Towards Theory for Real-World Data

Wim Martens
University of Bayreuth
Germany
wim.martens@uni-bayreuth.de

ABSTRACT

Fundamental research on data manipulation languages is often motivated by the search for balance between desirable properties, such as expressiveness, robustness, compositionality, the existence of efficient algorithms, etc. Real-world data can be helpful for this search in many different respects. Data sets may exhibit common structures that efficient algorithms can exploit. Query logs and schemas can give us an idea of single features that are used very often, or groups of features that are frequently used together. In this sense, they can guide us towards features or fragments of data manipulation languages that are common in practice and may therefore be worthy of deeper study. In other cases, we may even get a glimpse on features that are not well-understood by users, which may inspire us to redesign them or develop tools that increase their ease-of-use.

This tutorial aims to provide, first of all, an overview on several practical studies that have been conducted in the areas of tree-structured and graph-structured data, with a focus on cases with strong interaction between analysis of the data and fundamental research. Second, it aims to provide a set of lessons learned after the investigation of some large-scale logs consisting of more than 850 million queries.

CCS CONCEPTS

• Information systems → Query languages for non-relational engines; **• Theory of computation → Database query languages (principles)**; *Regular languages.*

KEYWORDS

Graph databases, tree-structured data, XML, JSON, RDF, SPARQL, GQL, Cypher, regular path queries, regular languages, query languages, schema languages

ACM Reference Format:
Wim Martens. 2022. Towards Theory for Real-World Data. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3517804.3526066

1 INTRODUCTION

Database theory studies the underlying mathematical principles of data management. Data management, in turn, studies essentially all aspects of computer science that deal with large amounts of data, which is clearly of huge practical relevance. To facilitate the symbiosis between theory and practice, the main database theory conferences are colocated with database systems conferences, which gives the theory community the opportunity to interact with the broader data management community, whether it is to draw inspiration from current applied research or to disseminate its results to a broad audience. Database theory is therefore a field in which the interaction between theory and practice is an important and challenging aspect.

Another important aspect in database theory is the search for language fragments with desirable properties. Despite their domain-specific nature, data manipulation languages in their entirety are often still too complex or expressive, which leads to undecidability of optimization or even evaluation problems. Therefore, we search for fragments for which we can identify a balance between many factors such as expressiveness, robustness, computational complexity, declarativity, compositionality, etc. Since some of these factors are mutually exclusive (increased expressiveness usually leads to worse computational complexity properties), desirable fragments may differ depending on their intended use.

These two aspects of database theory come together in practical studies that investigate real-world data in order to understand how data manipulation languages are used in practice. Such studies can help us understand how real-world data is organized, they can identify practically relevant features of query or schema languages, or they can identify subsets of languages that suffice to cover important cases that we see in the wild. In short, they can give database researchers yet another perspective in the search for a good balance between expressiveness, usefulness, and computational complexity.

The idea of using practical studies for identifying useful language fragments is not new. Already *conjunctive queries*, the most studied query language in the field (and a fragment of first-order logic or relational algebra) were at least partly motivated by a practical study [27]. In the early years of database theory, the language *Query-by-Example* [90], designed to appeal to the non-professional user with little mathematical background, was based on a core of conjunctive queries. A practical study of Thomas and Gould [85], performed on a group of 39 students, showed that this core is learnt and used most readily.

The advent and steep growth of the Web in the 21st century has made different kinds of practical studies possible. Ever since the Web made real-life data available for research in an unprecedented scale, we have seen practical studies that analyze this data, providing many valuable insights. This paper provides an overview of various practical studies on data sets, schemas, and queries for tree- and graph-structured data, together with a selection of insights that are relevant to fundamental research. It will also identify areas in which, to the best of the author's knowledge, such insights are still largely missing. We will see examples showing that

(1) a theoretical perspective can add significant value to practical studies and
(2) such studies can motivate new and exciting research questions, both theoretical and practical.

Concerning (1), theoreticians are trained to have a deep understanding of data manipulation languages and can pinpoint combinations of features that will lead to intractability or other undesired behavior. Concerning (2), the paper will provide several concrete examples how practical evidence motivated new research avenues. (There will be a focus on fundamental and theoretical questions.)

This paper will *not* deal with user studies or questionnaires, which are another important tool to get an idea for real-life usage of data. The focus on tree- and graph-structured data is due to the broad availability of such data on the Web. Despite the effort to be comprehensive, there will undoubtedly be studies that are not covered here.

We will discuss both real-world objects in this paper (e.g., when discussing practical studies) and their theoretical abstractions. The reader should be aware that theoretical abstractions are not intended to cover every feature of the real-life object but usually aim at presenting a simple and elegant definition that captures its essence. We will assume basic familiarity with the real-world objects treated in the paper.

The paper is structured as follows. After presenting preliminaries on regular expressions (Section 2), which are a recurring theme through the paper, we have two blocks (Sections 3–6 and 7–10) that give an overview on practical studies on tree-structured and graph-structured data, respectively. The structure of these two blocks is identical: we discuss data sets, schemas, queries, and conclude. Section 11 provides some lessons learned after having done a fair amount of practical studies. We wrap up in Section 12.

2 PRELIMINARIES

Let Lab denote a countably infinite set of *labels*. We use Lab as an abstraction for the set of labels that can be assigned to nodes in trees (such as *element names* in XML or *keys* in JSON) or to edges in graphs (such as *IRIs* in RDF). Following convention, we usually use the letter Σ to denote finite subsets of Lab and use the term *alphabet* to refer to such sets.

The set of regular expressions over Lab is inductively defined as follows. The expressions $\emptyset$ and ε are regular expressions over Lab, each symbol $a \in \text{Lab}$ is a regular expression over Lab and, if e_1 and e_2 are regular expressions over Lab, then so are $e_1 \cdot e_2$, $e_1 + e_2$, e_1^*, $e_1?$, and e_1^+. As usual, we often omit the concatenation symbol "$\cdot$". Notice that a regular expression e over Lab can only use a finite subset Σ of Lab. We therefore also refer to e as a regular expression over Σ. By RE we denote the class of all regular expressions.

A *word (over Σ)* is a sequence $a_1 \cdots a_n$ where $a_i \in \Sigma$ for every $i \in \{1, \dots, n\}$. By ε we denote the empty word (i.e., the case where $n = 0$). A *word language* is a set of words. Assume that L, L_1, and L_2 are word languages. We define $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1$ and $w_2 \in L_2\}$ and $L^n = \{w_1 \cdots w_n \mid w_i \in L$ for every $i \in \{1, \dots, n\}\}$.

The *language* $L(e)$ of a regular expression e over Σ is the set of words over Σ inductively defined as follows. For the base case, we have $L(\emptyset) = \emptyset$, $L(\varepsilon) = \{\varepsilon\}$, and $L(a) = \{a\}$. Furthermore, $L(e_1 \cdot e_2) = L(e_1) \cdot L(e_2)$, $L(e_1 + e_2) = L(e_1) \cup L(e_2)$, $L(e_1^*) = \cup_{i=0}^{n} L(e_1)^i$, $L((e_1?) = L(e_1) \cup \{\varepsilon\}$, and $e_1^+ = \cup_{i=1}^{n} L(e_1)^i$. The expressions e_1 and e_2 are *equivalent*, if $L(e_1) = L(e_2)$. A word language L is *regular* if there exists a regular expression e with $L = L(e)$.

3 TREE-STRUCTURED DATA SETS

We start with discussing practical studies for tree-structured data because of historic reasons. Data sets, schemas, and queries for *XML (eXtensible Markup Language)* started to become publicly available on the Web since the early 2000's. Today, we have similar material available for data in *JSON (JavaScript Object Notation)*. We treat both of these formats in Sections 3–6. There is an emphasis on XML because, to the best of the author's knowledge, the interaction between practical studies and fundamental research has been more pronounced for XML than for JSON. Due to the focus on underlying principles, the insights gained from analyzing XML data may also be relevant to JSON.

Theoretical research usually abstracts XML and JSON data as labeled trees. We denote a *node-labeled tree* as a tuple $T = (V, E, \text{lab})$ where

- V is its finite set of *nodes*,
- $E \subseteq V \times V$ is its *child relation*, and
- lab : $V \rightarrow$ Lab associates a label to each node.

A tree has a unique node r without incoming edges, which is its *root*. For XML data, the trees are formed by considering the nesting structure of their tags (also called "element names"). Here, the label of each node is the corresponding tag in the XML document. In this sense, the trees are always *ordered*, which means that the children of a node are always considered as an ordered sequence $u_1, \dots, u_n$ rather than a set. Although JSON data allows a mix between ordered and unordered content in its tree structure (arrays are ordered, while sets of objects are unordered), this is not crucial for this paper.

Example 3.1. Figure 1 contains an XML document, a JSON document containing similar data, and a labeled ordered tree showing the data's tree structure. We note that there is not a single "correct" way to model XML or JSON data as node-labeled trees. Some ambiguity is already present in Figure 1: depending on the theoretical properties one wants to study, one could also model the `pers_id` attribute name of Figure 1a as a child of the `person`-nodes in Figure 1c. Similarly, one could also add nodes that are labeled with the data values: the attribule value "1", the name "Aretha", etc.

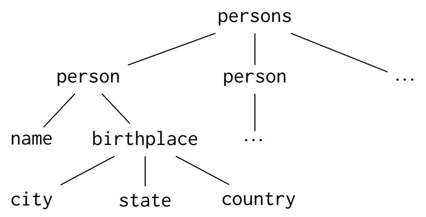
The abstraction of XML or JSON data as node-labeled trees is sufficient for modeling XML and JSON data for many purposes, but certainly not for all. In XML, for example, the accompanying schema may declare some attributes as ID / IDREF, which play a similar role as key / foreign key constraints. In this sense, one may want to consider an IDREF value as an edge to the referred node, in which case the underlying structure of the data may no longer be a tree. In the case of JSON, it has been argued that edge-labeled trees represent the data more naturally [24]. For this paper, this difference is not important.

```
<persons>
  <person pers_id="1">
    <name> Aretha </name>
    <birthplace>
      <city> Memphis </city>
      <state> Tennessee </state>
      <country> United States </country>
    </birthplace>
  </person>
  <person pers_id="2">
    ...
  </person>
  ...
</persons>
```

(a)

```
{"persons": [
  {"person":
    { "pers_id":"1",
      [{"name":"Aretha"},
       {"birthplace":
          [{"city":"Memphis"},
           {"state": "Tennessee"},
            {"country", "United States"}]}]],
    }
  },
  {"person":
    { "pers_id":"2", ...
  }, ...
}
```

(b)



(c)

Figure 1: An XML document, a JSON document, and a tree

3.1 Practical Studies

To the best of the author's knowledge, XML or JSON data sets have not yet been subjected to a large-scale practical study that focuses on their structure. However, real-life XML data has been extensively used for benchmarking, which led to rudimentary analysis of several prominent data sets, like XML versions of DBLP [35], UPenn's Treebank data [60], and Swissprot [84]. The focus usually was on explaining the difference between the data sets to justify possible run-time differences between experiments, rather than generating deep insights about the data. We know that DBLP has a depth of 7, Treebank has depth 37, and Swissprot has depth 6. Since the number of nodes in these data sets runs in the millions, their tree structure is broad and shallow. This structure of XML trees has been exploited in, e.g., XML document compression [26, 41].

A notable study by Grijzenhout and Marx [46] collected 180k unique XML files together with their schema (if available), and obtained that 85% of the XML files is well-formed. When studying which errors occur and to which extent they can be automatically repaired, they identified 74 different error categories, 9 of which were already responsible for 99% of the errors in well-formedness. This situation seems promising for (semi-)automatic repair. The three most prominent error types, already responsible for 79.9% of the errors, were opening and ending tag mismatch, premature end of data in a tag, or improper UTF-8 encoding of the file.

Concerning JSON, we see different usage patterns in practice than for XML, for instance because JSON is a popular format for APIs. Although large data sets are available in JSON format, JSON data is also available for large collections of small, similarly structured documents, such as tweets on Twitter. This property of JSON collections has been used to, e.g., develop efficient algorithms for analytics on JSON-structured data in databases [38].

4 SCHEMAS FOR TREE-STRUCTURED DATA

Another way to get insight in the structure of data is by analyzing its schema. Schema analysis has attracted a fair amount of attention, both for XML [16, 28, 61, 64, 76] and for JSON [9, 57].

The aforementioned study of Grijzenhout and Marx [46] discovered that only 25% of their XML files contain a reference to a valid schema. Furthermore, only just over 10% of the well-formed documents are valid with respect to their schema. Although this may sound alarming at first, it also needs a bit of perspective. For instance, it may be possible that a schema is referenced but cannot be retrieved. Since schemas can reference each other, this may already happen because one of the schemas referenced in the "main" schema file was replaced by a newer version under a different URL.

