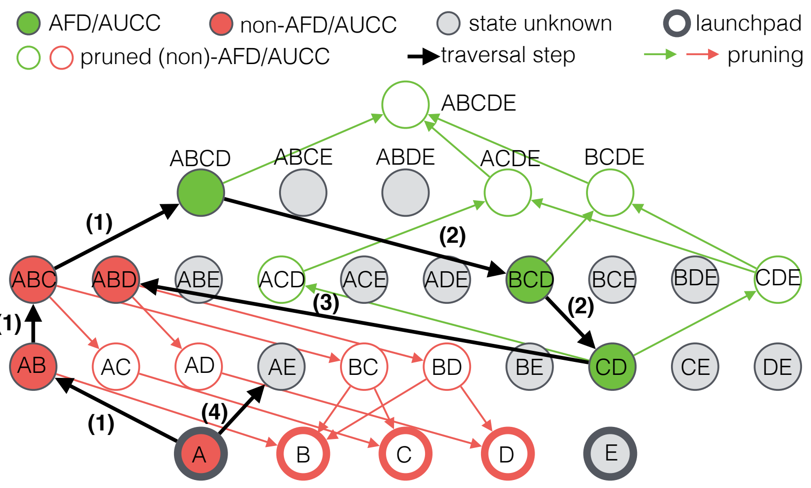


Figure 4.4: Example of a search space traversal round.

and then validates this estimate. Afterwards, considerable parts of the search space can be pruned without ever being visited.

Let us outline PYRO's traversal strategy more concretely with the example in Figure 4.4 before elaborating on its individual phases in detail. The traversal can be thought of as a firework display – which inspired the name PYRO. It consists of multiple rounds starting from the single attributes, the *launchpads*. In our example, PYRO selects A as launchpad[4] and *ascends* (like a skyrocket) in the search space until it detects the dependency $ABCD$ (Step (1), Section 4.5.1). From this dependency, called *peak*, PYRO *trickles down* (like an exploded skyrocket) and estimates the position of all *minimal* dependencies that generalize the peak (Step (2), Section 4.5.2), which is the case for CD. Then, it verifies the estimate by checking the *complementary*, untested dependency candidates, namely ABD (Step (3), Section 4.5.3).

This completes the first search round and, as shown in Figure 4.4, PYRO uses both discovered non-dependencies and dependencies to drastically narrow down the search space for subsequent search rounds. In fact, discovered (non-)dependencies are stored in a trie (cf. Section 4.4.1) to efficiently determine whether following dependency candidates are already pruned. Finally, in the next search round, PYRO might pick up a pruned launchpad; in Figure 4.4, we pick again A. In that case, PYRO *escapes* the launchpad into the unpruned part of the search space (Step (4), Section 4.5.4).

4.5.1 Ascend

The goal of the ascend phase is to efficiently determine some dependency in a given search space, which will then form the input to the subsequent trickle-down phase. Algorithm 4.4 depicts how PYRO achieves that. PYRO begins this search at the *launchpads*.

[4]Recall from Section 4.3 that we use dependency candidates and attribute sets synonymously. A can be an PUCC candidate or an PFD candidate FD$AR$ for some RHS R. As a result, PYRO's traversal algorithm can handle both dependency types.

These are minimal dependency candidates with an unknown state. Hence, the single attributes are the initial launchpads. Pyro estimates their error and picks the one with the smallest error, e. g., attribute A as in Figure 4.4, assuming it to lead to a dependency quickly (Line 1).

Algorithm 4.4: The ascend phase.

```
   Data: launchpads $L$, maximum error $e_{\max}$
1  $(X, \hat{e}_X) \leftarrow$ pick launchpad with smallest error estimate $\hat{e}_X$ from $L$
2  while True do
3      if $\hat{e}_X \leq e_{\max}$ then
4          if $\hat{e}_X$ is not known to be exact then
5              $\hat{e}_X \leftarrow$ calculate error of $X$
6          if $\hat{e}_X \leq e_{\max}$ then break
7      $A \leftarrow \arg\min_{A \in R \setminus X} \hat{e}_{XA}$ with $XA$ is not pruned
8      if no such $A$ then break
9      $X \leftarrow XA$
10     estimate $\hat{e}_X$
11 if $\hat{e}_X$ is not known to be exact then
12     $\hat{e}_X \leftarrow$ calculate error of $X$
13 if $\hat{e}_X \leq e_{\max}$ then
14     trickle down from $X$
15 else
16     declare $X$ a maxmimum non-dependency
```

Then, Pyro greedily adds that attribute to the launchpad that reduces the (estimated) dependency error the most (Line 7) until either a dependency is met (Line 6) or no attribute can be added anymore (Line 8). The latter case occurs when there simply is no attribute left to add or when all possible candidates are already known dependencies from previous search rounds. In this case, we declare it as a maximum non-dependency for pruning purposes and cease the current search round (Line 16). However, if we meet a dependency, as is the case for $ABCD$ in Figure 4.4, we proceed with the trickle-down phase starting from that dependency (Line 14).

4.5.2 Trickle down

Given a dependency from the ascend phase, called *peak* P, Pyro trickles down to estimate the position of *all minimal* dependencies that generalize P. Algorithm 4.5 outlines how Pyro performs this estimation. First, P is placed into a new priority queue that orders peaks by their estimated dependency error (Line 2). Pyro then takes the smallest element (which initially is P) without removal (Line 4), checks whether it is pruned by some already estimated minimal dependencies (Line 5) (which is initially not the case), and then invokes the function `trickle-down-from` with P whose purpose is to estimate the position of exactly one minimal dependency that generalizes P or return $\bot$ if P and none of its generalizations are estimated to be a dependency (Line 12).

Algorithm 4.5: Estimate position of minimal dependencies.

```
Data: peak $P$, maximum error $e_{\max}$
1  $\mathcal{M} \leftarrow$ new trie
2  $\mathcal{P} \leftarrow$ new priority queue with $P$
3  while $\mathcal{P}$ is not empty do
4      $P' \leftarrow$ peek from $\mathcal{P}$
5      $\mathcal{M}' \leftarrow$ look up subsets of $P'$ in $\mathcal{M}$
6      if $\mathcal{M}' \neq \emptyset$ then
7          remove $P'$ from $\mathcal{P}$
8          foreach $H \in$ hitting-set($\mathcal{M}'$) do
9              if $P' \setminus H$ is not an (estimated) non-dependency then
10                 add $P' \setminus H$ to $\mathcal{P}$
11     else
12         $M \leftarrow$ trickle-down-from($P', e_{\max}$)
13         if $M \neq \bot$ then add $M$ to $\mathcal{M}$ else remove $P'$ from $\mathcal{P}$
14 Function trickle-down-from($P', e_{max}$)
15     if $|P'| > 1$ then
16         $\mathcal{G} \leftarrow$ error-based priority queue with generalizations of $P'$
17         while $\mathcal{G}$ is not empty do
18             $G, \hat{e}_G \leftarrow$ poll from $\mathcal{G}$
19             if $\hat{e}_G > e_{\max}$ then break
20             $C \leftarrow$ trickle-down-from($G, e_{\max}$)
21             if $C \neq \bot$ then return $C$
22     $e_{P'} \leftarrow$ calculate error of $P'$
23     if $e_{P'} \leq e_{\max}$ then return $P'$
24     create and cache AS sample with focus $P'$
25     return $\bot$
```

In our example, we initially invoke `trickle-down-from` with our peak $P = ABCD$. It now creates a priority queue that orders the immediate generalizations of P, i. e., ABC, ABD etc., by their estimated dependency error (Line 16). These generalizations are potential minimal dependencies, therefore any of them with an estimated error of less than $e_{\max}$ is recursively trickled down from (Lines 17–21). If the recursion yields a minimal dependency candidate, PYRO immediately reports it. In Figure 4.4, we recursively visit BCD and then CD. Neither C nor D is estimated to be a dependency, so CD might be a minimal dependency. Eventually, we calculate the error of CD to make sure that it actually is a dependency and return it (Lines 22–23). If, in contrast, we had falsely assumed CD to be a dependency, we would create a focused sample on CD so as to obtain better error estimates for dependency candidates between CD and the peak $ABCD$ and continue the search at BCD.

Finally, we add CD to the estimated minimal dependencies $\mathcal{M}$ (Line 13) and peek again from the peak priority queue (Line 4), which still contains the original peak $ABCD$. However, now there is the alleged minimal dependency CD, which explains $ABCD$. Still, we must not simply discard this peak, because there might be further minimal dependencies generalizing it. For this purpose, we identify all maximal dependency candidates that are a subset of $ABCD$ but not a superset of CD. As detailed in the next paragraphs, PYRO determines those candidates by calculating the *minimal hitting sets* of CD, namely C and D, and removing them from $ABCD$ (Line 8–10), which yields ABD and ABC. These form the new peaks from which the search for minimal dependencies is continued. In our example, we estimate both to be non-dependencies and remove them from the queue (Line 13).

Let us now explain the hitting set calculation in more detail. Formally, a set is a hitting set of a set family $\mathcal{S}$ (here: a set of attribute sets) if it intersects every set $S \in \mathcal{S}$. It is *minimal* if none of its subsets is a hitting set. The calculation of minimal hitting sets is employed in the following traversal steps, too, and furthermore constitutes an NP-hard problem [Karp, 1972]. Facing this computational complexity, the problem should be solved as efficiently as possible. Algorithm 4.6 displays how PYRO calculates *all* minimal hitting sets for a set of attribute sets.

Algorithm 4.6: Calculate minimal hitting sets.

```
   Data: attribute sets $\mathcal{S}$, relation schema $R$
1  Function hitting-set($\mathcal{S}$)
2      $T \leftarrow$ set trie initialized with $\emptyset$
3      $L_\mathcal{S} \leftarrow$ list of elements in $\mathcal{S}$
4      sort $L_\mathcal{S}$ ascending by set size
5      foreach $S \in L_\mathcal{S}$ do
6          $\overline{S} \leftarrow R \setminus S$
7          $\mathcal{V} \leftarrow$ remove all subsets of $\overline{S}$ from $T$
8          foreach $V \in \mathcal{V}$ do
9              foreach $A \in S$ do
10                 if no subset of $VA$ is in $T$ then
11                     add $VA$ to $T$
12     return $T$
```

First, we initialize a set trie (cf. Section 4.4.1) with the empty set as initial solution (Line 2). Next, we order the attribute sets by size (Lines 3–4). If $\mathcal{S}$ contains two sets A and B with $A \subseteq B$, we want to process A first. The rationale for this is that any intermediate hitting set that intersects A will also intersect B. When processing B, we do not need to update the intermediate solution. Then, we iterate the ordered input attribute sets one after another (Line 5). Assume, we have $S = CD$ as above. Then, we remove all current hitting sets that do not intersect with S by looking up subsets of its inversion (Line 6–7); recall that we can perform subset lookups on tries efficiently. In our example, the inversion of CD is ABE and the initial solution in T, the empty set, is a subset of it. Eventually, we combine all removed sets with all attributes in S to re-establish the hitting set property (Line 8–11). For instance, combining $V = \emptyset$ with $S = CD$ yields the two new hitting sets C and D. However, these new hitting sets might not be minimal. Therefore, before adding a new hitting set H back to the trie, we check if there is an existing minimal hitting set in the trie that is a subset of H. Again, subset look-ups can be performed efficiently on tries. After all attributes from $\mathcal{S}$ have been processed, the trie contains all minimal hitting sets.

4.5.3 Validate

While the estimated set of minimal dependencies $\mathcal{M}$ from the trickle-down phase contains only verified dependencies, it is not known whether these dependencies are minimal and whether $\mathcal{M}$ is complete. Pyro validates the completeness of $\mathcal{M}$ with as few error calculations as possible. $\mathcal{M}$ is complete if and only if any dependency candidate $X \subseteq P$ is either a specialization of some allegedly minimal dependency $Y \in \mathcal{M}$, i. e., $X \supseteq Y$, or X is a non-dependency.

To test this, it suffices to test the *maximal* alleged non-dependencies "beneath" the peak, i. e., those dependency candidates that generalize P and whose specializations are all known dependencies. If these maximal candidates are indeed non-dependencies, then so are all their generalizations and $\mathcal{M}$ is complete. As in Algorithm 4.5, Pyro calculates these candidates, denoted as $\overline{\mathcal{M}}$, by calculating the minimal hitting sets of all elements in $\mathcal{M}$ and removing them from P. For our example, we have `hitting-sets`$(\{CD\}) = \{C, D\}$ and thus need to check $P \setminus C = ABD$ and $P \setminus D = ABC$.

Pyro checks all candidates in $\overline{\mathcal{M}}$ with two possible outcomes. If those candidates are non-dependencies, then $\mathcal{M}$ is indeed complete. If, however, $\overline{\mathcal{M}}$ contains dependencies, $\mathcal{M}$ is not complete. Nonetheless, we can use this result to narrow the focus our search.

Let $\mathcal{D} \subseteq \overline{\mathcal{M}}$ denote said dependencies in $\overline{\mathcal{M}}$ and assume that ABD turned out to be a dependency, i. e., $\mathcal{D} = \{ABD\}$. Any dependency not covered by $\mathcal{M}$ must be a generalization of some dependency in $\mathcal{D}$, because any candidate $X \subseteq P$ is either a superset of some element in $\mathcal{M}$ or a subset of some element $\overline{\mathcal{M}}$.

Further, let $\mathcal{N} = \overline{\mathcal{M}} \setminus \mathcal{D}$ denote the non-dependencies in $\overline{\mathcal{M}}$. In our modified example, we have $\mathcal{N} = \{ABC\}$. These are maximal w. r. t. the peak P, i. e., all their supersets that are also subsets of P are known dependencies. We can now determine the dependency candidates that are not subsets of any such maximal non-dependency in $\mathcal{N}$, denoted as $\overline{\mathcal{N}}$: We invert all elements in $\mathcal{N}$ w. r. t. P and calculate their minimal hitting sets. For $\mathcal{N} = \{ABC\}$, we get $\overline{\mathcal{N}} = \{D\}$. It follows that any dependency not covered by $\mathcal{M}$ is a specialization of some dependency candidate in $\overline{\mathcal{N}}$ and a generalization of some dependency in $\mathcal{D}$, i. e., in our example the unknown minimal dependencies must be a superset of D and a subset of ABD.

As a result, Pyro can create a search sub-space with exactly those dependency candidates and recursively process it, including all steps presented in this section. In addition, Pyro *boosts* the size of AS samples while working in this sub-space so as to decrease the probability of new mispredictions about the minimal dependencies. Still, in the case of another misprediction Pyro can recursively create new sub-spaces. The recursion is guaranteed to terminate, though, because the sub-spaces are continuously shrinking. However, in our experiments we rarely saw a recursion depth of even 2. After the recursion, Pyro eventually needs to check for which dependencies in $\mathcal{M}$ the recursion has not yielded a generalizing minimal dependency. Those dependencies were minimal all along and must be reported as such.

4.5.4 Escape

When a search round has been completed, it prunes great parts of the search space. As can be seen in Figure 4.4, also the launchpads might now be in the pruned part. Unless these launchpads have been found to be (minimal) dependencies, we must not discard them, though: There might still be undiscovered minimal dependencies that are supersets of that launchpad.

Whenever Pyro picks up a pruned launchpad, it needs to *escape* it out of the pruned search space part by adding a minimum amount of attributes to it. Let us assume that Pyro picks up the launchpad A once more. To determine whether it is pruned, Pyro determines all previously visited peaks that are supersets of A, which is $ABCD$ in our case. Again, a hitting set calculation can now determine minimum attribute sets to add to A, such that it is not a subset of $ABCD$ anymore: Pyro calculates the hitting sets of $R \setminus ABCD = E$, which is simply E. By adding E to A, we get the only minimal escaping (cf. Step (4) in Figure 4.4). Note that this operation is the exact inverse of the relocation of peaks in Algorithm 4.5. However, because we have the launchpad E, which is a subset of AE, we finally have to discard AE and, as a matter of fact, all unknown dependency candidates are indeed supersets of E.

Because Pyro maintains the launchpads, such that they form exactly the minimal untested dependency candidates, a search space is completely processed when it contains no more launchpads. As a result, Pyro will eventually terminate with the complete set of dependencies of the search space.

4.6 Parallelization

In Section 1.4, we suggest that it might be appropriate for algorithms that solve computationally hard problems to exploit as much hardware resources as possible through parallel and distributed execution. Furthermore, it is interesting from a research perspective to determine under which circumstances and to what extent parallelization improves the efficiency and scalability of an algorithm. As in the previous chapters, we propose a parallelization scheme for the algorithm described in this chapter, Pyro.

Let us at first focus on parallelizing Pyro within a single computer. Obviously, Pyro's search spaces can be processed independently of each other. Only the access to the PLI and AS cache has to be synchronized. This can be done via read-write-locks, which allow the more frequent reads to these data structures to take place simultaneously. Only the less frequent writes need exclusive access.

However, in general, it is possible that we have more cores available than there are search spaces – especially when the algorithm has fully processed the majority of search spaces of a given dataset. For this reason, PYRO can also have two or more cores work on a single search space, in which case it runs multiple search rounds in the same search space (starting from different launchpads) in parallel.

Of course it is also possible that there are fewer available launchpads in a search space than there are cores assigned to it. In this case, the first core that cannot retrieve a launchpad from its search space searches for a different search space with a free launchpad. Only if there is no such search space, the core goes to sleep for a short time and then again tries to pick up a launchpad from some search space. Notice that it is indeed possible that the number of launchpads in a search space shrinks and grows throughout the profiling process.

This parallelization scheme can be further extended to computer clusters: As depicted in Figure 4.5, we can achieve this with a *master* that assigns search spaces to *workers*. A worker runs on a dedicated machine with potentially multiple cores; it maintains its own PLI and AS caches, uses them to profile any assigned search space, and reports discovered dependencies back to the master. The master, in turn, continuously monitors the workers and dynamically (re-)assigns search spaces among workers whenever some worker has completed a search space.

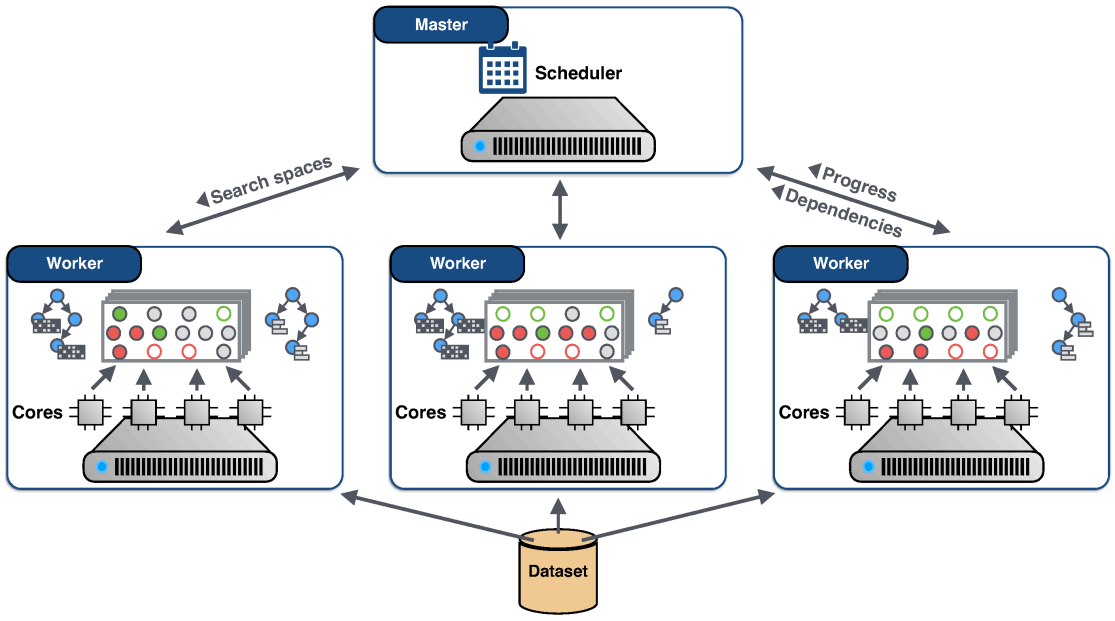


Figure 4.5: Distributed architecture of PYRO.

