Bridging Theory and Practice with Query Log Analysis

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

Tina Trautner
University of Bayreuth
tina.trautner@uni-bayreuth.de

ABSTRACT

Since large structured query logs have recently become available, we have a new opportunity to gain insights in the types of queries that users ask. Even though such logs can be quite volatile, there are various new observations that can be made about the structure of queries inside them, on which we report here. Furthermore, building on an extensive analysis that has been done on such logs, we were able to provide a theoretical explanation why *regular path queries* in graph database applications behave better than worst-case complexity results suggest at first sight.

1. INTRODUCTION

The recent availability of large logs of structured queries provides new research opportunities for the database community. With millions of queries available for analysis, we suddenly have a large amount of information that can help us to identify interesting characteristics of real-world database queries. Such characteristics can then guide our focus when we want to study certain aspects of query evaluation or optimization, or if we simply want to understand the types of questions that users find interesting.

Database research has traditionally always had a strong focus on searching for subclasses of query languages that exhibit favorable computational properties. Well-known examples are the focus on *conjunctive queries* instead of full-fledged query languages, Datalog as a subset of Prolog, myriads of fragments of XPath or XQuery in the times of XML research, or even set semantics of queries (i.e., *select distinct* queries), as opposed to bag semantics.

Now that query logs are becoming available, we are obtaining hard data against which we can test or justify the importance of some of the specific problems we have been studying, and in which we may be able to discover new interesting cases. A nice side-effect for researchers is that, once we find a specific property of queries to be very prominent in logs, we immediately have numbers that we can use to motivate research on this specific property.

However, there is also a snag: the query logs we currently have exhibit very different characteristics depending on the data source that is queried [5]. Due to the mixture of robotic and user-generated queries, these characteristics can even vary significantly on a day-to-day basis for the same query log [4]. This has some very important consequences that we need to keep in mind. First of all, we need to be careful that we don't focus too much on an aspect that is too specific, i.e., we would overfit our research. Needless to say, this is a delicate balance that is challenging to maintain. Second, studies on query log analysis should not be used to argue that some property of queries is not interesting. This is for the simple reason that, even though we may have "large" amounts of queries available, there is an even larger amount of queries that we do not have available (or will become important in the future) and we know nothing about. The old-fashioned elegance of a problem therefore remains extremely important for guiding research.

In this paper we report on some lessons learned from analysing over half a billion queries, coming from DBpedia, Semantic Web Dog Food, LinkedGeoData, BioPortal, OpenBioMed, British Museum, and WikiData. We also illustrate how some of the knowledge gained from this analysis could be used in theoretical research to give an explanation why certain types of queries seem to behave mostly unproblematically in practice even though their worst-case complexity is quite high.

2. ANALYSIS OF SPARQL LOGS

To the best of our knowledge, the first study on huge logs of structured queries was done by Bonifati et al. [5]. The study had a total of about 180M SPARQL queries, summarized in Table 1. The table mentions, for each of the logs, its total number of queries (*Total*) and the number of queries that could be parsed using Apache Jena 3.0.1 (*Valid*). From the latter set, duplicates were removed, resulting in the unique queries that could be parsed (*Unique*). The queries come from DBpedia, LinkedGeoData (LGD), BioPortal (BioP), OpenBioMed (BioMed), Semantic Web Dog Food (SWDF), British Museum (BritM), and WikiData. The `WikiData17` set is very small: it consists of the user-submitted example queries from Wikidata in February 2017. This first study has since been extended with 170M DBpedia queries [6] and 208M Wikidata queries [7], adding up to more than 550 million queries. In this paper we will

This paper is primarily based on the paper "Evaluation and Enumeration Problems for Regular Path Queries" (published in ICDT 2018), but also on "An Analytical Study of Large SPARQL Query Logs" (published in VLDB 2018)

`https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples`

Table 1: Query logs in the corpus of [5].

Source	*Total #Q*	*Valid #Q*	*Unique #Q*
DBpedia9/12	28,534,301	27,097,467	13,437,966
DBpedia13	5,243,853	4,819,837	2,628,005
DBpedia14	37,219,788	33,996,480	17,217,448
DBpedia15	43,478,986	42,709,778	13,253,845
DBpedia16	15,098,176	14,687,869	4,369,781
LGD13	1,841,880	1,513,868	357,842
LGD14	1,999,961	1,929,130	628,640
BioP13	4,627,271	4,624,430	687,773
BioP14	26,438,933	26,404,710	2,191,152
BioMed13	883,374	882,809	27,030
SWDF13	13,762,797	13,618,017	1,229,759
BritM14	1,523,827	1,513,534	135,112
WikiData17	309	308	308
Total	180,653,910	173,798,237	56,164,661

primarily focus on the query logs in Table 1, but we will report insights from the other studies [7, 6] whenever relevant.

Size of Queries.

A first important discovery that was made in the logs is about the size of queries. In [5], this was measured by counting the number of *subject-predicate-object* triples in the queries, which are the SPARQL counterpart of atoms (or relational predicates) in relational databases. The distribution in these logs is extremely skewed: if we look in the *Unique* queries, over 56% have only a single triple. Even though there are queries with up to 229 triples, it is the case that up to six triples are enough to capture over 90% of the queries and, with up to twelve triples, we capture over 99%. In Wikidata logs that were recently investigated [7], the distribution is less skewed, at least for the *non-robotic* queries. Here, only 13% of the unique queries have a single triple. Moreover, one needs up to 9 triples to capture over 90% and up to 16 triples to capture over 99% of the queries.

Cyclicity.

