How to execute SPARQL property path queries online and get complete results?

Julien Aimonier-Davat, Hala Skaf-Molli, Pascal Molli

How to execute SPARQL property path queries online and get complete results?

Julien Aimonier-Davat, Hala Skaf-Molli, and Pascal Molli

LS2N – University of Nantes, France
{Julien.Aimonier-Davat,hala.skaf,pascal.molli}@univ-nantes.fr

Abstract. SPARQL property path queries provide a concise way to write complex navigational queries over RDF knowledge graphs. However, the evaluation of these queries over online knowledge graphs such as DBPedia or Wikidata are often interrupted by quotas, returning no results or partial results. Decomposing SPARQL property path queries into triple pattern subqueries allows to get complete results. However, such decomposition generates a high number of subqueries, a large data transfer and finally delivers poor performances. In this paper, we propose an algorithm able to decompose SPARQL property path queries into Basic Graph Pattern (BGP) subqueries. As BGP queries are guaranteed to terminate on preemptable SPARQL servers, property path queries always deliver complete results. Experimental results demonstrate that our approach outperforms existing approaches in terms of HTTP calls, data transfer and query execution time.

1 Introduction

Context and motivation: Property path queries provide a concise way to write sophisticated navigational queries in Knowledge Graphs (KGs). SPARQL queries with property paths are largely used. They represent a total of 38% of the entire log of wikidata [7]. However, executing these complex queries against online public SPARQL services is challenging, mainly due to quotas enforcement that prevent queries to deliver complete results as pointed out in [18, 11, 16]. This raises the main issue of the paper: *How to execute SPARQL property path queries online and get complete results?*

Related Works: The decomposition of SPARQL property path queries into subqueries that may terminate under quotas allows to get complete results. However, ensuring the termination of any query under quotas is challenging [2]. Another option is to rely on restricted SPARQL servers that ensure the termination of supported SPARQL queries such as Triple Pattern Fragment (TPF) servers [24] or Preemptable servers [16] such as SaGe [1]. The granularity of the decomposition strongly impacts the execution time of the initial query, i.e. a decomposition of a property path query into triple patterns generates more subqueries than a decomposition into Basic Graph Patterns (BGPs) where a BGP is

[1] http://sage.univ-nantes.fr

```
select ?oeuvre ?inspiration
where {
  ?oeuvre wdt:P144 ?inspiration .
  ?oeuvre wdt:P31/wdt:P279* wd:Q17537576 .
  ?inspiration wdt:P136 wd:Q8253
}
```

(a) $Q1$: Creative works and the list of fiction works that inspired it on Wikidata

```
@prefix owl: <http://www.w3.org/2002/07/owl#>
@prefix foaf: <http://xmlns.com/foaf/0.1/>
select ?x ?o where {
  ?x foaf:name ?n .
  ?x owl:sameAs* ?o .
}
```

(b) $Q2$: list of similar entities on DBPedia

Fig. 1: Property path queries on online KGs

a set of triple patterns. Unlike TPF servers, Preemptable servers support BGPs. Unfortunately, there is currently no algorithm to decompose a property path query into BGP subqueries.

Approach and Contributions: In this paper, we propose an algorithm able to decompose a SPARQL property path query into BGP subqueries with filters and unions. As the generated subqueries are guaranteed to terminate when processed by a preemptable SPARQL server, the property path queries are executed online and always return complete results.

The contributions of the paper are the following: (i) We define an algorithm that computes a compressed automaton for SPARQL property path queries. The algorithm allows to decompose the SPARQL property path queries into BGP subqueries. (ii) We compare the performance of our approach with existing approaches (TPF and SaGe). Experimental results demonstrate that the compressed automata approach outperforms existing approaches by several orders of magnitude in terms of HTTP calls, execution time and data transfer.

This paper is organized as follows. Section 2 reviews related works. Section 3 introduces SPARQL property path queries and automata as property path expressions models. Sections 4 presents the automata compression approach in the context of the web preemption. Section 5 presents our experimental results. Finally, the conclusion is outlined in Section 6.