An interesting question is how to profile a single search space on multiple workers simultaneously. The key challenge is that synchronized access to data structures across computers would impose severe network latency. Therefore, we suggest that each worker maintain its very own set of discovered (non-)dependencies and regularly synchronize them with workers profiling the same search space. While this form of weak consistency might cause duplicate work among workers, it also decouples them and thus enables scalability. However, we leave a more thorough specification of this idea as well as its evaluation to future work.

4.7 Evaluation

Pyro's goal is to enable efficient and scalable discovery of partial dependencies in given datasets. To evaluate in how far Pyro attains this goal, we particularly investigate how Pyro compares to (modifications of) three state-of-the-art algorithms in terms of efficiency and scalability and furthermore conduct several in-depth analyses of Pyro. Note that we resort to an empirical evaluation: A theoretical comparison of the algorithms would be of limited value, because in the worst case pucc/pfd discovery is of exponential complexity w. r. t. the number of profiled attributes due to the possibly exponential numbers of dependencies [Abedjan et al., 2014b; Liu et al., 2012] – even a brute-force algorithm that checks every dependency candidate would meet that complexity. An average-case complexity analysis, on the other hand, would have to resort to strong, perhaps unjustifiable, assumptions on the data due to the adaptivity of the algorithms. That being said, we close the evaluation with an investigation of the interestingness of puccs and pfds.

4.7.1 Experimental setup

We have carefully (re-)implemented Pyro, Tane, Fdep, and Ducc/Dfd in Java, so that their runtimes are comparable. For easy repeatability of our experiments, all algorithms are integrated with the data profiling framework Metanome [Papenbrock et al., 2015a]. Pyro further monitors the memory usage of the PLI and AS caches and halves them when running low on memory. Also, Pyro is integrated with Akka 2.5.2[5] to implement the distribution scheme explained in Section 4.6. Additionally, we fixed a known issue regarding Tane's key-pruning [Papenbrock et al., 2015b] and extended both Tane and Fdep to output also puccs. For Fdep and Ducc/Dfd specifically, we consulted the original implementations whenever we found that their respective publications were not specific enough. Eventually, we modified Ducc/Dfd to discover partial dependencies as suggested in [Abedjan et al., 2014b]. To our knowledge, this modification was never implemented or evaluated. Therefore, we believe that the respective experiments are interesting in itself.

We conducted our experiments on a cluster that consists of a master node (IBM xServer x3500 with two Intel Xeon X5355 Quadcore (2.66 GHz, 8 MB cache), 32 GB RAM, and a 2.7 TB hard disk) and four workers (Dell PowerEdge R430 with an Intel Xeon E5-2630 v4 (2.2 GHz, 10 cores, 20 threads, 25 MB cache), 32 GB RAM, and 2 hard disks (7.2 kRPM, SATA)) running Ubuntu 16.04.1 and Oracle JDK 1.8.0_112 with a maximum heap size of 24 GB. Experiments that did not require distribution (which

[5]See `https://akka.io/` (accessed January 15, 2018).

actually applies to most experiments) were run on one of the workers only. Unless specified otherwise, we considered an error threshold of 0.01 for both PFDs and PUCCs and used an initial AS sample size of 1,000 and a boost factor of 2 with PYRO. The partition cache size for DUCC/DFD was set to 1,000 in accordance to the suggested value and our main memory size [Abedjan et al., 2014b]. Our datasets were stored as CSV files on disk. Their sizes can be found in Table 4.1. For repeatability purposes, our implementations and pointers to the datasets can be found on our webpage: `https://hpi.de/naumann/projects/repeatability/data-profiling/fds.html`.

Dataset			PFDs + PUCCs	
Name	Cols.	Rows	$e_{\max} = 0.01$	$e_{\max} = 0.05$
School results	27	14,384	3,408	1,527
Adult	15	32,561	1,848	1,015
Classification	12	70,859	119	1311
Reflns	37	24,769	9,396	3,345
Atom sites	31	32,485	79	78
DB status	35	29,787	108,003	45,617
Entity source	46	26,139	*(unknown)*	*(unknown)*
Bio entry	9	184,292	29	39
Voter	19	100,001	647	201
FDR-15	15	250,001	225	225
FDR-30	30	250,001	900	900
Atom	31	160,000	1,582	875
Census	42	199,524	*(unknown)*	*(unknown)*
Wiki image	12	777,676	92	74
Spots	15	973,510	75	79
Struct sheet	32	664,128	1,096	1,458
Ditag feature	13	3,960,124	187	260

Table 4.1: Overview of the evaluation datasets sorted by their size (columns $\times$ tuples).

4.7.2 Efficiency

Let us begin with a broad comparison of all four algorithms to show in how far PYRO advances the state of the art in PUCC and PFD discovery. For that purpose, we ran all algorithms on a single worker with the datasets from Table 4.1 and two different, typical error thresholds. Note that we report runtimes for PYRO in a parallel version (to show its best runtimes) and a non-parallel version (for comparison with the other algorithms). We also report runtimes for the logically flawed FDEP algorithm (see Section 4.2) to include an algorithm that is based on the comparison of all tuple pairs in a dataset. The results are shown in Figure 4.6.

Obviously, PYRO is the most efficient among the correct algorithms: While the non-parallel version is outperformed on some easy-to-process datasets by at most 0.6 seconds, PYRO outperforms its competitors by at least a factor of 2 in 59 % of the configurations.

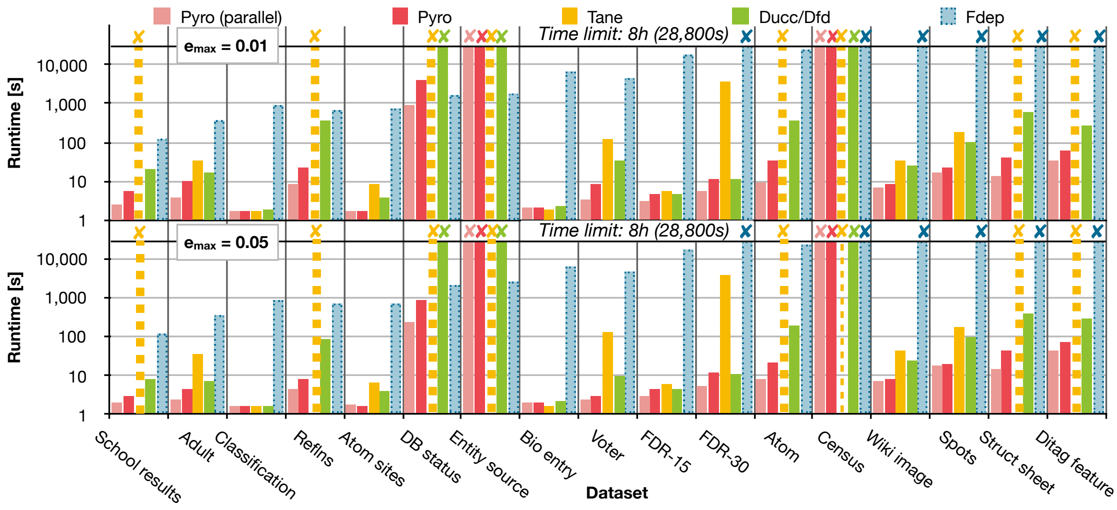


Figure 4.6: Runtime comparison of Pyro, Tane, Ducc/Dfd, and Fdep. The crosses indicate that an algorithm either ran out of time or memory.

For that matter, for hard-to-process datasets we observe the greatest speed-ups. For instance on the DB status dataset for $e_{\max} = 0.05$, Pyro outperforms the best competitor, Ducc/Dfd, *at least* by a factor of 33 (or 7.7 h in absolute terms) – for the parallel version, we even measured a speed-up factor of at least 123. The actual speed-up might even be greater, because Ducc/Dfd did not complete the profiling within the 8 h time limit. This shows that Pyro's approach to estimate and verify dependency candidates is much more efficient than Tane's breadth-first search and Ducc/Dfd's random walk search.

The above comparison omits Fdep, because it reports incorrect results (see Section 4.2). Nonetheless, we find it to be regularly inferior in terms of performance. The reason is that it compares all possible tuple pairs in a dataset, thereby incurring quadratic load in the number of tuples, which is prohibitive on larger datasets. On the other hand, the few scenarios where Fdep is faster than Pyro can be attributed to its logical flaw: For instance on the DB status for $e_{\max} = 0.01$, Fdep reports only 15,138 supposed dependencies – 7× fewer than there actually are. And what is more, 67 % of these supposed dependencies are incorrect. Correcting those results would require additional time. Note that, although Pyro also compares tuple pairs, it does not run into the quadratic trap because it requires only a constant number of comparisons (cf. Theorem 4.2) and reports correct results because it does not deduce the final dependencies from those tuple comparisons.

While Pyro is clearly the superior algorithm in terms of performance, no algorithm could profile the complete Entity source and Census datasets within the time limit. The next section shows the reason: These datasets entail tremendous dependency sets.

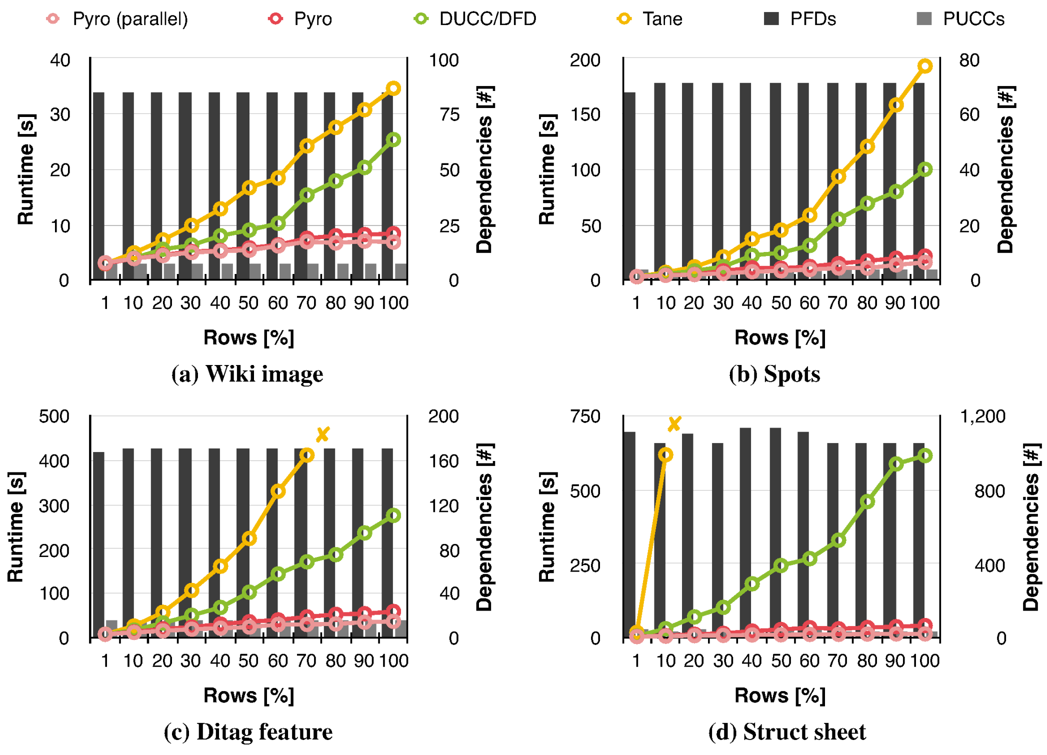


Figure 4.7: Row scalability of Pyro, Ducc/Dfd, and Tane ($e_{\max} = 0.01$). Crosses indicate that an algorithm ran out of memory.

4.7.3 Scalability

Our next experiment analyzes the algorithms' scalability, i. e., how they compare as the profiled datasets grow in the number of tuples and columns. Note that we again exclude Fdep from these and following detail analyses.

To determine the row scalability of the algorithms, we execute them on various samples of the four datasets with the most rows; Figures 4.7(a–d) depict the results. Because the number of dependencies is mainly stable across the samples, each algorithm tests more or less the same dependency candidates, regardless of the sample size. While all algorithms exhibit a somewhat linear scaling behavior, Ducc/Dfd is usually up to 20 times faster than Tane, and Pyro is up to 15 times faster than Ducc/Dfd. In both cases, the speed-up increases as the sample size increases, which is because on larger datasets the overhead, e. g., for initially loading the data and maintaining AS samples to avoid PLI-based error calculations, is more effectively redeemed.

In our column scalability experiments on the four datasets with the most columns, Pyro also shows the best scaling behavior, as can be seen in Figures 4.8(a–d): Ducc/Dfd is up to 22 times faster than Tane and Pyro up to 12 times faster than Ducc/Dfd – the more columns are considered, the greater is the speed-up. A particularly interesting observation is that all algorithms' runtime curves somewhat follow the number of dependencies. Such behavior is optimal, in the sense that any algorithm has to be at least of linear complexity in its output size; still the algorithms differ greatly in terms of

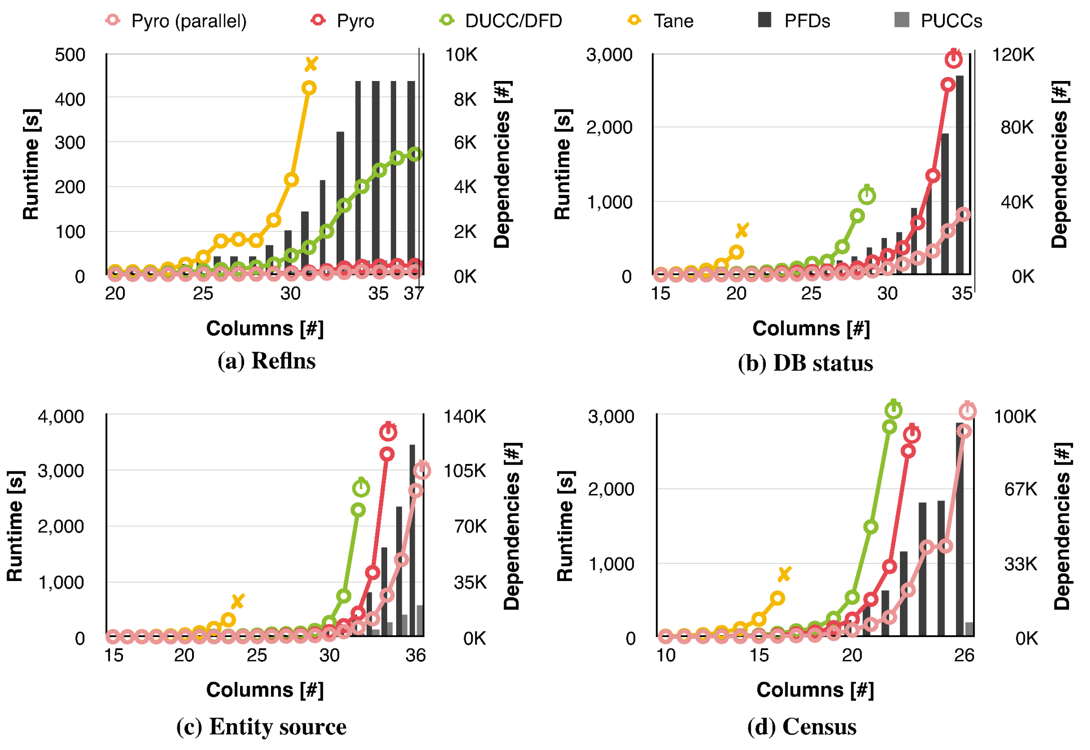


Figure 4.8: Column scalability of Pyro, Ducc/Dfd, and Tane ($e_{\max} = 0.01$). Crosses indicate that an algorithm ran out of memory and stopwatches mean that the algorithm exceeded the 1 h time limit.

absolute runtimes. Also, the number of columns that can be processed by each algorithm differs. Tane always fails first due to its high memory demand. In contrast, Pyro and Ducc/Dfd fail only when they exceed the 1 h time limit of this experiment. They are less susceptible to run out of main memory because of their dynamic caches. Still, Pyro outperforms Ducc/Dfd because of its high efficiency and, as a result, scales further.

Nonetheless, Pyro can only *mitigate*, but not *completely avoid* the effects of the exponential complexity of the dependency discovery problem, of course. Figure 4.9 breaks down Pyro's runtime from Figures 4.8(b–d), thereby showing that the growing number of dependencies requires Pyro to calculate more dependency errors, which dominates the runtime. Avoiding the exponential complexity altogether would require a relaxation of the problem itself, which is not the goal of this work. However, the above experiments demonstrate Pyro's improved scalability, thereby making it an excellent basis for relaxed algorithms.

4.7.4 Parallelization

Besides Pyro's scaling behavior on growing datasets, it is also relevant to understand how Pyro scales as it is provided with more hardware resources. For this purpose, we investigate Pyro's *scale-in* behavior (by varying the number of cores for its computation on a single machine) as well as its *scale-out* behavior (by varying the number of machines for its computation). To quantify these two properties, we consider the *speed-up* of PYRO in different hardware configurations: Let t_i be the runtime of PYRO using i cores or workers, respectively. Then, we calculate the speed-up for using i cores or workers as $\frac{t_1}{t_i}$.

Let us at first consider the scale-in behavior in Figure 4.10(a). We observe that the actual speed-up is highly dependent on the dataset: While on easy-to-process datasets the parallelization is virtually ineffective, datasets with many columns benefit the most from parallelization. We never observe a linear speed-up when using all 10 cores, though. The reasons for these observations are manifold. First, the initial loading of the data is not parallelized in our implementation, so that according to Amdahl's Law a linear speed-up is precluded. And second, we often encounter situations where there is not enough work to keep all cores busy. In fact, many datasets entail a few search spaces that cause considerably more computational effort than the other search spaces. As a result, after a short processing time PYRO is often left with only a few hard-to-process search spaces and has to parallelize their traversal.

In contrast to the parallelization *among* search spaces, the parallelization *within* search spaces does not break down the profiling into independent tasks. Therefore, this mode of parallelization is often less effective depending on (i) how many launchpads are available in the search space; (ii) how much duplicate work is performed by the cores when two simultaneously processed launchpads are "close" to each other in the search space; and (iii) how much synchronization overhead is incurred for managing classified dependency candidates and launchpads of the search space. However, these issues are mitigated for datasets with many columns that are in particular need of effective parallelization (cf. Section 4.7.3): Such complex datasets tend to have more hard-to-process search spaces; also each search space is much larger, so that it does not easily run out of launchpads and duplicate work among cores is less likely.

Let us now proceed to PYRO's scale-out behavior, depicted in Figure 4.10(b). In this experiment, each worker is restricted to using only a single core. That way, the overhead for the distribution in comparison to the number of available cores is maximized. Also, PYRO cannot parallelize the traversal of a single search space, as our prototype can process each search space on a single worker only. As for the scale-in experiment, we observe that the scale-out speed-up is dependent on the complexity of the profiled dataset – datasets with many columns tend to allow for higher speed-ups. In simple terms, the

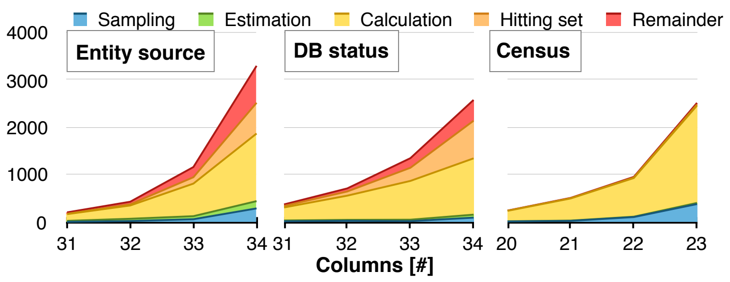
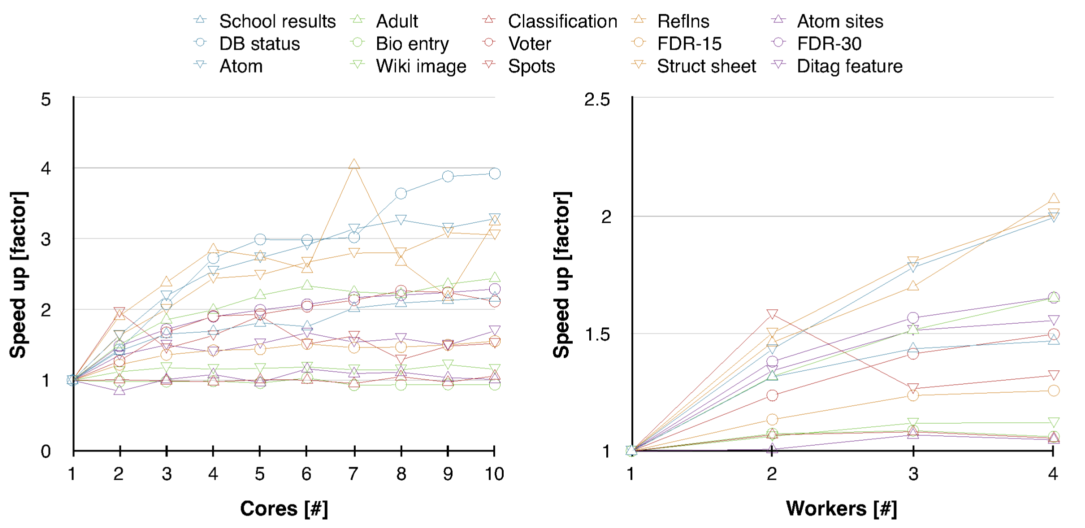


Figure 4.9: Runtime breakdown of PYRO for varying numbers of profiled columns ($e_{\max} = 0.01$).