These observations on the size of queries are important if we want to understand cyclicity. *Cyclicity* and *acyclicity* of queries is indeed a very important aspect of queries that has received a huge amount of attention in the literature (e.g., [11] and the references therein). An important reason is that queries in practice are assumed to be only *mildly cyclic*, which would be good news, since cyclic queries are more complex to evaluate. Since our standard definitions of cyclicity require a query to have at least three atoms to be cyclic [1], and since 78% of the unique queries in Table 1 only have up to two triples, we already know that the majority must be acyclic. But the assumption that real world queries are only mildly cyclic is also strongly confirmed when we look deeper. Out of all the conjunctive queries, even 99.9% are acyclic. Again, these numbers slightly shift when we look into the non-robotic Wikidata queries mentioned before, where around 97.8% of the unique queries investigated in [7] are acyclic. In terms of *treewidth*, we found queries in the logs with treewidth up to five (if one also allows property paths). An interesting real-world query with treewidth five can be found in Figure 1. (The subquery consisting of the variables is a six-clique, which has treewidth five.)

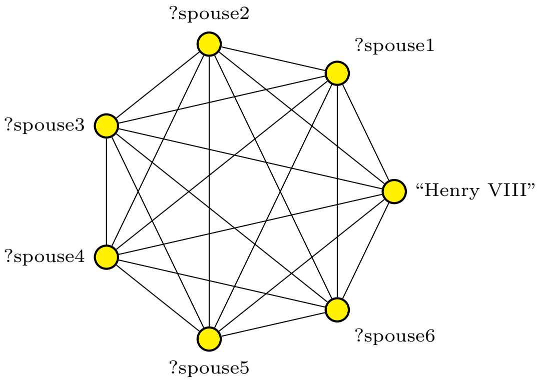


Figure 1: Graphical representation of a highly cyclic query: a 7-clique containing one constant and six variables. All edges connecting to Henry VIII are labeled "married-to" and all edges between the variables ?spouse1, . . . , ?spouse6 are labeled "! =". The query therefore searches for six different spouses of Henry VIIIth. The query is inspired on a query we found in the logs of Table 1.

Query Shapes.

Since so many queries in the logs are acyclic, it also makes sense to look more closely at their structure. More precisely, one can consider the graph structure induced by the subject-predicate-object triples in the queries by considering each triple (x, y, z) and turning it into two nodes x and z, connected by an undirected edge. (The construction of the graph is actually more subtle – sometimes *hypergraphs* are required. We refer to [5] for more details.)

For those queries that can be adequately represented as a graph, the *undirected* version of this graph was considered and it was investigated which fractions of the queries are a *single edge*, a *chain* (a connected, acyclic sequence of edges), a *star* (is a "central" node, to which chains can be attached), or a *tree*. These shapes are intended to be *cumulative*, so each shape generalizes the previous one. Considering the unique conjunctive queries, it turns out that around 78.98% are a single edge, 98.87% are a chain, 99.81% are a star, and 99.90% are trees. A visual inspection of the remaining queries showed that many of them can be seen as *flowers*, which are a central node, to which trees or *petals* can be attached. Here, a petal consists of two nodes u and v that are connected by chains. This generalization allowed to capture 99.94% of the queries. Most of the remaining queries consisted of multiple connected components. Generalizing from *flower* queries to *bouquet* queries allowed to capture essentially 100%. Here, a bouquet is a graph in which each connected component is a flower.

Differences Between Logs.

Even though some trends can be identified in the logs, there are also some drastic differences. This is a healthy warning for us: we should not declare that we now understand what users are interested in. For instance, in the

`BioP13` and `BioP14` logs, 79.66% and 40.48% of the unique queries use the GRAPH-operator, whereas this operator only occurs in 2.71% of the total queries. Bielefeldt et al. [4] observed huge differences in query volumes in Wikidata logs over different days, mainly due to automatically generated queries that, consequently, can have huge effect on the types of queries in the logs. Finally, in the data of Table 1, less than 1% of the queries use property paths, whereas this grows to 38% of the unique queries in [7].

3. CASE IN POINT: PATH QUERIES

Regular path queries (RPQs) are a crucial feature of graph database query languages, since they allow us to answer queries that involve arbitrarily long paths in graphs using regular expressions. We give an example. Consider the toy graph database in Figure 2, which is loosely inspired on a part of the Wikidata graph. Suppose that we want to find *artists who died at age 27*, we can easily do so using a regular path query. (These artists are known under the name "27 club". The club has famous members such as Kurt Cobain, Jimi Hendrix, Janis Joplin, Jim Morrison, and Amy Winehouse.) For instance, we can retrieve the persons who died at age 27 with a Cypher-like subquery of the form

```
CONSTRUCT (x)
  MATCH   (x:Person)-[:age-at-death]->(y:Integer)
    WHERE  y = 27
```

Likewise, artists can be found by the query

```
CONSTRUCT (x)
  MATCH   (x:Person)-[:occupation]->()
                    -[:subclassof*]->(y:Profession)
    WHERE  y.name = 'artist'
```

The second query asks for persons whose occupation is a profession that is connected with a `subclassof`-path to "artist". Here, we used the regular expression $\texttt{subclassof}^*$ to allow arbitrarily long paths in which every edge is labeled with `subclassof`. Since we may not know in advance how many `subclassof`-edges we have to consider, it is very comfortable to be able to use the regular path query $\texttt{subclassof}^*$. The example also illustrates the robustness of regular path queries. Even when the graph database changes (e.g., by introducing an additional profession such as "string instrumentalist"), the query still returns the correct results.