4.1 Document Type Definitions (DTDs)

Early practical studies on schemas for XML data focused on rudimentary properties of *Document Type Definitions* or *DTDs*. We provide a formal abstraction based on [4].

Definition 4.1. A *Document Type Definition (DTD)* over alphabet Σ is a triple $d = (\Sigma, \rho, S)$, where

- $\Sigma \subseteq \mathrm{Lab}$ is a finite set of *labels*,
- ρ is a function from Σ to the regular expression over Σ, and
- $S \subseteq \Sigma$ is a set of *start labels*.

We say that a labeled ordered tree $T = (V, E, \mathrm{lab})$ is *valid* w.r.t. d if $\mathrm{lab}(r) \in S$ (where r is the root of T) and, for every $v \in V$ with ordered sequence of children $v_1, \ldots, v_n$, the word $\mathrm{lab}(v_1) \cdots \mathrm{lab}(v_n)$ is in $L(e)$, where $e = \rho(\mathrm{lab}(v))$. In real-world DTDs, ρ is usually written as a set of rules rather than a function. We will follow this convention and write $a \longrightarrow e$ if $\rho(a) = e$.

Example 4.2. The tree in Figure 1c satisfies the DTD consisting of the rules

$$\begin{array}{lcl} \texttt{persons} & \longrightarrow & \texttt{person}^* \\ \texttt{person} & \longrightarrow & \texttt{name birthplace} \\ \texttt{birthplace} & \longrightarrow & \texttt{city state country?} \end{array}$$

and start labels $S = \{\texttt{persons}\}$. The set Σ is implicitly defined as the set of labels that occur in the rules of the DTD.

Whereas the present definition of DTDs has been widely used in research, it omits some aspects of the language. We will touch upon some of these later, such as *deterministic regular expressions.*

Early Studies. Early studies on the use of DTDs in practice were performed on small sets of schemas, consisting of twelve [76] or sixty [28] DTDs. Sahuguet [76] concluded, among others, that (1) even though DTDs should prescribe what are well-formed documents, many of them are erroneous themselves; and (2) it seemed that users wanted to express the unordered concatenation operator "&" from SGML[1] but, being absent in DTD, they used workarounds

[1]The unordered concatenation $a_1 \& \cdots \& a_n$ defines the set of words that consist of permutations of $a_1, \ldots, a_n$.

such as encoding $(a\&b\&c)$ as $(a+b+c)^*$. Notice that the latter is a drastic overapproximation of $(a\&b\&c)$, which is actually equivalent to $abc + acb + bac + bca + cab + cba$.

Choi [28] investigated somewhat deeper structural properties of DTDs such as *recursion*. Here, a DTD $d = (\Sigma, \rho, S)$ is *recursive*, if the directed graph with nodes Σ and edges $\{(a, b) \mid b$ appears in some word in $\rho(a)\}$ has a directed cycle. While 35 out of 60 DTDs in Choi's sample were recursive, the non-recursive DTDs allowed for trees up to depth 20. On the theoretical side, it is known that recursive DTDs can pose challenges, e.g., for streaming validation algorithms. Although the non-recursive DTDs are precisely those for which a constant-memory streaming validation algorithm exists [79], the picture is much less clear if we can already assume that the XML document is well-formed [78].

4.2 Regular Expressions in DTDs

DTDs mainly prescribe the structure of their data using the regular expressions e in the rules of the form $a \to e$. As such, a number of studies has focused on getting a clearer picture on the use of these expressions in practice [16, 50, 53, 54, 68].

4.2.1 Deterministic Regular Expressions. Choi [28] performed a first analysis on the regular expressions in DTDs. In his corpus, the parse depth of the regular expressions was 1 to 9, which tells us that these expressions are structurally not very complex. Furthermore, he observed that a number of DTDs use *non-deterministic* regular expressions, which is in violation of the XML standard [39, Appendix D]. Here, a regular expression is deterministic if, when reading a word from left to right without looking ahead, it is always clear to where the current symbol can be matched in the expression.[2] For example, $e = (a + b)^* a$ is not deterministic, because if a word starts with label a, we need to look ahead to see if we reached the end of the word in order to know if we need to match to the first or second a in expression e. On the other hand, $e' = b^*a(b^*a)^*$ is deterministic and equivalent to e.

Understanding determinism in regular expressions is theoretically challenging. Brüggemann-Klein and Wood discovered that deterministic expressions[3] cannot define all regular languages [25] and provided a characterization of the definable languages in terms of their minimal deterministic finite automaton. In particular, the expression $(a + b)^*a(a + b)$, which is very similar to the aforementioned expression e does not have an equivalent deterministic regular expression. In fact, it is PSPACE-complete to decide if a given regular expression can be transformed to a deterministic one [33, 56] and determinizing regular expressions can give rise to an unavoidable exponential blow-up [55]. The canonical translation from regular expressions to deterministic regular expressions goes through deterministic finite automata. While it is known that there are unavoidable exponential blow-ups from regular expressions to deterministic finite automata, and from deterministic finite automata to deterministic regular expressions [55], it is not known if there are cases in which the determinization of a regular expression can cause an unavoidable double exponential blow-up.

Finally, we note that determinism in regular expressions is also required in XML Schema [88, Section 3.8.6.4], the successor of DTD which we will discuss in Section 4.3. Here, the constraint is called the *unique particle attribution* constraint.

4.2.2 Sequential and Chain-like Regular Expressions. In 2004, Bex et al. [16] performed a detailed structural analysis on regular expressions in DTDs. They collected a sample of 103 DTDs and discovered that over 92% of the regular expressions used in DTDs have a highly constrained syntactical structure. We make this more precise.

Definition 4.3 (Sequential Regular Expression). A *simple factor (over Σ)* is a regular expression of the form $(a_1 + \cdots + a_k)$, $(a_1 + \cdots + a_k)?$, $(a_1 + \cdots + a_k)^*$, or $(a_1 + \cdots + a_k)^+$ where $\{a_1, \ldots, a_k\} \subseteq \Sigma$. A *sequential regular expression (over Σ)* is a regular expression of the form $f_1 \cdots f_n$, where each f_i is a simple factor.

For example, a^*abb^* and $(a+b)^*a(a+b)?$ are sequential regular expressions, where as $(a^* + b^*)$ is not. Notice that sequential regular expressions are not necessarily deterministic. Variants of such expressions were first studied in [62] under the name *simple regular expressions*, *chain regular expressions* [18], and *extended chain regular expressions* [63].

The discovery that the majority of expressions in practical schemas has a very restricted syntax can be theoretically interesting. For instance, fundamental problems on DTDs such as containment and intersection non-emptiness reduces to the same problems on regular expressions [63]. Furthermore, the usual worst-case complexity analysis for these problems on regular expressions does not work with these restricted expressions. We define these problems next and use $\mathcal{R}$ to denote a class of regular expressions.

$\mathcal{R}$-Containment	
Given:	Two regular expressions e_1 and e_2 from $\mathcal{R}$.
Question:	Is $L(e_1) \subseteq L(e_2)$?

$\mathcal{R}$-Intersection	
Given:	Regular expressions $e_1, \ldots, e_n$ from $\mathcal{R}$.
Question:	Is $L(e_1) \cap \cdots \cap L(e_n) \neq \emptyset$?

Both problems are PSPACE-complete for general regular expressions [51, 83]. Notice that the intersection problem considers an arbitrary number of regular expressions. If we would consider the intersection problem for a fixed number of expressions, it can always be solved in polynomial time by standard automata-theoretic methods (construct a product automaton for the intersection and test non-emptiness).

Motivated by understanding the complexity of containment, equivalence, and intersection non-emptiness of real-life schemas, Martens et al. [63] revisited the complexity of these problems for sequential regular expressions, where fragments with different types of factors were considered.[4] We denote such fragments as $\mathrm{RE}(f_1, \ldots, f_k)$, where $f_i \in \{a, a?, a^*, a^+, (+a), (+a)?, (+a)^*, (+a)^+\}$ for all $i \in \{1, \ldots, k\}$. Here, "a" always stands for an arbitrary symbol from Σ. The factor types $(+a)$, $(+a)?$, $(+a)^*$, and $(+a)^+$ allow disjunction in factors, whereas the types a, $a?$, a^*, a^+ do not. For instance, $\mathrm{RE}(a, a^*)$ denotes those expressions in which every factor is a single symbol $a \in \Sigma$ or is of the form a^* with $a \in \Sigma$. For instance, the expression ab^*a^*ab is of this form. The following is known:

Theorem 4.4 ([63]). *(a) $RE(a, a^+)$-Containment is in PTIME.*

[2] For a formal definition, we refer to [25, 33].

[3] They referred to determinism in regular expressions as *one-unambiguity*.

[4] The work also considers more expressive factors, which we do not treat here.

(b) $RE(a, (+a))$-Containment *is in PTIME.*
(c) $RE(a, a^*)$-Containment *is coNP-complete.*
(d) $RE(a, a?)$-Containment *is coNP-complete.*
(e) $RE(a, (+a)?)$-Containment *is coNP-complete.*
(f) $RE(a, (+a)^*)$-Containment *is PSPACE-complete.*
(g) $RE(a, (+a)^+)$-Containment *is PSPACE-complete.*

While (a–b) are easy to see because expressions in $\mathrm{RE}(a, a^+)$ and $\mathrm{RE}(a, (+a))$ can be translated to deterministic finite automata in polynomial time, the arguments for (c–g) are less trivial and shown in [63]. The hardness in items (c) and (d) is perhaps remarkable since the involved regular languages are so inexpressive. (We provide a proof sketch in Appendix A.) Remarkably, testing *equivalence* of expressions in $\mathrm{RE}(a, a^*)$ or $\mathrm{RE}(a, a?)$, i.e., testing if they define the same language, is in PTIME [63], despite Theorem 4.4(c–d). Finally, containment of $\mathrm{RE}(a?, (+a^*))$ is in PTIME [1]. The reason for the latter result is that, since the languages of such expressions are closed under taking subsequences, containment can be checked with a greedy strategy. Details can be found in Abdulla et al. [1].

Concerning intersection, the problem is PSPACE-complete for general regular expressions [51], but the complexities for sequential regular expressions are lower:

Theorem 4.5 ([63]). *(a)* $RE(a, a^+)$-Intersection *is in PTIME.*
(b) $RE(a, (+a))$-Intersection *is in PTIME.*
(c) $RE(a, a^*)$-Intersection *is NP-complete.*
(d) $RE(a, a?)$-Intersection *is NP-complete.*
(e) $RE(a, (+a)?)$-Intersection *is NP-complete.*
(f) $RE(a, (+a)^*)$-Intersection *is NP-complete.*
(g) $RE(a, (+a)^+)$-Intersection *is NP-complete.*

Here, the argument for (a) is based on a normal form of the expressions, from which intersection can easily be decided. Item (b) is easy to see, because all words in the language have the same length. The upper bounds for (c–g) hold because it is possible to guess a polynomial-size representation of a candidate witness word w in the intersection of the languages, and to test in polynomial time if w (which possibly has exponential length) is in each of the languages. The lower bounds for (c–d) and (g) are non-trivial.

4.2.3 k-Occurrence Regular Expressions. Another kind of expressions studied by [62] are now known as *k-occurrence regular expressions (k-OREs)*, which are regular expressions in which every alphabet symbol occurs at most k times. The basic decision problems for k-OREs have the following complexities.

Theorem 4.6 ([63]).
(a) *k-ORE*-Containment *is in PTIME for every fixed $k \in \mathbb{N}$.*
(b) *k-ORE*-Intersection *is PSPACE-complete for every fixed $k \geq 3$.*

Theorem 4.6(a) holds because a k-ORE over Σ can always be converted to a deterministic finite automaton with at most $|\Sigma| \cdot 2^k$ states. The upper bound in Theorem 4.6(b) already holds for general regular expressions, whereas the lower bound is less trivial.

By re-investigating the data of Bex et al. [16], it was discovered that over 99% of the regular expressions in DTDs and XML Schema Documents are 1-OREs, also known as *single-occurrence regular expressions* or *SOREs* [17].

SOREs and k-OREs turn out to be interesting for *schema inference.* Here, the task is, given a set of trees $\{T_1, \ldots, T_n\}$, to compute a DTD D such that $\{T_1, \ldots, T_n\} \subseteq L(D)$ and such that D is useful in a similar way as a hand-crafted schema would be. To this end, it is important that D is not too specific (we should not take $L(D) = \{T_1, \ldots, T_n\}$) and not too general (we should not take $L(D)$ to be the set of all trees).

Since above scenario for schema inference only provides *positive examples* for an inference algorithm, the notion of *learning in the limit* [43] has been adopted for schema inference.