(a) Scaling the number of cores (scale in). **(b) Scaling the number of workers (scale out).**

Figure 4.10: Scaling behavior of PYRO on different hardware deployments ($e_{\max} = 0.01$).

reason is again that complex datasets entail more work, which can in turn be distributed more evenly.

Apart from the similar scaling behaviors, the reader might notice that the DB status dataset, which benefits a lot from parallelized profiling in Figure 4.10(a), is not present in Figure 4.10(b). The reason is that in this configuration the workers send huge numbers of dependencies so rapidly to the master, that it has to drop incoming messages. In reaction, the Akka framework stalls the connection and PYRO fails. We are confident that this issue can be overcome by some engineering effort, but did not consider a technical solution in our implementation of PYRO.

Having discussed the effectiveness of scaling PYRO in and out, the question remains how well the two approaches work together. In pursuit of this matter, we executed PYRO with different numbers of workers, whereby each worker utilized all of its 10 cores. The results are essentially similar to those in Figure 4.10(b), but the speed-ups are slightly greater. In fact, we measured the highest speed-up for 4 workers with 10 cores compared to 1 worker with 1 core on the Reflns dataset. This speed-up amounts to a factor of 4 and is thus only an improvement of factor 2 over the speed-up when using only 1 core on each of the 4 machines (cf. Figure 4.10(a)). We conclude that PYRO's demand for computational resources is at some point saturated. As we explained above, that point of saturation is dependent on the profiled dataset, though.

4.7.5 Memory requirements

A particular concern of data profiling algorithms is their memory requirement, because computers vary greatly in the amount of available main memory. And after all, performance improvements might be just an advantage gained by using the available main memory more extensively. Thus, we compare Pyro's, Ducc/Dfd's, and Tane's memory requirements.

In detail, we execute all three algorithms with a maximum heap size of 32 MB and continuously double this value until the respective algorithm is able to process the given dataset. As can be seen in Figure 4.11, Pyro always requires the least amount of memory. In fact, we find Pyro to run out of memory only while loading the data. In contrast to Tane and Ducc/Dfd, Pyro pins only very few larger data structures in main memory, while managing PLIs and AS samples in caches that dynamically adapt to the amount of available main memory. This renders Pyro highly robust.

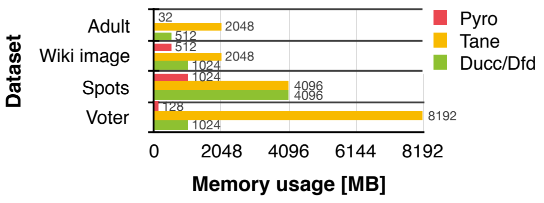


Figure 4.11: Memory requirements of Pyro, Tane, and Ducc/Dfd ($e_{\max} = 0.01$).

4.7.6 Sampling

Pyro's principal idea is to save costly dependency error calculations by estimating the error of dependency candidates. Even though Theorem 4.2 describes the accuracy of Pyro's estimator depending on the AS sample size, it is unknown, which accuracy works best to find the minimal dependencies in real-world datasets. To investigate this matter, we execute Pyro with different AS sample sizes on hard-to-profile datasets and display the results in Figure 4.12.

Apparently, not using AS samples (and thus error estimations) is always the worst option, while AS sample sizes of 1,000 and 10,000 work well on all tested datasets. That being said, using a tiny sample is consistently better than not using sampling at all to guide the search space traversal. For instance, on the Atom dataset, using a sample of only 10 agree sets reduces the number of dependency error calculations from 15,708 to 7,920. A sample size of 10,000 further reduces this value to 4,562. Note that the latter number leaves only little room for improvement: The Atom dataset has 1,582 minimal dependencies and about as many maximal non-dependencies, all of which need to be verified by an individual error calculation in the optimal case.

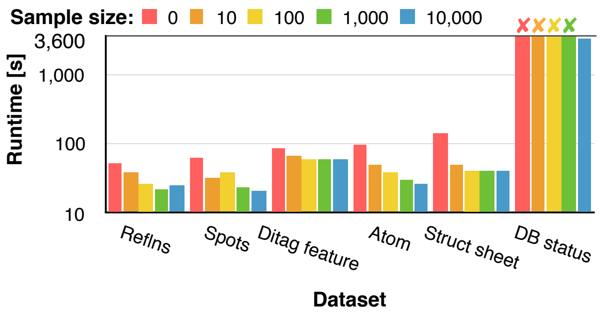


Figure 4.12: Comparison of Pyro's sampling strategies ($e_{\max} = 0.01$).

4.7.7 Exact dependency discovery

As we describe in Section 4.2, several techniques for the discovery of exact UCCs and FDs are not applicable to the partial case – in particular, the pruning of dependencies by counter-examples and the deduction of dependencies from individual dependency violations. In other words, there is a price to pay for relaxing the discovery of exact to partial dependencies and it is intriguing to determine that price.

To do so, we use PYRO to determine exact UCCs and FDs by setting $e_{\max} = 0$ (after all, exact dependency discovery is a special case of partial dependency discovery) and compare it to the competitors that are described in Section 4.7.1 and whose main focus is exact dependency discovery. Indeed, TANE can use an additional pruning rule for exact dependency discovery and FDEP's flaw regarding partial dependency discovery does not affect the exact case. Furthermore, we include HYFD [Papenbrock and Naumann, 2016] and HYUCC [Papenbrock and Naumann, 2017a], which are not applicable for partial dependency discovery. For those latter two algorithms, we use the original implementations in our experiments. Note that this comes with the caveat that HYFD and HYUCC waive potential to do common operations only once. In particular, they have to read the input data twice, which takes from about 1 second for small datasets to 30 seconds for the largest dataset. Because HYFD and HYUCC define a parallelization strategy, we report their runtimes for both the single-threaded and multi-thread mode of operation.

Figure 4.13 displays the results of this experiment. Interestingly, PYRO is a competitive algorithm for exact dependency discovery: For most datasets, it is among the fastest algorithm and in five cases PYRO *is* even the fastest algorithm. This is a remarkable result, as PYRO was not designed to discover exact dependencies. Nonetheless, on the datasets School results, Reflns, Atom sites, DB status, and Entity source, PYRO is outperformed by several factors by at least one competitor, usually HYFD and HYUCC. As can be seen in Table 4.1, these datasets have particularly many columns. Because HYFD's and HYUCC's distinguishing features are the focused search for dependency violations paired with a dependency deduction step and because they perform particularly well for the five above mentioned datasets, we conclude that these two features greatly improve the column scalability of exact FD and UCC discovery. Unfortunately, neither the focused violation search nor the dependency deduction are applicable to the partial case.

A second difference between exact and partial dependency discovery becomes apparent only when one compares the runtimes of Figure 4.13 to those in Figure 4.6: Observe that PYRO's runtimes for the dataset School results are orders of magnitudes faster in the partial case than in the exact case. And, vice versa, PYRO's runtimes for the datasets DB status, Entity source, and Census are orders of magnitudes faster in the exact case than in the partial case. These dramatic runtime differences can be explained by the location of exact and partial dependencies in the search space. The search spaces for exact and partial FDs and UCCs can be modeled as a power set lattice, such as in Figure 4.4 (page 93). Now we can color each node in this search space according to its dependency error. Figure 4.14 depicts such a coloring in a highly abstract manner.

In this figure, there are two key points to notice: At first, dependency candidates with a high error are "below" those with a low error due to the monotonicity of the error

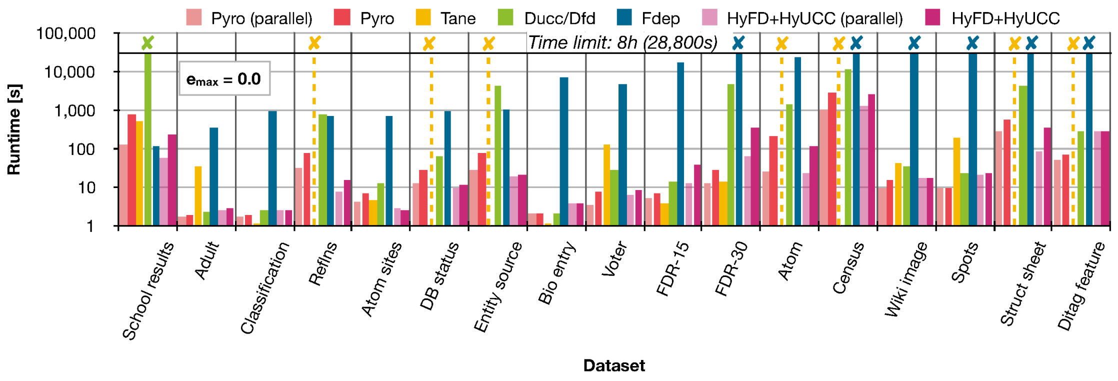
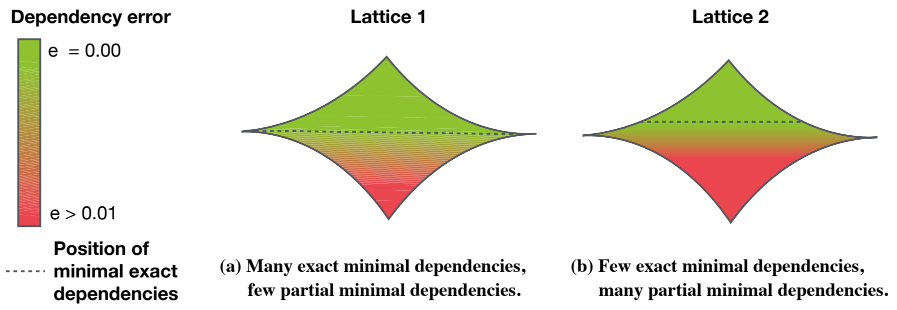


Figure 4.13: Comparison of Pyro, Tane, Ducc/Dfd, Fdep, as well as HyFD and HyUCC for exact dependency discovery. Crosses signal that an algorithm ran out of main memory or processing time (at most 8 hours).



Dependency error
e = 0.00
e > 0.01
Position of minimal exact dependencies
Lattice 1
Lattice 2
(a) Many exact minimal dependencies, few partial minimal dependencies.
(b) Few exact minimal dependencies, many partial minimal dependencies.

Figure 4.14: Location of exact and partial dependencies in two different lattices.

measures from Definition 4.1. And second, the vast majority of dependency candidates are situated in the middle layers of the search space, because the power set of an n-ary set comprises $\binom{n}{k}$ k-ary sets, which reaches its maximum for $k = \lfloor \frac{n}{2} \rfloor$. Depending on how a search space is colored, it may have many more exact than partial *minimal* dependencies (as indicated in Figure 4.14(a)) or other way around (as indicated in Figure 4.14(b)). Indeed, for the above mentioned datasets we observe vastly different numbers of partial and exact FDs and UCCs. Because any algorithm is bound to be at least of polynomial time complexity in its output (and Pyro is indeed highly output sensitive as shown in Section 4.7.3), discovering partial dependencies can be either more facile or more difficult than exact dependency discovery – depending on the dataset.

4.7.8 Ranking

Table 4.1 suggests that certain datasets entail an unwieldy number of partial dependencies. While some use cases, such as query optimization and feature selection (see Section 4.1), can make use of all those dependencies regardless of their semantics, other use cases, such as data cleaning, require semantically meaningful dependencies. This raises the question whether users can be supported in finding relevant dependencies.

At this point, ranking schemes can be a great help to order the discovered dependencies by their estimated interestingness. Indeed, several measures have been described in the literature, in particular for FDs [e. g., Dalkilic and Roberston, 2000; Giannella and Robertson, 2004; Pfahringer and Kramer, 1995; Piatetsky-Shapiro and Matheus, 1993; Sánchez et al., 2008]. Each of these measures rates dependencies w. r. t. a certain characteristic or application, such as the utility for data compression. To rank our PFDs, we developed a novel ranking scheme, specifically tailored to determine the *interestingness* of PFDs. This scheme takes into account how likely a PFD could have occurred just by chance and how accurate that PFD is.

Concretely, for a PFD $X \rightarrow A$ our measure models the overlap in tuple pairs agreeing in either X or A with the hypergeometric distribution and determines how many standard deviations the actual overlap and the mean overlap are apart – let h denote this value. Intuitively, if h is large, X and A strongly correlate, because then the tuple pairs agreeing in X and A coincide to a great extent. Additionally, we consider the conditional probability that a tuple pair agrees in A given it agrees in X, which is similar to the confidence of association rules – thus let c denote this probability. Now, we assign each PFD $c \cdot \operatorname{sgn} h \cdot \log |h|$ as ranking score, where sgn is the signum function.

We applied this ranking to the Voter dataset, for which we have column headers and can thus understand the column semantics, and report the highest ranked PFDs: *(1+2)* The PFDs voter id $\rightarrow$voter reg num and voter reg num $\rightarrow$voter id uncover that voter id and voter reg num are equivalent voter identifications. Note that they are not keys, though. *(3)* The PFD zip code $\rightarrow$city reflects a rule of the data domain and lends itself to data cleaning. *(4)* The PFD first name $\rightarrow$gender also exposes a latent rule of the data domain, thereby supporting knowledge discovery. These examples demonstrate that it is possible to quickly draw relevant dependencies from large dependency sets. However, notice that, although often only a small share of the discovered dependencies is meaningful, it is nonetheless necessary to discover *all* dependencies: Ranking criteria, such as the one presented above, do not possess the necessary monotonicity that would allow to leverage them for pruning during the discovery process (cf. Section 4.1).

As a side note, the above proposed ranking scheme can also be adapted to n-ary PUCCs for $n \geq 2$: As an example, the ternary PUCC ABC could be interpreted as three "anti-PFDs" $AB \not\rightarrow C$, $AC \not\rightarrow B$, and $BC \not\rightarrow A$. Each such anti-PFD basically states that if a tuple pair agrees in its LHS, then it should not agree in its RHS.[6] Now to rank PUCCs, one can adapt the above described correlation and accuracy calculation of PFDs for anti-PFDs and aggregate the resulting interestingness scores for all anti-PFDs entailed by a PUCC, e. g., by taking the maximum interestingness score. Having said that, ranking dependencies is not the focus of this work, so that we leave a thorough analysis of these proposed methods as well as a comparison to related measures for future work.

4.8 Summary

This chapter introduced PYRO, an efficient and scalable algorithm to discover all PFDs and all PUCCs in a given dataset. PYRO uses a separate-and-conquer discovery strategy that quickly approaches minimal dependencies via samples of agree sets, efficiently

[6]Interestingly, this view reconciles the error measures for PUCCs and PFDs from Definition 4.1.

verifies them, and effectively prunes the search spaces with the discovered dependencies. Furthermore, PYRO has a small memory footprint and benefits from parallel execution. In our evaluation with state-of-the-art dependency discovery algorithms, we found PYRO to be up to 33× more efficient than the best competitor. Parallelization regularly allows for even greater speed-ups.

As future work, it is particularly intriguing to investigate how dependency interestingness measures, such as the one proposed in this chapter, could be leveraged by PYRO to prune the search spaces of profiled datasets. Because oftentimes only a few among all discovered PUCCs and PFDs are really interesting, this might considerably improve the efficiency and scalability of our algorithm, whose performance appears to depend on the number of discovered dependencies in particular.

Chapter 5
Data Anamnesis with Metacrate

In the above chapters, we present algorithms to profile datasets for several types of data dependencies. While it is crucial to design efficient and scalable profiling algorithms, data profiling is of course not an end in itself. Rather than that, the discovered structural metadata is supposed to support various kinds of data management scenarios. Therefore, we provided references to, as well as novel ideas for such management applications throughout the previous chapters.

In particular, we proposed the notion of a *data anamnesis* [Kruse et al., 2016b] in Section 1.1 as a data management problem and as a motivation as to why data profiling (and dependency discovery in particular) should be considered in data-related projects. Recall that in a data anamnesis a user seeks to obtain a basic understanding of an uncharted dataset, which data it contains, how to query it, and whether it is useful for the project at hand. The data anamnesis should be based on the data itself (or rather its structural metadata) to avoid any dependence on external artifacts (e. g., a documentation), which might be incomplete, incorrect, outdated, or simply not available. Structural metadata lends itself to this task, not only because it can expose latent properties of a dataset, but also it is usually much smaller than the original data, thereby potentially allowing fast analyses for interactive workflows. In this chapter, we seek to substantiate this idea of a data anamnesis with first-hand experiences and to investigate if and to what extent a data anamnesis can be carried out with structural metadata collected by our data profiling algorithms.

With this goal in mind, this chapter makes two principal contributions. At first, we present our system Metacrate [Kruse et al., 2017a] along with the specific challenges, that it tackles, in Section 5.1. In few words, Metacrate is a database system for structural metadata. It allows to organize, analyze, and visualize data profiling results – hence, it serves us well as technical basis for data anamneses. As a second contribution, we describe a case study, in which we carry out actual steps of a data anamnesis on the real-world dataset of the MusicBrainz project [Swartz, 2002]. In detail, we profile this dataset to reconstruct its schema in Section 5.2; we explore ways to enable an easier understanding of the dataset contents in Section 5.3; and we assess the quality of the dataset schema in Section 5.4. Notice, it is not the intention of this chapter to develop novel data anamnesis techniques. For that matter, our data anamnesis adopts and adapts existing algorithms and measures where applicable. Instead, we aim to establish an understanding of the practical applicability of data profiling for data anamnesis – its prospects and its limitations. This second contribution is based on [Kruse et al., 2016b]. Eventually, we summarize this chapter in Section 5.5.

5.1 A Database System for Structural Metadata

When discussing structural metadata (or data profiles[1]) in the previous chapters – in particular data dependencies – we mostly regard them *in the context of their search spaces*, i. e., we focus on their interaction with other dependency candidates. While this point of view is the basis for many discovery algorithms, users in a data management scenario might prefer a different approach: For them, it might be more appropriate to regard data profiles *as annotations to the original data*, but also *in the context of other profiles describing the same portion of the data*. Indeed, the key concerns of data management practitioners are supposedly how to discover relevant metadata, how to integrate different types of metadata, and – ultimately – how to refine and abstract the metadata to insights that serve their very project, be it a data cleaning, a data integration, or a reverse engineering project. Henceforth, we also assume this latter viewpoint.

While there are several tools and approaches that incorporate data profiles for specific data management scenarios [e. g., Andritsos et al., 2002; Dasu et al., 2002; Fernandez et al., 2016], none of them offers a general solution for the management of metadata. Instead, at their core all those tools need to solve the common problem of how to store, organize, query, and integrate data profiles. Given the success of database systems to solve these tasks for relational data, it seems worthwhile to devise a database system for structural metadata that approaches above mentioned problems in a principled manner and that can replace ad-hoc solutions in specific data management tools. In fact, data management tools could then be easily written *on top of* such a "meta-database".

5.1.1 Challenges

We identify three key challenges that a system for the management of structural metadata needs to address, namely *volume*, *variety*, and *integration*. Let us outline each of these challenges in the following.

Volume

Traditionally, one might think of metadata as tiny, especially in comparison to the original data that is referenced by that metadata. Surprisingly, this is not necessarily the case for structural metadata. For instance, Papenbrock and Naumann [2016] found a specific dataset of only 2.4 MB to entail over 2.5 million FDs, which required already

[1]Recall from Chapter 1 that we use the terms "stuctural metadata" and "data profiles" synonymously.

