RDFind: Scalable Conditional Inclusion Dependency Discovery in RDF Datasets

ABSTRACT

Inclusion dependencies (INDs) form an important integrity constraint on relational databases, supporting data management tasks, such as join path discovery and query optimization. Conditional inclusion dependencies (CINDs), which define including and included data in terms of conditions, allow to transfer these capabilities to RDF data. However, CIND discovery is computationally much more complex than IND discovery and the number of CINDs even on small RDF datasets is intractable.

To cope with both problems, we first introduce the notion of *pertinent* CINDs with an adjustable relevance criterion to filter and rank CINDs based on their extent and implications among each other. Second, we present RDFIND, a distributed system to efficiently discover all pertinent CINDs in RDF data. RDFIND employs a lazy pruning strategy to drastically reduce the CIND search space. Also, its exhaustive parallelization strategy and robust data structures make it highly scalable. In our experimental evaluation, we show that RDFIND is up to 419 times faster than the state-of-the-art, while considering a more general class of CINDs. Furthermore, it is capable of processing a very large dataset of billions of triples, which was entirely infeasible before.

1. INCLUSIONS WITHIN RDF DATASETS

The *Resource Description Framework* (RDF) [18] is a flexible data model that shapes data as a set of subject-predicate-object triples. 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 [9, 12] and data integration systems [11] to scientific applications [29,32]. As a result, very large volumes of RDF data are made available, in particular in the context of the Linked (Open) Data movement [8]. It is expected that this expansion of RDF data will perpetuate, leading to enormous amounts of large heterogeneous datasets [21].

A major particularity of RDF is that, in contrast to relational databases, its schema (ontology) is not always available, and even if it is, the data may violate the schema constraints. This impedes the use of RDF datasets, e.g., it is harder to formulate sound queries and wrong assumptions about the data might be made. *Data profiling* copes with such problems by identifying dataset properties, such as structure and integrity constraints [1].

Table 1: University RDF data instance example.

	Subject s	Predicate p	Object o
t_1	patrick	rdf:type	gradStudent
t_2	mike	rdf:type	gradStudent
t_3	john	rdf:type	professor
t_4	patrick	memberOf	csd
t_5	mike	memberOf	biod
t_6	patrick	undergradFrom	hpi
t_7	tim	undergradFrom	hpi
t_8	mike	undergradFrom	cmu

One of the most important integrity constraints are *inclusion dependencies* (INDs). Basically, an IND describes the inclusion of a set of values or elements in another set. In the context of relational databases, where these sets are the values from specified columns, INDs have proven to be very useful in various data management scenarios, such as foreign key and join path discovery [33], query optimization [17], and schema (re)design [25].

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. In fact, 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 Table 1.

However, *conditional inclusion dependencies* (CINDs) allow to refine these sets. A CIND filters the including and included sets of an IND with conditions and requires only these filtered sets to satisfy the IND. These CIND semantics allow to describe meaningful inclusions within RDF data:

EXAMPLE 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* undergradFrom*. Table 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*.*

CIND applications. CINDs of RDF datasets are useful for many applications, such as *ontology reverse engineering*, *knowledge discovery*, and *query optimization*. In the follow-

ing, we exemplify this with real CINDs from DBpedia 2014[1]. More details can be found in Appendix B.

CINDs facilitate *ontology reverse engineering* by finding the domain and range of RDF predicates (e.g., the objects of the triples with predicate capitalPosition are included in the subjects of rdf:type gml:Feature) and revealing class hierarchies (e.g., subjects of rdf:type Leptodactylidae are a subset of the subjects of rdf:type Frog); CINDs also support *knowledge discovery* to give insights about rules that may apply to unknown RDF datasets (e.g., the subjects with areaCode 559 are included in the subjects that are partOf California).

Furthermore, CINDs can be employed for *SPARQL query optimization*. SPARQL queries very often consist of a considerable amount of self-joins [5], 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 Table 1 asking for the departments of graduate students and the institutes that they received their undergraduates from: `SELECT ?d ?u WHERE {?s rdf:type gradStudent . ?s memberOf ?d . ?s undergradFrom ?u .}` Note that the data in Table 1 satifies a CIND that states that all subjects of the triples having memberOf as predicate are included in the set of subjects of rdf:type gradStudent. This CIND tells us that all department members are graduate students and allows to remove the first query triple (`?s rdf:type gradStudent`) without affecting the final results. As a proof of concept, we executed LUBM query Q2 [27] on the RDF-3X dataset [30] once in its original form and once CIND-optimized. Query Q2 originally contains six query triples; exploiting CINDs, we reduced Q2 to three query triples. This speeds up query execution by a factor of 3 (see Appendix B for details).

Challenges. Despite this importance and benefits of knowing CINDs in RDF datasets, the problem of CIND discovery for RDF data has not been tackled so far. A reason might be the complexity of this problem: Because each CIND involves two conditions, the search space for CINDs is quadratic in the number of possible conditions. Amongst others, equivalence conditions can be formed for each distinct value in an RDF dataset, so a few millions of values typically found in RDF datasets yield trillions of CIND candidates – only for this type of condition. This huge search space poses two major 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 of trillions candidates. We acknowledge that several algorithms have been proposed for the discovery of INDs and CINDs on relational data and we discuss those as related work. However, none of them is suited or could be easily adapted to efficiently discover CINDs within datasets.

Contributions and structure. We study the problem of CIND discovery in RDF data and present RDFIND, the first system to efficiently solve this problem for large datasets. Specifically, we make three main contributions:

(1) We formally define the CIND discovery problem, study its complexity and explain why the discovery of *all* CINDs is not desirable (Section 2). In response, we introduce the new class of *pertinent* CINDs, with an adjustable relevance criterion to filter and rank CINDs based on their extent and implications among each other (Section 3).

[1] `http://wiki.dbpedia.org/Downloads2014`

(2) We introduce RDFIND, which efficiently discovers all pertinent CINDs in a *distributed* fashion (overview in Section 4). In particular, we propose a lazy pruning approach that reduces the CIND search space in two steps; we introduce a compact representation for RDF triples that facilitates enumerating and verifying CIND candidates; and we devise algorithms for efficiently extracting CINDs (Sections 5–7).

(3) We implemented RDFIND on top of Flink[2], a distributed data processing system. We compare it against the state-of-the-art algorithm for CIND discovery using different real-world and synthetic datasets. We show that RDFIND outperforms it by a factor of 419 and scales to large datasets that other solutions cannot compute (Section 8).

We conclude the paper with related work in Section 9 and future directions in Section 10.

2. THE CIND SEARCH SPACE

Before stating the problem of finding all CINDs, we first briefly review the RDF data model [18] and present a new CIND definition for RDF.

RDF data model. An RDF *triple* is a statement of the form (s, p, o), where s is the subject, p is the property, and o is the object. The subject and property is an RDF resource (URI), while the object can also be an RDF literal, i.e., a typed value. We treat blank nodes as URIs. 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, property, and object, respectively. In the remainder of this paper, we can interchangeably use the elements of triples, s, p, and o, in most definitions and algorithms. For clarity, we thus use the symbols α, β, and γ to denote any of these three elements.