2 Related Works

Property paths were introduced in SPARQL 1.1 [2] to add extensive navigational capabilities to the SPARQL query language. Property paths closely correspond to regular expressions and are crucial to perform non-trivial navigation in knowledge graphs. Regular expressions involve operators such as ' * ' (zero or more occurrences-kleene star), ' | ' (OR operator), ' / ' (sequence operator), ' $\wedge$ ' (inverse operator), ' ! ' (NOT operator) that allow to describe complex paths of arbitrary length. For instance, the query `SELECT ?x ?y WHERE ?x foaf:knows* ?y` require to compute the transitive closure of the relation $foaf : knows$ over all pairs x, y present in the KG. Many techniques [19, 6] proposed to compute such queries but, computing transitive closure over large graphs remains costly.

[2] https://www.w3.org/TR/sparql11-property-paths/

Breadth First Search or Depth First Search algorithms compute transitive closures with a time complexity in $\mathcal{O}(|E|+|V|)$ and a space complexity in $\mathcal{O}(|V|^2)$, with E and V the finite set of KG edges and vertices, respectively. Even if different optimisations have been proposed [25, 12] that greatly improve performances, a simple property path query evaluation over a large graph may require a large amount of CPU and memory to complete.

This makes the evaluation of property path queries challenging on online Knowledge Graphs such as DBPedia or Wikidata. To ensure a fair usage policy of resources, public SPARQL endpoints enforce quotas [9] in time and ressources for executing queries. As queries are stopped by quotas, many queries return no results or partial results. For instance, the query Q_1 Figure 1 returns no result on Wikidata because it has been stopped after running more than 60s. The Q_2 [3] on DBPedia returns partial results because it has been killed after delivering the first 10000 results.

To overcome quotas limitations, KG providers publish dumps of their data. However, re-ingesting billions of triples on local resources to compute SPARQL property path queries is extremely costly and raises issues with freshness. Moreover, it is an offline approach, and in this paper we want to execute property path queries online and get complete results.

To overcome quota limitations, it is also possible to decompose SPARQL queries into subqueries that may terminate under quotas [2]. However, finding such decomposition is hard in the general case, as quotas can be different from one server to another, both in terms of values and nature [2]. Consequently, there is no guarantee that subqueries terminate. Another option is to rely on restricted server interfaces to ensure that the execution of subqueries terminate, e.g. the Triple Pattern Fragments approach (TPF) [10, 24] or the preemptable server SaGe [16]. However, the granularity of the decomposition strongly impact the execution time of the initial query. Relying on the TPF interface, a property path query has to be decomposed into sequences of multiple triple pattern queries, while a preemptable server allows to decompose property path queries into BGP queries with union and filters.

The TPF client [10, 24] decomposes SPARQL queries into sequences of paginated triple pattern queries. As paginated triple patterns queries can be executed in bounded times, the server does not need quotas, i.e. all queries executed by the server have nearly the same duration. However, as the TPF server only processes triple pattern queries, property paths have to be decomposed into sequences of triple pattern queries. This requires to compute several joins on the client, especially to compute transitive closure expressions, which require a high number of HTTP calls and a large data transfer leading to poor query performance.

[3] Q_1 and Q_2 are executed at the public SPARQL endpoints of Wikidata, and DBPedia, respectively, at August 5 2020.

SaGe implements the web preemption [16] model. A preemptable server interrupts a SPARQL query execution after a quantum of time, returning partial results and the state of the SPARQL query (the query execution plan). The client can continue the query execution by sending the state of SPARQL query back to the preemptable server. Following the web preemption model, many queries may be virtually suspended, but the server remains stateless. As queries are suspended after a quantum, the server only processes queries of nearly the same duration and there is no need for quotas. SaGe server implements the evaluation of triple patterns, BGPs, filters and unions. Although, web preemption allows processing BGPs, property paths are still decomposed into sequences of triple pattern queries leading to poor query performance.

A BGP decomposition is much more efficient than a triple pattern decomposition, as it generates less subqueries and transfers less intermediate results. Unfortunately, there is no algorithm able to decompose property path into BGP queries. In this paper, we propose an algorithm of decomposition based on automaton compression. Similar automaton compression techniques have been already used in other domains, but not related to query processing [26].

3 Property Path Expressions and Automata

We recall briefly definitions related to the proposal of the paper.