4 GB of main memory in their experimental setup. Another example where data profiling results become huge can be found in this thesis in Section 3.2, where we describe that an RDF dataset of 72,445 triples entails 1.3 billion (non-pertinent) CINDs. While these are certainly extreme cases, they have a common cause: dependency search spaces quickly surge, often exponentially, as the number of considered attributes grows (and this issue further intensifies if the dependency type admits conditions, too). And larger search spaces usually contain more dependencies. This is a general issue, even if the outcome is not always as overwhelming as in the two above examples.

Another factor that potentially causes an abundance of data profiles is the size of the base data, in particular its schema. In other words, if a database has a complex schema, there are more structural entities that can be profiled. Complex (relational) schemata can contain hundreds to thousands of tables, each with tens to hundreds of columns. What is more, modern information systems often aggregate multiple databases – a trend that is somewhat reflected in the popular notions of *polyglot persistence* and *data lakes*. Nonetheless, the larger and the more unwieldy a data landscape is, the more pressing is the need to profile it for data management purposes.

Last but not least, there is a vast array of different data profile types. For instance, in a recent survey Caruccio et al. [2016] described 35 variations of (partial) FDs. Furthermore, many types of structural metadata can be parameterized, such as the number of buckets in a histogram or the error threshold for partial UCCs. Hence, there is essentially no limit on how many data profiles can be ascertained per schema element. On the face of the many factors pertaining to the volume of structural metadata, it seems only reasonable to assume that structural metadata cannot always be (efficiently) stored in and queried from main memory or flat files.

Variety

The above discussed variety of data profile types does not only have implications on the metadata volume, but also raises the question for a reasonable (logical) data model as well as a query language for structural metadata. Not only must the data model and query language be capable to serve a wide range of data profiles types (e. g., an n-dimensional histogram and a CIND are quite distinct in terms of their representations as data structures and in terms of their applications for data management), but they must also be extensible, as new data profile types or variations of existing types should be supported. At this point, the reader might wonder whether a traditional relational database provides the required flexibility. In our experience, it is possible but utterly impractical: most data profile types require multiple tables; a universal schema to accommodate all data profile types does not exist, so it must be continuously extended; and querying the database with SQL is complex, tedious, and error-prone – let alone performing complex analyses.

Integration

Let us emphasize as a third point, that the various data profiles must be *integrated.* To explain what that means, recall from above that a data management practitioner might view data profiles as annotations to the base data. As a data profile usually characterizes the underlying data w. r. t. one specific property, such as some statistic or relationship, it is likely that the practitioner needs to combine several types of data profiles to accomplish her task at hand. As an example, assume that this task is foreign key discovery. Rostin et al. [2009] propose to approach this task by combining INDs with several features (which can be drawn from other data profiles) and apply a machine learning model to predict, which INDs instantiate foreign keys. Concretely, this means that INDs need be combined with other data profiles that describe the dependent and referenced columns of those INDs (e. g., the number of `NULL` values or whether those columns form UCCs). What is more, some relevant data profiles, such as the number of tuples, might annotate tables rather than columns; it should be easily possible to include such data profiles, too. While this is only one example, we find in our data anamnesis that virtually any metadata-driven task requires a combination of various data profiles. In consequence, we consider the integration of data profiles to be a first-class requirement.

5.1.2 Architecture

Having laid out the motivation for and challenges of devising a database for metadata, let us describe our respective solution for relational databases: the open-source system METACRATE.[2] Generally speaking, METACRATE is a hub between data profiling algorithms (which produce the data profiles) and metadata-driven algorithms (which consume them). In addition to that, (i) we enable seamless storing of any kind of data profile; (ii) we expose the data profiles in a well-organized, integrated fashion; (iii) we provide a query layer to explore and exploit the contents of METACRATE; (iv) and we provide a library of basic data management and analysis algorithms to quickly build workflows on top of METACRATE and also as a proof of concept for our design. In summary, METACRATE not only seeks to answer the research question of whether it is possible to build a system that accommodates the above presented challenges; METACRATE also intends to contribute to the community a basis to facilitate the research and development of metadata-driven applications – especially w. r. t. advanced types of structural metadata, such as data dependencies, which is a particular blind spot of many industry tools [Eschrig, 2016]. For this purpose, METACRATE is tightly integrated with METANOME [Papenbrock et al., 2015a], which provides a wide host of cutting-edge algorithms[3] to discover data dependencies (including the algorithms presented in this thesis).

Let us now outline METACRATE's architecture and present its principal components and design decisions. Consider Figure 5.1 for that purpose. As can be seen, METACRATE interacts with *data profiling tools* that produce data profiles and with *metadata-driven*

[2]See `https://github.com/stratosphere/metadata-ms` (accessed November 28, 2017).

[3]See `https://hpi.de/naumann/projects/data-profiling-and-analytics/metanome-data-profiling/algorithms.html` (accessed November 28, 2017)

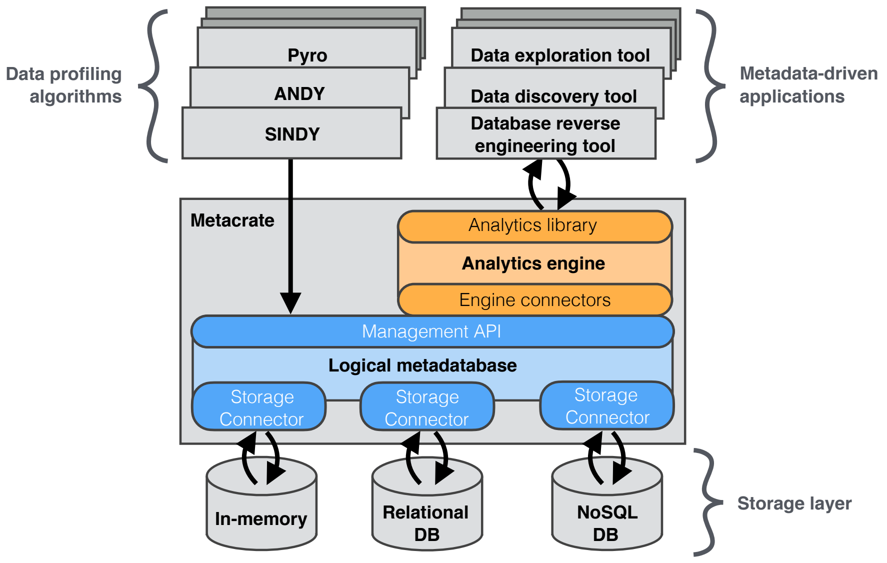


Figure 5.1: Overview of Metacrate.

tools that need access to the data profiles to accomplish their various tasks. In this chapter, we particularly focus on metadata-driven *data management tools*, e. g., for database reverse engineering or data discovery tools. In this ecosystem, Metacrate acts as a broker between those two parties, thereby taking care of storing and organizing the profiles (*logical metadatabase*) and offering them via a query interface (*analytics engine*).

Logical metadatabase

Metacrate's logical data model governs the principal manner in which data profiles are stored and organized. On the face of the variety of structural metadata, it aims to strike a balance: On the one hand, it should be abstract enough to deal with all kinds of data profiles and, on the other hand, it should be specific enough to be queried in a meaningful way. Figure 5.2 depicts our choice of such a logical data model as an entity-relationship diagram. On a high level, it can be separated into three parts parts, the *schema elements*, the *data profiles*, and *execution-related data*. Let us briefly iterate these parts.

The core entity to describe schema elements is called *target*, which is simply an abstraction for any kind of structural element in a relational database.[4] Concretely, we consider three types of targets. Of course, relational *tables* are targets. Those comprise a number of *columns*, which are also targets. Finally, we consider as top-level targets *schemata* as any collection of tables, which could be a complete relational database, a table space or schema within the database, or just a directory with CSV files. Furthermore, each target has a (not necessarily unique) name and an optional location. A distinctive feature of targets is the *ID* attribute, which serves several purposes. At first,

[4]The idea behind the naming *target* is that these schema elements are referenced (or targeted) by data profiles. The reader could also think of them as *schema elements*, but should beware of any confusion with *schemata*.

it resolves potential naming ambiguities among targets. Second, it is a highly compact representation of targets, which allows a compact representation of data profiles (because they employ those IDs when referencing a target), which in turn allows to analyze data profiles more efficiently. As a third and final point, the IDs also embed the inclusion relationship among schemata, tables, and columns that is depicted in Figure 5.2. Inspired by the hierarchical structuring of IP addresses, the leading bits in a target ID identify the schema, the middle bits identify the table, and the trailing bits identify the column. As a result, we can tell (i) which type of target an ID describes (i. e., schema, table, or column) and (ii) how the targets of two IDs relate (e. g., one includes the other), without having to consult some mapping table, just by observing those IDs.

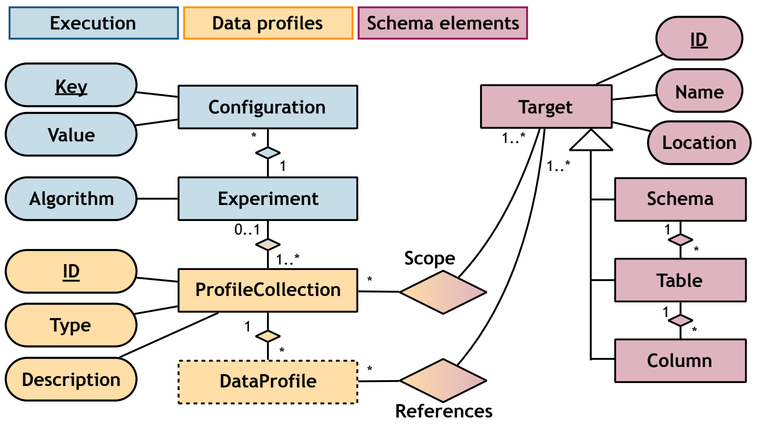


Figure 5.2: METACRATE's logical data model.

Let us move on to the data profile and execution parts of the logical schema. As shown in Figure 5.2, data profiles are organized in *profile collections*, thereby providing a natural means of organizing data profiles. In addition to that, the profile collections have various properties that users can use to discover them. In particular, each profile collection stores a specific *type* of data profiles and has a *scope* that declares which parts of the schema are described by those data profiles. These properties already allow for user queries, such as: *"Retrieve all profile collections with* INDs *for schema R."* Furthermore, a profile collection can be associated to a data profiling *experiment* that can further describe the contents of that collection. For instance, when a user executed the SANDY algorithm to discover partial INDs (see Chapter 2.3), it is important to know the configured error threshold in that execution of SANDY. As a further result, it is possible to have different "versions" of data profiles, e. g., when comparing the results of data profiling algorithms or when executing algorithms with different parameterizations.

Furthermore, the organization of data profiles in profile collections fulfills a second important purpose: This approach allows to model the *absence* of a certain data profile, which is particularly relevant for data dependencies. As an example, assume we have a profile collection with UCCs and a profile collection with FDs, both having the same table as scope. This allows us to reason, for which FDs there is an implying UCC and for which FDs there is no such UCC. Amongst others, this is important to discover FDs that might be employed for schema normalization [Papenbrock and Naumann, 2017b]. Note that this would not be possible without scopes and profile collections, as otherwise it would not be possible to tell whether a dependency is absent in METACRATE because it indeed does not hold on the data, or simply because it has not been stored in METACRATE.

It remains to describe how we represent actual *data profiles* in our logical data model. As indicated by the dashed line in Figure 5.2, we virtually do not make any restrictions on the structure of the data profiles. In fact, METACRATE technically allows data profiles to be any Java class; it is exactly this point, where we account for the great variety of data profiles. There are only two requirements that METACRATE imposes on the data profiles: (i) all data profiles within a profile collection should be of the same type; and (ii) whenever a data profile references a target (i. e., a schema element), it must do so via its ID. We impose these requirements, as it facilitates the development of applications on top of METACRATE. Specifically, the above described integration of data profiles and the declarative retrieval of profile collections are enabled by these design decisions.

Applications can interact with this logical data model via a *management API*, e. g., to create new schemata or profile collections. Nonetheless, an actual storage backend is needed to ultimately persist the inserted metadata. For this purpose, METACRATE implements three default storage backends: (i) an in-memory backend for simple ad-hoc tasks; (ii) a backend for relational databases as a persistent option; and (iii) a backend for Apache Cassandra as a scalable option (recall that structural metadata can indeed be voluminous).

Analytics engine

METACRATE complements its storage component with an analytics engine, which can be seen as the counterpart of query engines in relational databases. However, METACRATE does not adopt a closed algebra to formulate analytics, because we neither know the various data profile types apriori (cf. Figure 5.2) nor the actual analytical tasks to be performed by metadata-driven applications. And even if we knew all types of data profiles in advance, we learned that formulating SQL queries, for instance, to analyze data profiles is cumbersome and error-prone. Instead, METACRATE employs a Scala-based data flow API that supports user-defined functions and whose operator set is extensible. As we demonstrate in our data anamnesis in the following sections, this design choice allows to formulate rich analytics in a concise manner. We further provide an *analytics library* for structural metadata that offers common functionality for various data management tasks ([e.,g., Andritsos et al., 2004; Dasu et al., 2002; Fernandez et al., 2016; Kruse et al., 2016b; Rostin et al., 2009; Yang et al., 2009]). This library facilitates the development of data management tools and allows for complex ad-hoc queries to METACRATE. In fact, the library functionality is exposed via a Scala API that can be seamlessly weaved in with regular queries. Note that we present several concrete examples for METACRATE queries in the following sections.

Internally, METACRATE executes those queries on the cross-platform data processing system RHEEM [Agrawal et al., 2016; Kruse et al., 2018], which also is an open-source system[5] that we developed in a collabration with the Qatar Computing Research Institute.

[5]See `https://github.com/rheem-ecosystem/rheem` (accessed November 28, 2017).

In detail, Metacrate feeds schema elements and profile collections to Rheem using dedicated *engine connectors*. Rheem then selects an actual data processing platform to execute the query on, e. g., Java Streams[6] or Apache Spark[7]. In combination with the various storage engines, Metacrate can therefore either employ light-weight components for simple scenarios or make use of distributed components to scale to complex scenarios with millions and more of data profiles.

5.1.3 Appliations

As described above and as shown in Figure 5.1, Metacrate is a backend system to support metadata-driven applications, such as data managment. In the following sections, we employ Metacrate to carry out a data anamnesis. Although it is not our intention to develop a dedicated, easily operated data anamnesis application, we are in need of a concrete application to interact with the Metacrate backend. Therefore, we adopt an approach that has been recently very popular among data scientists: we integrate Metacrate with Jupyter notebooks.[8]

In few words, (Jupyter) notebooks are web-based applications that allow users to write ad-hoc code, usually small snippets, to analyze a dataset. The code is then executed on some remote, powerful computer and the results are reported back to the user, often with some kind of visualization. This approach is a good fit for our data anamnesis: We can write ad-hoc queries to Metacrate to perform simple tasks of the data anamnesis, or use Metacrate's analytics library for the more complex tasks. To complement this idea, Metacrate's integration with Jupyter provides a set of visualizations, many of which are dedicated visualizations of data profiles. An example of what this Jupyter integration looks like can be found in Figure 5.3. This screenshot depicts a Metacrate query that simply aggregates the number of columns in each table of some schema. Such queries and visualizations form the basis of our following data anamnesis.

5.2 Schema Discovery

Having conveyed an overview of Metacrate, we can now describe our data anamnesis case study. Let us initially remark that all the queries and analyses of this case study are captured in a Jupyter notebook that can be downloaded from the Metacrate website along with several other artifacts, e. g., a demonstration video: `https://hpi.de/naumann/projects/data-profiling-and-analytics/metacrate.html`

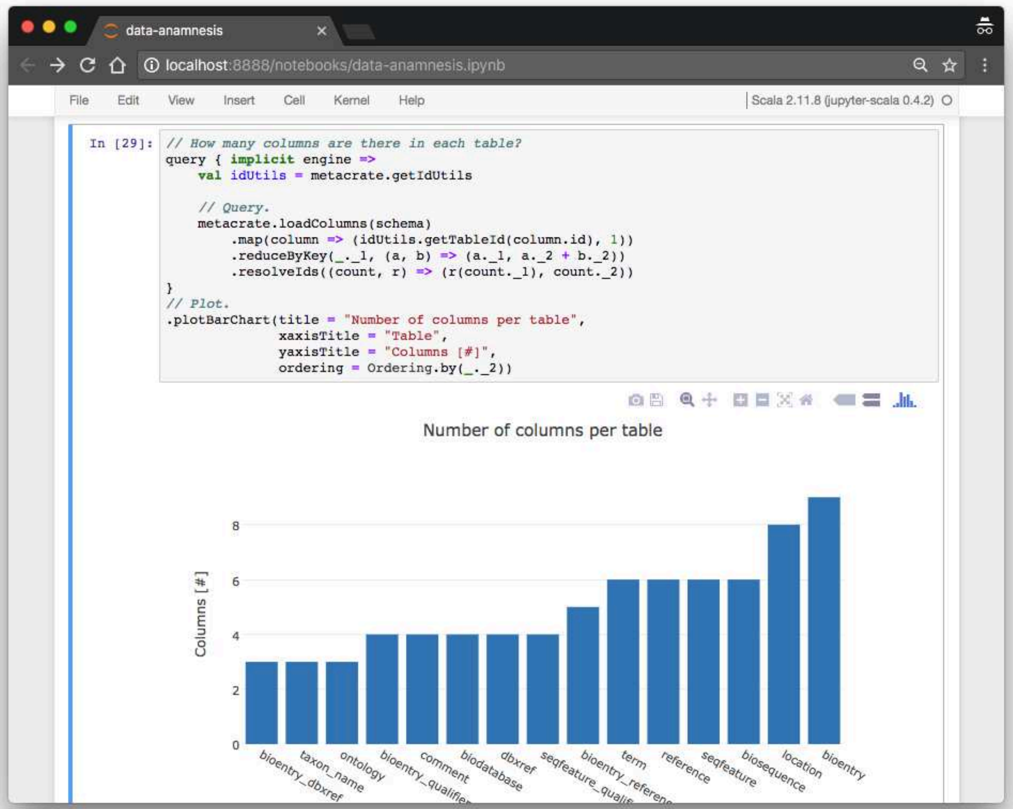
To make our study insightful, we chose a complex dataset, that cannot easily be apprehended by eye-balling: The MusicBrainz dataset[9] is an open, community-driven mu-

[6]See `https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html` (accessed November 28, 2017).

[7]See `http://spark.apache.org/` (accessed November 28, 2017).

[8]See `https://jupyter.org`; (accessed November 28, 2017).

[9]The MusicBrainz dataset can be downloaded from `ftp://ftp.eu.metabrainz.org/pub/musicbrainz/data/fullexport/`. However, this mirror regularly provides new versions of the dataset, while older versions are removed. The here described case study is based on the dataset version 20171125-001448.



```
In [29]: // How many columns are there in each table?
         query { implicit engine =>
             val idUtils = metacrate.getIdUtils

             // Query.
             metacrate.loadColumns(schema)
                 .map(column => (idUtils.getTableId(column.id), 1))
                 .reduceByKey(_._1, (a, b) => (a._1, a._2 + b._2))
                 .resolveIds((count, r) => (r(count._1), count._2))
         }
         // Plot.
         .plotBarChart(title = "Number of columns per table",
                       xaxisTitle = "Table",
                       yaxisTitle = "Columns [#]",
                       ordering = Ordering.by(_._2))
```

Figure 5.3: Screenshot of METACRATE running in a Jupyter notebook.

sic encyclopedia that provides discographical data for a large number of musical works. The database dump of the MusicBrainz dataset is fairly extensive: It comprises 260 tab-separated files, which together account for 51 GB of data. While we discovered an incomplete schema description on the MusicBrainz website[10] as well as several schema definition SQL files in a source code repository[11], the dump itself is not accompanied by a schema. In fact, it is easy to imagine (and frequently the case), that such additional artifacts are not available for a given dataset. And even in the presence of such artifacts, there is no guarantee that they are correct, complete, sufficiently specific, and consistent with the acquired data – especially if the data and the additional artifacts reside in different repositories. In consequence, it is reasonable to assume that oftentimes the understanding of the dataset has to be driven *by* that very dataset – which is the key principle of data anamnesis (see Section 1.1).

As a starting point of our data anamnesis, we profiled the data with METANOME. We used a cluster that consists of a master node (IBM xServer x3500 with two Intel Xeon

[10]See `https://musicbrainz.org/doc/MusicBrainz_Database/Schema` (accessed November 28, 2017).

[11]See `https://github.com/metabrainz/musicbrainz-server/tree/master/admin/sql` (accessed November 28, 2017).