Regular path queries or RPQs started as an academic idea in Cruz et al.'s seminal paper [8] and are nowadays part of SPARQL, Cypher and Oracle's PGQL. Although the main idea behind RPQs is always to match regular expressions against paths in a graph database, academic research and real-world systems do not always agree on how this should be done. The main difference lies in which paths should be considered for matching, and the most considered candidates are *all paths* or *paths without repeated nodes or edges*. Whereas academic research most commonly allows all paths (which allow polynomial time algorithms to test if a matching path exists between two given nodes), graph database systems usually revert to paths without repeated nodes or edges. There seem to be different reasons why this is so. First of all, this restriction always ensures that the number of paths that can match is finite, so one does not have to deal with infinity. Second, paths without repeated nodes or edges gives the semantics that some users seem to prefer [Lindaaker, personal communication]. From a theoretical point of view, however, such paths very quickly lead to intractability. Even testing if a matching path exists between nodes is NP-complete, see Theorem 1.

3.1 Complexity of Simple Paths and Trails

We briefly want to explain some of the fundamental results about RPQ evaluation against paths without repeated nodes or edges. We use edge-labeled graphs as abstractions for graph databases. To this end, let Σ be a set of *labels*. A graph database (with labels in Σ) is a pair $G = (V, E)$, where V is the finite set of *nodes* of G and $E \subseteq V \times \Sigma \times V$ is the set of *edges*. We say that edge $e = (u, a, v)$ is *from node u to node v* and *has label a*. Notice that this definition allows graphs to have self-loops and multiple edges from u to v if they have different labels. The *size* of a graph G, denoted by $|G|$, is defined as $|G| = |V| + |E|$.

A *path* from node u to node v in G is a sequence

$$p = (v_0, a_1, v_1)(v_1, a_2, v_2) \cdots (v_{n-1}, a_n, v_n)$$

of edges in G such that $u = v_0$ and $v = v_n$. By $\text{lab}(p)$ we denote the sequence $a_1 \cdots a_n$ of labels on the edges of p. The length of a path p is its number of edges. A path p is *simple* if it has no repeated nodes, that is, all nodes $v_0, \ldots, v_n$ are pairwise different. It is a *trail* if it has no repeated edges, that is, every edge appears only once in p.

Regular path queries are abstracted as regular expressions. Here, ε, and every Σ-symbol is a regular expression; and if r and s are regular expressions, then so are $(r \cdot s)$, $(r + s)$, and (r^*). (To improve readability, we use associativity and the standard priority rules to omit braces in regular expressions. We usually also omit the outermost braces.) We use $r?$ to abbreviate $r + \varepsilon$. The *size* $|r|$ of a regular expression is the number of occurrences of Σ-symbols in r. For example, $|((a \cdot b) \cdot a)^*| = 3$. We define the *language* $L(r)$ of r as usual. A path p *matches* r if $\text{lab}(p) \in L(r)$, that is, the sequence of labels on the edges of p is in the language of r. The following two decision problems are central to evaluation of regular path queries over simple paths and trails.

$\mathsf{SimPath}(\mathcal{R})$

Given: A graph $G = (V, E)$, two nodes $x, y \in V$, and an RPQ $r \in \mathcal{R}$.

Question: Is there a simple path from x to y in G that matches r?

$\mathsf{Trail}(\mathcal{R})$

Given: A graph $G = (V, E)$, two nodes $x, y \in V$, and an RPQ $r \in \mathcal{R}$.

Question: Is there a trail from x to y in G that matches r?

We parameterized the problems with a class $\mathcal{R}$ of regular expressions, so that we can discuss variants of these problems. (If $\mathcal{R}$ is just a single regular expression r, then we simply write $\mathsf{SimPath}(r)$ instead of $\mathsf{SimPath}(\{r\})$, and analogously for Trail.)

Notice that any algorithm that is able to answer RPQs (i.e., compute all matching paths) while considering simple paths and trails, is able to solve these decision problems. So, the complexity of these decision problems is important. Notice that both problems are trivially in NP. Mendelzon

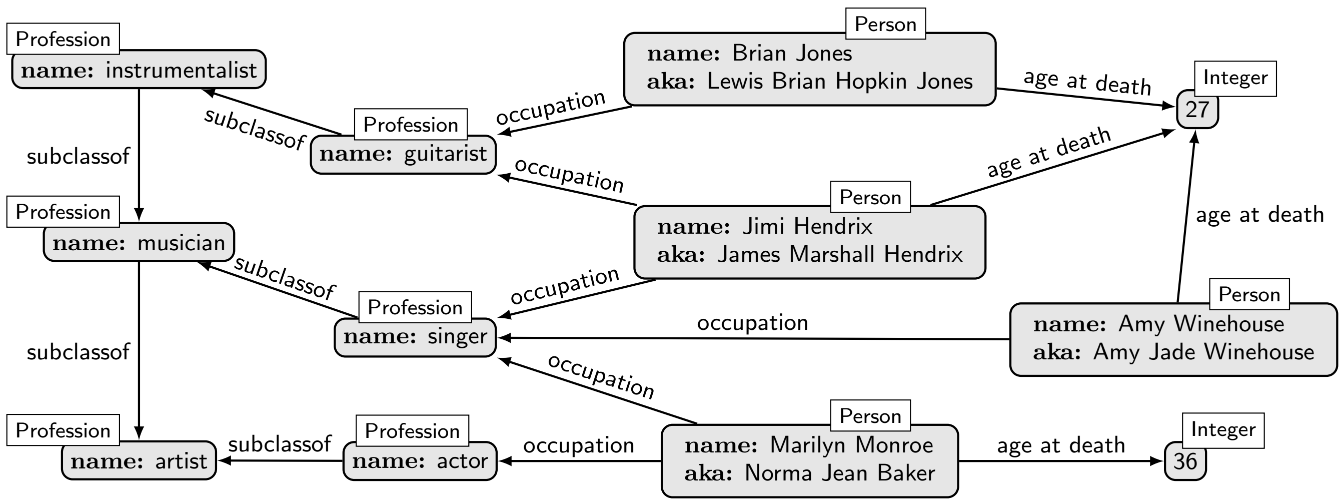


Figure 2: A graph database (as a *property graph*), inspired on a fragment of WikiData

and Wood studied SimPath and discovered that it becomes NP-hard very quickly [16]. Items (a) and (b) of the following theorem come from their work:

Theorem 1. *The following problems are NP-complete:*
(a) SimPath$((aa)^*)$ *(b)* SimPath(a^*ba^*)
(c) Trail$((aa)^*)$ *(d)* Trail(a^*ba^*)

Items (c) and (d) can be obtained by easy reductions from the *two disjoint paths* problem, using the standard split-graph construction from Perl and Shiloach [18] or LaPaugh and Rivest [12] and the same reductions as for simple paths used by Mendelzon and Wood [16].

So, not only are SimPath and Trail NP-complete, they are even NP-complete in cases where the regular path query is fixed. Furthermore, the expressions for which this NP-completeness holds can be very small. It is therefore no surprise that, from a worst-case complexity perspective, it seems to be a bad idea to build a query language for graph databases on simple path or trail semantics. We note that it is understood for which fixed regular expressions SimPath and Trail are NP-complete [3, 13].

3.2 What About Query Logs?

Once query logs became available, we have been able to analyze what kind of RPQs actually occur. The study of Bonifati et al. [5] had 247k SPARQL property paths in unique queries, which gave us a first impression. Syntactically, SPARQL property paths are extensions of RPQs. This is important, because it means that the types of regular expressions we will see are not syntactically constrained by the query language. On top of the ordinary operators for RPQs, SPARQL allows operators for wildcards and for following edges in the reverse direction. This would not be the case for Cypher, for example. (In Neo4j's Cypher 3.2 manual, only single labels or wildcards were allowed below Kleene stars [17]. Cypher 9 is becoming more liberal and allows disjunction below a Kleene star, see [10, Figure 3: Syntax of Cypher patterns]. In the near future, Cypher plans to support full regular path queries [10].)

In Table 2, we provide a summary of the types of property paths found in the data of [5]. That is, Table 2 is not the table appearing in [5], but we went over the raw data again and aggregated the types of expressions slightly differently. We use the following conventions: (1) lower case letters denote single symbols, (2) upper case letters denote sets of symbols, (3) we denote a wildcard test by $\sqcup$, (4) we do not distinguish between following an edge in the forward or backward direction, (5) each expression type also encompasses its symmetric form. For instance, when we write a^*b, we count the expressions of the form a^*b and ba^*. We always list the variant that occurred most often in the data. That is, a^*b occurred more often than ba^*. These conventions are the same as in our conference paper [14].

Under *Expression Type*, the table summarizes which types of expressions are in Bonifati et al.'s data set, sometimes parameterized by a number ℓ for which the next column describes the values that were found. *Relative* describes which percentage of the 247,404 expressions fall into this expression type. We discuss *STE?* in the next section.

In Table 2 we can immediately observe that the property paths found in the query logs of Bonifati et al. are not very complex and that the expressions mentioned in Theorem 1 only occur very rarely. In fact, the query $(ab)^*$ occured only once and we found out that this query was posed by a theoretician testing the robustness of the engine [Vrgoč, personal communication].

Another thing to keep in mind is how to interpret the classification in Table 2. After all, property paths do not occur often in the logs of Table 1: only about 0.4% of the queries have them. However, this seems to be an artifact of the underlying data. Most of the property paths appear in DBpedia queries, but DBpedia was designed when property paths were not yet part of SPARQL. In a more recent study on Wikidata query logs, containing 35 million unique queries, a drastically larger 38.94% of the queries use property paths [7]. Moreover, the structure of these property paths shows a picture similar to what we see in Table 2 [7].

4. SIMPLE TRANSITIVE EXPRESSIONS

We now define a class of RPQs called *simple transitive expressions (STEs)*, with the intent of capturing the vast majority of the expressions in Table 2, while avoiding the problems discussed in Section 3.1. Intuitively, simple tran-

Expression Type	ℓ	*Relative*	*STE?*
$(a_1 + \cdots + a_\ell)^*$	2–4	29.10%	yes
$\sqcup$		25.48%	yes$^{(*)}$
a^*		19.66%	yes
$a_1 \cdots a_\ell$	2–6	8.66%	yes
a^*b		7.73%	yes
$(a_1 + \cdots + a_\ell)$	1–6	6.61%	yes
$(a_1 + \cdots + a_\ell)^+$	1–2	1.54%	yes
$a_1?a_2? \cdots a_\ell?$	1–5	1.15%	yes
$a(b_1 + b_2)?$		0.01%	yes
$a_1a_2? \cdots a_\ell?$	2–3	0.01%	yes
$a^*b?$		$< 0.01\%$	yes
abc^*		$< 0.01\%$	yes
$A_1 \cdots A_\ell$	2–6	$< 0.01\%$	yes
$(a_1 + a_2)?$		$< 0.01\%$	yes
$\sqcup^*$		$< 0.01\%$	yes$^{(*)}$
$\sqcup b^*$		$< 0.01\%$	yes$^{(*)}$
$\sqcup?$		$< 0.01\%$	yes$^{(*)}$
$(ab^*) + c$		$< 0.01\%$	no
$a^* + b$		$< 0.01\%$	no
$a + b^+$		$< 0.01\%$	no
$a^+ + b^+$		$< 0.01\%$	no
$(ab)^*$		$< 0.01\%$	no