Definition 4.7. A *sample* is a finite set of words over Σ. Let $\mathcal{R}$ be a class of regular expressions. An algorithm A that maps samples to expressions in $\mathcal{R}$ is said to *learn $\mathcal{R}$ from positive data* if (1) $S \subseteq L(A(S))$ for every sample S and (2) to every $e \in \mathcal{R}$ we can associate a so-called *characteristic sample* $S_e \subseteq L(e)$ such that, for each sample S with $S_e \subseteq S \subseteq L(e)$, we have that $A(S)$ is equivalent to e.

A class $\mathcal{R}$ of regular expressions is then said to be *learnable from positive data* if an algorithm exists that learns $\mathcal{R}$ from positive data. Gold [43] already showed that the class of regular expressions is not learnable in the limit. Bex et al. [15, Theorem 1.4] show:

Theorem 4.8. *The class of deterministic regular expressions is not learnable from positive data.*

For this reason, also DTDs are not learnable from positive data. However, this is different for k-OREs.

Theorem 4.9. *For each fixed k, the class of deterministic k-OREs is learnable from positive data.*

The algorithm to prove this result, however, is not very practical as it does not provide a method to automatically determine the best value of k for the given sample and, in some cases, needs exponentially many samples in the alphabet of the target expression [15].

But the result did encourage to dig deeper. Bex et al. [15] develop and experimentally evaluate an algorithm iDRegEx which learns deterministic k-OREs for increasing values of k. The method is probabilistic, based on Hidden Markov Models, and it first learns an automaton for the sample S, which is then translated into a k-ORE. In a companion paper [18], Bex et al. present an inference algorithm for regular expressions that are both SOREs and sequential regular expressions, which still constitute over 90% of the regular expressions in the corpus of [16]. The algorithm performs well in practice, even in scenarios with little data available [18].

4.3 XML Schema

The second major schema language for XML is *XML Schema* [88]. In a nutshell, XML Schema adds to DTDs (a) a wider range of *simple types*, such as dates, floats, integers, etc. and (b) the ability to use *complex types*. In our theoretical abstraction, we focus on the latter, which we define in two steps.

Definition 4.10. An *extended DTD (EDTD)* is a tuple $D = (\Sigma, \Gamma, \rho, S, \mu)$, where

- $\Sigma \subseteq \mathrm{Lab}$ is a finite set of *labels*,
- $\Gamma \subseteq \mathrm{Lab}$ is a finite set of *types*,
- (Γ, ρ, S) is a DTD over alphabet Γ, and
- $\mu : \Gamma \to \Sigma$.

A labeled ordered tree $T = (V, E, \mathrm{lab})$ is *valid* w.r.t. an EDTD if there exists an assignment of types to the nodes in V such that the typed tree is valid w.r.t. the underlying DTD (Γ, ρ, S). Formally, for a tree T^Γ using labels in Γ, let us denote by $\mu(T^\Gamma)$ the tree obtained from T^Γ by replacing each label $a \in \Gamma$ with $\mu(a) \in \Sigma$. Hence, $\mu(T^\Gamma)$ only uses labels in Σ. We now say that T *satisfies* D if there exists a tree T^Γ such that $T^\Gamma \in L((\Gamma, \rho, S))$ and $\mu(T^\Gamma) = T$. We call T^Γ a *witness* for T. Again, we denote the set of trees satisfying D by $L(D)$.

Example 4.11. Consider the EDTD consisting of the rules

```
persons            →  person*
person             →  name (birthplace-US + birthplace-Intl)
birthplace-US      →  city state country?
birthplace-Intl    →  city state country
```

with μ(birthplace-US) = μ(birthplace-Intl) = birthplace. This schema assigns different types to birthplaces: birthplace-US to birthplaces for which country is optional, and birthplace-Intl for the others. The tree in Figure 1c is in the language of the schema.

The schema in Example 4.11, however, does not satisfy XML Schema's *Element Declarations Consistent* constraint, which forbids the occurrence of different types associated to the same element name in a regular expression $\rho(t)$. In the example, both birthplace-US and birthplace-Intl are indeed on the same right-hand side of a rule and are associated to the same element name birthplace.

Definition 4.12. Let $D = (\Sigma, \Gamma, \rho, S, \mu)$ be an EDTD. A regular expression r over Γ is *single-type* if it does not contain distinct types t_1 and t_2 with $\mu(t_1) = \mu(t_2)$. We say that D is a *single-type EDTD (stEDTD)* when every regular expression $\rho(t)$ is single-type and S does not have distinct types t_1 and t_2 with $\mu(t_1) = \mu(t_2)$.

Structurally, XML Schema corresponds closely to single-type EDTDs. The most substantial difference between single-type EDTDs and XML Schema is that XML Schema has an additional restriction on regular expressions in ρ, called *Unique Particle Attribution*, which mimics the determinism constraint from DTDs.

Problems such as Intersection and Containment for XML Schema or single-type EDTDs are known to reduce to the corresponding problems for regular expressions [63], which we already discussed in Section 4.2. Similarly, although XML Schema inference is more involved than DTD inference, the regular expression inference which we discussed in Section 4.2 remains at its core [18].

4.4 XML Schema Complex Types in Practice

The usage of XML Schema in practice has been investigated in several studies [16, 52, 61, 64]. Bex et al. [16] investigated the use of complex types in 30 XSDs. They concluded that 25 out of these 30 are structurally equivalent to a DTD, whereas the remaining five use complex types beyond the power of DTDs. These five schemas use types such that a node is always assigned with a type that depends on its label, its parent's label, or its grandparent's label. An example of such a schema can be found in Figure 2a, where we use the types $\Gamma = \{a, b, c, d^1, d^2, e, f, g, h^1, h^2, i, j, k\}$ and define μ as $\mu(d^1) = \mu(d^2) = d$, $\mu(h^1) = \mu(h^2) = h$, and the identity elsewhere. Similar findings were confirmed on larger sets of schemas [61, 64].

The observation of how complex types are used in practical schemas gave rise to the development of a schema language called *BonXai* [61], which is equivalent to XML Schema (and thus includes

$$\begin{array}{rcl} a & \longrightarrow & b + c \\ b & \longrightarrow & ed^1f \\ c & \longrightarrow & ed^2f \\ d^1 & \longrightarrow & gh^1i \\ d^2 & \longrightarrow & gh^2i \\ h^1 & \longrightarrow & j \\ h^2 & \longrightarrow & k \end{array}$$

(a)

$$\begin{array}{rcl} a & \longrightarrow & b + c \\ b & \longrightarrow & edf \\ c & \longrightarrow & edf \\ d & \longrightarrow & ghi \\ //b//h & \longrightarrow & j \\ //c//h & \longrightarrow & k \end{array}$$

(b)

Figure 2: A single-type extended DTD and an equivalent *pattern-based schema* [64]

all bells and whistles concerning, e.g., XML Schema simple types), but aims to facilitate the use of complex types. The main conceptual idea behind BonXai is to specify the Figure 2a schema as the set of rules in Figure 2b. The philosphy behind the schema in Figure 2b is that we have a set of rules $\varphi \longrightarrow e$, where φ selects a set of nodes in a tree T and e is a regular expression. For instance, the left-hand side a of the first rule selects all nodes labeled a, and the XPath expression $//b//h$ selects all h-labeled nodes with a b-labeled ancestor. A tree T satisfies the schema if, for every node v of T we have that (1) v is selected by some left-hand-side φ and (2) for every rule $\varphi \longrightarrow e$ such that v is selected by φ, the children of v match e, as in a DTD. The conceptual advantage of a schema as in Figure 2b is that one does not need the set of types Γ. The entire schema is specified using labels that actually appear in T, which can be conceptually easier to do for users.

4.5 JSON Schema

The interaction between theory and practical studies on JSON Schema is not yet as mature as for DTD or XML Schema. For this reason, we do not provide a formal definition for JSON schema but refer the reader to [24]. In a nutshell, whereas DTD or XML Schema make heavy use of regular expressions, JSON schema follows a logic-based approach [24, Section 5]. JSON Schemas are logical combinations of assertions that describe classes of constraints on objects, arrays, and base values [9].

Maiwald et al. [57] collected 159 JSON schemas from SchemaStore and performed a study in the style of Choi [28], investigating the size of schema, the usage of types, recursion, and maximal nesting depth of nonrecursive schemas. The study found 26 recursive schemas. The non-recursive ones allowed for maximum depths ranging from 3 to 43, wich an average of 11.

They also looked at the usage of JSON's *schema-full* vs *schema-mixed* feature. Here, schema-full means that the schemas only allow documents that have the properties as specified in the schema. Schema-mixed means that the properties specified in the schema must be present in the document (properties can be declared as optional), but additional properties which are not even mentioned by the schema are allowed. JSON Schemas are schema-mixed by default, and the schema-full mode was explicitly used in 8 schemas.

This observation is in stark contrast to what practical studies have seen in DTDs. In DTD, one can argue that the ANY-type is a way to specify arbitrary additional content. In a study with 103 DTDs, ANY was only used in one schema [16].

A recent study by Baazizi et al. collected 11.5k unique schemas from GitHub looked at the usage of negation in JSON Schema [9]. Negation is used in 2.6% of these files and, in many cases, is a work-around for features that seem to be missing in the language (such as a keyword *forbidden* as a dual to *required* or an implication $x \Rightarrow y$ which is encoded as $\neg x \vee y$).

Concerning connections to theory, there is early work on JSON schema containment [42], but the area still seems very young. This contrasts with the work on containment for schemas for XML data, where many principled approaches exist (e.g., [29–31, 63]). JSON schema inference is also starting to attract attention, with systems being developed (e.g., [7, 82]) and theoretical underpinnings such as the design of a first type system that allows the user to configure levels of precision and succinctness [8].

5 QUERIES FOR TREE-STRUCTURED DATA

There are many query languages for tree-structured data, including XPath, XQuery, XSLT, JSONiq, and JSONPath. A recent study by Baelde et al. [11] considered a corpus of 21.1k XPath queries and analyzed these concerning their size, navigational features, and relationship to fragments that were considered in work on XPath satisfiability. Their size analysis shows a power law on the number of nodes in their syntax trees, where a majority of the queries has size at most 13, but still 256 queries with size at least 100.

Navigation in XPath is done using so-called *axes*. These are *child*, *descendant(-or-self)*, *parent*, *ancestor(-or-self)*, *attribute*, *following*, *following-sibling*, *preceding*, *preceding-sibling*, *self*, and *namespace*. In their corpus, such axes were used in 46.5% of the XPath expressions, the most popular being child (31.1%), attribute (17.1%), descendant(-or-self) (3.6%), and ancestor(-or-self) (3.6%).

Baelde et al. study fragments of XPath that were considered in the literature, such as *positive XPath*, *Core XPath 1.0*, and *downward XPath*. Such fragments are indeed prominent in their logs (around 25%–30% of the expressions for positive XPath, Core XPath 1.0, and downward XPath). However, since most of these fragments were intended for theoretical investigation, they further investigated how many queries are *expressible* in each of these fragments, in which case the coverage grew significantly (to e.g., 60% for positive XPath, 70% for Core XPath 1.0, and 35% for downward XPath).

In terms of less expressive fragments, a master's thesis from 2009 by Pasqua [72] gathered 218 XQuery files, and 1882 XSLT documents, from which around 95k XPath-expressions were extracted. In contrast to Baelde et al. [11], who removed duplicate expressions within each source, Pasqua only removed duplicate XQuery- and XSLT-files. Pasqua purely focused on the navigational features of the expressions (the usage of axes) and stripped away everything else. He concluded that over 90% of the expressions in his corpus are *tree patterns* [34, 67] (aka *twig queries*). If we consider the 10% largest XPath expressions, then this fragment drops to 68%, whereas it drops to 58% in the top 5% largest expressions.

6 CONCLUDING TREE-STRUCTURED DATA

Although the structure of tree-structured data sets and queries has not been subject of intense research, there has been a lot of investigation on schemas for tree-structured data. Here, most of the theory-practice interaction has taken place around the analysis of regular expressions in schemas and the usage of complex types, leading to algorithms for containment, intersection non-emptiness, and inference for DTD and XML Schema (Sections 4.2), as well as the development of a possibly user-friendlier alternative for XML Schema (Section 4.4).

Concerning JSON, the theory still seems less developed than for XML. While formal models exist [24], JSON standards are still evolving and the formal models may need to be revisited at some point. We are also seeing recent practical studies going hand-in-hand with fundamental work in areas such as schema containment and inference, indicating interesting opportunities for active research in this area (Section 4.5). Since much of the fundemental research on XML focused on the broader concept of tree-structured data, it may be leveraged to contribute to research on JSON as well.