3.1 SPARQL property path queries

SPARQL Queries : We follow the notation from [17, 20] and consider three disjoint sets I (IRIs), L (literals) and B (blank nodes) and denote the set T of RDF terms $I \cup L \cup B$. An RDF triple $(s, p, o) \in (I \cup B) \times I \times T$ connects subject s through predicate p to object o. An RDF graph $\mathcal{G}$ (called also RDF dataset) is a finite set of RDF triples. We assume the existence of an infinite set V of variables, disjoint with previous sets. A mapping μ from V to T is a partial function $\mu : V \to T$, the domain of μ, denoted $dom(\mu)$ is the subset of V where μ is defined.

A SPARQL graph pattern expression P is defined recursively as follows.

1. A tuple from $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is a triple pattern.
2. If $P1$ and $P2$ are graph patterns, then expressions (P1 AND P2), (P1 OPT P2), and (P1 UNION P2) are graph patterns (a conjunction graph pattern, an optional graph pattern, and a union graph pattern, respectively).
3. If P is a graph pattern and R is a SPARQL built-in condition, then the expression (P FILTER R) is a graph pattern (a filter graph pattern).

SPARQL Property Path Queries: The SPARQL 1.1 language [21] introduces property paths. We adopt the same syntax as [14] to define operator property

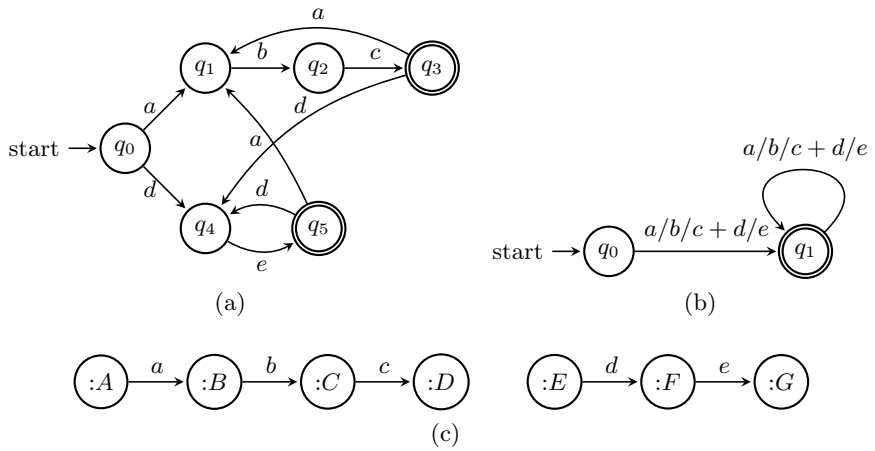


Fig. 2: For the path expression $P = ((a \cdot b \cdot c) + (d \cdot e))^+$: (a) **mono-predicate** automaton of P; (b) minimal **multi-predicate** automaton of P; (c) graph G_1

path expressions, i.e. the inverse path is denoted by e^- and alternative path $e_1 + e_2$ [4].

Definition 1 (Property Path Expressions [14]).

Property path expressions are defined by the grammar:

$$e := a \mid e^- \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e^+ \mid e^* \mid e? \mid !a_1, \ldots, a_k \mid !a_1^-, \ldots, a_k^-,$$

*where a, a_1, ..., a_k are properties, i.e. $IRIs$ in I. A single property is called a **predicate path expression**. It is the smallest path expression and it can only match paths of length one, i.e. triple patterns. Expressions of the forms $(e_1 \cdot e_2)$, $(e_1 + e_2)$, (e^{+*}) and $(e?)$ are respectively called **sequence, alternative, transitive and optional path expressions**. Expressions of the last two forms (i.e. starting with !) are called negated property sets. The set of all property paths expressions is denoted by PP.*

Definition 2 (Property Path [14]). *A property path pattern is a triple in $(I \cup L \cup V) \times PP \times (I \cup L \cup V)$.*

Property path patterns are incompatible with triple patterns, because they allow property path expressions in predicate positions but forbid variables in these positions.