CINDs in RDF. In the relational data model, CINDs have been defined by means of an IND, which is embedded in the CIND and is often partially violated, and a pattern tableau that removes the violating tuples [28]. However, the notion of embedded INDs is not appropriate for RDF data. As discussed in Section 1, 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 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 2.1 (CONDITION). *A unary condition is a predicate* $\phi(t) := t.\beta{=}v_1$ *and a binary condition is a predicate* $\phi(t) := t.\beta{=}v_1 \wedge t.\gamma{=}v_2$ *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* $\phi := \beta{=}v_1$ *whenever it is clear.*

DEFINITION 2.2 (CAPTURE). *A capture* $c := (\alpha, \phi)$ *combines the unary or binary condition* ϕ *with a projection attribute* α*, which must not be used in* ϕ*. The interpretation of* c *on an RDF dataset* T *is* $\mathcal{I}(T, c) := \{t.\alpha : t \in T \wedge \phi(t)\}$.

[2] `https://flink.apache.org/`

EXAMPLE 2. *The binary condition* $\phi := p$=rdf:type $\wedge$ o=gradStudent *holds for the triples* t_1 *and* t_2 *from the example dataset in Table 1. From it, we can define the capture* (s, ϕ) *with the interpretation* {patrick, 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 2.3 (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. We say then that an* RDF *dataset* T *satisfies the* CIND *if and only if* $\mathcal{I}(T, c) \subseteq \mathcal{I}(T, c')$.

EXAMPLE 3. *A valid* CIND *for the dataset in Table 1 is* $(s, p$=rdf:type $\wedge$ o=gradStudent$) \subseteq (s, p$=undergradFrom$)$, *because the interpretation of the dependent capture* {patrick, mike} *is a subset of the referenced capture's interpretation* {patrick, mike, tim}.

Naïve CIND discovery problem. A naïve formulation of the CIND discovery problem would be to discover all CINDs in the sense of Definition 2.3 that are satisfied by a given RDF dataset. However, finding all CINDs entails both a huge search space and a large solution set, which (i) renders the problem computationally extremely complex and (ii) bloats results with a sheer amount of highly specific CINDs, ineligible for the applications presented in Section 1. For instance, even one of our smallest datasets, Diseasome[3], with only 72,446 triples comprises over 50 billion CIND candidates and over 1.3 billion (!) actual CINDs. Relational CIND discovery approaches mitigate these issues by considering only a single combination of dependent and referenced projection attributes, by neglecting self-joins, and by considering only referenced conditions [7,16]. As a matter of fact, this reduces the CIND discovery problem to a condition discovery problem. Neither of these simplifications is appropriate for RDF data and consequently RDFIND faces new challenges. Before going into algorithmic details, we define *pertinent* CINDs in the next section to reduce the CIND search space while still covering all of its projection and condition attribute configurations. Then, we revise our problem statement.

3. TAMING THE CIND SEARCH SPACE

The discovery of all CINDs in an RDF dataset must tackle two problems: First, and as explained in the previous section, the search space is enormous, so that pruning strategies are needed. Furthermore, the result space often is also enormous, making the interpretation and usage of CINDs difficult. Again, pruning strategies could help. Therefore, we narrow the search space to *pertinent* CINDs only (Section 3.1), which helps reducing both the search space and the set of results CINDs. Furthermore, we show that association rules are a special class of CINDs and help to further prune the CIND search space (Section 3.2). We then re-define the CIND discovery problem using these two concepts (Section 3.3).

3.1 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

[3] http://datahub.io/dataset/fu-berlin-diseasome

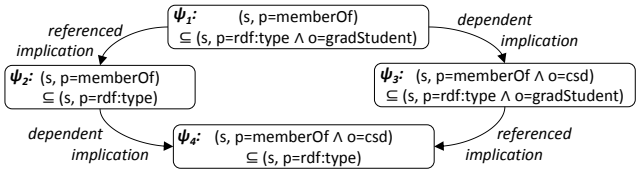

Figure 1: Extract from the cind search space for Table 1. The nodes are cind candidates and the arrows implications.

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.

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 $(s, p$=memberOf$) \subseteq (s, p$=rdf:type $\wedge$ o=gradStudent$)$ from Table 1 is minimal and implies various non-minimal CINDs.

Specifically, we consider two inference rules: the *dependent* and *referenced* implications [28]. Intuitively, tightening the dependent condition of a valid CIND (by making a binary condition unary) or relaxing the referenced condition (by making the unary condition binary) yields a new valid CIND. For instance, $(s, p$=memberOf $\wedge$ o=csd$) \subseteq (s, p$=rdf:type $\wedge$ o=gradStudent$)$ and $(s, p$=memberOf$) \subseteq (s, p$=rdf:type$)$ are also valid, but not minimal, because they can be inferred from the above CIND.

Formally, we denote with $\phi \prec \phi'$ the fact that a unary condition ϕ implies a binary condition ϕ', i.e., the predicate of ϕ is one of the two predicates of ϕ'. For instance, $\phi :=$ p=memberOf implies $\phi' :=$ p=memberOf $\wedge$ o=csd. Consequently, $(\alpha, \phi') \subseteq (\alpha, \phi)$ is a valid CIND if $\phi \prec \phi'$. 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 $\phi_1 \prec \phi_1'$ 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 $\phi_2' \prec \phi_2$ also holds in T, because $(\alpha, \phi_1) \subseteq (\beta, \phi_2) \subseteq (\beta, \phi_2')$. Therefore, we formally define a minimal CIND $(\alpha, \phi_1) \subseteq (\beta, \phi_2)$ as that one where neither ϕ_1 (i.e., its dependent condition) can be relaxed to a condition ϕ_1' nor ϕ_2 (i.e., its referenced condition) can be tightened to a condition ϕ_2' without violating the CIND.

EXAMPLE 4. *Figure 1 depicts four* CINDs *for the dataset from Table 1. The* CIND ψ_1 *implies* ψ_2 *and* ψ_3, *which in turn imply* ψ_4, *respectively. Hence, only* ψ_1 *is minimal.*

Broad CINDs. A broad CIND describes the inclusion of a sufficient number of distinct values. For instance, if we require the inclusion of at least two values, then $(s, p$=rdf:type $\wedge$ o=gradStudent$) \subseteq (s, p$=undergradFrom$)$ 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., $(s, p$=memberOf$\wedge o$=geod$) \subseteq (s, p$=rdf:type$\wedge o$=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., $(o, s$=patrick $\wedge$ p=rdf:type$) \subseteq$ $(o, s$=mike $\wedge$ p=rdf:type$)$). We define this number of distinct values as *support*, inspired from the widely adopted measure for association rules [4]:

Definition 3.1 (Support). *Given an* RDF *dataset* T*, the support of a* CIND $\psi := c \subseteq c'$ *is defined as* $supp(\psi) := |\mathcal{I}(T, c)|$.

Example 5. *The* CIND $(s, p{=}memberOf \wedge o{=}csd) \subseteq (s, p{=}undergradFrom \wedge o{=}hpi)$ *has a support of* 1 *in Table 1. This is because its dependent capture* $(s, p{=}memberOf \wedge o{=}csd)$ *selects a single value. This implies that this* CIND *describes a rather specific insight that pertains only to* patrick.