7 GRAPH-STRUCTURED DATA SETS

Graph databases today are mostly available in Resource Description Framework (RDF) format. An RDF data set G is a set of triples (s, p, o), where we refer to s as *subject*, p as *predicate*, and o as *object*. The RDF specification [75] states that s, p, and o can come from pairwise disjoint sets $\mathcal{I}$ (*IRIs*), $\mathcal{B}$ (*blank nodes*), and $\mathcal{L}$ (*literals*) as follows: $s \in \mathcal{I} \cup \mathcal{B}$, $p \in \mathcal{I}$, and $o \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$. For this paper, the precise definition of IRIs, blank nodes, and literals is not important. Technically, we assume that $\mathcal{I} \subseteq \mathsf{Lab}$, i.e., literals are *labels*.

Abstractly, RDF data is often seen as an edge-labeled directed graph, where each triple (s, p, o) corresponds to an edge from node s to node o with label p. Notice that, per specification, RDF triples can use a predicate p in the subject or object position of another triple, which means that the abstraction as edge-labeled directed graphs would be inaccurate (an edge can leave or go to a label of another edge). A practical study by Fernandez et al. [40, Table 3], however, shows that this is not very common. More precisely, they showed that, in their corpus, the ratios $|P_G \cap S_G|/|P_G \cup S_G|$ and $|P_G \cap O_G|/|P_G \cup O_G|$ (where S_G, P_G, and O_G are the sets of subjects, predicates, and objects in a dataset G) are often zero and, if not, in the order of 10^{-7} to 10^{-3}.

7.1 Practical Studies

The structure of RDF data has been studied mostly in the Semantic Web community [10, 40], with a notable exception in the Database Theory community [59]. We first discuss main findings that focus only on structure before moving to a study that includes labels [40].

7.1.1 Structure. Power law distributions have been discovered in a wide range of metrics for RDF data. Ding and Finin [37] observed a power law for the number of triples per RDF document in their collection of 1.7 million RDF documents. Bachlechner and Strang [10] focused on 1.6 million Friend-of-a-Friend (FOAF3) documents, reaching similar conclusions for the in- and out-degree distributions of nodes (number of triples related to a subject and the number of triples related to an object, respectively). The skewed distribution is illustrated by a maximum degree of 7739, whereas the average is 9.56. Fernandez et al. [40] observed power law distributions for in-degrees and out-degrees of nodes in various data sets (Jamendo, LinkedMDB, Dbtune, Flickr, SWDF, DBLP, 2011 Australian Census, 2000 US Census, Wordnet 3.0, DBpedia, AEMET, Ike, Linked Geo Data, Affymetrix).

Dataset	*#nodes*	*#edges*	*lower tw*	*upper tw*
HongKong	321,210	409,038	32	145
Paris	4,325,486	5,395,531	55	521
Wikipedia	252,335	2,427 434	1,007	19,876
Gnutella	65,586	147,892	244	9,374
Royal	3,007	4,862	11	24

Table 1: Upper and lower bounds for the treewidth (tw) of some real-world datasets considered by Maniu et al. [59]

Maniu et al. [59] studied the *treewidth* of real-world graph data, which is a very challenging metric to compute. Indeed, determining if the treewidth of a given graph is at most k is NP-complete when the graph and k are the input [6]. Since Maniu et al. dealt with graphs with millions of nodes and values of k ranging in the thousands, they can only provide intervals for the treewidth of the data, despite long computation times and serious optimization efforts.

Maniu et al. considered 25 data sets from eight different domains, namely infrastructure networks (road networks, public transportation, power grid), social networks (explicit as in social networking sites, or derived from interaction patterns), web-like networks, a communication network, data with a hierarchical structure (genealogy trees), knowledge bases, traditional OLTP data, as well as a biological interaction network. We present a selection of their results in Table 1. Here, HongKong and Paris are road networks, Wikipedia is a web-like network, Gnutella is a communication network, and Royal is a hierarchical network extracted from a genealogy of royal families in Europe.

They obtain that, *generally*, the treewidth of the data is too large to be able to directly use treewidth-based algorithms with any hope of efficiency. However, partial decompositions may still be helpful, since complex networks often have a dense core together with a tree-like fringe structure [71], which allows the use of tree decomposition methods on the fringe. This is in stark contrast with tree-likeness of *queries* for graph-structured data, which we'll discuss in Section 9.

7.1.2 *Distribution of Labels.*

From a database perspective, it is important to take label information into account when investigating labeled graph-structured data. This is because the role of edge labels in graph databases is often similar to the role of relation names in relational data.

Fernandez et al. [40, Section 3.5] study *predicate lists* per subject s. If we denote by S_G and O_G the set of subjects and objects in an RDF data set G respectively, then these lists are defined as $L_G = \{L_s \mid s \in S_G\}$ where L_s for a subject s is the set $\{p \mid \exists z \in O_G, (s,p,z) \in G\}$. They discover that subjects almost always have the same set of labels in outgoing edges, i.e., in around 99% of the cases. They also show that each pair (s,p) of subject and predicate is mostly related to a unique object o and each pair (p,o) is often related to one s, but there is a high standard deviation of the latter, which means that the distribution is skewed. In their data set, the number of predicates per object is close to 1, which means that objects o very often just have one incoming edge.

8 SCHEMAS FOR GRAPH-STRUCTURED DATA

The languages that play a similar role to database schemas in RDF (i.e., for validation purpuses) are ShEx [81] and SHACL [80]. However, the author is not aware of any systematic study of the real-world usage of these schema languages, similar to those that we saw in Section 4. A first study on the theoretical underpinnings of schema inference for graph-structured data, based on the framework of grammatical inference was done by Groz et al. [47].

9 QUERIES FOR GRAPH-STRUCTURED DATA

Queries for graph-structured data have mostly been analyzed in the form of SPARQL query logs. Such logs became accessible to the research community in the early 2010's through the USEWOD workshop series [86], which provided query logs for BioPortal, OpenBioMed, LinkedGeoData, and DBpedia.

We assume familiarity with SPARQL, but recall the basics of the language for our discussion. Our presentation is strongly based on Picalausa and Vansummeren [74] and Bonifati et al. [21]. Let $\mathcal{V} = \{?x, ?y, ?z, ?x_1, \ldots\}$ be an infinite set of variables, disjoint from $\mathcal{I}$ (IRIs), $\mathcal{B}$ (blank nodes), and $\mathcal{L}$ (literals). As in SPARQL, we always prefix variables by a question mark. A *SPARQL query* Q can be seen as a tuple of the form

$$(\textit{query-type}, \textit{pattern } P, \textit{solution-modifier}).$$

Here, *query-type* is one of Select, Ask, Construct, and Describe, possibly with additional features. (E.g., Select is usually accompanied with a list of variables, for which the query is to return the results.) The *pattern* is the central component of the query, which we will discuss in more detail next, and the *solution-modifier* is for performing aggregation, grouping, sorting, duplicate removal, and returning only a specific window (e.g., the first ten) of the solutions returned by the pattern.

Patterns. A *triple pattern* is an element of $(\mathcal{I} \cup \mathcal{B} \cup \mathcal{V}) \times (\mathcal{I} \cup \mathcal{V}) \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L} \cup \mathcal{V})$. A *property path* is a regular expression over the set $\mathcal{I}$ and is SPARQL's version of a *regular path query* (see, e.g., [4, 14, 32] for basics on regular path queries). A *property path pattern* is an element of $(\mathcal{I} \cup \mathcal{B} \cup \mathcal{V}) \times pp \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L} \cup \mathcal{V})$, where pp is a property path. A *SPARQL pattern* P is an expression generated from the following grammar:

$$\begin{aligned} P ::= \quad & t \mid pp \mid Q \mid P_1 \textsf{ And } P_2 \mid P \textsf{ Filter } R \mid P_1 \textsf{ Union } P_2 \mid \\ & P_1 \textsf{ Optional } P_2 \mid \textsf{Bind } X \textsf{ AS } v \mid \textsf{Service } n\ P \mid \textsf{Values } tup\ T \end{aligned}$$

Here, t is a triple pattern, pp is a property path pattern, Q is again a SPARQL query, and R is a so-called *SPARQL filter constraint*. SPARQL filter constraints R are built-in conditions which can have unary predicates, (in)equalities between variables, and Boolean combinations thereof. Bind associates a unary expression to a single variable v. Service calls a remote service with name n and sends it a pattern P. Finally, Values binds a tuple tup to values in a given table T. Property paths (pp) and subqueries (Q) in the above grammar are new features since SPARQL 1.1. We refer to [73, 87] for further details.

We illustrate by example how parts of our definition correspond to real SPARQL queries. The following query comes from Wikidata's example queries ("Locations of archaeological sites") [89].

```
SELECT ?label ?coord ?subj
WHERE { ?subj wdt:P31/wdt:P279* wd:Q839954 .
        ?subj wdt:P625 ?coord .
        ?subj rdfs:label ?label FILTER(lang(?label)="en") }
```

The query uses the property path `wdt:P31/wdt:P279*`, the literal `wd:Q839954`, and the triple pattern `?subj wdt:P625 ?coord`. It also uses a filter constraint. Notice that SPARQL denotes concatenation in regular expressions by "`/`" and the And operator by a dot.

In Sections 9.1 and 9.2, we give an overview of studies that happened on SPARQL 1.0 and SPARQL 1.1 queries. In Sections 9.3–9.6, we look at specific types of analyses that have been done on SPARQL queries [21, 22], using the largest logs to date. To make the (sometimes relatively dry) material more interesting, we will interleave the presentation with a storyline that aims at shedding light on the following questions that are relevant to database theory:

(1) How prominent are *conjunctive queries*?
(2) What is the structure of conjunctive queries?
(3) How prominent are *conjunctive regular path queries*?
(4) What is the structure of conjunctive regular path queries?
(5) Which kinds of regular path queries are used?

We refer to [4, 14] for basic definitions of conjunctive queries and conjunctive regular path queries.

Several studies [21, 22, 74] investigated both the multiset of syntactically correct queries that are in original query logs (*Valid*) and the set of queries obtained from those by duplicate elimination (*Unique*). For a given analysis, it is often interesting to present results with respect to both of these datasets. Therefore, whenever we report a number or a percentage as X (Y), the number X refers to the *Valid* and the number Y to the *Unique* set of queries.

9.1 Studies on SPARQL 1.0

Early studies on SPARQL queries in 2010–2016 [2, 5, 48, 49, 69] mostly focused on rudimentary properties of queries, such as the number of triple patterns per query and the identification of popular features of the languague.

Arias et al. [5] already observed that small queries are common in logs, that star- and chain-shaped queries are common, and that SPARQL's Union and Optional operators are used in a significant amount of queries. The significant use of Optional is important, because it has a high impact on the complexity of pattern matching.

Pattern matching for SPARQL queries using various subsets of operators was studied by Perez et al. [73], who define the *evaluation* problem as follows.[5]

Evaluation	
Given:	An RDF dataset D, a SPARQL pattern P, and a candidate answer μ.
Question:	Is μ an answer to P over D?

Crucially, Perez et al. prove that Evaluation can be solved in linear time combined complexity for SPARQL patterns using only the And and Filter operators [73, Theorem 3.1],[6] while the problem becomes PSPACE-complete if Optional is additionally allowed [73, Theorem 3.3]. This high increase in complexity led them to the notion of *well-designed* SPARQL patterns, which use OPTIONAL in a restricted fashion such that their evaluation is coNP-complete [73, Theorem 4.6].

[5]Perez et al. also provide a formal definition of when μ is an answer to P over D, which we omit here.
[6]We note that SPARQL patterns do not have projection, which would make Evaluation NP-complete for these queries, as for conjunctive queries.

Source	*Total #Q*	*Valid #Q*	*Unique #Q*
DBpedia9-12	28,651,075	27,622,233	13,437,966
DBpedia13	5,243,853	4,819,837	2,628,000
DBpedia14	37,219,788	33,996,486	17,217,416
DBpedia15	43,478,986	42,709,781	13,253,798
DBpedia16	15,098,176	14,687,870	4,369,755
DBpedia17	169,110,041	164,297,723	34,440,636
LGD13	1,927,695	1,531,164	357,843
LGD14	1,999,961	1,951,973	628,640
BioP13	4,627,270	4,624,449	687,773
BioP14	26,438,932	26,404,716	2,191,151
BioMed13	883,375	882,847	27,030
SWDF13	13,853,604	13,670,550	1,229,759
BritM14	1,555,940	1,545,643	135,112
WikiRobot/OK	207,538,912	207,498,419	34,527,051
WikiOrganic/OK	676,297	665,472	260,723
WikiRobot/TO	33,616	33,465	3,168
WikiOrganic/TO	14,528	14,087	8,729
Total	558,352,049	546,956,715	125,404,550

Table 2: Queries in the logs of [21, 22].