Table 2: Structure of the 247,404 SPARQL property paths that were also used in the query logs investigated by Bonifati et al. [5]. The structure is sometimes in terms of a variable $\ell \in \mathbb{N}$, for which the second column indicated the values that were found in the logs. *Relative* indicates which percentage of the 247,404 property paths have this structure.

sitive expressions aim at capturing very basic navigation in graphs: first do some *local navigation*, followed by an optional *transitive step*, and finally again some *local navigation*. The rationale is that, if we want to connect entities in a graph database, then this is a natural way to navigate. Let us again consider our running example of artists that died at the age of 27. When we want to find out if a Person is an artist, we first need to do some local navigation (following an `occupation`-edge) and then perform a transitive reflexive step (following an arbitrarily long path of `subclassof`-edges). More precisely, simple transitive expressions allow to:

1. first follow a path of length *exactly* k_1 or *at most* k_1 (for some $k_1 \in \mathbb{N}$),
2. then do a (reflexive) transitive closure step,
3. finally, follow a path of length *exactly* k_2 or *at most* k_2 (for some $k_2 \in \mathbb{N}$).

All three steps are subject to label tests. Furthermore, any step can be omitted, so a simple transitive expression can also express that paths must have length between k_1 and $k_1 + k_2$. In the following definition, we use sets $A = \{a_1, \ldots, a_\ell\} \subseteq \Sigma$ to abbreviate disjunctions $(a_1 + \cdots + a_\ell)$.

Definition 2. *An* atomic expression *is of the form $A \subseteq \Sigma$ with $A \neq \emptyset$. A* bounded expression *is a regular expression of the form $A_1 \cdots A_k$ or $A_1? \cdots A_k?$, where $k \geq 0$ and each A_i is an atomic expression. Finally, a* simple transitive expression (STE) *is a regular expression*

$$B_{pre} T^* B_{suff},$$

where B_{pre} and B_{suff} are bounded expressions and T is ε or an atomic expression.

A minor technicality is that we can take $T = \varepsilon$. This means that T^* will only match the empty word, and therefore the STE defines a finite language. In Table 2 the column *STE?* indicates whether the expression is an STE. Here, we write "yes$^{(*)}$" to indicate that the expression is an STE if a wildcard is treated the same as a set of labels A. (Our algorithms indeed can be generalized to incorporate wildcards.)

In total, we saw that only 20 property paths are not STEs or trivially equivalent to an STE (by taking $T = \varepsilon$ in the definition of STEs, for example). For instance, the expression type $a_1a_2? \cdots a_\ell?$ is equivalent to an STE where $B_{\text{pre}} = a_1$, $T = \varepsilon$, and $B_{\text{suff}} = a_2? \cdots a_\ell?$. In this sense, 99.992% of the property paths in Table 2 correspond to STEs.

In fact, *all* expressions in the table except for $(ab)^*$ are unions of STEs. Unions of STEs can actually be handled in the same way than STEs, by applying the STE evaluation algorithm to each part of the union.

4.1 Dichotomies for STEs

Our main technical results are two dichotomies for evaluating STEs under simple path and trail semantics. That is, we precisely characterize for which classes $\mathcal{R}$ of STEs the problems SimPath for $\mathcal{R}$ and Trail for $\mathcal{R}$ are easy and for which classes these problems are difficult. Here, "easy" and "difficult" refer to complexities in parameterized complexity, namely *fixed-parameter tractable* and W[1]-hard. Our results will imply that SimPath and Trail are "easy" for the types of expressions in Table 2 — except for $(ab)^*$. Furthermore, the parameters on which the complexity can exponentially depend are small.

Some Examples and Intuition.

We give a bit of intuition about our results. Throughout the example, we use the following notation. The input graph is always denoted as G, and it has n nodes and m edges. We always denote the start and end nodes in the input of the SimPath problem by x and y, respectively. We will abbreviate long concatenations with a power notation, that is, we use r^k to denote a sequence of k times the expression r. For instance a^4 denotes the expression $aaaa$. Let a^k denote the class $\{a^k \mid k \in \mathbb{N}\}$ of STEs. We define the classes $(a?)^k$, a^ka^*, ba^ka^*, and a^kba^* analogously.

We now discuss the complexities of SimPath for these classes. As a first example, we consider SimPath for $(a?)^k$. This problem is easy to solve: one can simply use an algorithm that tests reachability with a-labeled edges. The crux is that loops do not matter: if there is a path from x to y that matches $(a?)^k$ then there is also a simple such path, since removing loops does not change matching $(a?)^k$.

This technique does not work for our second example: SimPath for a^k. However, Alon et al.'s color coding technique [2] can solve this problem in time $2^{O(k)} m \log n$. Color coding therefore shows that SimPath for a^k is fixed-parameter tractable, where the parameter is the size k of the RPQ: it is an algorithm with complexity $f(k) \cdot p(|G| + k)$, where f is a computable function and p is a polynomial. The function f is even single exponential in this case. Notice that, if P

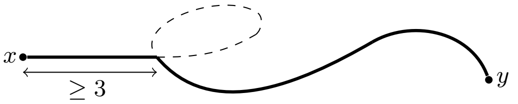


Figure 3: Intuition behind cuttability, using $bbba^*$

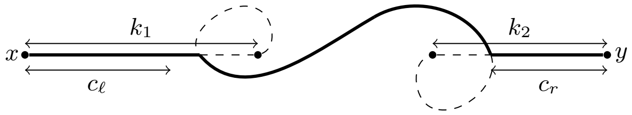
$\neq$ NP, we cannot hope for f to be a polynomial function, because SimPath for a^k is at least as difficult as the Hamiltonian Path problem. (Indeed, the cases of SimPath for a^k where we give a graph G with only a-labeled edges and the RPQ a^{m+1} are equivalent to the Hamiltonian Path problem for G.)