X5355 Quadcore (2.66 GHz, 8 MB cache), 32 GB RAM, and a 2.7 TB hard disk) and four PowerEdge R430 workers with an Intel Xeon E5-2630 v4 (2.2 GHz, 10 cores, 20 threads, 25 MB cache), 32 GB RAM, and 2 hard disks (7.2 kRPM, SATA) running Ubuntu 16.04.1 and Oracle JDK 1.8.0_112 with a maximum heap size of 24 GB. To control the overall duration of the profiling, we limited each individual algorithm execution[12] to a maximum time of 8 hours.

Table 5.1 summarizes the profiling process: It lists the aggregated runtimes for each algorithm (including failed runs); it reports the number of failed (and total) executions of each algorithm; and it displays the number of discovered data profiles of each data profile type. The algorithms Andy and Pyro are described in the Chapters 2 and 4 of this thesis and the algorithms HyFD [Papenbrock and Naumann, 2016], HyUCC [Papenbrock and Naumann, 2017a], and Scdp are further algorithms from the Metanome project. Last but not least, the reservoir sampling is provided by Metacrate. All these algorithms yield a somewhat moderate number of data profiles.[13] However, it is important to note that, with the exception of Andy and the reservoir sampling, no algorithm could profile the whole dataset, e. g., due to too long running times (HyFD) or exhausted main memory (HyFD, HyUCC, Pyro, Scdp). In consequence, we have to carry out our data anamnesis with incomplete structural metadata.

Data profile type	Algorithm	Runtime	Failures	Data profiles
Inds	Andy	0:16:06	0/1	74,010
Ars				123,194
Fds	HyFD	33:14:58	35/260	4,328
Uccs	HyUCC	3:52:05	2/260	538
Partial fds	Pyro	3:44:41	10/260	4,544
Partial uccs				789
Column statistics	Scdp	3:30:50	1/260	1,564
Samples	Reservoir sampling	0:23:29	0/260	260
All		45:02:09	48/1,301	209,227

Table 5.1: Insights on the profiling process of the MusicBrainz dataset.

Also notice that neither the number of failures nor the runtimes of the algorithms are meant to be compared. As a matter of fact, every algorithm solves a distinct problem, operates on different hardware resources, and fails on different input files. That being said, we still provide these data points to convey an idea of how long it takes state-of-the-art algorithms to profile a larger dataset and how robust they are.

Having comprehensively profiled the MusicBrainz dataset, we can now proceed to describe the reconstruction of its schema. Concretely, in our case study we attempted to infer (i) data types, (ii) `NOT NULL` constraints, (iii) primary keys, and (iv) foreign keys.

[12]Apart from Andy, which profiles the complete dataset in a single execution, all other algorithms conduct one execution per table.

[13]It shall be remarked that without Andy's ars we would have to cope with millions of inds.

To assess the quality of our results, we compared them to the constraints and data types defined in the above mentioned SQL files. Unfortunately, those SQL files do not cover all 260 tables, so that we restrict the comparison to the 209 covered tables only.

Column descriptions

To reconstruct the data types and `NOT NULL` constraints, we used straight-forward approaches. In fact, the SCDP algorithm mentioned in Table 5.1 outputs for each profiled column the most specific data type that accommodates all values in that column, which we directly proposed as data type. Furthermore, SCDP outputs the number of `NULL` values for each column and we simply proposed a `NOT NULL` constraint for all columns without any `NULL` values.

According to the definitions in the SQL files, our data type inference determined the correct data type for 497 columns (37.0 %) and determined a more specific data type (e. g., `SMALLINT` instead of `INT`) or a closely related data type (e. g., `VARCHAR(32)` instead of `CHAR(32)`) for 697 columns (51.9 %). For the residual 150 columns (11.2 %) we failed to predict the data type mainly because 144 columns contain only `NULL` values or empty strings. The other 6 failures can be attributed to rather rare data types (e. g., `CUBE`) that are not supported by SCDP and to the algorithm failure shown in Table 5.1: SCDP only failed on the edit_data table, which alone comprises 29 GB of data.

Furthermore, we proposed 1,112 `NOT NULL` constraints of which 836 are also declared in the SQL files, yielding a precision of approximately 0.752. For that matter, we also missed 2 `NOT NULL` constraints because of the one failure of SCDP on the edit_table, so that we obtained a recall value of 0.997.

In summary, we find that our simple data-driven constraint inference yielded similar, but tighter constraint sets than those specified by the schema designers: Our proposed data types *approximate* the originally specified data types quite well (with many of them being even more specific) and the proposed `NOT NULL` constraints essentially form a superset of the originally specified ones. This is completely fine if the inspected dataset will not change anymore. In contrast, if the data is going to change, e. g., because the user regularly downloads the latest version of the MusicBrainz database, then the proposed constraints might need to be revised. This is because the data-driven approach proposes constraints that accommodate *one specific* database instance, while a human schema designer can also incorporate assumptions on *all* possible database instances when declaring constraints.

Primary and foreign keys

Designating primary and foreign keys is rather different from designating data types and `NOT NULL` constraints: Instead of processing every column of the dataset individually, we need to consider additional restrictions. In fact, there can be only one primary key per table and each column may at most reference one other column, which in turn needs to be a primary key.[14] Furthermore, primary and foreign keys are highly critical in relational datasets, as they essentially define how tables relate and can be joined meaningfully.

To accommodate these additional requirements, we approached the designation of primary and foreign keys as machine learning problems: We assessed for each discovered UCC and IND how likely it is to be a primary or a foreign key, respectively, based on features of the dependencies themselves as well as other structural metadata. Access to the original dataset was not required, though. We would then select mutually consistent primary and foreign keys with a high assessed probability.

Concretely, to assess how likely a UCC constitutes a primary key, we adapted features described by Papenbrock and Naumann [2017b] to build a logistic regression model. Similarly, we used features adapted from Rostin et al. [2009] and Zhang et al. [2010] along with logistic regression to determine potential foreign keys among the INDs. The concrete features are listed in Table 5.2. In addition, we narrowed our considerations only to those INDs whose referenced attributes constitute a primary key. In consequence, we selected the primary keys before the foreign keys. In summary, we designated our primary and foreign keys by (i) considering those candidates for which we assessed a probability of at least 0.5 to constitute a key; (ii) sorting them by this probability in descending order; and (iii) promoting one after another as a key unless it would conflict with a previously designated key. Note that this classification algorithm is part of METACRATE's analytics library.

Our machine learning approach raises two questions. At first, it is important how well our features and classifier (logistic regression) can fit the problem at hand in the first place. And second, it is interesting to learn whether we could find a general model that fits most datasets well. To find indications for these questions, we designated the primary and foreign keys with two different models. One model was learned on a completely unrelated, biological dataset with the BioSQL schema[15] and is referred to as *foreign model*; the other model was learned on the MusicBrainz dataset directly and is referred to as *local model*. Let us also remark that we used oversampling while learning those models to even out the highly skewed distribution of keys and non-keys among all UCCs and INDs.

The feature weights of both the local and foreign models are compared in Table 5.2. A manual inspection shows that the primary key models are somewhat similar – their cosine similarity amounts to 0.98. This suggests that the primary key models are indeed general and not overfitted to a single dataset. In contrast, the two foreign key models are quite different and have a cosine similarity of only 0.35. In particular, the feature weights pertaining to table and column names are contradictory among the two models, which might reflect different naming schemes on the respective training datasets.

These hypotheses are also confirmed when comparing the predicted primary and foreign keys to those defined in the SQL files, as displayed in Table 5.3. Both the foreign and the local model predict the primary keys pretty accurately (and in fact make the

[14]The SQL standard also admits foreign keys that reference columns with a `UNIQUE` constraint, but in our experience such constructs are so rare that we decided to exclude them.

[15]See `http://biosql.org/` (accessed December 12, 2017).

Primary key assessment	Weight	
Feature	Foreign model	Local model
Fill status	+7.3	+233.2
Arity	-2.3	40.9
Maximum value length	-51.6	-4,977.1
Columns to the left	-8.8	-580.0
Gaps between attributes	-8.8	-191.6
Offset	-1.7	+221.2
Foreign key assessment	Weight	
Feature	Foreign model	Local model
Unorderedness of referenced columns	+13.4	-12.9
Gaps between dependent columns	-1.0	+7.0
Table name overlap (w/o equality)	-430.6	+4,984.2
Column name overlap	+395.3	-10,168.5
Distributional distance	-337.6	-5,843.7
Value coverage of the referenced columns	+126.4	+5,722.3
Offset	-865.6	-12,204.4

Table 5.2: Features and weights for the primary and foreign key assessment.

Constraint type	INDs/UCCs	Gold	Model	Precision	Recall
Primary keys	445	209	Foreign	0.99	0.99
			Local	0.99	0.99
Foreign keys	52,482	387	Foreign	0.50	0.00
			Local	0.34	0.34

Table 5.3: Features and weights for the primary and foreign key assessment.

same three mispredictions and miss the same two primary keys) with the caveat that in 13 cases the selected UCCs form a subset of the actual primary key. In fact, for those tables the actual primary keys do not constitute minimal keys.

In contrast to that, the foreign key prediction appears to be a much more intricate task. The foreign model proposes only two foreign keys of which one is not correct. The local model performs much better, but still not as good as the primary key prediction. This shows that the learned models underfit the problem and do not generalize very well – even though we adopted state-of-the-art techniques. An investigation of other features and classifiers is out of the scope of our case study, though. Also, as expected for a large dataset, the ratio of foreign keys to INDs is quite small (cf. Section 2.5.2), which makes their discovery a search for the "needle in the haystack". In fact, the *MusicBrainz* dataset is much more extensive than the datasets analyzed in related work, thus our findings question the scalability of current foreign key designation approaches.

* * *

In summary, we find that many schema discovery tasks can be performed almost automatically just by inspecting and combining various data profiling results. The corresponding analyses took at most a few seconds. An exception to that is the foreign key discovery, which not only turned out to be a particularly hard task but also was computationally most expensive. In fact, training a foreign key model took around 30 minutes, which can be attributed to the large number of INDs for the most part. In addition, we want to remark that there are many other interesting schema discovery tasks that we have not considered, e. g., restoring table and column names or proposing `CHECK` constraints. If and in how far data profiling results can support such tasks, e. g., in combination with machine learning algorithms, is interesting and relevant future work.

5.3 Conceptualization

When analyzing an unknown dataset, it is not only important to obtain a technical (relational) schema for that data. It is also crucial that users familiarize themselves with the dataset, so as to put its contents to actual use. We refer to this process of building up an understanding of the dataset as *conceptualization*. Clearly, this is a much less clear-cut task than the reconstruction of technical schema features. What is more, the conceptualization is for the largest part a cognitive process of the user rather than a purely technical process – in consequence, conceptualization can hardly be measured.

To narrow down the discussion, we enacted three concrete conceptualization aids in our data anamnesis: (i) As a beginning, we conduct a break-down analysis of the *dataset size*. Then, we apply (ii) a *table importance scoring* and (iii) a *table clustering* algorithm proposed by [Yang et al., 2009]. As a matter of fact, we adapted these algorithms to operate exclusively on data profiles and implemented them as part of METACRATE's analytics library. Eventually, we used standard plots to visualize the results of these algorithms. In the following, we present the outcomes of these conceptualization aids and, where applicable, compare them to the documentation of the MusicBrainz dataset.

Dataset size

Before a user delves into the details of a dataset, such as the structure of individual tables, it might be appropriate to gain an overview of how large a dataset is. An understanding of the dataset's extent allows, amongst others, to select an appropriate database deployment for the data, to apprehend how rich the data is terms of instances and features, and to estimate the time that is needed to adopt the new dataset [cf. Kruse et al., 2015b].

To this end, we broke down the number of the dataset's columns and rows to the table level and visualized the result in a scatter plot. To achieve this with METACRATE, a simple query suffices that joins (via the `coGroup` operator) the number of columns (obtained by counting) and rows (loaded from a profile collection) for each table and feeds the outcome in the respective plotting function (namely `plotScatterChart`). The complete query is as follows:

```
// Global variables: metacrate, schema
query { implicit engine =>
 // Load rows and columns per table.
 val cols = metacrate.loadColumns(schema)
  .map(col => (getTableId(col.id), 1))
  .reduceByKey(_._1, (cnt1, cnt2) => (cnt1._1, cnt1._2+cnt2._2))
 val rows = metacrate.loadConstraints[TupleCount](schema)
 // Join rows and columns.
 cols.keyBy(_._1).coGroup(rows.keyBy(_.getTableId))
  .map { coGroup =>
   val (cols, rows) = (coGroup.field0, coGroup.field1)
   (cols.head._2, // columns
    rows.map(_.getNumTuples.toInt).headOption getOrElse 0, // rows
    cols.head._1) // table ID
  }
  // Convert table ID to readable name
  .resolveIds((counts, r) => (counts._1, counts._2, r(counts._3)))
}
// Plot the resulting triples in a scatter plot.
.plotScatterChart(title = "Number of rows vs. number of columns",
                  xaxisTitle = "Columns [#]",
                  yaxisTitle = "Rows [#]")
```

Figure 5.4 depicts the result of this query and immediately conveys an intuition of the dataset size: The tables of the MusicBrainz dataset comprise at most 19 columns, but most contain fewer than 10. Also, there are many tables with millions of tuples and a few with even tens of millions of tuples. Furthermore, this visualization reveals a pattern between the number of row and columns in tables that we observed across very many different datasets, too: While the majority of tables has (relatively) few rows and columns, there are always some tables that contain *either* exceptionally many rows *or* exceptionally many columns. The former ones typically are join tables, e. g., in our dataset the upper left tables are called `edit_recording` and `edit_artist`. On the other hand, the tables with many columns tend to model central entities of the dataset. In Figure 5.4 the right-most tables are called `artist` and `event`.

Table importance scoring

The above dataset size breakdown gives a first clue as to which tables form the principal entities in the dataset. Knowing those important tables can be very helpful for the conceptualization: Usually, they represent easy-to-grasp concepts and summarize the dataset. Therefore, those tables also support the understanding of the other tables; in fact, other tables regularly are either join or dimension tables for those important tables and can thus be understood more easily when put into perspective.

However, identifying important tables only by their number of columns is not a very sophisticated approach. In particular, it ignores the structure of the schema, i. e., the foreign key relationships among tables, as an important criterion. To this end, we adapted a well-known table importance scoring algorithm [Yang et al., 2009]. The algorithm's basic idea is to interpret the dataset's schema as a graph, where the tables form the vertices and foreign key relationships form the edges. Then each vertex is given an initial score

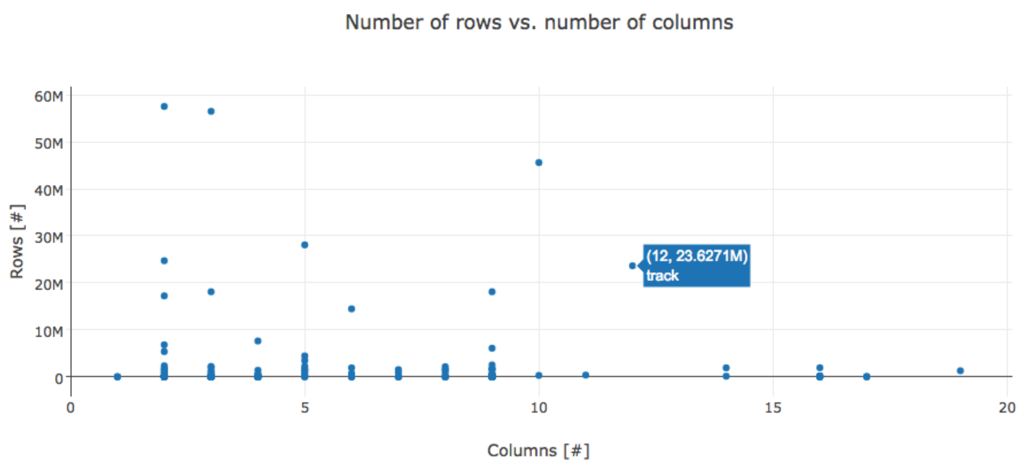


Figure 5.4: Screenshot of breaking down the dataset size to the tables of the MusicBrainz dataset with METACRATE. Hovering over a data point displays additional information, as exemplified for the track table.

based on the amount of information contained in its respective table; and each edge is qualified by how strongly it bonds the two adjacent tables. Then, an adapted PageRank algorithm [Brin and Page, 1998] is executed and the final “page ranks” serve as table importance scores.

We assessed the table importances with this algorithm and visualized the result as a schema graph, where vertices represent nodes, edges represent foreign key relationships, and the size of the nodes corresponds to the assessed table importances. Figure 5.5 depicts an excerpt of this visualization. Arguably, the graph is rather complex and to some extent overwhelming; however, the large vertices provide orientation and guidance so that this visualization is clearly more valuable than a plain schema graph without variable vertex sizes. That being said, further (schema) graph summarization techniques could be used to additionally produce smaller, more abstract graph visualizations [e. g., Wang et al., 2012; Yu and Jagadish, 2006].

In order to make a qualitative statement on the importance scores, we compared the top-ranked tables to the “primary entities” listed in the schema documentation of the MusicBrainz.[16] We find that those 11 primary entities are among the 14 top-ranked tables. The other 3 top-ranked tables, edit, link, and annotation, apparently do not represent entities from the discographical domain – in contrast to the primary entities. Instead, these tables form central metadata entities, that are, amongst others, responsible for the version management of the dataset. One might hence argue that these tables are also of particular interest for several applications. However, regardless of these three latter tables it is safe to say that the data profiling results indeed proved helpful to identify important tables in the MusicBrainz dataset.

[16]See `https://musicbrainz.org/doc/MusicBrainz_Database/Schema` (accessed December 15, 2017).

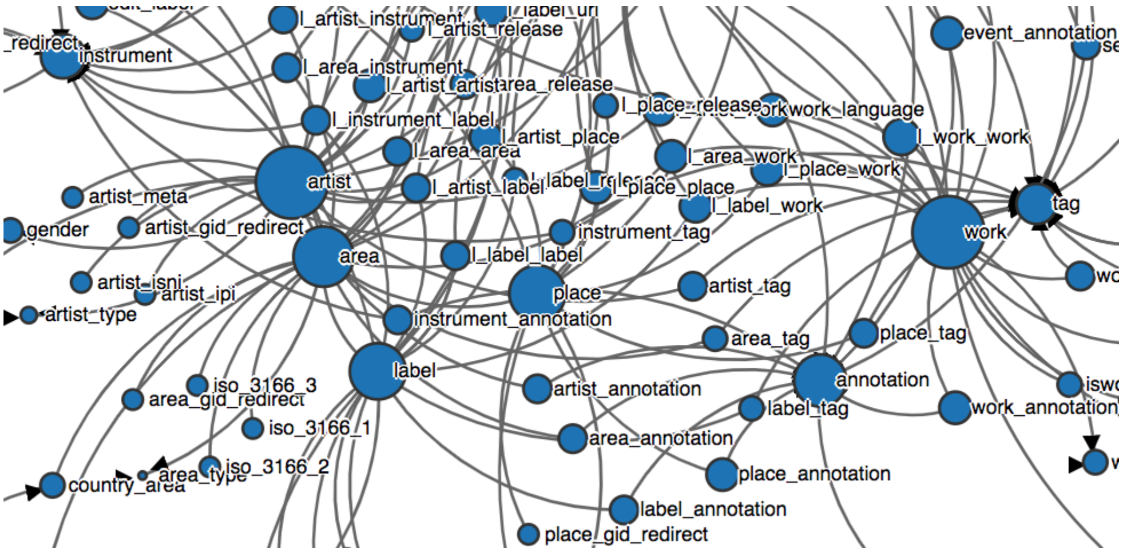


Figure 5.5: Screenshot of displaying the assessed table importances of the MusicBrainz dataset as a schema graph with METACRATE (excerpt). For better exploration, users can interact with the graph.

Table clustering

The above paragraph supposes that further summarization techniques might be helpful for users to cope with complex schema graphs, such as the one depicted in Figure 5.5. One such summarization technique is clustering: Clustering the tables in a schema can support users in understanding the context of non-important tables and efficiently identify relevant parts of the schema for their task at hand.

In fact, Yang et al. [2009] propose a schema clustering algorithm that prefers important tables as clustroids and uses a topology-based similarity measure to assign tables to clusters. We implemented this approach as part of METACRATE's analytics library, too; again taking care to derive all the relevant clustering exclusively from metadata. In consequence, the algorithm runs with sub-second performance. This is a great advantage, because like k-means the clustering algorithm requires a user-defined number k of clusters. We could easily test different k values and settled for $k = 12$, for which we deemed the resulting clustering to be most appropriate in terms of the abstraction level of its clusters.