One early study, by Picalausa and Vansummeren [74], performed a fairly deep structural analysis, which was a major inspiration for later work [21, 22]. They gathered around three million queries from DBpedia 2010 logs, containing approximately one million unique queries. Their study discovered (independently from [5]) that the operators Union and Optional occur very often. More precisely, in their corpus, 65.7% (51.3%) of the patterns, a sizable portion only use the operators And and Filter (and are therefore closely related to the conjunctive queries in Database Theory). Union and Optional was discovered in 20.9% (26.9%) and 29.6% (46.4%) of the queries.

They therefore analyzed the structure of queries with And, Filter, Union, and Optional more deeply. Out of all such patterns that use Optional, they discovered that roughly 50% are unions of well-designed patterns. In fact, out of all the patterns that use Optional or Union, again roughly 50% satisfy an even stronger condition, called *well-behavedness*, which makes Evaluation tractable. In their total logs, 83.8% (75.7%) of the patterns are well-behaved.

9.2 Studies Including SPARQL 1.1

The introduction of *property paths* in SPARQL 1.1 brought the language significantly closer to graph database languages as they have been researched since the late 1980s [32]. Property paths are SPARQL 1.1's version of *regular path queries*, which are the crucial feature that allows queries to navigate over long distances in graphs, and which is being developed further in Cypher [70] and GQL [36].

Logs with SPARQL 1.1 queries have become available since the mid 2010s from several sources. LSQ [77] provided logs from DBpedia, Linked Geo Data, Semantic Web Dog Food, and the British Museum, as did later incarnations of USEWOD [86]. Finally, the

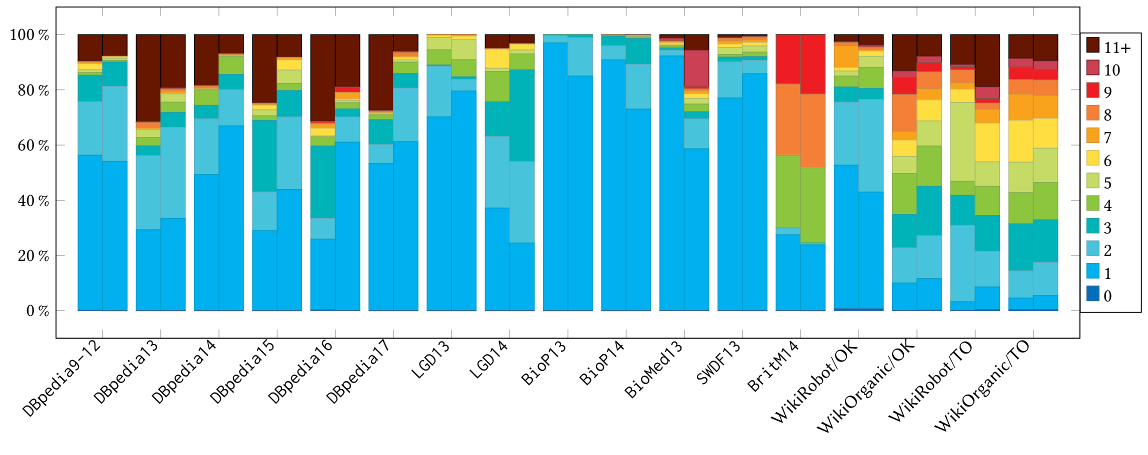


Figure 3: Percentages queries having a certain number of triples (in colors) for the Valid (left hand side of each bar) and Unique (right hand side of each bar) queries of [21, 22].

work of Malyshev et al. [58] was accompanied with a huge release of SPARQL queries for Wikidata.

Our discussion on practical studies in Sections 9.2–9.6 mainly focuses on the query analysis by Bonifati et al. [21, 22], for which the details of the data sets are summarized in Table 2. We have all queries (*Total*), the ones that parse (*Valid*), and the unique queries (*Unique*). The Wikidata query logs (prefixed with Wiki) are studied in [21] and the others in [22]. `DBpedia9-12` contains the DBpedia logs from USEWOD'13, which are from 2009–2012. All other `DBpediaX` sets contain the query logs from the year `'X`. Concerning Wikidata queries, there is a division between robotic (Robot) and organic (Organic) queries, made by Bielefeldt et al. [19]. The queries are also divided in those that timed out (TO) versus those that did not (OK). Only 0.3% (0.8%) of the queries are organic, which means that human activity in query logs is likely to be hidden by bots [19].

Wikidata logs are particularly interesting due to their common usage of property paths. Indeed, around 24.0% (38.9%) of the Wikidata queries use property paths [21], whereas this is only around 0.4% for the others [22]. The main reasons are that data sets like DBpedia have been designed when property paths were still absent from SPARQL and that Wikidata makes extensive use of property paths for ontological reasoning. For this reason, we will sometimes distinguish between the Wikidata logs and the others in the following. We use *DBpedia–BritM* to refer to the non-Wikidata logs.

9.3 Size of Queries

Figure 3 gives an overview of the number of triple patterns in the queries in the Table 2 logs. It illustrates how queries containing 0 to 11+ triples are distributed over each of the data sets. The figure only considers valid Select, Ask and Construct queries. Describe queries (which constitute around 0.1% (0.5%) of queries in the Wikidata logs and 4.9% (3.4%) in the others) are omitted, because their semantics is implementation-dependent and the overwhelming majority does not contain a SPARQL pattern [22].

The figure immediately shows that queries with 0 to 2 triple patterns are extremely common. Indeed, 51.2% (52.6%) of the queries have at most one triple pattern, whereas 66.1% (75.9%) have at most two. However, this skew can drastically differ between data sets (compare `BioP13`/`BioP14` with `BritM14`)[7] or even between logs for the same data set in different years (compare `LGD13` with `LGD14`). In the Wikidata sets, the figure clearly shows that (1) organic queries tend to have more triple patterns than robotic ones and (2) so do timeout queries compared to OK queries.

The largest queries in the entire corpus have around 200–230 triples and can be found in `DBpedia15`–`DBpedia17` and `BioMed13`. There is a fairly significant outlier in `DBpedia17`, with 19.728 (4.669) queries with 105 triple patterns. (In all other cases, given a number k from 100 to 230, the logs have at most 15 queries with k triples.)

While a study on the number of triple patterns in queries is interesting in its own right, it is also important to put other results in the right context. Indeed, in order for a query to have a join, it needs at least two triple patterns. In order to be cyclic, it needs at least three.

9.4 Feature Analysis

Many studies on SPARQL query logs perform a form of feature analysis, which studies the keywords that are used in the queries (e.g., [19, 74]). At least two kinds of such analyses are interesting.

The first kind considers individual features or keywords, and investigates in which percentage of queries it occurs. Table 3 shows the result of such an analysis on the data of Table 2. (The columns with "V" represent results for the *Valid* queries, and the columns with "U" for the *Unique* queries. All relative number are w.r.t. the Select, Ask, and Construct queries.) Already from this simple analysis we can see that there are fundamental differences between the DBpedia–BritM and Wikidata logs. For example, the use of calls to external services (Service) is negligible in DBpedia–BritM (only in ~1k (~500) out of the 339M (91M) queries), but not uncommon in Wikidata, amounting to 8.39% (13.00%) of the queries. The most common usage of Service in Wikidata is due to the use of the service `wikibase:label`, which fetches labels in the specified language. We see a similar striking difference in the usage of property paths.

[7] `BritM14` is a collection of queries with fixed templates.

	DBpedia–BritM				Wikidata			
SPARQL operator	*AbsoluteV*	*RelativeV*	*AbsoluteU*	*RelativeU*	*AbsoluteV*	*RelativeV*	*AbsoluteU*	*RelativeU*
Distinct	96,055,379	29.83%	29,973,843	34.24%	15,945,026	7.67%	3,795,334	10.96%
Limit	46,442,906	14.42%	17,043,642	19.47%	38,406,877	18.47%	5,894,510	17.03%
Offset	8,651,003	2.69%	4,112,837	4.70%	13,886,400	6.68%	3,955,496	11.43%
Order By	3,480,878	1.08%	1,609,784	1.84%	18,265,458	8.78%	701,109	2.03%
Filter	148,681,834	46.17%	34,609,238	39.53%	37,026,908	17.80%	5,175,973	14.95%
And	129,524,401	40.22%	26,737,126	30.54%	74,324,520	35.74%	8,220,701	23.75%
Optional	107,447,774	33.37%	13,119,328	14.99%	31,726,714	15.25%	3,530,184	10.20%
Union	85,024,735	26.40%	15,761,740	18.00%	19,037,334	9.15%	875,177	2.53%
Graph	27,556,055	8.56%	1,523,675	1.74%	20		9	
Values	7,595,570	2.36%	5,086,020	5.81%	66,474,972	31.96%	1,591,804	4.60%
Not Exists	2,527,430	0.78%	1,096,077	1.25%	431,819	0.21%	54,248	0.16%
Minus	2,199,143	0.68%	1,664,350	1.90%	1,795,060	0.86%	601,604	1.74%
Exists	13,964		7,831	0.01%	98,557	0.05%	5,143	0.01%
Group By	9,100,289	2.83%	3,887,124	4.44%	922,759	0.44%	137,456	0.40%
Count	924,467	0.29%	653,749	0.75%	41,822	0.02%	3,917	0.01%
Having	197,455	0.06%	40,393	0.05%	20,972	0.01%	5,053	0.01%
Avg	7,713		730		1,700		135	
Min	7,038		3,747		1,379		452	
Max	6,500		3,792		3,424		789	
Sum	2,767		784		1,663		367	
Service	1,066		466		17,445,279	8.39%	4,499,817	13.00%
property paths (RPQs)	1,419,195	0.44%	335,816	0.38%	49,971,258	24.03%	13,480,433	38.94%

Table 3: Use of individual features in the queries of Table 2. Empty cells denote 0.00%**.**

	DBpedia–BritM			
Operator Set	*AbsoluteV*	*RelativeV*	*AbsoluteU*	*RelativeU*
none	107.285.001	33,32%	31.785.829	36,31%
And	15,106,733	4.69%	7,769,125	8.87%
Filter	30,679,568	9.53%	14,822,989	16.93%
And, Filter	9,583,469	2.98%	4,176,565	4.77%
CQ+F subtotal	162,654,771	50.51%	58,554,508	66.89%

Table 4: Queries that only use the And and Filter operators in the DBpedia–BritM logs.

	Wikidata			
Operator Set	*AbsoluteV*	*RelativeV*	*AbsoluteU*	*RelativeU*
none	81,505,534	15.38%	11,298,828	9.25%
And	12,961,835	2.45%	359,843	0.29%
Filter	4,243,990	0.80%	2,537,531	2.08%
And, Filter	6,477,005	1.22%	72,615	0.06%
CQ+F subtotal	105,188,364	19.85%	14,268,817	11.68%
2RPQ	19,101,741	3.60%	9,259,665	7.58%
And, 2RPQ	58,659,802	11.07%	2,144,013	1.76%
Filter, 2RPQ	47,402	0.01%	1,672	
And, Filter, 2RPQ	765,205	0.14%	138,050	0.11%
C2RPQ+F subtotal	183,762,514	34.67%	25,812,217	21.13%

Table 5: Queries that only use And, Filter, and property paths (2RPQs) in the Wikidata logs.

Whereas property paths are rare in DBpedia–BritM (only in 0.44% (0.38%) of the queries), they occur in 24.03% (38.94%) of the Wikidata queries, which is quite prominent.

A second important feature analysis focuses on identifying fragments of the query logs that only use a given group of features. Table 4 summarizes for how many DBpedia–BritM queries the SPARQL pattern doesn't use any features at all (e.g., a single triple pattern without Filter), uses the And-operator but nothing else, etc. To this end, let us classify a SPARQL query Q as CQ if its pattern P only uses the operator And. Likewise, a query in CQ+F is a query for which its pattern P uses only the operators And and Filter.

This table gives us a rough idea for how many queries in DBpedia–BritM the body closely corresponds to a conjunctive query (so, the query can still use additional aggregation or grouping). More precisely, the CQs ("none"+And) constitute around 38.01% (45.18%) of the queries in the DBpedia–BritM logs.

CQ+F queries, i.e., queries that only use And and Filter, are very prominent in the logs, constituting 50.51% (66.89%) in the DBpedia–BritM corpus (Table 4). Since Filter expressions often are just a simple unary condition on a variable, it can make sense to still consider the body of a query to be "conjunctive" if it only makes use of Filter in a limited way. We come back to this idea in Section 9.5.