Formally, a broad CIND has a support equal to or above a given threshold h. The choice of this threshold depends on the use case and its datasets. For instance, according to our experiences $h{=}1,000$ is a reasonable choice the for query minimization use case and $h{=}25$ for the knowledge discovery use case. In Appendix B, we present some more CINDs and their support. Without any loss of generality, we assume this threshold to be given in this paper. 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.

3.2 CINDs as Association Rules

CINDs are natural extensions of regular INDs, but they also share some properties with exact *association rules*[4] (ARs), 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 Table 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 example AR implies the CIND $(s, o{=}$gradStudent$) \subseteq (s, p{=}$rdf:type $\wedge\, o{=}$gradStudent$)$. In contrast, the inverse implication is not necessarily correct, e.g., adding a triple (patrick, status, gradStudent) would invalidate the AR but not the CIND. Also note that ARs can only imply CINDs of the above mentioned kind. In particular, all example CINDs in Section 1 are not implied by ARs. Given this, ARs can replace some CINDs, thereby enhancing the result's understandibility and enabling further applications, such as selectivity estimation [19]. Moreover, AR discovery is less complex than CIND discovery, so we exploit ARs to further prune the CIND search space and improve the efficiency of our CIND discovery algorithm.

3.3 Pertinent CIND Discovery Problem

We now revise the naïve problem statement from Section 2 with the above introduced concepts. For a given dataset T and a user-defined CIND support threshold h, we want to efficiently discover all pertinent CINDs that hold on T, that is, all CINDs that are both minimal and broad. Moreover, if a CIND ψ is implied by an AR r, we want to provide r instead of ψ due to its stronger semantics.

4. SYSTEM OVERVIEW

The typical scenario for RDFind is as follows. RDFind which constitutes a distributable data flow, is implemented and deployed on a distributed data processing system. The input RDF datasets reside in a distributed storage engine, such as HDFS or a triple store. Further assumptions, e.g., on data partitioning, are not made.

[4]The association rules considered in this paper are different from the ones used in [3]. See Section 9 for details.

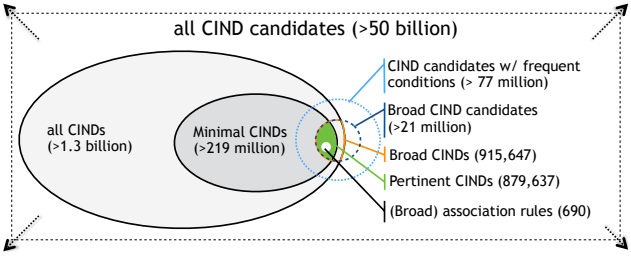

Figure 2: CIND search space for the Diseasome dataset (72,445 triples) and support threshold 10.

Let us now give the overall idea of how RDFind efficiently discovers all pertinent CINDs in large RDF datasets. Intuitively, we need to find the sets of broad CINDs, which satisfy the user-defined support threshold, and the minimal CINDs. Then, their intersection yields the pertinent CINDs. However, in practice, the set of minimal CINDs is often considerably larger than the set of broad CINDs. For example, the Diseasome dataset has approximately 219 million minimal CINDs but fewer than 1 million broad CINDs for a support threshold of 10 (see Figure 2).

Therefore, we devise a *lazy pruning* strategy that reduces the search space in two phases. Figure 3 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 details in the remaining sections. For the implementation details, see Appendix C.

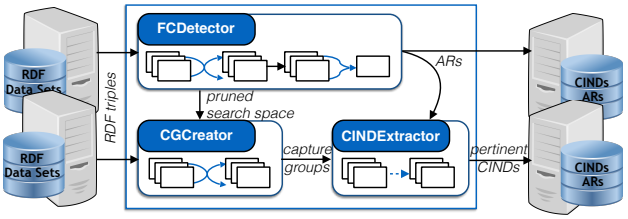

Figure 3: Overview of the RDFind system.

FCDetector. Before initiating the actual search for CINDs, RDFind 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 (see Section 5). RDFind also exploits frequent conditions to easily derive association rules and, consequently, to further prune the search space.

CGCreator. Next, RDFind transforms all RDF triples in the previously pruned search space into compact representations, called *capture groups*, from which it can efficiently extract CINDs (see Section 6). A capture group is a set of captures whose interpretations have a certain value in common. For example, in Table 1 the value patrick spawns a capture group containing, amongst others, the capture

$(s, p{=}\text{rdf:type} \wedge o{=}\text{gradStudent})$.

CINDExtractor. Given the capture groups, RDFind reduces the search space of CINDs with frequent conditions to the set of broad CIND candidates. This is the second phase of our lazy pruning strategy. Subsequently, our system extracts the broad CINDs and their support from the capture groups. As CIND extraction is usually the most expensive step, RDFind 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 (see Section 7).

5. 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 (Definition 2.1) 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 (Section 3.2) 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.

5.1 Why Frequent Conditions?

A frequent condition is that condition whose *condition frequency* (its number of satisfying triples) is not below the user-defined support threshold (Definition 3.1). The support of a CIND is tightly connected with the condition frequency of its dependent and referenced captures, as we assert in the following lemma (proof in Appendix A).

LEMMA 1. *Given a* CIND $\psi := (\alpha, \phi) \subseteq (\beta, \phi')$ *with support* $supp(\psi)$*, the condition frequencies of* ϕ *and* ϕ' *are equal to or greater than* $supp(\psi)$.

Frequent condition pruning. With Lemma 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 4 shows that, for real world-datasets, the vast majority of the conditions are satisfied by only very few triples. For instance, in the DBpedia dataset, 86% of the conditions have a frequency of 1, i.e., they hold only for a single triple, 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.

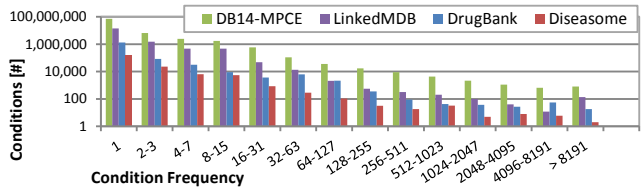

Figure 4: Number of conditions by frequency in real-world datasets of varying size from ~72k (Diseasome) to ~33M (DBP14-MPCE) triples.

Association rule pruning. In addition, frequent conditions allow RDFind to easily derive ARs. As discussed in Section 3.2, the system can use ARs to further prune the CIND search space. 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, Table 1 contains $o{=}\text{gradStudent} \rightarrow p{=}\text{rdf:type}$, 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.

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 $(\alpha, \beta{=}v_1 \wedge \gamma{=}v_2)$ are equal. Accordingly, our above example captures $(s, o{=}\text{gradStudent})$ and $(s, p{=}\text{rdf:type} \wedge o{=}\text{gradStudent})$ 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 $(\alpha, \beta{=}v_1 \wedge \gamma{=}v_2)$; they are equivalent to the candidates involving the capture $(\alpha, \beta{=}v_1)$.