3.2 Modeling property path expressions via automata

A finite automaton is often used to represent a property path expression [6, 8]. For example, the property path $P = ((a \cdot b \cdot c) + (d \cdot e))^+$ can be represented by the

[4] SPARQL 1.1 uses symbols $\hat{}e$ and $e_1|e_2$ for inverse and alternative path, respectively

automaton described in Figure 2a. We call such automaton a **mono-predicate** automaton.

To evaluate the query *Q=select * where {:A P ?y}* over the graph G_1 in Figure 2c, an automaton-based approach [6, 5] processes as follows:

1. A search is initialized from the configuration c_0=(q_0, :A), where q_0 is the initial state of the automaton, and :A is the subject of the property path pattern of Q .
2. From the configuration c_0, states q_1 and q_4 could be reached. q_1 is reached as the evaluation of $[\![$:A a ?$y]\!]_{G_1} = \{?y \rightarrow$:B$\}$. Therefore, the configuration $c_1 = (q_1,$:B) is built. q_4 is not reached as $[\![$:A d ?$y]\!]_{G_1} = \emptyset$.
3. From the configuration c_1, the state q_2 is reached as $[\![$:B b ?$y]\!]_{G_1} = \{?y \rightarrow$:C$\}$, the configuration $c_2 = (q_2,$:C) is built.
4. From the configuration c_2, q_3 is reached with $[\![$:C c ?$y]\!]_{G_1} = \{?y \rightarrow$:D$\}$, $c_3 = (q_3,$:D) is built. As q_3 is a final state, $\{?y \rightarrow$:D$\}$ is a solution to the query Q.
5. The process continues from c_3, but no more solutions can be found. The algorithm terminates when all the configurations have been found.

As we can see, all evaluations performed on G_1 correspond to triple pattern queries. By this way, the automaton-based approach decomposes the property path expressions into triple pattern subqueries. We call such automaton *a mono-predicate automaton.*

4 Compression of Property Path Automata

The mono-predicate automaton in Figure 2a is equivalent to the automaton presented in Figure 2b, i.e. they both recognize the same language. Compared to the mono-predicate automaton, transitions in the second automaton are labeled with more complex expressions such as sequences and alternatives. We call this automaton *a multi-predicate automaton.*

If both automata are equivalent, evaluating the path expression with the multi-predicate automaton generates BGP subqueries with union, which is much more efficient than evaluating triple pattern subqueries. Obviously, the second automaton is a compressed version of the first one. *The scientific problem is to write an algorithm able to transform any mono-predicate automaton into an equivalent* ***minimal*** *multi-predicate automaton according to servers capabilities.*

A multi-predicate automaton is said to be minimal if it does not exist another equivalent multi-predicate automaton with less states and transitions, according to a set of operators supported by a server, i.e. server capabilities.

For example, a mono-predicate automaton is a minimal multi-predicate automaton for a TPF server, as TPF only supports triple pattern queries. For a server

supporting BGP and union such as SaGe, the multi-predicate automaton of Figure 2b is minimal, while the multi-predicate automaton presented in Figure 3d is not.

In this paper, we only consider *predicate, sequence, alternative and transitive path expressions*. Optionals are ignored as they are naturally rewritten as alternatives when a property path expression is converted into a finite automaton. Concerning negated property sets and inverse path expressions, they can be treated as special cases of predicate path expressions. An inverse path expression can be rewritten as a triple pattern whose subject and object have been reversed, while a negated property set can be rewritten as a triple pattern whose predicate is a variable that is associated to a "not in" filter condition to exclude unwanted properties.

4.1 Algorithm for compressing path expression automata

In this section, we describe an algorithm to transform a mono-predicate automaton into a multi-predicate automaton. The algorithm is composed of two parts:

1. We first build an automaton for *sequence path expressions*, i.e. not considering alternatives. This produces a first compressed automaton that may be not minimal.
2. Second, we compress the automaton produced by the previous step considering *alternatives* to produce a minimal compressed automaton.