We now turn to *conjunctive (two-way) regular path queries* or *C2RPQs* [4, 14] in the Wikidata logs. Table 5 summarizes in a similar way to Table 4 how property paths (2RPQ), And, and Filter are used in the Wikidata logs. Here, CQ and CQ+F queries are much less prominent than in DBpedia–BritM, mostly due to the

DBpedia–BritM: CQ

	AbsoluteV	*RelativeV*	*AbsoluteU*	*RelativeU*
FCA	117,669,751	96.14%	36,786,572	93.00%
htw ≤ 1	118,245,505	96.61%	37,211,970	94.08%
htw ≤ 2	122,391,723	100.00%	39,554,945	100.00%
htw ≤ 3	122,391,734	100.00%	39,554,954	100.00%
Total	122,391,734	100.00%	39,554,954	100.00%

DBpedia–BritM: CQ+F

	AbsoluteV	*RelativeV*	*AbsoluteU*	*RelativeU*
FCA	152,870,307	93.98%	53,393,206	91.19%
htw ≤ 1	157,167,276	96.63%	55,954,287	95.56%
htw ≤ 2	162,654,760	100.00%	58,554,499	100.00%
htw ≤ 3	162,654,771	100.00%	58,554,508	100.00%
Total	162,654,771	100.00%	58,554,508	100.00%

Table 6: Hypertreewidth and free-connex acyclicity for conjunctive queries in the DBpedia–BritM logs.

usage of features such as Service, Values, and property paths. "Pure" C2RPQs, only using And and property paths, constitute around 32.50% (18.88%) of the queries. When we additionally allow Filter, we arrive at 37.67% (21.13%) of the queries, which is a sizable fraction.

Another important fragment of SPARQL queries are the *well-designed* queries [73], already mentioned in Section 9.1. These only use And, Filter, and Optional, but in a restricted way. In total, 62.31% (74.09%) of the queries in DBpedia–BritM and 27.72% (44.24%) in Wikidata only use And, Filter, and Optional, so we again see a big difference between these two data sets. The fraction of well-designed queries, compared to those that only use And, Filter, and Optional is very similar though, which is 98.74% (98.18%) in DBpedia–BritM and 94.18% (98.50%) for Wikidata.

9.5 Shape Analysis

From a database theory perspective, the first question to answer on the structure of conjunctive queries is probably how many of them are acyclic. And if they are not, then what is their hypertreewidth?

In order to discuss this, recall that a *hypergraph* is a pair $H = (V, E)$ where V is its finite set of *nodes* and $E \subseteq 2^V$ its set of *hyperedges*. Consider a SPARQL pattern P of a query q for which the body only uses And and Filter. The *triple hypergraph of* q is the hypergraph $H_q = (V, E)$, where $E = \{\{x, y, z\} \cap (\mathcal{V} \cup \mathcal{B}) \mid (x, y, z)$ is a triple pattern in $q\}$ and $V = \{x \mid \exists e \in E$ such that $x \in e\}$. Notice that we treat SPARQL blank nodes as variables. The *canonical hypergraph* of q is obtained from H_q by adding a hyperedge $\{x_1, \ldots, x_k\}$ for each filter constraint in P that uses precisely the k variables $x_1, \ldots, x_k$ (and adding the respective elements to V).

Table 6 reports on the *hypertree width (htw)* of the canonical hypergraphs of CQ+F queries from DBpedia–BritM, which was computed using det-k-decomp [45]. The top table considers the the patterns that only use And, and the bottom table considers the CQ+F queries. The row "FCA" reports on the amount of queries that are *free-connex acyclic*. (For definitions of hypertree width and free-connex acyclicity, we refer to [13, 44].) From the table, we can conclude that, indeed, in the DBpedia–BritM logs, the hypertree width of conjunctive queries is low and that most queries are acyclic, and even free-connex.

More Precise Shapes. We now dive into more specific shapes by focusing on those CQ+F queries that can structurally be considered as graphs. Indeed, most SPARQL patterns P do not use variables as predicates, that is, they use triple patterns (s, p, o) where p is an IRI. We call a SPARQL pattern P a *graph pattern* if, for every triple pattern (s, p, o) in P, either p is an IRI or p does not appear in another triple pattern in P. (In the latter case, if p is a variable, it can be considered to be a wildcard.) The *triple graph* of graph pattern P is the graph $G = (V, E)$ where $E = \{\{x, y\}) \mid (x, p, y)$ is a triple pattern in P and $p \in \mathcal{I} \cup \mathcal{V}\}$ and $V = \{x \mid \{x, y\} \in E\}$. Notice that the triple graph of a pattern includes nodes that correspond to values in $\mathcal{I}$. Such nodes would correspond to "constant values" in Database Theory terminology. We will look at the graph shapes of queries with and without such constants.

We now turn to the extent in which we can include filter constraints. Let us say that a Filter condition is *safe* if it is either a unary condition on a variable or it is of the form $?x = ?y$. It is *simple* if it is unary or binary. In on the data from Table 4, around 59.5% of the unique queries only use And and safe filters. The bodies of these queries can still be considered to be very close to conjunctive queries. If one would additionally allow simple filters (which goes beyond conjunctive queries, since this allows conditions such as $?x \neq ?y$ or $?x < ?y$), one arrives at about 63.9% of the unique queries.

We call a CQ+F query q *suitable for graph analysis* if it is a graph pattern and all filter constraints are simple, i.e., at most binary. By graph-CQ+F we denote those CQ+F queries that are suitable for graph analysis. If q is in graph-CQ+F, we define its canonical graph as its triple graph, to which we add an edge $\{x, y\}$ for each filter constraint that uses the two variables x and y.

We say that an undirected graph $G = (V, E)$ is a *chain (of length* k*)* if $k = 0$ and G is emtpy or isomorphic to the graph $(v_0, \{\})$; or $k > 0$ and G is isomorphic to the undirected graph with edges $\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{k-1}, v_k\}$, where all v_i are different. A *tree* is an undirected graph such that, for every pair of nodes x and y, there exists exactly one undirected path between x and y. (Hence, every chain is also a tree.) A *forest* is a graph in which every connected component is a tree. A *star* is a tree for which there exists at most one node with more than two neighbors, that is, there is at most one node u such that there exist u_1, u_2, and u_3, all pairwise different and different from u, for which $\{u, u_i\} \in E$ for each $i = 1, 2, 3$.

Table 7 summarizes the structure of the canonical graphs of graph-CQ+F queries, with constants (top) and without (bottom). The canonical graph "without constants" is obtained from the canonical graph by deleting the nodes that correspond to IRIs and their incident edges. This analysis considers more and more general shapes until all queries are classified. Although the vast majority of the graphs are forests, some queries have treewidth two (tw ≤ 2) or three (tw ≤ 3). Perhaps most striking is the huge fragment of queries for which the graph has at most one edge. Beyond this class, simple shapes reign supreme, with star queries already constituting 98% or 99% of the queries. A second striking fact is the number of queries for which the canonical graph without constant nodes doesn't even have an edge anymore.

graph-CQ+F/ with constants

Shape	*AbsoluteV*	*RelativeV*	*AbsoluteU*	*RelativeU*
no edge	73.155	0,05%	1.284	0,00%
$\leq$ 1 edge	137.634.741	87,56%	46.480.555	83,05%
chain	151.963.570	96,68%	54.131.513	96,72%
star	155.324.994	98,82%	55.416.976	99,02%
tree	155.716.239	99,07%	55.487.740	99,15%
forest	155.748.614	99,09%	55.509.368	99,19%
tw $\leq$ 2	157,183,687	100.00%	55,965,063	100.00%
tw $\leq$ 3	157,183,691	100.00%	55,965,066	100.00%
total	157,183,691	100.00%	55,965,066	100.00%

graph-CQ+F/ without constants

Shape	*AbsoluteV*	*RelativeV*	*AbsoluteU*	*RelativeU*
no edge	136.357.782	86,75%	47.047.994	84,07%
$\leq$ 1 edge	150.954.909	96,04%	52.939.341	94,59%
chain	155.832.003	99,14%	55.596.702	99,34%
star	156.787.074	99,75%	55.898.030	99,88%
tree	157.146.828	99,98%	55.944.813	99,96%
forest	157.167.276	99,99%	55.954.287	99,98%
tw $\leq$ 2	157,183,689	100.00%	55,965,065	100.00%
tw $\leq$ 3	157,183,691	100.00%	55,965,066	100.00%
total	157,183,691	100.00%	55,965,066	100.00%

Table 7: Cumulative shape analysis of graph patterns in CQ+F for the DBpedia–BritM logs, for shapes with constants (top) and without (bottom). The relative numbers are w.r.t. the queries that are suitable for graph analysis.

Expression Type	*AbsoluteV*	*RelativeV*	*AbsoluteU*	*RelativeU*
a^*	27,850,487	50.48%	1,392,865	9.87%
ab^*, a^+	9,417,166	17.07%	2,816,134	19.96%
ab^*c^*	823,153	1.49%	67,502	0.48%
A^*	328,895	0.60%	51,860	0.37%
ab^*c	122,286	0.22%	1,680	0.01%
a^*b^*	62,784	0.11%	608	
abc^*	27,287	0.05%	4,083	0.03%
$a?b^*$	15,893	0.03%	11,999	0.09%
A^+	4,674	0.01%	2,043	0.01%
Ab^*	1,562		674	
Other transitive	1,643		161	
$a_1 \cdots a_k$	13,382,005	24.26%	9,368,442	66.41%
A	3,043,725	5.52%	381,434	2.70%
$A?$	31,150	0.06%	296	
$a_1 a_2? \cdots a_k?$	25,872	0.05%	5,940	0.04%
$\hat{}a$	21,202	0.04%	471	
$abc?$	7,620	0.01%	8	
Other non-transitive	697		289	
Total	55,168,101	100%	14,106,489	100%

Table 8: Property path structure of robotic queries of [21].

9.6 Navigation with Regular Path Queries

The syntactic structure of property paths has been investigated in several works [21, 22]. We present the main findings for Wikidata queries [21], since property paths are most prominent there. In this data set, 49,971,258 (13,480,433) queries use property paths, which amounts to a total of 24.03% (38.94%) of the entire logs. The logs contain 165,343 (82,764) property paths in organic queries and 55,168,101 (14,106,489) in robotic ones. (The same query can contain multiple property paths.)

Due to space reasons, we focus on property paths in robotic queries, which contain 64 different *types*. Here, the *type* of a property path is obtained as follows. We replace each variable or IRI by letters from the alphabet in increasing order. (If a variable or IRI is repeated in the property path, we replace it by the same alphabet letter.) For example, `wdt:P31*/wdt:P279*` is of the type a^*b^* and `wdt:P31/wdt:P31*/wdt:P279*` is of the type aa^*b^*.

Table 8 summarizes the most common types of property paths in robotic queries. For succinctness, we aggregated different types together. For example, we aggregated each type with its reverse type. For instance, the row for ab^* also contains the expressions of the form a^*b. Furthermore, $\hat{}a$ ("follow an a-edge in reverse direction") is treated the same as a single label. (The operator $\hat{}$ is used in 0.80% (1.10%) of robotic and 2.03% (3.18%) of organic queries.) Finally, disjunctions are grouped together, denoted by capital letters. In Table 8, a capital letter A denotes a subexpression that matches a *disjunction of at least two symbols*. Empirically, an A either denotes an expression of the form $!a$, $(a|!a)$, or a disjunction of the form $(a_1|\cdots|a_k)$ with $k > 1$ (SPARQL denotes disjunction with '|'). We divided Table 8 into *transitive expressions* (top) and *non-transitive expressions* (bottom). Transitive expressions are those that match arbitrarily long paths (i.e., they use the operators * or $^+$). The empty cells represent values that round down to 0.00%. We note that the relative percentages differ drastically between the *Valid* and the *Unique* queries.

The *structure* of the property paths in these logs are remarkably similar to those in DBpedia-BritM [22]. Martens and Trautner [66] defined the class of *simple transitive expressions*, which are syntactically very restricted, but covered over 99% of the property paths in the DBpedia-BritM queries [22]. In the Wikidata logs, 1.61% (0.48%) of the robotic and 3.83% (2.72%) of the organic property paths are not simple transitive expressions. The most significant reason why property paths fall out of this fragment is the use of a^*b^* as a subexpression, whereas simple transitive expressions only use one subexpression with Kleene star. Furthermore, all property paths except 198 (98) are in $\mathsf{C}_{\mathsf{tract}}$and all except 93 (14) are in $\mathsf{T}_{\mathsf{tract}}$. Here, $\mathsf{C}_{\mathsf{tract}}$is a broader class introduced by Bagan et al. [12] and which precisely characterizes the set of regular path queries for which the data complexity under *simple path semantics* is tractable if P $\neq$ NP. $\mathsf{T}_{\mathsf{tract}}$is the corresponding class for which the data complexity under *trail semantics* is tractable if P $\neq$ NP [65].

9.7 Future Languages: GQL

Beyond query languages for RDF, there is currently a lot of activity around query languages for *property graphs*, leading to the development of the Graph Query Language (GQL). Practical studies on GQL queries are very premature at this point, since the GQL standard is not yet publicly available at the time of writing this paper. The basics of the language are presented in an industry paper in SIGMOD 2022 [36]. Initial ideas about key constraints for the data model have been considered in [3].