5.2 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]. Using the state-of-the-art Apriori algorithm [4] to find all frequent unary and binary conditions, however, is inefficient as it was designed for single-node settings. Furthermore, it does not scale to large amounts of frequent itemset candidates, because it needs to keep all candidates in memory. Therefore, we devise a fully distributed algorithm that scales to arbitrary amounts of candidates by *checking candidates on-demand* using space-efficient indices. Figure 5 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:

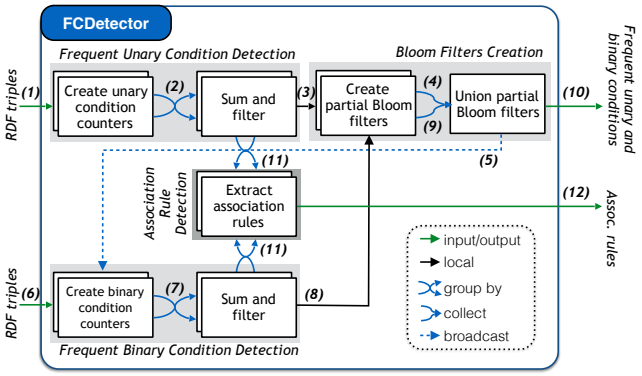

Figure 5: Data flow of the FCDetector.

Frequent unary conditions. We assume that all triples are distributed among the RDFind workers of a cluster, e.g., by means of a distributed triple store. Each worker can then process an independent horizontal partition of an 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 5). For instance, a worker creates the three unary conditions-counters

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

($\langle s$=patrick$\rangle$, 1), ($\langle p$=rdf:type$\rangle$, 1), and ($\langle o$=gradStudent$\rangle$, 1) for the triple (patrick, rdf:type, gradStudent) in Table 1. All workers then run a global `GroupBy`[6] on the conditions of the unary conditions-counters and add up the counters for each resulting group (Step (2)). As this global `GroupBy` 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.

Compacting unary frequent conditions. Once all the frequent unary 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 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)).

Frequent binary conditions. A binary condition can be frequent only if its two embedded unary conditions are frequent [4]. 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 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, csd) from Table 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.

Compacting binary frequent conditions. As for the frequent unary conditions, RDFind encodes all frequent binary conditions in a Bloom filter in order to speed up the CIND pruning phase (Steps (8) and (9)). As a result of this process, RDFind outputs the set of frequent unary and binary conditions, which implicitly represent the pruned CIND search space (Step (10)).

5.3 Extracting Association Rules

As in [4], 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, ($\langle p$=rdf:type$\rangle$, 3) and ($\langle o$=gradStudent$\rangle$, 2) both join with ($\langle p$=rdf:type$\wedge o$=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 ($\langle o$=gradStudent$\rangle$, 2) and ($\langle p$=rdf:type $\wedge$ o=gradStudent$\rangle$, 2), hence, the responsible worker derives the association rule o=gradStudent $\rightarrow$ p=rdf:type with a support of 2. RDFind uses the association rules to further prune the CIND search (see Section 5.1) and additionally includes them in the final result for users (Step (12)), because they are a special class of CINDs (see Section 3). In particular, the association rule support is equal to the support of its implied CINDs according to the following lemma (proof in Appendix A).

Algorithm 1: Create counters for binary conditions

Data: *RDF triples* T, *unary condition Bloom filter* B_u

```
1 foreach t ∈ T do
2     probe ⟨s=t.s⟩, ⟨p=t.p⟩, and ⟨o=t.o⟩ in B_u;
3     foreach (α, β) ∈ {(s,p), (s,o), (p,o)} do
4         v_α ← t.α; v_β ← t.β;
5         if ⟨α=v_α⟩ and ⟨β=v_β⟩ are frequent then
6             forward (⟨α=v_α ∧ β=v_β⟩, 1);
```

LEMMA 2. *The support s of the association rule α=v $\rightarrow$ β=v' is equal to the support of its implied* CIND $(\gamma, \alpha$=$v) \subseteq (\gamma, \alpha$=$v \wedge \beta$=$v')$.

6. 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 groups*. A capture group is a set of captures (Definition 2.2) 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.

6.1 Capture Groups

Our system creates capture groups in two steps. 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 creating the capture groups.

Creating Capture Evidences. A capture evidence is a statement that a certain value exists in a capture (interpretation). 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 2.1 and 2.2). For example, the triple (patrick, memberOf, csd) entails, amongst others, the capture evidences patrick $\in$ (s, p=memberOf) and patrick $\in$ (s, p=memberOf$\wedge o$=csd). One might think that this is an unaffordable task as it would increase the input data volume

[6]This is referred to as *reduce* on Map/Reduce-like systems.

Algorithm 2: Creating capture evidences

Data: *triples* T, *unary condition Bloom filter* B_u, *binary condition Bloom filter* B_b, ARs AR
Result: *Evidences of relevant captures* C

```
1  foreach t ∈ T do
2      C ← ∅;
3      foreach α ∈ {s, p, o} do
4          {β, γ} ← {s, p, o} \ {x};
5          v_α ← t.α; v_β ← t.β; v_γ ← t.γ;
6          if ⟨β = v_β⟩ ∈ B_u then
7              if ⟨γ = v_γ⟩ ∈ B_u then
8                  if ⟨β = v_β ∧ γ = v_γ⟩ ∈ B_b
9                  ∧ ⟨β=v_β → γ=v_γ⟩ ∉ AR
10                 ∧ ⟨γ=v_γ → β=v_β⟩ ∉ AR then
11                     C ← C ∪ {⟨v_α ∈ (α, β=v_β∧γ=v_γ]⟩};
12                 else C ← C ∪ {⟨v_α ∈ (α, β=v_β)⟩, ⟨v_α ∈
                       (α, γ=v_γ)⟩};
13             else C ← C ∪ {⟨v_α ∈ (α, β=v_β)⟩};
14         else if ⟨γ=v_γ⟩ ∈ B_u then
               C ← C ∪ {⟨v_α ∈ (α, γ=v_γ)⟩} ;
```

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 5). 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, csd) and its capture $(s, p{=}\text{memberOf} \wedge o{=}\text{csd})$. The resulting capture evidence patrick $\in (s, p{=}\text{memberOf} \wedge o{=}\text{csd})$ subsumes both patrick $\in (s, p{=}\text{memberOf})$ and patrick $\in (s, o{=}\text{csd})$. Hence, it suffices to keep the first binary capture evidence.

Algorithm 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 5). 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 (Line 3), e.g., $\alpha{=}s$, and two condition attributes (Line 4), e.g., $\beta{=}p$ and $\gamma{=}o$. Then, for the two emerging unary conditions ($p{=}$memberOf and $o{=}$csd), it checks whether they might be frequent using the unary Bloom filter (Lines 4–7). If so, it also checks whether the emerging binary condition ($p{=}\text{memberOf} \wedge o{=}\text{csd}$) is also frequent[7] (Line 8) and does not embed a known AR (Line 9). In this case, it creates a capture evidence with the binary condition only (Line 11; (patrick $\in (s, p{=}\text{memberOf} \wedge o{=}\text{csd})$). 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).

Creating capture groups. RDFind aggregates all capture evidences with the same value using a global `GroupBy` 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 Table 1, a minimum support of 2, and the value patrick, we have the capture evidences patrick $\in (s, o{=}\text{gradStudent})$, patrick $\in (s, p{=}\text{memberOf})$, and patrick $\in (s, p{=}\text{undergradFrom})$. The aggregation combines them into the capture group $\{(s, o{=}\text{gradStudent}), (s, p{=}\text{memberOf}), (s, p{=}\text{undergradFrom})\}$. Note that the capture groups are distributed among the workers after this step and can be processed distributedly in the following.