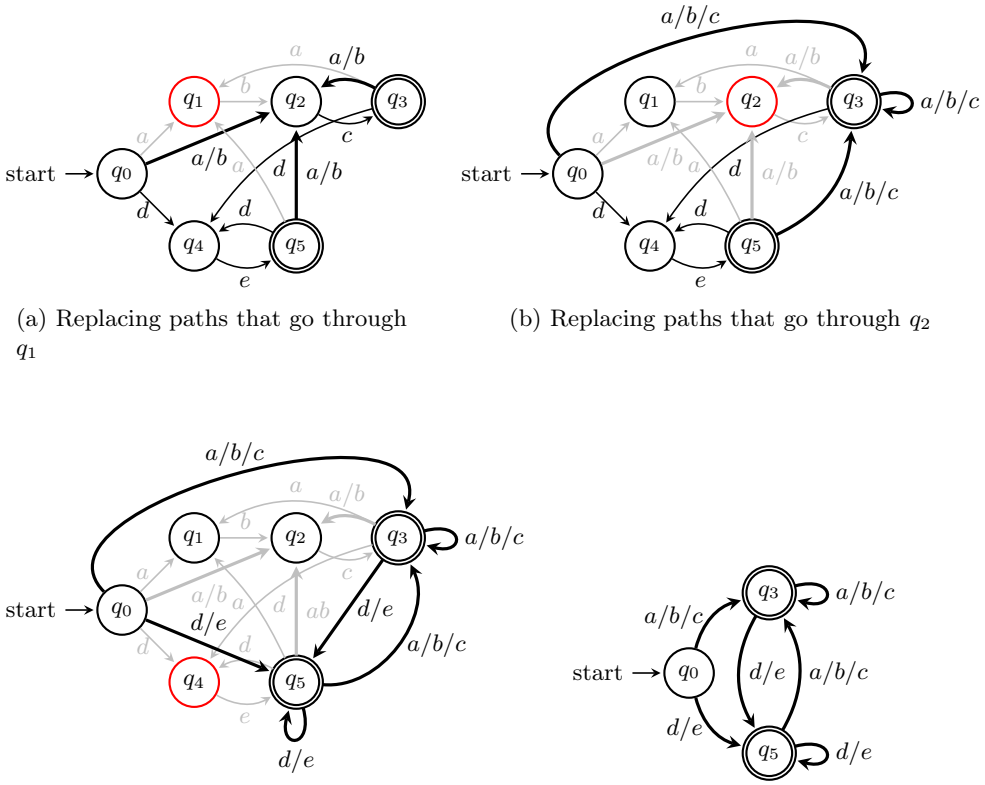
4.1.1 Processing sequence path expressions

When a property path expression is converted into a mono-predicate automaton, sequences without transitive closures are converted into paths of mono-predicate transitions, such that (1) consecutive transitions are connected together by an **intermediate state**, (2) paths start from a non-intermediate state, (3) paths go to a non-intermediate state.

Definition 3 (Intermediate state). *A state is called an intermediate state if and only if (1) it is not a start state, (2) it is not a final state, (3) it has no self transition.*

For example, in the mono-predicate automaton of the property path $P = ((a \cdot b \cdot c) + (d \cdot e))^+$, the sequence $(a \cdot b \cdot c)$ corresponds to the paths $<(q_0, a, q_1), (q_1, b, q_2), (q_2, c, q_3)>$, $<(q_3, a, q_1), (q_1, b, q_2), (q_2, c, q_3)>$, $<(q_5, a, q_1), (q_1, b, q_2), (q_2, c, q_3)>$ where q_0, q_3 and q_5 are non-intermediate states, while q_1 and q_2 are intermediate states.

Consequently, the first step to build a minimal multi-predicate automaton is to replace these paths by single transitions that are labeled with the corresponding sequence path expressions. As only paths extremities are non-intermediate



(a) Replacing paths that go through q_1

(b) Replacing paths that go through q_2

(c) Replacing paths that go through q_4

(d) Result of replacing paths between non-intermediate states by the shortest paths

Fig. 3: Running Algorithm 1 on the automaton depicted in Figure 2a

states, a simple solution to achieve this is to compute the shortest paths between the non-intermediate states, before removing the intermediate states and all transitions that are connected to one of them. Algorithm 1 follows this procedure by using a Floyd-Warshall based approach to compute the shortest paths between the non-intermediate states.