To inspect the clustering result, we utilized two different visualizations. At first, we extended the graph from Figure 5.5 by encoding cluster membership as vertex colors. Second, we displayed the above mentioned similarity measure in a square matrix whose rows and columns correspond to tables. Specifically, the opacity of each matrix tile reflects the similarity of the tables that are associated with the corresponding row and column. If row and column table are also in the same cluster, then the tile color additionally encodes that cluster.

The colored schema graph is depicted in Figure 5.6. We found the coloring to be highly valuable. By spotting important tables, we could get a grasp of the associated

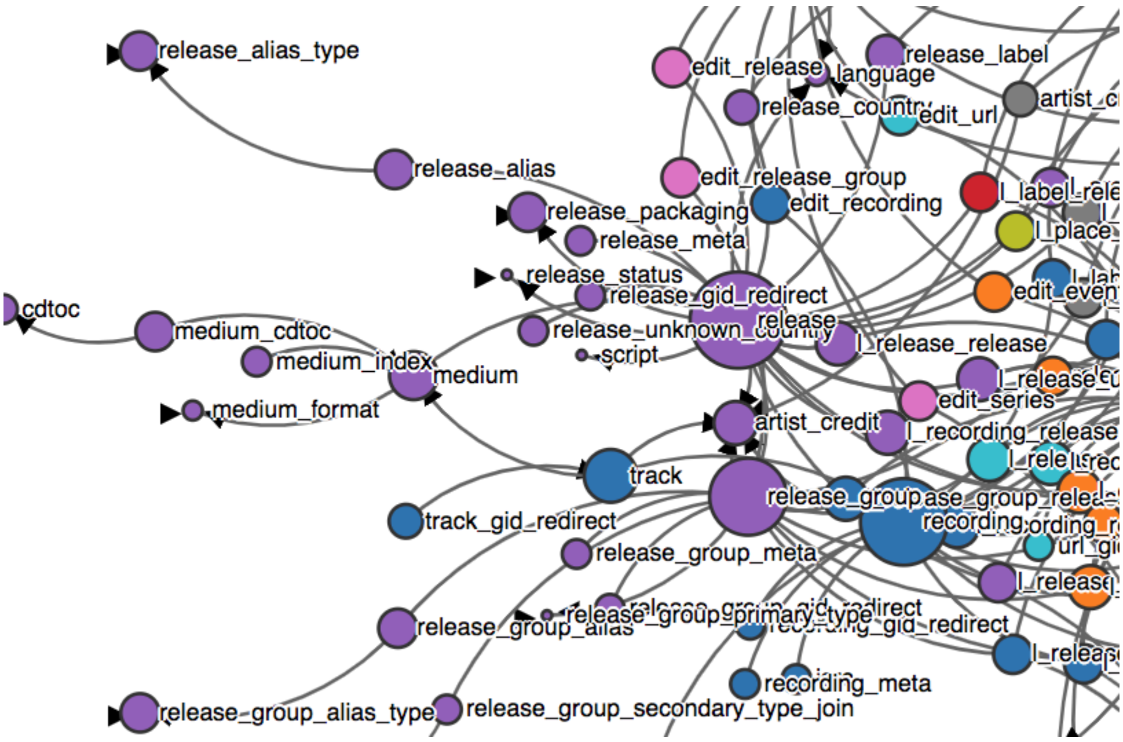


Figure 5.6: Screenshot of displaying the assessed table importances together with the table clustering of the MusicBrainz dataset as a schema graph with METACRATE.

cluster's contents. If that cluster appeared to be interesting, then we could quickly explore it by identifying tables that are of the same color. For instance, the purple vertices in Figure 5.6 obviously describe releases of musical works, as indicated by the two important tables release and release_group. If a user would be interested in release data in particular, she could immediately identify many relevant tables, such as medium, which apparently describes the physical medium of a release.

In contrast to the graph visualization, we found the matrix visualization in Figure 5.7 to convey completely different insights. This visualization is rather well-suited to identify similar tables (w. r. t. the similarity measure proposed by Yang et al. [2009]). Therefore, it was easy to see that many clusters are not very cohesive: Most clusters have one or two tables that are similar to the majority of tables in the same cluster, but most tables of a cluster are pairwise dissimilar. Furthermore, this visualization lends itself to spot links between clusters: Whenever two tables are similar, but do not reside in the same cluster, they form a bridge between the two clusters. In the matrix visualization, such table pairs form outliers and are hence easy to spot. In fact, Figure 5.7 highlights such an outlier between the release and the annotation cluster.

* * *

The three above explored conceptualization techniques can support users to get an overview of an unknown dataset. When potentially interesting tables have been identified, the next logical step would be to inspect individual tables in more detail. While

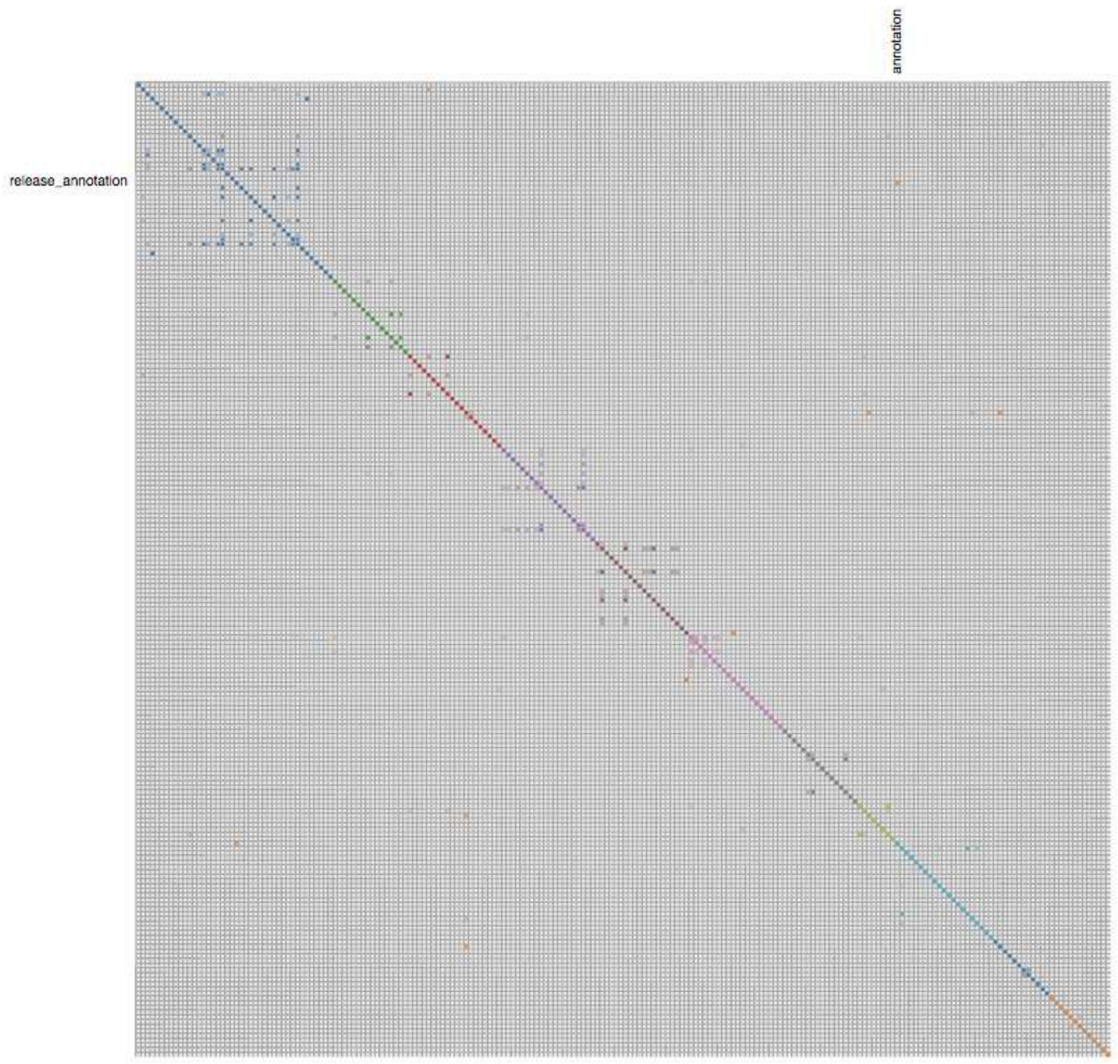


Figure 5.7: Screenshot of the visualization of the table clustering and the underlying similarities of the MusicBrainz dataset as a tile matrix with METACRATE. Hovering over a tile displays the tables associated to the respective row and column (note that the cluster colors do not match those in Figure 5.6).

we did not consider such techniques in our case study, industry data profiling tools, such as the IBM InfoSphere Information Analyzer[17] or Talend Data Quality[18], address this problem. Typically, those tools provide a wide host of column summaries, such as value distributions or textual patterns. As a future direction, it might be interesting to complement the summaries with data dependencies, such as (approximate) functional dependencies (see Chapter 4) or denial constraints [Bleifuß et al., 2017] to expose relationships among the individual columns.

[17]See `https://www.ibm.com/us-en/marketplace/infosphere-information-analyzer` (accessed on December 19, 2017).

[18]See `http://www.talend.com/products/data-quality` (accessed on December 19, 2017).

5.4 Schema Quality Assessment

The above two sections describe how to approach datasets with missing (relational) schemata or documentation. An additional concern with the adoption of new datasets is the question whether the data is actually "fit" for its intended use. One main consideration of such a fitness check is whether the dataset provides sufficient entities and models relevant features. Another major aspect is the quality of the dataset.

The research area of data cleaning investigates the methods to detect and repair data quality issues [e. g., Ganti and Sarma, 2013; Ilyas and Chu, 2015; Rahm and Do, 2000]. In particular, *instance-based* data cleaning has been (and continues to be) in the focus, i. e., methods to detect and correct erroneous or inconsistent tuples. Prominent data cleaning problems are deduplication [e. g., Christen, 2012; Kolb and Rahm, 2012; Whang et al., 2013] and constraint repairs [e. g., Bleifuß et al., 2017; Bohannon et al., 2007; Rekatsinas et al., 2017]. For the latter problem, discovery algorithms for partial dependencies as we describe them in this thesis form an invaluable asset, as they help to find violated database constraints.

In contrast to these works, our case study dealt with the orthogonal aspect of detecting quality problems at the schema level rather than at the instance level. Research on this topic is rather scarce, although it is apparently a relevant issue: In a study with numerous open-source software databases, Coelho et al. [2012] attested largely poor schema quality to them. Arguably, Codd [1971b] pioneered the consideration of schema quality by proposing integrity rules and database normal forms that give the database desirable properties, such as reduced redundancy and diminished susceptibility to inconsistencies. Apart from such technical constraints, other works have focused on judging schema quality from the application view point. As an example, Batini et al. [1991] define completeness, extensibility, and readability, amongst others, as schema quality dimensions.

In a data anamnesis scenario, such *top-down* approaches might be of limited applicability; especially if the user is not knowledgeable of the underlying data domain. To this end, we propose a *bottom-up* schema quality assessment – that is, we assessed the schema quality from the viewpoint of the data. Such a bottom-up assessment also seems applicable to revise initial assumptions made during the design of a schema, once a database is in production. In our case study, we studied two data-driven schema quality aspects, namely *conciseness* and *normality*, as we describe in the following.

Conciseness

While instance-based data cleaning methods treat missing values as a deficiency of the data, one might as well attribute greater numbers of missing values as a consequence of an overly complex or poorly designed schema. In fact, a schema might model properties that apply to only few instances, that are exceptionally hard to ascertain, or that are simply not relevant. Highly general and complex schemata are not only hard to understand and to maintain; they also might come with increased storage costs, their transactions and queries might become more complex and, in consequence, their execution could slow down. Hence, it would be desirable if the schema is only as complex as necessary, i. e., *concise*. Given this, our proposed approach to determine the conciseness of a schema is to look at the data and check whether we could also express it with a simpler schema. To do so, we identified several metadata-based indicators for lacking conciseness.

One obvious indicator for a lack in schema conciseness are (mostly) unused schema elements, i. e., empty tables and columns. Such schema elements might be pruned to enhance the schema conciseness. Our database dump of the MusicBrainz dataset did not contain any empty tables. However, the SQL files define 323 tables of which 114 were not present in the dump, which provides an opportunity to greatly simplify the schema.

Besides empty tables, we also analyzed (almost) empty columns with the following query:

```
// Global variables: metacrate, schema
query { implicit engine =>
  metacrate.loadConstraints[ColumnStatistics](schema)
    .map(stats => (stats.getFillStatus, 1))
    .reduceByKey(_._1, { case ((fill, a), (_, b)) => (fill, a+b) })
}.plotBarChart(title = "Fill status of all columns",
                  xaxisTitle = "Ratio of non-null values",
                  yaxisTitle = "Columns [#]")
```

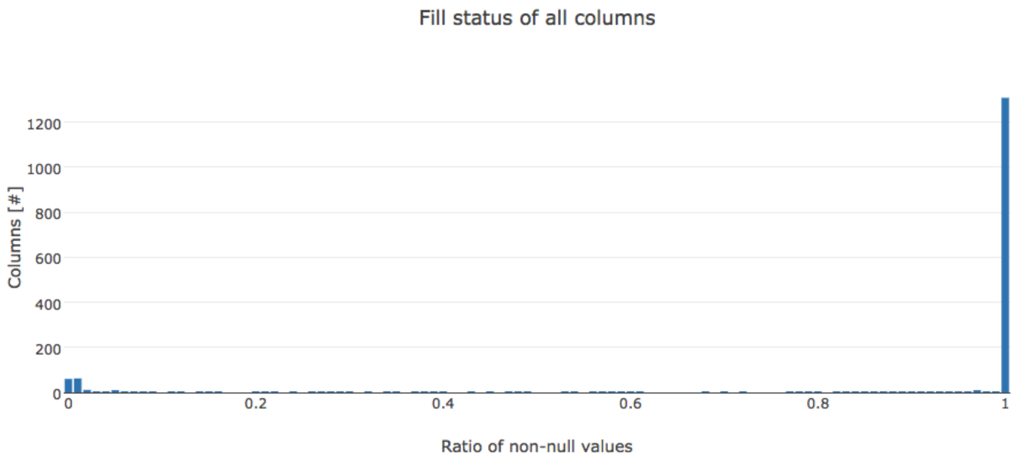


Figure 5.8: Screenshot of a bar chart summarizing the fill statuses of the MusicBrainz dataset with METACRATE.

The resulting bar chart is displayed in Figure 5.8. Apparently, 60 columns are completely empty and another 62 columns have a fill status below 0.01. To list those empty and sparse columns, we executed the following query:

```
// Global variables: metacrate, schema
query { implicit engine =>
  metacrate.loadConstraints[ColumnStatistics]("scdp-column-stats")
    .filter(_.getFillStatus <= 0.01)
    .resolveIds((stats, r) => (r(stats.getColumnId, withTable=true), stats.
        getFillStatus))
}.printTable(
  ordering = Ordering.by(_._2),
  columns = Seq("Table" -> (e => e._1),
                "Fill status" -> (e => f"${e._2}%.2f")))
```

The list of resulting columns was indeed insightful. Most empty columns are named either description or parent and belong to tables that have the name suffix _type in common. A closer look reveals that those tables are basically static tables that list types of certain entities. Apparently, users of the database did not see a necessity to provide descriptions for those types. Furthermore, the parent attribute allows to declare sub-types of other types, but apparently many entity types do not form hierarchical relationships.

We also learned that the majority of non-empty but highly sparse columns contain temporal data. As it seems, the MusicBrainz schema allows to specify time intervals for all kinds of entities, e. g., when and how long an artist used a certain alias. The majority of instances does not provide those time intervals, probably because it is very hard to ascertain and not that relevant. In particular, times are represented as separate year, month, and day attributes, of which the latter are particular susceptible to missing values. Overall we found that from the 1,566 columns in the MusicBrainz dataset roughly 10 % contain little or no data at all. A database administrator might consider whether to prune them.

Besides superfluous schema elements, we also analyzed the schema for unnecessarily complex relationships between entities. In relational databases, one-to-one relationships between entities can be represented with only a single table, while one-to-many relationships can be modeled with a simple join, and many-to-many relationships require an additional join table. To keep the schema as simple as possible, a concise schema should express the relationship with the most specific relational representation.

In our case study, we discovered one-to-one relationships that are modeled as one-to-many relationships by looking for foreign keys that embed a UCC. This can be done with the following METACRATE query:

```
// Global variables: metacrate, schema, resolveAll(...)
query { implicit engine =>
  val fks = metacrate.loadConstraints[InclusionDependency]("sql-foreign-
      keys")
  val uccs = metacrate.loadConstraints[UniqueColumnCombination]("hyucc-uccs
      ")

  fks.keyBy(_.getDependentColumnIds.toSet).join(uccs.keyBy(_.getColumnIds.
      toSet))
     .assemble { case (fk, ucc) => fk }
     .resolveIds { (fk, r) =>
         (resolveAll(fk.getDependentColumnIds, r), resolveAll(fk.
             getReferencedColumnIds, r))
     }
}.printList(before = "FKs that implement 1:1 relationships",
            format = { case (dep, ref) => s"$dep &rarr; $ref" })
```

Indeed, we found 32 such unnecessary one-to-many relationships. Judging by the involved table names (e. g., release and release_meta, link_type and orderable_link_type), it seems reasonable to merge the table pairs into single tables.

Furthermore, we detected unnecessary many-to-many relationships by searching join tables that embed an FD between its foreign key attributes. In detail, we looked for pairs of foreign keys, such that the referencing column of the one foreign key functionally determines the referencing column of the other foreign key, which can be achieved by two subsequent joins. The corresponding query looks as follows:

```
// Global variables: metacrate, schema, resolveAll(...)
query { implicit engine =>
  val fks = metacrate.loadConstraints[InclusionDependency]("sql-foreign-keys")
    .filter(_.getArity == 1)
    .map(fk => (fk.getDependentColumnIds()(0), fk.getReferencedColumnIds()(0)))
  val fds = metacrate.loadConstraints[FunctionalDependency]("hyfd-fds")
    .filter(_.getArity == 1)
    .map(fd => (fd.getLhsColumnIds()(0), fd.getRhsColumnId))

  val fksWithFds = fks.keyBy(_._1).join(fds.keyBy(_._1))
   .assemble { case ((dep, ref), (lhs, rhs)) => (dep, ref, rhs) }
   .keyBy(_._3).join(fks.keyBy(_._1))
   .assemble { case ((dep1, ref1, rhs), (dep2, ref2)) => (dep1, ref1, dep2, ref2) }

  fksWithFds.resolveIds { case ((dep1, ref1, dep2, ref2), r) =>
    (resolveAll(dep1, r), resolveAll(ref1, r), resolveAll(dep2, r), resolveAll(ref2, r))
  }
}.printList(before = "Join tables that implement 1:n relationships",
            format = { case (dep1, ref1, dep2, ref2) => s"$ref1 &rarr; $dep1 &rarr; $dep2 &rarr; $ref2" })
```

This query reported 12 modeled many-to-many relationships, that are only instantiated as one-to-many relationships by the data. As an example, we found that the within the l_instrument_table the attribute entity0 (which references the table instrument) functionally determines the attribute entity1 (which references the table label). In other words, the join table associates at most one label with each instrument. As a result, one might discard the join table and instead add a (nullable) foreign key in the instrument table that references the label table.

Normality

As we point out in the beginning of this section, an often desired schema property is the adherence to a normal form, in particular the Boyce-Codd normal form (BCNF) [Codd, 1974; Heath, 1971]. Normal forms are essentially best practices for data modeling that provide schemata with useful properties, such as effective queryability or – in the case of the BCNF – reduced redundancy. Concretely, the BCNF requires that schemata do not embed any FD unless its left-hand side is a candidate key, i. e., a UCC.

Originally, schema designers had to specify those FDs as rules of the associated data domain. Of course, designers might miss some FDs or deliberately (but perhaps incorrectly) omit them in anticipation of any exceptional cases that could violate certain FDs.