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

6.2 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 2.3) with the standard IND definition by extending the IND criterion described in [13]. Intuitively, if a capture group contains the referenced capture, then it also contains the dependent capture. Formally:

Lemma 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 $supp(\psi) = |\{G \in \mathcal{G}\colon c \in G\}|$.*

We prove this lemma in Appendix A. 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 1. *Given an RDF dataset T with its capture groups $\mathcal{G}$ and a support threshold h, any valid* CIND ψ *with support $supp(\psi) \geq h$ can be extracted from $\mathcal{G}$.*

Proof. This trivially holds from Lemmata 1 and 3. □

7. FAST CIND EXTRACTION

As final step, in the CINDExtractor component (see Figure 6), RDFind extracts pertinent CINDs from the previously created capture groups. Overall, 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 a simple approach based on an existing IND discovery mechanism to extract broad CINDs is inadequate for RDF (Section 7.1). We then show how RDFind extracts broad CINDs from capture groups efficiently (Section 7.2). We finally show how our system extracts the pertinent CINDs from broad CINDs (Section 7.3).

7.1 Trouble of a Direct Extraction

A simple broad CIND extraction mechanism could be based on an existing IND discovery mechanism (such as [24]). It would work as follows: Because a valid CIND's dependent capture is in the same capture groups as its referenced capture (Lemma 3), all workers enumerate all CIND candidates within their capture groups. They also initialize a support count for each CIND candidate as shown in Example 6.

Example 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 `GroupBy` on the dependent capture of the CIND candidate sets and aggregates them

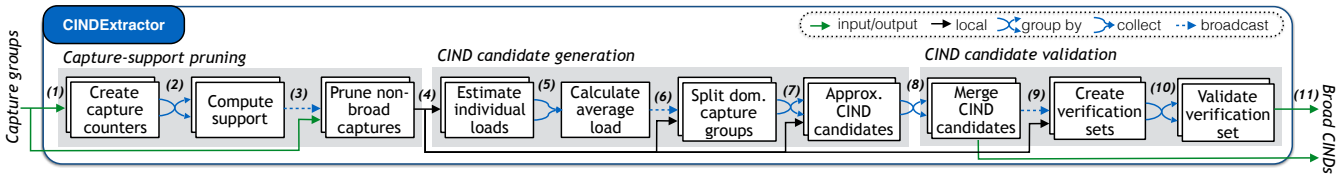

Figure 6: Data flow of the CINDExtractor.

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 [23], 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 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.

7.2 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 6 depicts the complete process.

Capture-Support Pruning. 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 5) 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 6), then broadcasts the prunable captures with a support less than the user-defined support threshold h to each worker, and eventually each worker removes these captures from its capture groups (Step (3)). For example, assuming that h is set to 2, we can then identify in Example 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 capture groups: $G_1' = \{c_a, c_b, c_c, c_d\}$, $G_2 = \{c_a, c_b\}$, and $G_3 = \{c_c, c_d\}$.

CIND Candidate Generation. 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 can identify 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 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 $\mathcal{O}(n)$, where n is the number of captures in a work unit. We experimentally observed that $k = 64\,B$ yields the best performance.

In Example 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.

CIND Candidate Validation. 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 shows this aggregation process, where Boolean functions f_1, f_2, and f_3 mark if 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

Algorithm 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 $\quad C_3 \leftarrow C_1 \cap C_2$;
3 **else if** $hasBloomFilter(C_1) \wedge hasBloomFilter(C_2)$ **then**
4 $\quad C_3 \leftarrow C_1$ & C_2;
5 **else**
6 $\quad C_1' \leftarrow C_i$ where $\neg isBloomFilter(C_i)$;
7 $\quad C_2' \leftarrow C_j$ where $isBloomFilter(C_j)$;
8 $\quad C_3 \leftarrow \{c \in C_1' : c \in C_2'\}$;
9 $count_3 \leftarrow count_1 + count_2$;
10 $f_3 \leftarrow (f_1 \wedge f_2) \vee isEmpty(C_3)$;

as in the basic extraction (Lines 1 & 2 in Algorithm 3). As a result, the system gets *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 $(c_a \sqsubseteq Bloom(c_b, c_c, c_d), 1)^*$ 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 9)). 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 only their 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 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)).

7.3 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 Figure 1). It 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 $\Psi_{2:2}$). Second, it then removes non-minimal CINDs from the latter two by consolidating them with CINDs that have a unary dependent condition and a binary referenced condition ($\Psi_{1:2}$).

We explain the consolidation process by example for $\psi = (s, p{=}\mathsf{memberOf}) \subseteq (s, p{=}\mathsf{rdf{:}type}) \in \Psi_{1:1}$ from Table 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.

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

Table 2: 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

8. EXPERIMENTS

We implemented RDFIND on top of 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 *deal with 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 http://... (double-blind reviewing).

8.1 Experimental Setup

Datasets. 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 2.

Systems. We compare RDFIND to CINDERELLA [7], 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 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 [7], because CINDERELLA is shown to be faster, and not to *Data Auditor* [16], 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 only differ in the conditions generation.

Hardware. We have conducted all experiments on a commodity hardware cluster consisting of a master node (8× 2 GHz, 8 GB RAM) and 10 worker nodes (2× 2.2 GHz, 8 GB RAM each). All nodes are interconnected via Gigabit Eth-

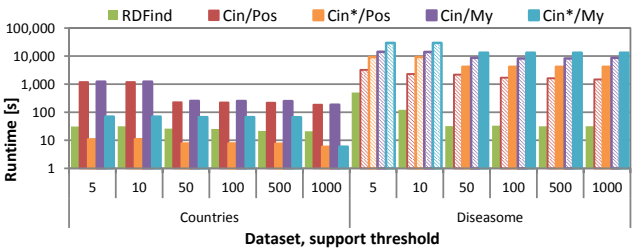

Figure 7: RDFind vs. standard and optimized Cinderella on MySQL and PostgreSQL. Hollow bars indicate algorithm failures and therefore present lower bounds on the execution time.

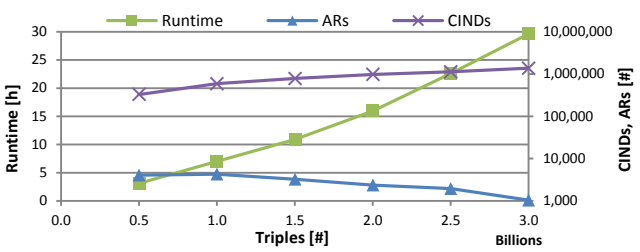

Figure 8: RDFind when increasing the number of input triples with a support threshold of 1,000.

ernet 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.

8.2 RDFind vs. Cinderella

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 [7] on different datasets. As Cinderella is not a distributed algorithm, we ran our experiments only on 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 7 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 of RDFind's relatively high start-up costs, which are dominating on this very small dataset. The larger *Diseasome* dataset yields a different picture, 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.

8.3 Scalability

We proceed to study the scalability of RDFind in terms of both the number of input triples and compute machines.

Scaling the number of triples. We aim at stressing RDFind's robustness with this experiment, 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 8 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, and thereby discovers 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.

Scaling out. We now evaluate the scalability of our system when increasing the number of machines. We consider the medium-size LinkedMDB dataset with a varying support threshold h and number of machines. As Flink allows for task parallelism in a single node through multi-threading, we consider a case with 1, 2, 4, 8, and 10 nodes running a single thread per node. Additionally, we consider a case with 10 nodes running two threads per node.