To illustrate, consider the automaton in Figure 2a. Algorithm 1 starts by considering all intermediate states. In this example q_1, q_2 and q_4. For each of them, the algorithm searches all pairs of transitions that are consecutive through it. Two consecutive transitions can be seen as the two operands of a join operator, where the label of the first transition is the right operand, while the label of the second transition is the left operand. Consequently, the two transitions can be merged together into a new transition, labeled with the concatenation of the two operands. A delimiter (/) is used to be able to parse the expression, in order to convert it into a BGP query, during the evaluation of the property path expression. Figure 3a presents the automaton obtained after considering the intermedi-

Algorithm 1: Compression sequence of path expressions

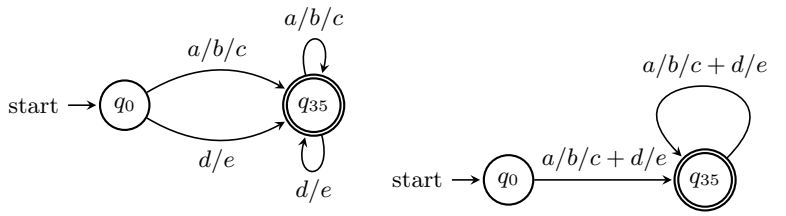
```
   Input: A mono-predicate automaton
   Output: A multi-predicate automaton that defines BGP decomposition
 1 begin
 2   foreach k ∈ intermediate state do
 3     foreach i ∈ state and ∃ transition(i,k) do
 4       foreach j ∈ state and ∃ transition(k,j) do
 5         create new transition (i,j)
 6         l = concat(label (i,k),label(k,j))
 7         addLabel(l, transition (i,j))
 8         add transition (i,j)
 9       end
10     end
11   end
12   remove the original transitions for which the source or the destination is
       an intermediate state
13   remove intermediate states
14 end
```

ate state q_1. At this step, paths that go through state q_1 are found and replaced. For example, the two consecutive transitions $(q_0, a, q_1), (q_1, b, q_2)$, where predicate path expressions a and b are the operands of the sequence $(a \cdot b)$, are replaced by the transition $(q_0, a/b, q_2)$. In Figure 3b it is paths that go through state q_2 that are found and replaced. Of course, transitions introduced in the previous steps are considered. Thus, the two consecutive transitions $(q_0, a/b, q_2), (q_2, c, q_3)$ are replaced by the transition $(q_0, a/b/c, q_3)$. At this step, we can see that the expression $(a \cdot b \cdot c)$ is now complete. No more transition is labeled with a subexpression of $(a \cdot b \cdot c)$. Finally, Figure 3c presents the automaton obtained after considering the last intermediate state q_4. At the end, all paths that go through states q_1, q_2 or q_4 have been found and replaced by single transitions labeled with the corresponding sequence expressions. After removing the intermediate states, we obtain the automaton described in Figure 3d. This multi-predicate automaton is minimal if we consider a server that only supports triple patterns and BGPs.

4.1.2 Processing alternative path expressions

Although, evaluating alternative expressions on the server-side does not improve the data transfer, it can significantly reduce the number of subqueries. To illustrate, if the property path $P = (a_1 + ... + a_n)$ is evaluated on the client, then it requires to send n subqueries to the server. However, only one call is required to evaluate this expression on the server.

Definition 4 (Equivalent states). *Two states in a finite automaton M are equivalent if and only if for every string x, if M is started in either state with x as input, it either accepts in both cases or rejects in both cases.*



(a) Minimizing the automaton

(b) Merging transitions with the same sources and destinations

Fig. 4: Compressing alternative expressions on the automaton depicted in Figure 3d

When sequences have been processed, the n clauses of a same alternative expression are represented by n transitions, such as they start from the same state and go to the same or equivalent states. For example, in Figure 3d the two transitions $(q_0, a/b/c, q_3)$ and $(q_0, d/e, q_3)$ are the two clauses of the expression $(a \cdot b \cdot c) + (d \cdot e)$. Consequently, the last step to build a multi-predicate automaton is just to merge transitions that share the same sources and equivalent destinations. This also requires to merge equivalent states. A simple solution is to merge equivalent states by using a minimization algorithm such as [13]. Then, transitions that are part of the same alternative expression can be safely merged, knowing that in a minimized automaton, n transitions represent the n clauses of a same alternative expression if they share the same source and destination.