10 CONCLUSIONS FOR GRAPH-STRUCTURED DATA

In the world of RDF, Shacl, and SPARQL, we still know relatively little about the structure of real-world data, and it seems that we especially need to know more about the data when taking predicates (or edge labels) into account. Likewise, database schemas for graph-structured data have seen little study to the author's knowledge.

SPARQL queries for graph-structured have received a fair amount of attention, giving us insights about several prominent practical fragments such as CQs and C2RPQs. Due to the wide array of features that SPARQL offers, the amount of queries that can be considered to be "CQ-like" is larger than what we have seen here. In a sense, keywords such as Values can be included Service and have been considered in [21, 22]. For these queries, simple forms (chains, star-shapes) are extremely common (Table 7). This study confirms that these simple types of queries are an interesting class to study, but we cannot conclude that complex queries are not. Public query logs don't necessarily reflect the interests of "power users" and don't have data on the importance of single queries. It could therefore be interesting to perform a study that focuses on the most complex or largest queries in logs.

Furthermore, conjunctive-like queries are only one fragment. A quick look on the feature analysis of SPARQL queries tells us that the already intensely studied Optional, but also Union, for instance, is quite prominent in the logs. Furthermore, features such as Limit and Order By are quite common, depending on the concrete log.

11 LESSONS LEARNED

This section treats some lessons learned after analyzing a fair amount of real-world schemas and queries. In the last years, we built up most of our experience with SPARQL query log analysis, which we do using SHARQL [23]. Our current database holds around 850 million queries (around 300 million more than Table 2).

Don't Expect a Quick and Easy Process. Doing a high-quality practical study takes planning, patience, and learning. As is often the case in research, you will not be on the right path the first time. Especially researchers with a theory background may need to adjust to the dirtiness of real-world data. That said, the process can be very rewarding.

Getting Data. There is no rule of thumb for obtaining data. In the past, we have used Google Web Search and Google Custom Search for finding sources all over the Web. GitHub is a common source of a wide variety of files, from which queries or schemas can sometimes be extracted. Often, the old-fashioned method of talking to our colleagues at conferences has given us the best results.

Organizing Your Data. Whereas small data sets can be dealt with using a set of scripts, this approach does not scale once you deal with hundreds of millions of queries, each of which is subjected to ~120 analytical tests. Our query log data is managed in (surprise, surprise!) a database, which we found offers great flexibility, especially when combined with Jupyter scripts and even spreadsheets.

Keep Your Unaggregated Data Around. Whereas you may write your "practical study" paper with a certain goal in mind, chances are very high that you will need a different perspective later. As a few concrete examples, we have needed a differently aggregated version of the expressions in Table 8 in [65], and even studied a completely new aspect of queries (thresholds) in [20].

The Right Perspective. It is instrumental that your results are put into the right perspective. For example, if 90% of the queries in your corpus only has one atom (or triple pattern), then it is not surprising that 95% of the queries are conjunctive. It may even be misleading to say that 95% of the queries are conjunctive, without mentioning that this is because 90% only have one atom.

Should Every Fragment Now Be Motivated by a Practical Study? Of course not. There should be space in the research landscape for beautiful theory that is not immediately practically motivated. On top of that, data management has such a wide range of research topics that is will not always be possible to discover what real users want using a practical study. That said, you can sometimes really improve your research with a practical study.

12 CONCLUSIONS AND OUTLOOK

This paper presents an overview of practical studies on data sets, schemas, and queries, both for both tree- and graph structured data. We have seen several examples of tight interaction between theory and practice, especially in the area of schemas for tree-structured data, and many examples that illustrate how a theoretical perspective can add value to a practical study. Although we have touched upon many practical studies overall, there is still a wide range of areas that seem interesting for further study. These are (1) structural aspects of data sets when including label information, both for tree-structured and graph-structured data, (2) schemas for JSON data, (3) queries for tree-structured data, (4) schemas for graph-structured data, and (5) queries for graph-structured data, especially concerning Cypher or GQL.

Especially in the realm of queries for graph-structured data, we have seen that different data sets can have wildly different properties. We should therefore always be careful to draw general conclusions from a specific corpus. Furthermore, query logs give us a means to identify some important fragments, but not to identify unimportant fragments or features. For instance, the use of aggregation in the queries in Table 3 is relatively rare, but it is well-known that aggregation is very important in a myriad of data science activities.

Acknowledgments. I am deeply indebted to Frank Neven, who showed me how practical studies can go hand in hand with theoretical research. I thank Aidan Hogan, Stefanie Scherzinger, and Sławek Staworko for valuable pointers to the literature. This paper was written with the aim of being useful to the research community but without a strategy in mind for attracting citations. Since the usefulness of a paper is judged too often by its number of citations, I thank anyone in advance who finds an excuse to cite this paper. Wim Martens is supported by DFG grant 36911683.

REFERENCES

[1] Parosh Aziz Abdulla, Aurore Collomb-Annichini, Ahmed Bouajjani, and Bengt Jonsson. 2004. Using Forward Reachability Analysis for Verification of Lossy Channel Systems. *Formal Methods in System Design* 25, 1 (2004), 39–65.

[2] Saud Aljaloud, Markus Luczak-Roesch, Tim Chown, and Nicholas Gibbins. 2014. Get All, Filter Details — On the Use of Regular Expressions in SPARQL Queries. In *Workshop on Usage Analysis and the Web of Data (USEWOD)*.

[3] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W. Hare, Jan Hidders, Victor E. Lee, Bei Li, Leonid Libkin, Wim Martens, Filip Murlak, Josh Perryman, Ognjen Savkovic, Michael Schmidt, Juan F. Sequeda, Slawek Staworko, and Dominik Tomaszuk. 2021. PG-Keys: Keys for Property Graphs. In *International Conference on Management of Data (SIGMOD)*. ACM, 2423–2436.

[4] Marcelo Arenas, Pablo Barceló, Leonid Libkin, Wim Martens, and Andreas Pieris. 2022. *Principles of Databases*. Book in progress. Available at https://github.com/pdm-book/community/raw/main/pdm-public.pdf.

[5] Mario Arias, Javier D. Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. 2011. An Empirical Study of Real-World SPARQL Queries. *CoRR* abs/1103.5043 (2011). http://arxiv.org/abs/1103.5043

[6] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. 1987. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods* 8, 2 (1987), 277–284.

[7] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Parametric schema inference for massive JSON datasets. *VLDB J.* 28, 4 (2019), 497–521.

[8] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. A Type System for Interactive JSON Schema Inference (Extended Abstract). In *International Colloquium on Automata, Languages, and Programming (ICALP)*. 101:1–101:13.

[9] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2021. An Empirical Study on the "Usage of Not" in Real-World JSON Schema Documents. In *Conceptual Modeling (ER)*. 102–112.

[10] Daniel Bachlechner and Thomas Strang. 2007. Is the Semantic Web a Small World?. In *Intl. Conference on Internet Technologies and Applications (ITA)*.

[11] David Baelde, Anthony Lick, and Sylvain Schmitz. 2019. Decidable XPath Fragments in the Real World. In *ACM Symposium on Principles of Database Systems (PODS)*. ACM, 285–302.

[12] Guillaume Bagan, Angela Bonifati, and Benoît Groz. 2013. A Trichotomy for Regular Simple Path Queries on Graphs. In *ACM Symposium on Principles of Database Systems (PODS)*. ACM, 261–272.

[13] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *International Workshop on Computer Science Logic (CSL)*. 208–222.

[14] Pablo Barceló. 2013. Querying graph databases. In *ACM Symposium on Principles of Database Systems (PODS)*. ACM, 175–188.

[15] Geert Jan Bex, Wouter Gelade, Frank Neven, and Stijn Vansummeren. 2010. Learning Deterministic Regular Expressions for the Inference of Schemas from XML Data. *ACM Trans. Web* 4, 4 (2010), 14:1–14:32.

[16] Geert Jan Bex, Frank Neven, and Jan Van den Bussche. 2004. DTDs versus XML Schema: A Practical Study. In *International Workshop on the Web and Databases (WebDB)*. ACM, 79–84.

[17] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Karl Tuyls. 2006. Inference of Concise DTDs from XML Data. In *International Conference on Very Large Data Bases (VLDB)*. ACM, 115–126.

[18] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Stijn Vansummeren. 2010. Inference of concise regular expressions and DTDs. *ACM Trans. Database Syst.* 35, 2 (2010), 11:1–11:47.

[19] Adrian Bielefeldt, Julius Gonsior, and Markus Krötzsch. 2018. Practical Linked Data Access via SPARQL: The Case of Wikidata. In *Workshop on Linked Data (LDOW)*.

[20] Angela Bonifati, Stefania Dumbrava, George Fletcher, Jan Hidders, Matthias Hofer, Wim Martens, Filip Murlak, Joshua Shinavier, Slawek Staworko, and Dominik Tomaszuk. 2022. Threshold Queries in Theory and in the Wild. *Proceedings of the VLDB Endowment (PVLDB)* 15 (2022).

[21] Angela Bonifati, Wim Martens, and Thomas Tim. 2019. Navigating the Maze of Wikidata Query Logs. In *The World Wide Web Conference (WWW)*. ACM, 127–138.

[22] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *VLDB J.* 29, 2-3 (2020), 655–679.

[23] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. SHARQL: Shape Analysis of Recursive SPARQL Queries. In *International Conference on Management of Data (SIGMOD)*. ACM, 2701–2704.

[24] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoc. 2017. JSON: Data model, Query languages and Schema specification. In *ACM Symposium on Principles of Database Systems (PODS)*. ACM, 123–135.

[25] A. Brüggemann-Klein and D. Wood. 1998. One-Unambiguous Regular Languages. *Information and Computation* 142, 2 (1998), 182–206.

[26] Peter Buneman, Martin Grohe, and Christoph Koch. 2003. Path Queries on Compressed XML. In *International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann, 141–152.

[27] Ashok K. Chandra and Philip M. Merlin. 1977. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *ACM Symposium on Theory of Computing (STOC)*. ACM, 77–90.

[28] Byron Choi. 2002. What are real DTDs like?. In *International Workshop on the Web and Databases (WebDB)*. 43–48.

[29] Dario Colazzo, Giorgio Ghelli, Luca Pardini, and Carlo Sartiani. 2013. Almost-linear inclusion for XML regular expression types. *ACM Trans. Database Syst.* 38, 3 (2013), 15:1–15:45.

[30] Dario Colazzo, Giorgio Ghelli, Luca Pardini, and Carlo Sartiani. 2013. Efficient asymmetric inclusion of regular expressions with interleaving and counting for XML type-checking. *Theor. Comput. Sci.* 492 (2013), 88–116.

[31] Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2009. Efficient inclusion for a class of XML types with interleaving and counting. *Inf. Syst.* 34, 7 (2009), 643–656.

[32] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. 1987. A Graphical Query Language Supporting Recursion. In *International Conference on Management of Data (SIGMOD)*. 323–330.

[33] Wojciech Czerwinski, Claire David, Katja Losemann, and Wim Martens. 2013. Deciding Definability by Deterministic Regular Expressions. In *Foundations of Software Science and Computation Structures (FOSSACS)*. Springer, 289–304.

[34] Wojciech Czerwinski, Wim Martens, Matthias Niewerth, and Pawel Parys. 2018. Minimization of Tree Patterns. *J. ACM* 65, 4 (2018), 26:1–26:46.

[35] DBLP 2022. The DBLP data set. https://dblp.uni-trier.de/xml/.

[36] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Hannes Voigt, Oskar van Rest, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *International Conference on Management of Data (SIGMOD)*.

[37] Li Ding and Tim Finin. 2006. Characterizing the Semantic Web on the Web. In *International Semantic Web Conference (ISWC)*. 242–257.

[38] Dominik Durner, Viktor Leis, and Thomas Neumann. 2021. JSON Tiles: Fast Analytics on Semi-Structured Data. In *International Conference on Management of Data (SIGMOD)*. ACM, 445–458.

[39] Extensible Markup Language (XML) 1.1 (Second Edition) 2006. https://www.w3.org/TR/xml11/.

[40] Javier D. Fernández, Miguel A. Martínez-Prieto, Pablo de la Fuente Redondo, and Claudio Gutiérrez. 2018. Characterising RDF data sets. *J. Inf. Sci.* 44, 2 (2018), 203–229.

[41] Markus Frick, Martin Grohe, and Christoph Koch. 2003. Query Evaluation on Compressed Trees (Extended Abstract). In *Symposium on Logic in Computer Science (LICS)*. 188–197.

[42] Michael Fruth, Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2020. Challenges in Checking JSON Schema Containment over Evolving Real-World Schemas. In *Advances in Conceptual Modeling (ER) Workshops*. 220–230.

[43] E. Mark Gold. 1967. Language Identification in the Limit. *Inf. Control.* 10, 5 (1967), 447–474.