Figure 9 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 can leverage any provided additional resources.

8.4 Impact of Support Threshold

Having shown the high efficiency and scalability of our system, we now focus on evaluating the efficiency of our system when the number of pertinent CINDs increases. We run an experiment with different support thresholds for a

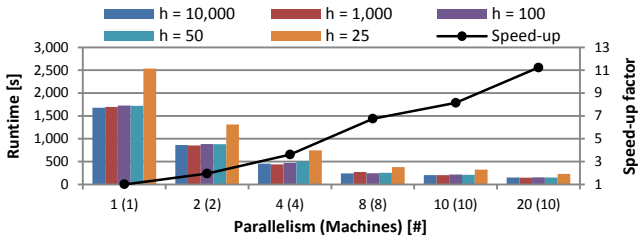

Figure 9: RDFind when increasing the number of machines on LinkedMDB with varying support threshold h.

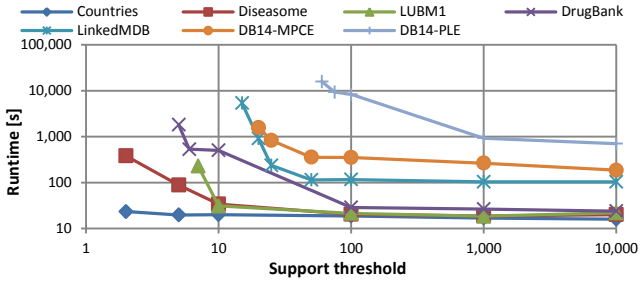

Figure 10: RDFind with different datasets and support thresholds.

variety of datasets.

Impact on runtime. We first evaluate how the number of pertinent CINDs impacts the runtime of our system.

Figure 10 shows the runtimes for our system when discovering pertinent CINDs on multiple datasets and different support thresholds h. The results reveal 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 h below 10. The reason for this is the distribution of conditions w.r.t. their frequencies: As shown in Figure 4, most conditions in datasets hold only on 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 with respect to capture group sizes. However, as we show in the following experiments, the result sizes grow enormously for low support thresholds. Thus, such low support thresholds are presumed to rather be the exception than the rule.

Impact on result. We now evaluate how the number of pertinent CINDs impacts the result of our system. Figure 11 displays the complete number of minimal CINDs (including ARs) that RDFIND discovers for multiple support thresholds. We observe that the number of minimal CINDs is to some extent inversely proportional to the support threshold. Decreasing the support theshold 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 10–20% of all 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

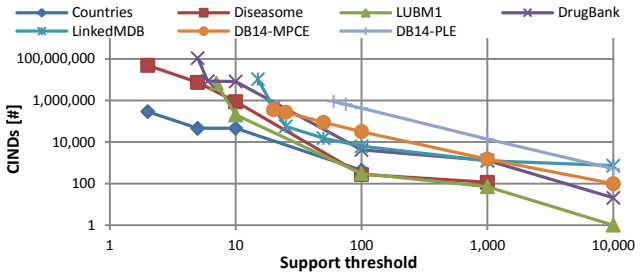

Figure 11: Number of CINDs for different datasets and supports.

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 a support of 41,300. This suggests that the associatedBand property is a subproperty of associatedMusicalArtist and hence it is a hint to revise the ontology. On the other side, low-support CINDs are also important for some datasets. 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 different support thresholds for different use cases. Thus, users need to set their support threshold according to their applications.

8.5 Pruning Effectiveness

We also investigate the effectiveness of our pruning techniques and algorithmic innovations. We compare RDFIND with two simplified versions of RDFIND: RDFIND-DE and RDFIND-NF. RDFIND-*DE* (Direct Extraction) extracts CINDs from capture groups without the capture-support pruning, load balancing, and approximate-first extraction. RDFIND-*NF* (No Frequent Conditions) additionally leaves out any operation related to frequent conditions. Note that we do not separately evaluate the efficiency impact of the implication-based pruning techniques because their main goal is to output the smallest possible, non-redundant result to the user.

Figure 13 shows the results. At a first glance, we observe that RDFIND-DE is slightly more efficient than RDFIND for large support thresholds in four of five cases. This is because the overhead for coping with dominant capture groups in RDFIND is often not redeemed in these scenarios. Note, however, that RDFIND-DE's speed up for the large support thresholds is only 1.07 and RDFIND's accumulated runtime is still 4.6 minutes shorter than RDFIND-DE's runtime. In contrast, for the small support thresholds, RDFIND is much more efficient in all experiments with 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. For example, for the larger DB14-PLE, RDFIND achieves a speed-up of 8.82 at a support threshold

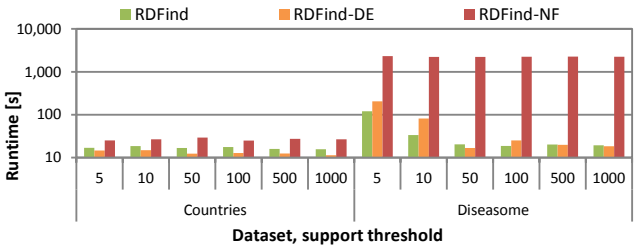

Figure 12: RDFind vs RDFind-DE vs RDFind-NF.

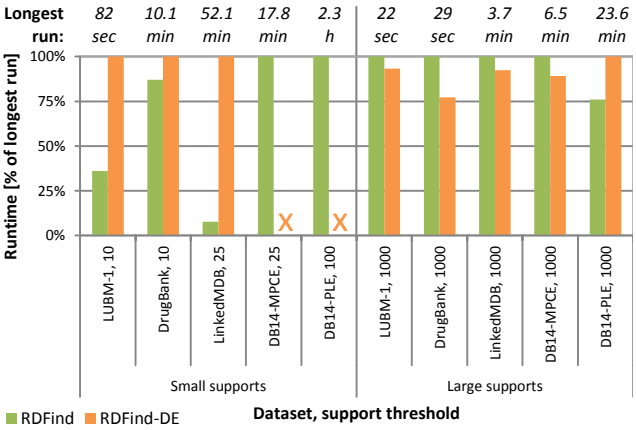

Figure 13: RDFind vs. RDFind-DE for small and large supports. Longest run refers to the execution time of the slower algorithm, respectively. The crosses represent algorithm failures.

of 100, reducing the execution time from over 26 hours to less than 3 hours. Furthermore, these numbers only pertain to the three smaller datasets, because RDFind-DE was not able to handle the larger DB14-MPCE and DB14-PLE due to main memory requirements. These results show the high efficiency and low overhead to cope with dominant capture groups of RDFind.

8.6 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 3.1. We implemented this idea by 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 2, so the overhead of this strategy is greater than its savings. This clearly shows the efficiency of the strategy we follow in RDFind.

9. RELATED WORK

Data profiling is a well-established ongoing research area that seeks to discover different kinds of integrity constraints (ICs) and other metadata, such as functional and inclusion dependencies [1]. However, conditional ICs [10, 15] have attracted only little attention so far, even though they are a valuable extension to plain ICs [1]. We believe this is due to the additional computational complexity of condition discovery aggravating the already complex IC discovery tasks.