As a third example, we consider SimPath for $a^k a^*$. This problem requires yet another technique, since color coding is designed to work for fixed-length paths. It can be solved in time $2^{O(k)}(n^2 + mn)$, however, using the representative sets technique of Fomin et al. [9]. The representative sets technique is nontrivial and addresses the following problem. Assume that we try to deal with $a^k a^*$ naively by considering all simple paths P of length k that start in x. For each such path P, assuming it ends in some node x_P, we could then test reachability from x_P to y while avoiding the nodes of P. But this algorithm is too inefficient. We may have up to n^k different possibilities for P, which means that the running time is not of the form $f(k) \cdot p(|G| + k)$ for a polynomial p and computable function f. In other words, it does not show that the problem is fixed-parameter tractable. This is where the representative sets technique is useful. It shows that the number of different paths P we have to consider can be limited to $2^{O(k)}n$, which makes the problem fixed-parameter tractable. The representative sets technique can even be adapted so that it *enumerates* all the simple paths.

We turn to two cases where the edge labels become important. First, consider SimPath($ba^k a^*$). Here, we can simply enumerate all b-edges that start in x and then use the algorithm for SimPath($a^k a^*$) from there (and making sure that we don't visit x). This shows that SimPath($ba^k a^*$) is fixed-parameter tractable.

Second, take SimPath($a^k ba^*$). At its core, this problem is a variant of the Two Disjoint Paths problem. We are essentially searching for two nodes x' and y' such that there is a path P_1 of length k from x to x' and a path P_2 from y' to y. Moreover, P_1 and P_2 should be node-disjoint and there should be a b-edge from x' to y'. Since we can prove that this Two Disjoint Paths problem (with parameter k) is W[1]-hard [14], it turns out that SimPath($a^k ba^*$) is hard as well.

The central notion in our dichotomy for SimPath is *cut borders* of STEs. We explain this notion intuitively, based on two simple examples. Consider the expressions $r_1 = aaaa^*$ and $r_2 = aaab^*$. Assume that, as in Figure 3, we found a path p (that may contain a loop) from x to y that matches r_1. Intuitively, if we want to test if the simple path p' obtained from p by deleting all loops still matches r_1, we just need to test if p' has length at least three. For r_2, however, we additionally need to test that the loop does not occur in the prefix of length 3 of p. For this reason, the cut border of r_2 will be equal to 3. We can prove that this notion of cut border is indeed the crucial one for the complexity of SimPath.



c_ℓ : left cut border
c_r : right cut border

Figure 4: Assume $r = A_1 \cdots A_{k_1} T^* A'_{k_2} \cdots A'_1$ has left and right cut borders c_1 and c_2, respectively. Assume that an arbitrary path from s to t matches r such that its length k_1 prefix and length k_2 suffix do not have loops and are node disjoint. If, after removing all loops, (1) the length c_1 prefix and length c_2 suffix are still the same and (2) the path still has length at least $k_1 + k_2$, then it matches r.

Dichotomy for Simple Paths.

We now state our main result on SimPath and explain the cut borders, cuttability, and the sampling condition after its statement. (We only require the condition that $\mathcal{R}$ can be sampled for the lower bound proof in part (b).)

THEOREM 3. *Let $\mathcal{R}$ be a class of STEs that can be sampled.*

(a) If $\mathcal{R}$ is cuttable, then SimPath($\mathcal{R}$) is solvable in time $2^{O(s)}n^{c+3}m$, where c is the cut border of $\mathcal{R}$.

(b) Otherwise, SimPath($\mathcal{R}$) cannot be solved in time $f(s) \cdot (n + m)^c$ for a constant c and a computable function f, unless FPT = W[1].

Here, n and m are the number of nodes and edges in the graph, respectively, and s is the size of the regular expression.

Here, FPT is the class of problems that is *fixed-parameter tractable*. It is a standard assumption in parameterized complexity theory that FPT $\neq$ W[1]. This assumption has a similar calibre as the P $\neq$ NP assumption in terms of decision problems.

We now explain cut borders, cuttability, and the condition that $\mathcal{R}$ can be sampled. To this end, the *left* (resp., *right*) *cut border* of an STE $r = A_1 \cdots A_{k_1} T^* A'_{k_2} \cdots A'_1$ is the largest value i such that T has a symbol that is not in A_i (resp., A'_i). If we have $A_1? \cdots A_{k_1}?$ (resp., $A'_{k_2}? \cdots A'_1?$), then the left (resp., right) cut border is 0. The *cut border* of r is the sum of its left and right cut border. A class $\mathcal{R}$ of STEs is *cuttable* if there exists a $c \in \mathbb{N}$ such that the cut border of each expression $r \in \mathcal{R}$ is at most c. The intuition of cut borders is explained in Figure 4: they characterize parts of paths in which it is not allowed to remove loops to obtain a simple path that still matches the expression.

Finally, we say that $\mathcal{R}$ *can be sampled* if there exists an algorithm that, given a number k in unary, returns an expression from $\mathcal{R}$ that has cut border at least k. Notice that this is a very weak restriction on $\mathcal{R}$.

Notice that the difference between cuttable and non-cuttable classes of STEs can be subtle. Using the same notation as with our previous examples, the classes $a^k b^*$ and $a^k(a + b)^*$ are not cuttable, but $(a+b)^k a^*$ is. Looking back at Table 2, we see that abc^* is 2-bordered and all other STEs are either 0-bordered or 1-bordered. It therefore seems that cut borders in practice are small and over 99% of the expressions fall on the tractable side of Theorem 3.