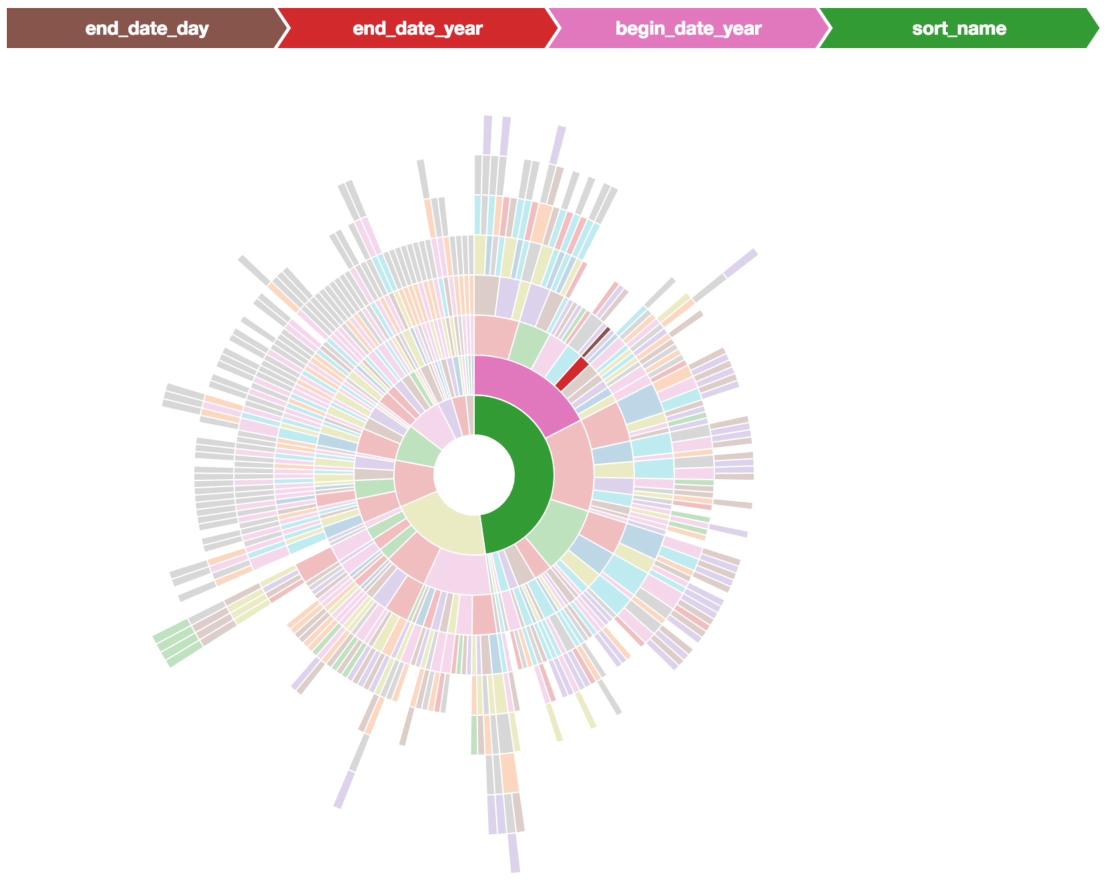


Figure 5.9: Screenshot of METACRATE's FD sunburst plot summarizing all those FDs from the artist table that are not implied by a UCC. Each section of the innermost ring represents a right-hand side attribute of several FDs, while the stacked outer sections represent left-hand side attributes of FDs. Hovering over a section displays the associated FD as a breadcrumb (here: end_date_day, end_data_year, begin_date_year→sort_name).

However, when the data is already present, such as in our case study, it is possible to efficiently and exactly determine the FDs of a dataset with modern discovery algorithms and use those FDs to check or enforce adherence to the BCNF [Papenbrock and Naumann, 2017a]. This comes with the caveat, though, that many FDs occur spuriously and should therefore not be considered as violations of the BCNF.

Separating non-spurious (partial) FDs is a challenging task whenever the data entails dependencies in abundance, as is the case for the artist table in the MusicBrainz dataset. To visually emphasize this point, we queried from METACRATE all FDs from the artist table, whose left-hand side attributes do not form a UCC, and had them displayed as a sunburst plot. The resulting chart, shown in Figure 5.9, gives an intuition of how tedious it would be to manually inspect the 241 relevant FDs. Note that this number is still moderate: tables sometimes entail millions of (partial) FDs or more.

In our case study, we tried to experimentally overcome this issue and identify interesting FDs that potentially constitute BCNF violations. First of all, we did not use the exact FDs and UCCs discovered by HyFD and HyUCC but the partial FDs and UCCs discovered by Pyro. The rationale is as follows: If the schema had violated the BCNF, then errors or inaccuracies might have sneaked into the data – those that the BCNF is actually supposed to avoid. In other words, FDs that violate the BCNF might be violated themselves. If we would encounter such violated and violating FDs, then we could use them for data cleaning [Bohannon et al., 2005; Thirumuruganathan et al., 2017] before normalizing the schema.

Second, we ranked the partial FDs that are not implied by a partial UCC. As a matter of fact, there are several measures to distinguish interesting from uninteresting FDs [e. g., Giannella and Robertson, 2004; Kruse and Naumann, 2018; Piatetsky-Shapiro and Matheus, 1993]. In our case study, we selected the ranking values provided by Pyro (see Section 4.7.8).

The ranking proved indeed useful and promoted several interesting dependencies. For instance, in the release_meta table, the attribute amazon_asin partially determines the info_url attribute (with a g_1 error of 0.006 according to Definition 4.1, page 81), which is sensible as those URLs embed the ASIN identifier. In a way, this finding is also representative of the other top-ranked partial FDs: Although most of the top-ranked rules reflect actual logical rules in the dataset, none of them seems to have sufficiently strong semantics for schema normalization.

Another grain of salt is that the ranking scores are not necessarily suitable to compare (partial) FDs across tables. And to our knowledge, there is no ranking scheme that overcomes this problem. As a result, one might instead resort to surveying the (partial) FDs for each table individually rather than just using one global ranking. However, because we did not find BCNF violations in the MusicBrainz dataset, we are not able to make any qualitative comparison of the global and local rankings within this case study.

In spite of these problem, we could still learn in a data-driven fashion that the MusicBrainz dataset is well normalized. That is, there are no (partial) FDs violating the BCNF. As a side effect, we cannot conclude how good or bad our ranking approach would have detected such violations, though.

*　　*　　*

The two analyzed aspects, conciseness and normality, already give an idea of the quality of the MusicBrainz schema: This schema has been modeled with very many details. Also, it is highly normalized. While both aspects make the schema very general, they also contribute to the complexity and extent of the schema, thereby hampering its understandablity. That being said, other data-driven schema quality dimensions could further complement this picture. For instance, one might use (partial) INDs to expose duplicate attributes and tables. Also, column summaries might expose columns with heterogeneous data – a sign that the attribute is frequently misused, which can happen when the schema is not extensive enough to model all the data it is supposed to store.

5.5 Summary

This chapter exemplified the utility of structural metadata in two steps. At first, we introduced Metacrate, a scalable system to store, refine, and visualize data profiles. Second, we performed a case study, in which we employed Metacrate to reconstruct the schema of the complex MusicBrainz dataset, familiarized ourselves with the dataset, and detected some short-comings of the schema from the viewpoint of the data. Essentially, each step in the data anamnesis benefited greatly from combining *different* types of structural metadata to derive deeper insights into the datasets.

As we point out throughout this chapter, we adapted several data analysis techniques from existing works to operate on structural metadata and additionally proposed several new metadata-based analyses. Because almost all those analyses and queries completed in less than a second, metadata-based algorithms lend themselves to data management scenarios that require user interaction. Thus, this field provides a promising branch for future research and we identify several challenges that could be solved using metadata, e. g., proposing column names and identifying overly simple parts of a schema.

Furthermore, in discussions with data scientists, we learned that an adaptation of Metacrate to unstructured data, such as texts or images, would be invaluable for their respective application domains. Arguably, profiling and analyzing such data is far beyond this thesis' scope. Nevertheless, profiling such types of data (i. e., extracting informative metadata) and use that metadata to perform a data anamnesis of the dataset, could be a fruitful research topic.

Chapter 6
Conclusions

It is almost a truism that data is a key enabler for many branches in science and industry. However, all too often the fact remains unattended that it can be extremely challenging to harness and overcome the size and complexity of datasets in order to properly leverage their contents. Data profiling addresses this serious problem by extracting summaries and structural properties from datasets that can then effectively support data management.

Along these lines, this thesis introduces several algorithms to discover various types of data dependencies from relational and RDF data: SINDY, SANDY, and ANDY discover (partial) inclusion dependencies (Chapter 2), RDFIND extracts conditional inclusion dependencies (Chapter 3), and PYRO detects both partial unique column combinations and partial functional dependencies (Chapter 4) in given datasets. The main objective of these algorithms is to push the envelope of data profiling algorithms in terms of efficiency and scalability. We attained this goal in a dual approach, in which we not only propose novel algorithmic strategies, but also target modern hardware setups with (clusters of) multi-core computers. Another eminent point of most of our algorithms is that they consider *partial* dependencies – that is, they reckon and actively cope with quality issues in the profiled datasets.

In addition to the discovery of data dependencies, we addressed the question of how to leverage the results of our profiling algorithms in actual use cases (Chapter 5). This question is a particularly relevant one, given that some datasets entail huge numbers of data dependencies. Besides proposing several dependency-specific interestingness measures, such as the support of conditional inclusion dependencies or a ranking score for partial functional dependencies, we also introduced the system METACRATE, which constitutes a principled approach to store, query, and analyze structural metadata.[1] In a case study, we demonstrated the applicability of METACRATE to data management by performing a data anamnesis on a complex real-world dataset. With a mixture of well-known reverse engineering algorithms as well as ad-hoc analyses, we were able to reconstruct interesting aspects of that dataset, thereby also confirming the utility of structural metadata.

[1]As mentioned in Section 5.1.2, METACRATE's analytics layer is based on the cross-platform data analytics system RHEEM [Agrawal et al., 2016; Kruse et al., 2018]. This system, which has been developed by the author of this thesis to a great extent, is complex in itself. However, it has not been presented in greater detail in this work, as it is not a dedicated data profiling system.

Apart from these positive results, we would also like to critically reflect on the limitations of our work and propose interesting future works. For that purpose, let us point out two basic, related issues: Our data profiling problems are of exponential complexity and their output sizes can also be exponential w. r. t. the number of profiled attributes. We identify two important consequences arising from these issues.

The first consequence concerns the efficiency and scalability: Our proposed algorithms clearly outperform previous state-of-the-art methods, sometimes by orders of magnitudes. However, that does not imply that we could automatically scale to datasets that are orders of magnitudes larger or solve every data profiling problem just by provisioning enough computational power. In fact, the inherent computational complexity of our data profiling problems essentially limits the scalability of all data profiling algorithms. To this end, it might be reasonable to abandon the path of exact and complete dependency discovery for further efficiency and scalability improvements.

One potential way to elude the above predicament could be to *waive guarantees* on exactness and completeness of data profiling results. We proposed two dependency discovery algorithms that *approximate* the exact and complete set of dependencies in a given dataset [Bleifuß et al., 2016; Kruse et al., 2017b] (we did not discuss these algorithms in this thesis, though). We found that approximation methods can indeed improve the efficiency of data profiling results by several factors while relinquishing only little result quality. However, the output of these algorithms can still be of exponential size and, thus, approximation techniques are also condemned to be of exponential complexity.

This brings us to the second consequence of the exponential nature of dependency discovery problems: The number of dependencies entailed by certain datasets is immensely large. At the same time, most of these dependencies are spurious or at least not relevant w. r. t. to the designated use case. In consequence, users need some form of automated support to process data profiling results, i. e., to separate the wheat from the chaff among thousands or even millions of discovered dependencies. We addressed this problem, e. g., by proposing a ranking measure for partial functional dependencies (Chapter 4) and studying several data anamnesis techniques (Chapter 5). The literature offers a few other ranking measures for various dependencies, but this problem deserves more attention. Inspiration could be drawn from the research on data mining that is facing similar problems – very well known representatives are the support and confidence measures for association rules.

That being said, these two issues – the theoretically limited algorithm scalability and the huge profiling results – bear the potential for an elegant synthesis: Of course, any profiling algorithm is bound to be of polynomial complexity in terms of its output size. However, if future algorithms manage to identify the few interesting data dependencies in profiled datasets already in the discovery process, then these algorithms might overcome the inherent exponential complexity of exact and complete data profiling algorithms. Essentially, such algorithms would kill two birds with one stone: They are not subject to unwieldy result sets, but at the same time could bear unparalleled profiling efficiency and scalability. The inherent drawback of such algorithms, on the other hand, would be that they need be tailored to specific applications of their dependencies. That is, such algorithms would incorporate some notion of dependency interestingness, thereby mixing application semantics into the data profiling. As a result, there might then be multiple application-specific dependency discovery algorithms. In that case, there would be no need to re-invent the wheel, though; instead our proposed algorithms provide a sophisticated starting point towards specifically tailored derivatives.

Despite all this criticism, our exact and complete profiling algorithms still scale very well to large datasets and their results are indeed of practical value, as we found in our case study. And yet it is important to learn the lessons from our results and then look ahead towards new challenges, because as Aristotle said:

"For the things we have to learn before we can do them, we learn by doing them."

Bibliography

Ziawasch Abedjan and Felix Naumann. Improving RDF data through association rule mining. *Datenbank-Spektrum*, 13(2):111–120, 2013.

Ziawasch Abedjan, Toni Grütze, Anja Jentzsch, and Felix Naumann. Mining and profiling RDF data with ProLOD++. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1198–1201, 2014a.

Ziawasch Abedjan, Patrick Schulze, and Felix Naumann. DFD: efficient functional dependency discovery. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 949–958, 2014b.

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

Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level.* Addison-Wesley, 1995. ISBN 0201537710.

Divy Agrawal, Lamine Ba, Laure Berti-Equille, Sanjay Chawla, Ahmed Elmagarmid, Hossam Hammady, Yasser Idris, Zoi Kaoudi, Zuhair Khayyat, Sebastian Kruse, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Mohammed J. Zaki. Rheem: enabling multi-platform task execution. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 2069–2072, 2016.

Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 487–499, 1994.

Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere platform for big data analytics. *VLDB Journal*, 23(6):939–964, 2014.

Chris Anderson. The end of theory: the data deluge makes the scientific method obsolete. 2008. Online; accessed July 4, 2017.

Periklis Andritsos, Ronald Fagin, Ariel Fuxman, Laura M. Haas, Mauricio A. Hernández, Howard Ho, Anastasios Kementsietsidis, Phokion G Kolaitis, Renée J Miller, Felix Naumann, et al. Schema management. *IEEE Data Engineering Bulletin*, 25(3):32–38, 2002.

Periklis Andritsos, Renée J. Miller, and Panayiotis Tsaparas. Information-theoretic tools for mining database structure from large data sets. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 731–742, 2004.

W. William Armstrong. Dependency structures of database relationships. *Information Processing*, 74(1):580–583, 1974.

Jalal Atoum. Mining approximate functional dependencies from databases based on minimal cover and equivalent classes. *European Journal of Scientific Research*, 33(2): 338–346, 2009.

Sören Auer, Jan Demter, Michael Martin, and Jens Lehmann. LODStats – an extensible framework for high-performance dataset analytics. In *Proceedings of the International Conference on Knowledge Engineering and Knowledge Management (EKAW)*, pages 353–362, 2012.

Carlo Batini, Stefano Ceri, and Shamkant B. Navathe. *Conceptual Database Design: An Entity-Relationship Approach.* Addison-Wesley, 1991. ISBN 0-8053-0244-1.

Jana Bauckmann. *Dependency discovery for data integration.* PhD thesis, University of Potsdam, 2013.

Jana Bauckmann, Ulf Leser, and Felix Naumann. Efficiently computing inclusion dependencies for schema discovery. In *ICDE Workshops (ICDEW)*, pages 2–2, 2006.

Jana Bauckmann, Ulf Leser, and Felix Naumann. Efficient and exact computation of inclusion dependencies for data integration. Technical Report 34, Hasso-Plattner-Institut, Universität Potsdam, 2010.

Jana Bauckmann, Ziawasch Abedjan, Ulf Leser, Heiko Müller, and Felix Naumann. Discovering conditional inclusion dependencies. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 2094–2098, 2012.

Catriel Beeri, Martin Dowd, Ronald Fagin, and Richard Statman. On the structure of Armstrong relations for functional dependencies. *Journal of the ACM*, 31(1):30–46, 1984.

Siegfried Bell and Peter Brockhausen. Discovery of data dependencies in relational databases. Technical report, Universität Dortmund, 1995.

George Beskales, Ihab F. Ilyas, and Lukasz Golab. Sampling the repairs of functional dependency violations under hard constraints. *Proceedings of the VLDB Endowment*, 3(1-2):197–207, 2010.

Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data – the story so far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(3): 1–22, 2009.

Thomas Bläsius, Tobias Friedrich, and Martin Schirneck. The parameterized complexity of dependency detection in relational databases. In *Proceedings of the International Symposium on Parameterized and Exact Computation (IPEC)*, pages 6:1–6:13, 2017.

Tobias Bleifuß, Susanne Bülow, Johannes Frohnhofen, Julian Risch, Georg Wiese, Sebastian Kruse, Thorsten Papenbrock, and Felix Naumann. Approximate discovery of functional dependencies for large datasets. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 1803–1812, 2016.

Tobias Bleifuß, Sebastian Kruse, and Felix Naumann. Efficient denial constraint discovery with Hydra. *Proceedings of the VLDB Endowment*, 11(3):311–323, 2017.

Philip Bohannon, Wenfei Fan, Michael Flaster, and Rajeev Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 143–154, 2005.

Philip Bohannon, Wenfei Fan, and Floris Geerts. Conditional functional dependencies for data cleaning. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 746–755, 2007.

Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building an efficient RDF store over a relational database. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 121–132, 2013.

Loreto Bravo, Wenfei Fan, and Shuai Ma. Extending dependencies with conditions. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 243–254, 2007.

Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

Paul G. Brown and Peter J. Hass. BHUNT: automatic discovery of fuzzy algebraic constraints in relational data. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 668–679, 2003.

Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodriguez-Muro, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. The MASTRO system for ontology-based data access. *Journal of Semantic Web (SWJ)*, 2(1):43–53, 2011.

Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. Relaxed functional dependencies – a survey of approaches. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 28(1):147–165, 2016.

Marco A. Casanova, Ronald Fagin, and Christos H. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 171–176, 1982.

Ashok K. Chandra and Moshe Y. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM Journal on Computing*, 14(3):671–677, 1985.

Qi Cheng, Jarek Gryz, Fred Koo, T. Y. Cliff Leung, Linqi Liu, Xiaoyan Qian, and K. Bernhard Schiefer. Implementation of two semantic query optimization techniques in DB2 Universal Database. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 687–698, 1999.

Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An efficient SQL-based RDF querying scheme. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1216–1227, 2005.

Peter Christen. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Springer Verlag, 2012. ISBN 9783642311635.

E. F. Codd. Normalized data base structure: A brief tutorial. In *Proceedings of the ACM SIGFIDET Workshop on Data Description, Access and Control*, pages 1–17, 1971a.

Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

Edgar F. Codd. Further normalization of the data base relational model. Technical Report RJ909, IBM, San Jose, California, 1971b.

Edgar F. Codd. Recent investigations in relational data base systems. In *IFIP Congress*, pages 1017–1021, 1974.

Edgar F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems (TODS)*, 4(4):397–434, 1979.

Fabien Coelho, Alexandre Aillos, Samuel Pilot, and Shamil Valeev. On the quality of relational database schemas in open-source software. *International Journal on Advances in Software*, 4(3):11, 2012.

Mehmet M. Dalkilic and Edward L Roberston. Information dependencies. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 245–253, 2000.

Tamraparni Dasu, Theodore Johnson, S. Muthukrishnan, and Vladislav Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 240–251, 2002.

Christopher J. Date. Referential integrity. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 2–12, 1981.

Kathi H. Davis and Peter H. Alken. Data reverse engineering: a historical survey. In *Proceedings of the Working Conference on Reverse Engineering*, pages 70–78, 2000.

Fabien de Marchi. CLIM: Closed inclusion dependency mining in databases. In *International Conference on Data Mining Workshops (ICDMW)*, pages 1098–1103, 2011.

Fabien de Marchi and Jean-Marc Petit. Zigzag: a new algorithm for mining large inclusion dependencies in databases. In *Proceedings of the International Conference on Data Mining (ICDM)*, pages 27–34, 2003.