CIND discovery. The only existing works on CINDs discovery are the Cinderella/Pli algorithm [7] and *Data Auditor* [16]. However, these works differ from RDFind, because not only do they target relational data, but also their results and algorithmic foundations are fundamentally different. As input they take a *partial* IND, that holds only on a part of the underlying data. Then, they look for conditions that select only the included tuples on the dependent side of the partial IND, potentially allowing for a certain error of the condition. In contrast to RDFind, both algorithms are not directly taking into account conditions on the referenced side of CINDs. This is a simplification of the actual CIND discovery problem, as the choice of the referenced condition influences the set of possible dependent conditions.

IND discovery. IND discovery algorithms for relational data, in particular Mind [13], Binder [31], and Sindy [24], find INDs by (implicitly) joining all columns in a dataset and successively extracting IND candidates with each tuple of the join product. They scale well with the number of tuples in the inspected database but not with the number of IND candidates. RDFind also employs a (distributed) join-extract strategy (with captures instead of columns). However, we enhance and speed up this process by employing lazy pruning and using multi-pass CIND extraction. These enhancements let RDFind scale to huge amounts of CIND candidates, which distinguishes it from all IND discovery algorithms. Moreover, RDFind's load-balancing ensures good scale-out even in presence of dominant capture groups.

RDF data profiling. Given the recent popularity of the LOD initiative [8], there is a plethora of tools for analyzing and profiling RDF data. Most of these systems focus on either RDFS/OWL schema discovery [22, 26] or on gathering statistics of Linked Data [6, 20]. ProLOD++ [2] comprises a wider variety of data profiling and mining tasks for RDF data, such as schema discovery and key discovery. Past research mines *association rules* (different from our ARs) from RDF data that are akin to CINDs of the form $(\alpha, \beta{=}v_1) \subseteq (\alpha, \beta{=}v_2)$ for user-defined α and β [3]. This work focusses on using these rules for RDF data management tasks, such as synonym detection, but not on the discovery itself. While RDFind's CINDs can also support these data management tasks, we show that the much broader class of CINDs can also serve many further use cases.

10. CONCLUSION & FUTURE WORK

This paper 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 and is robust enough to handle large RDF datasets that were not possible to handle before.

For the future, it is intriguing to investigate how the proposed techniques for CIND discovery on RDF data can also be leveraged on relational data and other tasks, such as conditional functional dependency discovery. In any case, supporting the user in finding appropriate support thresholds

and considering other tools, such as the local closed world assumption [14], while pruning is a worthwhile research objective. Finally, 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.

11. REFERENCES

[1] Z. Abedjan, L. Golab, and F. Naumann. Profiling relational data: A survey. *VLDB Journal*, 24(4):557–581, 2015.

[2] Z. Abedjan, T. Grütze, A. Jentzsch, and F. Naumann. Mining and profiling RDF data with ProLOD++. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1198–1201, 2014. Demo.

[3] Z. Abedjan and F. Naumann. Improving RDF data through association rule mining. *Datenbank-Spektrum*, 13(2):111–120, 2013.

[4] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 487–499, 1994.

[5] M. Arias, J. Fernández, M. Martínez-Prieto, and P. de la Fuente. An Empirical Study of Real-World SPARQL Queries. In *International Workshop on Usage Analysis and the Web of Data (USEWOD)*, 2011.

[6] S. Auer, J. Demter, M. Martin, and J. 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.

[7] J. Bauckmann, Z. Abedjan, U. Leser, H. Müller, and F. Naumann. Discovering conditional inclusion dependencies. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 2094–2098, 2012.

[8] C. Bizer, T. Heath, and T. Berners-Lee. Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(3):1–22, 2009.

[9] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. 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.

[10] L. Bravo, W. Fan, and S. Ma. Extending dependencies with conditions. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 243–254, 2007.

[11] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodriguez-Muro, R. Rosati, M. Ruzzi, and D. F. Savo. The MASTRO system for ontology-based data access. *Semantic Web Journal (SWJ)*, 2(1):43–53, 2011.

[12] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1216–1227, 2005.

[13] F. De Marchi, S. Lopes, and J.-M. Petit. Efficient algorithms for mining inclusion dependencies. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 464–476. 2002.

[14] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *Proceedings of the International Conference on Knowledge discovery and data mining (SIGKDD)*, KDD '14, pages 601–610, New York, NY, USA, 2014. ACM.

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

[16] L. Golab, F. Korn, and D. Srivastava. Efficient and Effective Analysis of Data Quality using Pattern Tableaux. *IEEE Data Engineering Bulletin*, 34(3):26–33, 2011.

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

[18] P. Hayes and P. F. Patel-Schneider. RDF 1.1 Semantics. W3C Recommendation, February 2014. `https://www.w3.org/TR/rdf11-mt/`.

[19] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. Cords: automatic discovery of correlations and soft functional dependencies. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 647–658. ACM, 2004.

[20] T. Käfer, A. Abdelrahman, J. Umbrich, P. O'Byrne, and A. Hogan. Observing linked data dynamics. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 213–227, 2013.

[21] Z. Kaoudi and I. Manolescu. RDF in the clouds: a survey. *VLDB Journal*, 24(1):67–91, 2015.

[22] S. Khatchadourian and M. P. Consens. ExpLOD: Summary-based exploration of interlinking and RDF usage in the linked open data cloud. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 272–287, 2010.

[23] L. Kolb and E. Rahm. Parallel Entity Resolution with Dedoop. *Datenbank Spektrum*, 13(1):23–32, 2012.

[24] S. Kruse, T. Papenbrock, and F. Naumann. Scaling out the discovery of inclusion dependencies. In *Proceedings of the Conference Datenbanksysteme in Business, Technologie und Web Technik (BTW)*, pages 445–454, 2015.

[25] M. Levene and M. W. Vincent. Justification for inclusion dependency normal form. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 12(2):281–291, 2000.

[26] H. Li. Data Profiling for Semantic Web Data. In *Proceedings of the International Conference on Web Information Systems and Mining (WISM)*, pages 472–479, 2012.

[27] LUBM. http://swat.cse.lehigh.edu/projects/lubm/.

[28] S. Ma, W. Fan, and L. Bravo. Extending inclusion dependencies with conditions. *Theoretical Computer Science*, 515:64–95, 2014.

[29] M. S. Marshall, R. Boyce, H. F. Deus, J. Zhao, E. L. Willighagen, M. Samwald, E. Pichler, J. Hajagos, E. Prud'hommeaux, and S. 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*, 2012.

[30] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB Journal*, 19(1):91–113, 2010.

[31] T. Papenbrock, S. Kruse, J.-A. Quiané-Ruiz, and F. Naumann. Divide & conquer-based inclusion dependency discovery. *Proceedings of the VLDB Endowment*, 8(7):774–785, 2015.

[32] N. 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.

[33] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava. On multi-column foreign key discovery. *Proceedings of the VLDB Endowment*, 3(1-2):805–814, 2010.

APPENDIX

A. PROOFS

We provide here the proofs to our Lemmata 1–3.