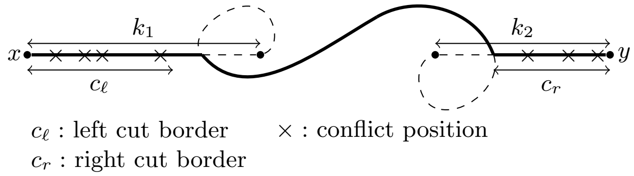


Figure 5: Visualization of the effect of conflict positions in a path that matches an STE r. If we would start with an arbitrary path and remove loops, we mainly need to be careful about labels behind the cut borders that can be identical to labels in the transitive part.

Dichotomy for Trails.

We now present a similar dichotomy for trails, obtained in [15]. The dichotomy is, perhaps surprisingly, different from the one in Theorem 3 in the sense that more classes fall on the tractable side. For instance, $\mathsf{SimPath}(a^kb^*)$ is intractable, whereas $\mathsf{Trail}(a^kb^*)$ is fixed parameter tractable because the a-path and the b-path can be evaluated independent of each other (no a-edge will be equal to a b-edge). We explain *conflict positions*, *almost conflict-freeness*, and *conflict-sampling* after the theorem statement. (The condition that $\mathcal{R}$ can be conflict-sampled is only needed for (b).)

Theorem 4. *Let $\mathcal{R}$ be a class of STEs that can be conflict-sampled.*

(a) If $\mathcal{R}$ is almost conflict free, then $\mathsf{Trail}(\mathcal{R})$ is solvable in time $2^{O(s)} \cdot m^{c+6}$, where c is the number of conflict positions in $\mathcal{R}$.

(b) Otherwise, $\mathsf{Trail}(\mathcal{R})$ cannot be solved in time $f(s)(n+m)^c$ for a constant c and a computable function f, unless FPT = W[1].

Here, n and m are the number of nodes and edges in the graph, respectively, and s is the size of the regular expression.

If $r = A_1 \cdots A_{k_1} T^* A'_{k_2} \cdots A'_1$ is an STE, we say that a position left of the left cut border (resp., right of the right cut border) is a *conflict position* if T and A_i (resp., A'_i) have a common symbol or, equivalently, have a non-empty intersection. If we have $A_1? \cdots A_{k_1}?$ (resp., $A'_{k_2}? \cdots A'_1?$), then the left (resp., right) cut border is 0 and therefore there are no cut positions. In Figure 5 we give a visual intuition about the meaning of conflict positions. A class $\mathcal{R}$ of STEs is *almost conflict free* if there exists a constant $c \in \mathbb{N}$ such that each expression $r \in \mathcal{R}$ has at most c conflict positions. The class a^kb^* is not cuttable, but it is conflict-free because $\{a\}$ and $\{b\}$ have an empty intersection. The point is that an edge labeled by some symbol in $\{a\}$ can never be the same than an edge labeled by some symbol in $\{b\}$, since their labels must be different. Therefore, we can evaluate a^k and b^* separately.

A class $\mathcal{R}$ of STEs *can be conflict-sampled* if there exists an algorithm that, given a number k in unary, returns an expression $r \in \mathcal{R}$ with at least k conflict positions.

Extension to Enumeration of Paths.

Real-life graph databases are usually not primarily interested in solving decision problems, but in computing the answers to a query. *Enumeration algorithms* can be seen as a theoretical framework in which such problems can be studied. Such algorithms typically consider the *preprocessing time* and *delay* for computing the answers of a query. Here, the *preprocessing time* is the time required before producing the first answer (and possibly build a data structure so that consecutive answers can be generated quickly) and the *delay* is the time required between two consecutive answers. In this framework, the requirement is usually that each answer is returned only once.

The tractability results from Theorems 3(a) and 4(a) can be extended to enumeration problems. Using an adaptation of Yen's algorithm [19] that works with labeled simple paths (resp., labeled trails), it can be shown that the *paths that match the expressions* can also be enumerated in such a way that the *delay* between the answers roughly corresponds to the upper bounds in Theorems 3(a) and 4(a).

Theorem 5. *Let $G = (V, E)$ be a graph, x and y be two nodes in V, and $\mathcal{R}$ a set of STEs.*

If $\mathcal{R}$ is cuttable, then the simple paths from x to y that match r, for a given $r \in \mathcal{R}$ can be enumerated with $2^{O(s)} \cdot n^{c+3}m$ preprocessing time and $2^{O(s)} \cdot n^{c+4}m$ delay.

If $\mathcal{R}$ is almost conflict free, then the trails from x to y that match r, for a given $r \in \mathcal{R}$ can be enumerated with $2^{O(s)} \cdot m^{c+6}$ preprocessing time and $2^{O(s)} \cdot m^{c+7}$ delay.

Core Techniques.

At the core of our tractability results lies the representative sets technique of Fomin et al. [9]. This technique can be used to find simple paths and trails of length at least k in time $2^{O(k)}(n^2 + nm)$, given a graph and the number k. If regular path queries are involved, the technique is only compatible with certain languages, such as cuttable or conflict-free STEs. The compatible languages have the property that we only need to guard a constant number of nodes/edges at the beginning and at the end of the path, to make sure that the rest of the path does not re-use the same nodes/edges.

Indeed, we can show that for languages violating this property, the problem becomes intractable. The reason is that it becomes at least as hard as a parameterized version of the two-disjoint paths problem. This parameterized problem asks: given a graph G, node pairs (x_1, y_1) and (x_2, y_2), and parameter $k \in \mathbb{N}$, are there two disjoint paths p_1 from x_1 to y_1 and p_2 from x_2 to y_2 such that p_1 has length k. (One can consider node-disjoint or edge-disjoint paths here.) We prove that this problem is W[1]-hard, both when node- or edge disjointness is required.