Fabien de Marchi, Stéphane Lopes, and Jean-Marc Petit. Efficient algorithms for mining inclusion dependencies. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 464–476, 2002.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

Xin Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. Knowledge vault: a web-scale approach to probabilistic knowledge fusion. In *Proceedings of the International Conference on Knowledge discovery and data mining (SIGKDD)*, pages 601–610, 2014.

Johannes Eschrig. Data Profiling Benchmark and Tool Assessment. Master's thesis, Hasso-Plattner-Institut, Universität Potsdam, 2016.

Ronald Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM Transactions on Database Systems (TODS)*, 2(3):262–278, 1977.

Ronald Fagin. Normal forms and relational database operators. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 153–160, 1979.

Ronald Fagin. A normal form for relational databases that is based on domains and keys. *ACM Transactions on Database Systems (TODS)*, 6(3):387–415, 1981.

Wenfei Fan. Dependencies revisited for improving data quality. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 159–170, 2008.

Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Transactions on Database Systems (TODS)*, 33(2):6:1–6:48, 2008.

Raul Castro Fernandez, Ziawasch Abedjan, Samuel Madden, and Michael Stonebraker. Towards large-scale data discovery: position paper. pages 3–5, 2016.

Peter A. Flach and Iztok Savnik. FDEP source code. `http://www.cs.bris.ac.uk/~flach/fdep/`.

Peter A. Flach and Iztok Savnik. Database dependency discovery: a machine learning approach. *AI Communications*, 12(3):139–160, 1999.

Flink. Apache Flink. `http://flink.apache.org/`; accessed August 11, 2017.

FOR 1306. Stratosphere – information management on the cloud. `http://gepris.dfg.de/gepris/projekt/132320961`. Online; accessed July 11, 2017.

Mario A. Gallego, Javier D. Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. An empirical study of real-world SPARQL queries. In *International Workshop on Usage Analysis and the Web of Data (USEWOD)*, 2011.

Venkatesh Ganti and Anish Das Sarma. *Data Cleaning: A Practical Perspective.* Morgan & Claypool, 2013. ISBN 9781608456789.

Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. The LLUNATIC data-cleaning framework. *Proceedings of the VLDB Endowment*, 6(9):625–636, 2013.

Chris Giannella and Edward Robertson. On approximation measures for functional dependencies. *Information Systems*, 29(6):483–507, 2004.

Seymour Ginsburg and Richard Hull. Order dependency in the relational model. *Theoretical Computer Science*, 26(1–2):149–195, 1983.

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

Lukasz Golab, Flip Korn, and Divesh Srivastava. Efficient and effective analysis of data quality using pattern tableaux. *IEEE Data Engineering Bulletin*, 34(3):26–33, 2011.

Jarek Gryz. An algorithm for query folding with functional dependencies. In *Proceedings of the International Symposium on Intelligent Information Systems*, pages 7–16, 1998a.

Jarek Gryz. Query folding with inclusion dependencies. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 126–133, 1998b.

Jarek Gryz. Query rewriting using views in the presence of functional and inclusion dependencies. *Information Systems (IS)*, 24(7):597–612, 1999.

Dimitrios Gunopulos, Roni Khardon, Heikki Mannila, Sanjeev Saluja, Hannu Toivonen, and Ram Sewak Sharma. Discovering all most specific sentences. *ACM Transactions on Database Systems (TODS)*, 28(2):140–174, 2003.

Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.

Marc Gyssens. Database dependencies. In *Encyclopedia of Database Systems*, volume 1. Springer Verlag, 2009. ISBN 978-0-387-39940-9.

Hadoop. Apache Hadoop. `http://hadoop.apache.org/`; accessed August 10, 2017.

Jean-Luc Hainaut, Jean Henrard, Didier Roland, Jean-Marc Hick, and Vincent Englebert. Database reverse engineering. In *Handbook of Research on Innovations in Database Technologies and Applications: Current and Future Trends*, pages 181–189. 2009.

Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1–12, 2000.

P. Hayes and Peter F. Patel-Schneider. RDF 1.1 semantics. `https://www.w3.org/TR/rdf11-mt/`, February 2014. Online; accessed on September 20, 2017.

I. J. Heath. Unacceptable File Operations in a Relational Data Base. In *Proceedings of the ACM SIGFIDET Workshop on Data Description, Access and Control*, pages 19–33, 1971.

Arvid Heise, Jorge-Arnulfo Quiané-Ruiz, Ziawasch Abedjan, Anja Jentzsch, and Felix Naumann. Scalable discovery of unique column combinations. *Proceedings of the VLDB Endowment*, 7(4):301–312, 2013.

Jun Hong, Weiru Liu, David Bell, and Qingyuan Bai. Answering queries using views in the presence of functional dependencies. In *British National Conference on Databases*, pages 70–81, 2005.

Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. TANE: an efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100–111, 1999.

Ihab F. Ilyas and Xu Chu. Trends in cleaning relational data: consistency and deduplication. *Foundations and Trends in Databases*, 5(4):281–393, 2015.

Ihab F. Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. CORDS: automatic discovery of correlations and soft functional dependencies. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 647–658, 2004.

David S. Johnson and Anthony Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *Journal of Computer and System Sciences*, 28 (1):167–189, 1984.

Theodore Johnson. Data profiling. In *Encyclopedia of Database Systems*, volume 1, pages 604–608. Springer Verlag, 2009. ISBN 978-0-387-39940-9.

Tobias Käfer, Ahmed Abdelrahman, Jürgen Umbrich, Patrick O'Byrne, and Aidan Hogan. Observing linked data dynamics. In *Proceedings of the European Semantic Web Conference (ESWC)*, pages 213–227, 2013.

Paris C. Kanellakis. Elements of relational database theory. Technical report, 1989.

Paris C. Kanellakis, Stavros S. Cosmadakis, and Moshe Y. Vardi. Unary inclusion dependencies have polynomial time inference problems. In *Proceedings of the Symposium on Theory of Computing (STOC)*, pages 264–277, 1983.

Martti Kantola, Heikki Mannila, R. Kari-Jouko, and Harri Siirtola. Discovering functional and inclusion dependencies in relational databases. *International Journal of Intelligent Systems*, 7(7):591–607, 1992.

Zoi Kaoudi and Ioana Manolescu. RDF in the clouds: a survey. *VLDB Journal*, 24(1): 67–91, 2015.

Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations*, pages 85–103, 1972.

Shahan Khatchadourian and Mariano P. Consens. ExpLOD: summary-based exploration of interlinking and RDF usage in the Linked Open Data cloud. In *Proceedings of the European Semantic Web Conference (ESWC)*, pages 272–287, 2010.

Zuhair Khayyat, Ihab F. Ilyas, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. BigDansing: a system for big data cleansing. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1215–1230, 2015.

Ronald S King and James J Legendre. Discovery of functional and approximate functional dependencies in relational databases. *Journal of Applied Mathematics and Decision Sciences*, 7(1):49–59, 2003.

Jyrki Kivinen and Heikki Mannila. Approximate dependency inference from relations. *Proceedings of the International Conference on Database Theory (ICDT)*, pages 86–98, 1992.

Jyrki Kivinen and Heikki Mannila. Approximate inference of functional dependencies from relations. *Theoretical Computer Science*, 149(1):129–149, 1995.

Andreas Koeller and E. A. Rundensteiner. Discovery of high-dimensional inclusion dependencies. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2002.

Andreas Koeller and Elke A. Rundensteiner. Journal on Data Semantics V. chapter Heuristic Strategies for the Discovery of Inclusion Dependencies and Other Patterns, pages 185–210. Springer Verlag, 2006. ISBN 978-3-540-31426-4.

Henning Köhler, Sebastian Link, and Xiaofang Zhou. Possible and certain SQL keys. *Proceedings of the VLDB Endowment*, 8(11):1118–1129, 2015.

Solmaz Kolahi and Laks V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 53–62, 2009.

Lars Kolb and Erhard Rahm. Parallel entity resolution with Dedoop. *Datenbank Spektrum*, 13(1):23–32, 2012.

Sebastian Kruse and Felix Naumann. Efficient discovery of approximate dependencies. *Proceedings of the VLDB Endowment*, 11(7):759–772, 2018.

Sebastian Kruse, Thorsten Papenbrock, and Felix Naumann. Scaling out the discovery of inclusion dependencies. In *Proceedings of the Conference Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*, pages 445–454, 2015a.

Sebastian Kruse, Paolo Papotti, and Felix Naumann. Estimating data integration and cleaning effort. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 61–72, 2015b.

Sebastian Kruse, Anja Jentzsch, Thorsten Papenbrock, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. RDFind: scalable conditional inclusion dependency discovery in RDF datasets. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 953–967, 2016a.

Sebastian Kruse, Thorsten Papenbrock, Hazar Harmouch, and Felix Naumann. Data anamnesis: admitting raw data into an organization. *IEEE Data Engineering Bulletin*, 39(2):8–20, 2016b.

Sebastian Kruse, David Hahn, Marius Walter, and Felix Naumann. Metacrate: organize and analyze millions of data profiles. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 2483–2486, 2017a.

Sebastian Kruse, Thorsten Papenbrock, Christian Dullweber, Moritz Finke, Manuel Hegner, Martin Zabel, Christian Zoellner, and Felix Naumann. Fast approximate discovery of inclusion dependencies. In *Proceedings of the Conference Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*, pages 207–226, 2017b.

Sebastian Kruse, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, Sanjay Chawla, Felix Naumann, and Bertty Contreras. RHEEMix in the data jungle - A cross-Platform query optimizer. `http://arxiv.org/abs/1805.03533`, 2018.

Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

Ulf Leser and Felix Naumann. *Informationsintegration - Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen.* dpunkt.verlag, 2007. ISBN 783898644006.

Mark Levene and Millist W. Vincent. Justification for inclusion dependency normal form. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 12(2):2000, 1999.

Huiying Li. Data profiling for semantic web data. In *Proceedings of the International Conference on Web Information Systems and Mining (WISM)*, pages 472–479, 2012.

Weibang Li, Zhanhuai Li, Qun Chen, Tao Jiang, and Zhilei Yin. Discovering approximate functional dependencies from distributed big data. In *Asia-Pacific Web Conference*, pages 289–301, 2016.

Tok Wang Ling and Cheng Hian Goh. Logical database design with inclusion dependencies. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 642–649, 1992.

Jixue Liu, Jiuyong Li, Chengfei Liu, and Yongfeng Chen. Discover dependencies from data – a review. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 24(2):251–264, 2012.

Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. Functional and approximate dependency mining: database and FCA points of view. *Journal of Experimental & Theoretical Artificial Intelligence*, 14(2-3):93–114, 2002a.

Stéphane Lopes, Jean-Marc Petit, and Farouk Toumani. Discovering interesting inclusion dependencies: application to logical database tuning. *Information Systems (IS)*, 27 (1):1–19, 2002b.

Claudio L. Lucchesi and Sylvia L. Osborn. Candidate keys for relations. *Journal of Computer and System Sciences*, 17(2):270–279, 1978.

Shuai Ma, Wenfei Fan, and Loreto Bravo. Extending inclusion dependencies with conditions. *Theoretical Computer Science*, 515:64–95, 2014.

Heikki Mannila and Kari-Jouko Raiha. Inclusion dependencies in database design. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 713–718, 1986.

Heikki Mannila and Hannu Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery (DMKD)*, 1(3):241–258, 1997.

M. Scott Marshall, Richard Boyce, Helena F. Deus, Jun Zhao, Egon L. Willighagen, Matthias Samwald, Elgar Pichler, Janos Hajagos, Eric Prud'hommeaux, and Susie Stephens. Emerging practices for mapping and linking life sciences data using RDF - a case series. *Web Semantics: Science, Services and Agents on the World Wide Web*, 14:2–13, 2012.

Victor Matos and Becky Grasser. SQL-based discovery of exact and approximate functional dependencies. *ACM SIGCSE Bulletin*, 36(4):58–63, 2004.

Mozhgan Memari. *Partial referential integrity in SQL database.* PhD thesis, University of Auckland, 2016.

Renée J Miller, Mauricio A. Hernández, Laura M. Haas, Ling-Ling Yan, C. T. Howard Ho, Ronald Fagin, and Lucian Popa. The Clio project: managing heterogeneity. *SIGMOD Record*, 30(1):78–83, 2001.

Ullas Nambiar and Subbarao Kambhampati. Mining approximate functional dependencies and concept similarities to answer imprecise queries. In *Proceedings of the ACM Workshop on the Web and Databases (WebDB)*, pages 73–78, 2004.

Felix Naumann. Data profiling revisited. *SIGMOD Record*, 42(4):40–49, 2014.

Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB Journal*, 19(1):91–113, 2010.

ODF 2017. Open Discovery Framework. `https://github.com/apache/incubator-atlas/tree/feature-odf/odf`. Online; accessed July 5, 2017.

Michael Palmer. Data is the new oil. `http://ana.blogs.com/maestros/2006/11/data_is_the_new.html`, 2006. Online; accessed July 4, 2017.

Thorsten Papenbrock and Felix Naumann. A hybrid approach to functional dependency discovery. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 821–833, 2016.

Thorsten Papenbrock and Felix Naumann. A hybrid approach for efficient unique column combination discovery. In *Proceedings of the Conference Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*, pages 195–204, 2017a.

Thorsten Papenbrock and Felix Naumann. Data-driven schema normalization. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 342–353, 2017b.

Thorsten Papenbrock, Tanja Bergmann, Moritz Finke, Jakob Zwiener, and Felix Naumann. Data profiling with Metanome. *Proceedings of the VLDB Endowment*, 8(12): 1860–1863, 2015a.

Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. Functional dependency discovery: an experimental evaluation of seven algorithms. *Proceedings of the VLDB Endowment*, 8(10):1082–1093, 2015b.

Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. Divide & conquer-based inclusion dependency discovery. *Proceedings of the VLDB Endowment*, 8(7):774–785, 2015c.

G. N. Paulley and Per-Ake Larson. Exploiting uniqueness in query optimization. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research: Distributed Computing*, pages 804–822, 1993.

J-M Petit, Farouk Toumani, J-F Boulicaut, and Jacques Kouloumdjian. Towards the reverse engineering of renormalized relational databases. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 218–227, 1996.

Bernhard Pfahringer and Stefan Kramer. Compression-based evaluation of partial determinations. In *Proceedings of the International Conference on Knowledge discovery and data mining (SIGKDD)*, pages 234–239, 1995.

Peter Pham. The impacts of big data that you may not have heard of. `https://www.forbes.com/sites/peterpham/2015/08/28/the-impacts-of-big-data-that-you-may-not-have-heard-of/`, 2015. Online; accessed July 4, 2017.

Gregory Piatetsky-Shapiro and Christopher J. Matheus. Measuring data dependencies in large databases. In *Proceedings of the International Conference on Knowledge Discovery in Databases (AAAIWS)*, pages 162–173, 1993.

Gil Press. Cleaning big data: most time-consuming, least enjoyable data science task, survey says. `https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says/`, 2016. Online; accessed July 4, 2017.

Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 10(4):334–350, 2001.

Erhard Rahm and Hong Hai Do. Data cleaning: problems and current approaches. *IEEE Data Engineering Bulletin*, 23(4):3–13, 2000.

Nicole Redaschi and UniProt Consortium. UniProt in RDF: tackling data integration and distributed annotation with the semantic web. In *Proceedings of the International Biocuration Conference*, 2009.

David Reinsel, John Gantz, and John Rydning. Data age 2025: the evolution of data to life-critical. `http://www.seagate.com/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf`, 2017. Online; accessed July 4, 2017.

Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. HoloClean: holistic data repairs with probabilistic inference. *Proceedings of the VLDB Endowment*, 10 (11):1190–1201, 2017.

Jorma Rissanen. Independent components of relations. *ACM Transactions on Database Systems (TODS)*, 2(4):317–325, 1977.

Alexandra Rostin, Oliver Albrecht, Jana Bauckmann, Felix Naumann, and Ulf Leser. A machine learning approach to foreign key discovery. In *Proceedings of the ACM Workshop on the Web and Databases (WebDB)*, 2009.

Perry Rotella. Is data the new oil? `https://www.forbes.com/sites/perryrotella/2012/04/02/is-data-the-new-oil/`, 2012. Online; accessed July 4, 2017.

Daniel Sánchez, José María Serrano, Ignacio Blanco, Maria Jose Martín-Bautista, and María-Amparo Vila. Using association rules to mine for strong approximate dependencies. *Data Mining and Knowledge Discovery (DMKD)*, 16(3):313–348, 2008.

Max Schmachtenberg, Christian Bizer, and Heiko Paulheim. State of the LOD cloud 2014. `http://lod-cloud.net/state/state_2014/`, 2014. Online; accessed on September 21, 2017.

Nuhad Shaabani and Christoph Meinel. Scalable inclusion dependency discovery. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 425–440, 2015.

Nuhad Shaabani and Christoph Meinel. Detecting maximum inclusion dependencies without candidate generation. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*, pages 118–133, 2016.

David Simmen, Eugene Shekita, and Timothy Malkemus. Fundamental techniques for order optimization. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 57–67, 1996.

Tom Simonite. Intel puts the brakes on Moore's law. `https://www.technologyreview.com/s/601102/intel-puts-the-brakes-on-moores-law/`, 2016. Online; accessed July 11, 2017.

Yannis Sismanis, Paul Brown, Peter J. Haas, and Berthold Reinwald. GORDIAN: efficient and scalable discovery of composite keys. In *Proceedings of the VLDB Endowment*, pages 691–702, 2006.

Spark. Apache Spark. `http://spark.apache.org/`; accessed August 11, 2017.

SPARQL 1.1. SPARQL 1.1 Overview. `https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/`, 2013. Online; accessed on September 20, 2017.

E. Sperner. Ein Satz über Untermengen einer endlichen Menge. *Mathematische Zeitschrift*, 27:544–548, 1928. In German.

Aaron Swartz. MusicBrainz: a semantic Web service. *IEEE Intelligent Systems*, 17(1): 76–77, 2002.

Hee Beng Kuan Tan and Yuan Zhao. Automated elicitation of inclusion dependencies from the source code for database transactions. *Journal of Software Maintenance*, 15 (6):379–392, 2003.

Bernhard Thalheim. *Entity-Relationship Modeling: Foundations of Database Technology.* Springer Verlag, 2013. ISBN 9783540654704.

Saravanan Thirumuruganathan, Laure Berti-Equille, Mourad Ouzzani, Jorge Arnulfo Quiané-Ruiz, and Nan Tang. UGuide: user-guided discovery of FD-detectable errors. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1385–1397, 2017.

Joris Toonders. Data is the new oil of the digital economy. `https://www.wired.com/insights/2014/07/data-new-oil-digital-economy/`, 2014. Online; accessed July 4, 2017.

Boris Trakhtenbrot. The impossibility of an algorithm for the decidability problem on finite classes. In *Proceedings of the USSR Academy of Sciences*, 1950. In Russian.

Fabian Tschirschnitz, Thorsten Papenbrock, and Felix Naumann. Detecting inclusion dependencies on very many tables. *ACM Transactions on Database Systems (TODS)*, 1(1):1–30, 2017.

Xue Wang, Xuan Zhou, and Shan Wang. Summarizing large-scale database schema using community detection. *Journal of Computer Science and Technology*, 27(3):515–526, 2012.

S. E. Whang, D. Marmaros, and H. Garcia-Molina. Pay-as-you-go entity resolution. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 25(5):1111–1124, 2013.

Xiaoyan Yang, Cecilia M. Procopiuc, and Divesh Srivastava. Summarizing relational databases. *Proceedings of the VLDB Endowment*, 2(1):634–645, 2009.

Cong Yu and HV Jagadish. Schema summarization. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 319–330, 2006.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 15–28, 2012.

Carlo Antonio Zaniolo. *Analysis and Design of Relational Schemata for Database Systems.* PhD thesis, University of California, Los Angeles, 1976.

Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M. Procopiuc, and Divesh Srivastava. On multi-column foreign key discovery. *Proceedings of the VLDB Endowment*, 3(1-2):805–814, 2010.

Jakob Zwiener. Quicker ways of doing fewer things: improved index structures and algorithms for data profiling. Master's thesis, Hasso-Plattner-Institut, Universität Potsdam, 2015.

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Doktorarbeit mit dem Thema:

Scalable Data Profiling – Distributed Discovery and Analysis of Structural Metadata

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ferner wurde die vorliegende Arbeit an keiner anderen Hochschule eingereicht.

Potsdam, den 19. Januar 2018