LEMMA 1. *Given a* CIND $\psi := (\alpha, \phi) \subseteq (\beta, \phi')$ *with support* $supp(\psi)$*, the condition frequencies of* ϕ *and* ϕ' *are equal to or greater than* $supp(\psi)$*.*

PROOF. *From the support definition, we have that the interpretation of the dependent capture* (α, ϕ) *contains* $supp(\psi)$ *values. Thus, the referenced capture* (β, ϕ')*, which is a superset, contains at least* s *values. As each value in the interpretation of a capture must be found in at least one triple and each triple yields only 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* $\geq supp(\psi)$*.* □

LEMMA 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.* □

LEMMA 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* $supp(\psi) = |\{G \in \mathcal{G}\colon c \in G\}|$*.*

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*.* □

B. USE CASES

CINDs can play an important role on versatile RDF data management scenarios, as INDs do on relational databases. In the following, we demonstrate three of such use cases, namely *query minimization*, *knowledge discovery* and *ontology reverse engineering*, using examples from real-world datasets.

Query optimization. As RDF data spans a graph, queries involve many joins. With joins being a very costly operation, this impacts the performance of RDF queries. As stated in the introduction, CINDs allow to remove unnecessary joins. For instance, knowing that $(s, p{=}\text{teaching}) \subseteq (s, p{=}\text{rdf:type} \wedge o{=}\text{professor})$ would allow us to remove the second query triple in the following query: `SELECT ?s ?o WHERE {?s teaching ?o . ?s rdf:type professor}`. As a result of such a minimization, the query can be answered with a simple scan of the data, avoiding an expensive join operation. Hence, queries can obtain significant performance gains from query minimization. For example, Figure 14 demonstrates the performance improvement when minimizing and running LUBM query Q2 [27] on RDF-3X [30]. We observe that the minimised version of query $Q2$ is three times faster than the original one.

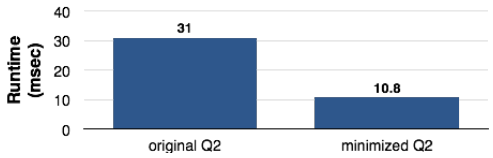

Figure 14: Effect of query minimization using CINDs.

Knowledge discovery. CINDs might reveal unknown facts about data instances that cannot be inferred from the ontology itself. As already mentioned, the CINDs $(s, p{=}\text{writer} \wedge o{=}\text{Angus\_Young}) \subseteq (s, p{=}\text{writer} \wedge o{=}\text{Malcolm\_Young})$ (support: 26) and, vice versa, $(s, p{=}\text{writer} \wedge o{=}\text{Malcolm\_Young}) \subseteq (s, p{=}\text{writer} \wedge o{=}\text{Angus\_Young})$ (support: 26), reveal that the AC/DC members, Angus and Malcolm Young, have written all their songs together. This reveals a fact that is not explicitly stated in DBpedia. A second example is the CIND $(s, p{=}\text{areaCode} \wedge o{=}559) \subseteq (s, p{=}\text{partOf} \wedge o{=}\text{California})$ (support: 98) meaning that cities with code 559 are located in California. Another example from the DrugBank dataset is the CIND $(o, s{=}\text{drug00030} \wedge p{=}\text{target}) \subseteq (o, s{=}\text{drug00047} \wedge p{=}\text{target})$ (support: 14), which reveals that anything cured by drug00030 is also cured by drug00047. There are many more facts that can be unveiled from CINDs.

Ontology reverse engineering. RDF data is not always accompanied by an ontology and even if it is, it does not always follow the ontology constraints. CINDs can provide general insights and reveal statements not reflected by the ontology (if it exists one). For instance, our ARs can be used to discover the RDF classes used in a dataset, e.g., the AR $o{=}\text{lmdb:performance} \rightarrow p{=}\text{rdf:type}$ (support: 197,271) reveals that the URI lmdb:performance is a class. Using CINDs we can also suggest to ontology engineers about class and predicate relationships such as: (i) class hierarchies, (ii) predicate hierar-

chies, and (iii) the domain and range of predicates. For instance, the two CINDs $(s, p{=}\text{associatedBand}) \subseteq (s, p{=}\text{associatedMusicalArtist})$ (support: 33,296) and $(o, p{=}\text{associatedBand}) \subseteq (o, p{=}\text{associatedMusicalArtist})$ (support: 41,300) suggest that the associatedBand property is a subproperty of associatedMusicalArtist. This is because, according to the RDF semantics [18], the set of subjects and the set of objects of the predicate associatedBand is a subset of the set of subjects and set of objects of associatedMusicalArtist, respectively. Similarly, the CIND $(s, p{=}\text{rdf:type} \wedge o{=}\text{Leptodactylidae}) \subseteq (s, p{=}\text{rdf:type} \wedge o{=}\text{Frog})$ (support: 1,010) reveals that the class Leptodactylidae could be a subclass of class Frog. The CIND $(o, p{=}\text{movieEditor}) \subseteq (s, p{=}\text{rdf:type} \wedge o{=}\text{foaf:Person})$ (support: 3,210) reveals that the domain of predicate movieEditor can be the class foaf:Person. Another interesting CIND, such as $(s, p{=}\text{classificationFunction} \wedge o{=}\text{“hydrolase activity”}) \subseteq (s, p{=}\text{classificationFunction} \wedge o{=}\text{“catalytic activity”})$ (support: 812) states that the classification function hydrolase activity is a catalytic activity. This gives hints to the ontology engineer that the activities should be classes and not string literals, so that class hierarchies can be exploited.

Notice that ARs as considered in [3] cannot express all above CINDs as they contain binary conditions and different projection and selection attributes.

C. IMPLEMENTATION DETAILS

We have implemented RDFind on top of the distributed data processing system Flink 0.9.0, which gives our prototype the flexibility both to scale up via multi-threading and scale out via distributed execution. Also, Flink's execution engine provides robust out-of-core execution for joins and aggregations. RDFind preserves the robustness by adding only data structures with a small memory footprint, e.g., Bloom filters. This allows RDFind to run in environments with low memory footprint, yet leveraging any available resources. Furthermore, RDFind accepts N-Triples[9] files as inputs. For the comparison with Cinderella (Section 8.2) these files resided in a local filesystem. For all other experiments (Sections 8.3, 8.4, and 8.5), the files resided in Hadoop Distributed File System, specifically in HDFS 2.7, without any specific partitioning of the triples.

RDFind uses a single Flink job, i.e., a single data flow plan, that implements all three components: FCDetector, CGCreator, and CINDExtractor. The detailed plan can be obtained from the implementation on our repeatability page (see Section 8). Here, we only want to convey the general strategy to implement RDFind on Flink. Each operation in RDFind, e.g., *Create unary condition counters* in Figure 5, is mostly implemented with a single Flink operator and a user-defined function that implements the respective algorithm logic. The communication between adjacent steps is handled by Flink and designated by the type of the receiving operator. We implemented our communication channels in Figures 5 and 6 as follows: (i) *local* communication using Flink's `Map` and `FlatMap` operators, (ii) *group by* using a combination of `GroupBy`, `GroupCombine`, and `GroupReduce`, (iii) *collect* using `GlobalReduce`, (iv) *joins* (multiple *group by* inputs) using a combination of `GroupBy` and `CoGroup`, and (v) *broadcast* using the `Broadcast` command. The only exception is the *group by* for load balancing of dominant capture groups. Here, we use a `FlatMap` operator to split them into work units and distribute the work units with Flink's `Repartition` operator.

[9] https://www.w3.org/TR/n-triples/