4.2 What Does This Mean?

If we interpret Theorems 3 and 4 in the light of the real world property paths in Table 2 we can observe the following. Let n and m be the number of nodes and edges of the graph, respectively.

Concerning simple paths, Theorem 3 gives us a running time of $2^{O(s)}n^{c+3}m$ for regular path query evaluation, where s is the size of the regular path query and c is the cut border. This result, together with the observation that the largest cut border in Table 2 is two, and therefore very small, can be seen as an explanation why, in practice, simple path semantics usually does not bring systems to their knees, even though this would theoretically be possible using regular expressions such as $(aa)^*$. Since the evaluation problem under simple path semantics generalizes the Hamilton Path problem (if $s = n - 1$), we cannot hope for a significantly better complexity unless P = NP.

One should keep in mind that this is a worst-case bound. In most practical settings, we expect that the run-time of even more naive evaluation algorithms will not come close to requiring n^{c+3} time for these simple expressions. For instance, the n^c factor comes from considering all paths that start in a given node x and obey a label constraint. For instance, for the expression abc^*, these are just the paths that start in x and are labeled ab. While this can, in the worst case, be n^2 many paths, we expect this to be much less in real databases.

The story for trails is similar. Here our upper bound admittedly gives less efficiency guarantees than the one for simple paths, but this is mainly because we have developed our methods for simple paths and then adapted them for trails. Furthermore, the dichotomy shows that it is easier to deal with trails than with simple paths: for every class of queries for which we have fixed-parameter tractable algorithms for simple path semantics, we also have them for trail semantics, but not vice versa.

5. CONCLUSIONS

The results in Section 4 can be seen as a theoretical explanation why evaluating certain queries (regular path queries against simple paths and trails) in graph databases seems to be less problematic in practice than theoretical results seem to suggest. The main reason is that, in the query logs that were considered in Section 2, the parameters that have a drastic impact on the complexity of evaluation remain relatively small.

In this sense, this paper showcases a line of work in which query log analysis was useful. However, a lot of work still remains to be done. First of all, the analysis of the logs in Section 2 showed much more than just the distribution of regular path queries. For instance, the shapes of queries found in the logs may be useful to generate realistic benchmarks. Second of all, the query log analysis from Section 2 itself is challenging too. For instance, in all the query logs we have seen until now, the distribution of the queries (and of interesting properties of queries) is extremely skewed. It is not clear how we balance finding interesting aspects of queries in logs with the fact that so many queries are extremely small, e.g., only have a single triple.

Furthermore, apart from having investigated successful and timeout queries for Wikidata [7], we do not know much about the combination of queries and data. For instance, it could be very interesting to study which parts of the graph are used for the evaluation of a query, and how large intermediate results become. Such studies must be left to future work.

Acknowledgments

In terms of methodology, we were heavily inspired by a line of work initiated by Frank Neven. In 2004, Frank had the idea to do a practical study on the shapes of regular expressions in schemas for XML data. This study motivated theoretical work on simple regular expressions (later called chain regular expressions), k-occurrence regular expressions and later work on schema inference. We acknowledge the grant 4938/4-1 by the Deutsche Forschungsgemeinschaft (DFG).

6. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the ACM*, 42(4):844–856, 1995.

[3] G. Bagan, A. Bonifati, and B. Groz. A trichotomy for regular simple path queries on graphs. In *PODS*, pages 261–272, 2013.

[4] A. Bielefeldt, J. Gonsior, and M. Krötzsch. Practical linked data access via SPARQL: the case of wikidata. In *LDOW@WWW*, 2018.

[5] A. Bonifati, W. Martens, and T. Timm. An analytical study of large SPARQL query logs. *PVLDB*, 11(2):149–161, 2017.

[6] A. Bonifati, W. Martens, and T. Timm. An analytical study of large SPARQL query logs. *The VLDB Journal*, 2019. Full version of [5], to appear.

[7] A. Bonifati, W. Martens, and T. Timm. Navigating the maze of wikidata query logs. In *WWW*, pages 127–138, 2019.

[8] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In *SIGMOD Conference*, pages 323–330, 1987.

[9] F. V. Fomin, D. Lokshtanov, F. Panolan, and S. Saurabh. Efficient computation of representative families with applications in parameterized and exact algorithms. *Journal of the ACM*, 63(4):29:1–29:60, 2016.

[10] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *SIGMOD Conference*, pages 1433–1445, 2018.

[11] G. Gottlob, G. Greco, N. Leone, and F. Scarcello. Hypertree decompositions: Questions and answers. In *PODS*, pages 57–74, 2016.

[12] A. S. LaPaugh and R. L. Rivest. The subgraph homeomorphism problem. *Journal of Computer and System Sciences*, 20(2):133 – 149, 1980.

[13] W. Martens, M. Niewerth, and T. Trautner. A trichotomy for regular trail queries. *CoRR*, abs/1903.00226, 2019.

[14] W. Martens and T. Trautner. Evaluation and enumeration problems for regular path queries. In *ICDT*, pages 19:1–19:21, 2018.

[15] W. Martens and T. Trautner. Dichotomies for evaluating simple regular path queries. *ACM Transactions on Database Systems*, 2019. Full version of [14], to appear.

[16] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258, 1995.

[17] Neo4j. The neo4j developer manual v3.2. `https://neo4j.com/docs/developer-manual/3.2/`, 2017.

[18] Y. Perl and Y. Shiloach. Finding two disjoint paths between two pairs of vertices in a graph. *Journal of the ACM*, 25(1):1–9, 1978.

[19] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.