[44] Georg Gottlob, Gianluigi Greco, and Francesco Scarcello. 2014. Treewidth and Hypertree Width. In *Tractability: Practical Approaches to Hard Problems*. Cambridge University Press, 3–38.

[45] Georg Gottlob and Marko Samer. 2008. A backtracking-based algorithm for hypertree decomposition. *ACM J. Exp. Algorithmics* 13 (2008).

[46] Steven Grijzenhout and Maarten Marx. 2013. The quality of the XML Web. *J. Web Semant.* 19 (2013), 59–68.

[47] Benoît Groz, Aurélien Lemay, Slawek Staworko, and Piotr Wieczorek. 2022. Inference of Shape Expression Schemas Typed RDF Graphs. In *International Conference on Database Theory (ICDT)*. To appear.

[48] Xingwang Han, Zhiyong Feng, Xiaowang Zhang, Xin Wang, Guozheng Rao, and Shuo Jiang. 2016. On the statistical analysis of practical SPARQL queries. In *International Workshop on Web and Databases (WebDB)*. ACM, 2.

[49] Jochen Huelss and Heiko Paulheim. 2015. What SPARQL Query Logs Tell and Do Not Tell about Semantic Relatedness in LOD Or: The Unsuccessful Attempt to Improve the Browsing Experience of DBpedia by Exploiting Query Logs. In *Workshop on Negative or Inconclusive Results in Semantic Web (NoISE)*.

[50] Yasunori Ishihara, Nobutaka Suzuki, Kenji Hashimoto, Shougo Shimizu, and Toru Fujiwara. 2013. XPath Satisfiability with Parent Axes or Qualifiers Is Tractable under Many of Real-World DTDs. In *International Symposium on Database Programming Languages (DBPL)*.

[51] Dexter Kozen. 1977. Lower bounds for natural proof systems. In *Symposium on Foundations of Computer Science (FOCS)*. 254–266.

[52] Alberto H. F. Laender, Mirella M. Moro, Cristiano Nascimento, and Patrícia Martins. 2009. An X-ray on web-available XML schemas. *SIGMOD Rec.* 38, 1 (2009), 37–42.

[53] Yeting Li, Xinyu Chu, Xiaoying Mou, Chunmei Dong, and Haiming Chen. 2018. Practical Study of Deterministic Regular Expressions from Large-scale XML and Schema Data. In *International Database Engineering & Applications Symposium (IDEAS)*. ACM, 45–53.

[54] Yeting Li, Xiaolan Zhang, Feifei Peng, and Haiming Chen. 2016. Practical Study of Subclasses of Regular Expressions in DTD and XML Schema. In *Asia-Pacific Web Conference (APWeb)*. 368–382.

[55] Katja Losemann, Wim Martens, and Matthias Niewerth. 2016. Closure properties and descriptional complexity of deterministic regular expressions. *Theor. Comput. Sci.* 627 (2016), 54–70.

[56] Ping Lu, Joachim Bremer, and Haiming Chen. 2015. Deciding Determinism of Regular Languages. *Theory Comput. Syst.* 57, 1 (2015), 97–139.

[57] Benjamin Maiwald, Benjamin Riedle, and Stefanie Scherzinger. 2019. What Are Real JSON Schemas Like? - An Empirical Analysis of Structural Properties. In *Advances in Conceptual Modeling (ER) Workshops*. 95–105.

[58] Stanislav Malyshev, Markus Krötzsch, Larry González, Julius Gonsior, and Adrian Bielefeldt. 2018. Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph. In *International Semantic Web Conference (ISWC)*. 376–394.

[59] Silviu Maniu, Pierre Senellart, and Suraj Jog. 2019. An Experimental Study of the Treewidth of Real-World Graph Data. In *International Conference on Database Theory (ICDT)*, Pablo Barceló and Marco Calautti (Eds.). 12:1–12:18.

[60] Mitchell P. Marcus, Beatrice Santorini, Mary Ann Marcinkiewicz, and Ann Taylor. [n.d.]. Treebank-3. https://doi.org/10.35111/gq1x-j780, year = 1999.

[61] Wim Martens, Frank Neven, Matthias Niewerth, and Thomas Schwentick. 2017. BonXai: Combining the Simplicity of DTD with the Expressiveness of XML Schema. *ACM Trans. Database Syst.* 42, 3 (2017), 15:1–15:42.

[62] Wim Martens, Frank Neven, and Thomas Schwentick. 2004. Complexity of Decision Problems for Simple Regular Expressions. In *International Symposium on Mathematical Foundations of Computer Science (MFCS)*. 889–900.

[63] Wim Martens, Frank Neven, and Thomas Schwentick. 2009. Complexity of Decision Problems for XML Schemas and Chain Regular Expressions. *SIAM J. Comput.* 39, 4 (2009), 1486–1530.

[64] Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. 2006. Expressiveness and complexity of XML Schema. *ACM Trans. Database Syst.* 31, 3 (2006), 770–813.

[65] Wim Martens, Matthias Niewerth, and Tina Trautner. 2020. A Trichotomy for Regular Trail Queries. In *International Symposium on Theoretical Aspects of Computer Science (STACS)*. 7:1–7:16.

[66] Wim Martens and Tina Trautner. 2019. Dichotomies for Evaluating Simple Regular Path Queries. *ACM Trans. Database Syst.* 44, 4 (2019), 16:1–16:46.

[67] Gerome Miklau and Dan Suciu. 2004. Containment and equivalence for a fragment of XPath. *J. ACM* 51, 1 (2004), 2–45.

[68] Manizheh Montazerian, Peter T. Wood, and Seyed R. Mousavi. 2007. XPath Query Satisfiability is in PTIME for Real-World DTDs. In *International XML Database Symposium (XSym)*. 17–30.

[69] Knud Möller, Michael Hausenblas, Richard Cyganiak, Siegfried Handschuh, and Gunnar Grimnes. 2010. Learning from Linked Open Data Usage: Patterns & Metrics. In *Web Science Conference (WebSci)*.

[70] Neo4j. 2022. The Neo4j Cypher Manual v4.4. https://neo4j.com/docs/cypher-manual/4.4/.

[71] M. E. J. Newman, S. H. Strogatz, and D. J. Watts. 2001. Random graphs with arbitrary degree distributions and their applications. *Phys. Rev. E* 64 (Jul 2001), 026118. Issue 2.

[72] Daniel Pasqua. 2009. *Theoretische und Praktische Untersuchungen zur Komplexität von XPath.* Master's thesis. Technische Universität Dortmund.

[73] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3 (2009), 16:1–16:45.

[74] François Picalausa and Stijn Vansummeren. 2011. What are real SPARQL queries like?. In *International Workshop on Semantic Web Information Management (SWIM)*. ACM, 7.

[75] RDF 2014. RDF 1.1 Primer. https://www.w3.org/TR/rdf11-primer/. World Wide Web Consortium.

[76] Arnaud Sahuguet. 2000. Everything You Ever Wanted to Know About DTDs, But Were Afraid to Ask. In *International Workshop on the Web and Databases (WebDB)*. 69–74.

[77] Muhammad Saleem, Muhammad Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2015. LSQ: The Linked SPARQL Queries Dataset. In *International Semantic Web Conference (ISWC)*. 261–269.

[78] Luc Segoufin and Cristina Sirangelo. 2007. Constant-Memory Validation of Streaming XML Documents Against DTDs. In *International Conference on Database Theory (ICDT)*. Springer, 299–313.

[79] Luc Segoufin and Victor Vianu. 2002. Validating Streaming XML Documents. In *ACM Symposium on Principles of Database Systems (PODS)*. ACM, 53–64.

[80] Shapes Constraint Language (SHACL). 2017. https://www.w3.org/TR/shacl/.

[81] SHEX—Shape Expressions. 2022. http://shex.io/.

[82] William Spoth, Oliver Kennedy, Ying Lu, Beda Christoph Hammerschmidt, and Zhen Hua Liu. 2021. Reducing Ambiguity in Json Schema Discovery. In *International Conference on Management of Data (SIGMOD)*. ACM, 1732–1744.

[83] L. J. Stockmeyer and A. R. Meyer. 1973. Word Problems Requiring Exponential Time: Preliminary Report. In *ACM Symposium on Theory of Computing (STOC)*. ACM, USA, 1–9.

[84] Swissprot data set 2022. https://www.uniprot.org/downloads.

[85] John C. Thomas and John D. Gould. 1975. A psychological study of query by example. In *National Computer Conference*. 439–445.

[86] USEWOD. 2011–2016. Workshop on Usage Analysis and the Web of Data. http://usewod.org/.

[87] W3C Sparql 2013. SPARQL 1.1 Query Language. https://www.w3.org/TR/sparql11-query/. World Wide Web Consortium.

[88] W3C XML Schema [n.d.]. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. https://www.w3.org/TR/xmlschema11-1/.

[89] Wikidata 2022. wikidata.org.

[90] Moshé M. Zloof. 1975. Query by Example. In *National Computer Conference*. 431–438.

A A PROOF SKETCH

We briefly sketch why the containment problem for regular expressions in $\mathrm{RE}(a, a^*)$ and $\mathrm{RE}(a, a?)$ is coNP-hard. A full proof can be found in [63].

The proof is by reduction from the validity problem of propositional formulas. In our proof sketch, we will just show how validity of an example formula is translated to the containment problem. Consider the formula $\varphi = (x_1 \land \neg x_2 \land x_3) \lor (\neg x_1 \land x_3 \land \neg x_4) \lor (x_2 \land \neg x_3 \land x_4)$ with $n = 4$ variables and $m = 3$ clauses.

For $\mathrm{RE}(a, a?)$, validity for φ is translated to containment of the expression e_1, defined as

$$\#a\$a\$a\$a\#a\$a\$a\$a\#\; a?a?\$a?a?\$a?a?\$a?a?\; \#a\$a\$a\$a\#a\$a\$a\$a\#$$

in the expression e_2, defined as

$$\begin{gathered}\#?a?\$?a?\$?a?\$?a?\#?a?\$?a?\$?a?\$?a?\#?\\ aa?\,\$\,a?\$\,aa?\,\$\,a?a?\,\#\,a?\,\$\,a?a?\,\$\,aa?\,\$\,a?\,\#\,a?a?\,\$\,aa?\,\$\,a?\,\$\,aa?\,\#\\ \#?a?\$?a?\$?a?\$?a?\#?a?\$?a?\$?a?\$?a?\#?\end{gathered}$$

Here, the word aa should be interpreted as "true" and ε as "false".

Expression e_1 first has $m-1$ "buffer" blocks containing $a\$a\$a\$a$, followed by one block that is intended to generate all truth assignments to the four variables, and concludes with the same $m-1$ buffer blocks. The expression e_2 also begins with $m-1$ blocks (containing $a?\$?a?\$?a?\$?a?$), then m blocks that encode φ, and concludes with the same $m-1$ blocks that it started with.

Notice that the start- and end-blocks of e_2 also match ε. This is crucial for the following reason. Every word that matches e_1 consists of $2m-1$ blocks. Since the start- and end-blocks of e_2 match ε, and since e_2 only has $m-1$ start blocks and end blocks, the middle block of each such word must be matched by one of the three middle blocks of e_2 that encode the clauses in φ. This structure of e_1 and e_2 ensure that each truth assignment "generated by" e_1 needs to match on one of the clauses encoded by e_2. The reader can now check that, using aa to encode "true" and ε to encode "false", we indeed have that φ is valid if and only if $L(e_1) \subseteq L(e_2)$.

For $\mathrm{RE}(a, a^*)$, validity for φ is translated to containment of the expression e_1, defined as

$$\begin{gathered}\#a\$a\$a\$a\#a\$a\$a\$a\#\\ a^*b^*a^*\$a^*b^*a^*\$a^*b^*a^*\$a^*b^*a^*\\ \#a\$a\$a\$a\#a\$a\$a\$a\#\end{gathered}$$

in the expression e_2, defined as

$$\begin{gathered}\#^*a^*\$^*a^*\$^*a^*\$^*a^*\#^*a^*\$^*a^*\$^*a^*\$^*a^*\#^*\\ aa^*b^*a^*\,\$\,b^*a^*\,\$\,aa^*b^*a^*\,\$\,a^*b^*a^*\#\\ b^*a^*\,\$\,a^*b^*a^*\,\$\,aa^*b^*a^*\,\$\,b^*a^*\#\\ a^*b^*a^*\,\$\,a^*b^*a^*\,\$\,b^*a^*\,\$\,aa^*b^*a^*\\ \#^*a^*\$^*a^*\$^*a^*\$^*a^*\#^*a^*\$^*a^*\$^*a^*\$^*a^*\#^*\end{gathered}$$

The general principle is exactly the same as for $\mathrm{RE}(a, a?)$. Here, the word ab encodes "true" and ba encodes "false".