To illustrate, consider the automaton in Figure 2a and imagine that sequences have already been processed, resulting in the automaton in Figure 3d. To perform the second transformation, i.e. merge the equivalent states, first, the automaton should be minimized. In Figure 4a, the two equivalent states q_3 and q_5 are merged into a new state q_{35}. Then, transitions that share the same sources and destinations are merged together. For example, the two transitions (q_0, abc, q_{35}), (q_0, de, q_{35}) are part of the same alternative expression $((a \cdot b \cdot c) + (d \cdot e))$, consequently, they are merged into a new transition $(q_0, (a/b/c+d/e), q_{35})$. Here, we use the delimiter $(+)$ to be able to rewrite this expression as an union of BGPs. Finally, the resulting automaton in Figure 4b is the corresponding minimal multi-predicate automaton of the property path $((a \cdot b \cdot c) + (d \cdot e))^+$. The decomposition defines by this automaton is effectively minimal in the context of a server that supports triple patterns, joins and unions.

5 Experimental Study

We want to empirically answer the following questions: Does compressed automata approach follow the W3C semantics of SPARQL property paths? Does compressed automata approach outperform mono-predicate automata approach in terms of query execution time, number of HTTP calls and data transfer? Does compressed automata approach outperform existing client-side approaches in terms of query execution time, number of HTTP calls and data transfer?

We implemented our multi-predicate automata compression approach as an extension of the SaGe query engine framework. All extensions and experimental results are available at https://github.com/JulienDavat.

5.1 Experimental setup

Dataset and Queries: We used BeSEPPI benchmark and gMark framework. We used **BeSEPPI benchmark** to study the compliance of our approach with the W3C semantics. BeSEPPI [1] is a benchmark designed to test the different semantics aspects of SPARQL property path expressions. BeSEPPI has 236 queries (73 ASK queries and 163 SELECT queries) and a dataset of 29 triples. The dataset is kept small in order to make the verification and the creation of new queries simple. According to [1], an approach follows the W3C semantics if each of the 236 queries returns a *complete and correct result*. In 2012, for complexity reasons [15, 3], the evaluation of transitive closure expressions has changed from a multi-set semantics to a set semantics. Because none of the 236 queries allow to check if this change has been taken into account, we added 6 new SELECT queries and 30 new triples.These queries are designed around the clique test introduced in [3]. We used **gMark framework** [4] to compare our approach with SaGe-Jena and Comunica. We generate a workload of 30 property path queries with complex path expressions on a dataset of 1M triples using the default "Shop" scenario of the framework.

Approaches: We compare the following approaches: (1) *SaGe-AC*: implements the automata compression approach on the SaGe smart client. (2) *SaGe-A*: for comparison, implements a traditional automaton-based approach with a mono-predicate automaton on the SaGe smart client. (3) *SaGe-Jena*: is implemented as an extension of Apache Jena[5], consequently, property path expressions are evaluated as defined in Jena, i.e. property path expressions are decomposed into sequences of triple patterns. (4) *Comunica*: a TPF smart client.

Servers configurations: We run the experimentations on a machine with a Processor Intel® Core™ i7-6700HQ CPU @ 2.60GHz x 8 and 16GB of RAM. To be able to run SaGe-Jena and our approach, we run a SaGe server with a time quantum of 75ms, a page-size of 2000 mappings and HDT files as backend. For Communica (version 1.12.1), we run a TPF server (version 2.2.5) with HDT files as backend and the same settings as SaGe.

Evaluation Metrics: (1) *Compliance with W3C semantics*: check whether the 236 return complete and correct results as defined in [1], i.e. produce the same results. We also checked manually the compliance of the results of our six defined queries. (2) *Data transfer*: is the total number of bytes transferred to the client when executing a query. (3) *Number of http calls*: is the total number of HTTP calls issued by the client when executing a query. (4) *Execution time*: is the total

[5] https://jena.apache.org/

Table 1: Number of queries that returned incomplete, incorrect, incomplete and incorrect or complete and correct result sets, or threw an error

Property path expression	Comunica					SaGe-Jena					SaGe-AC					
	Incompl. & Correct	Complete & Incor.	Incompl. & Incor.	**Complete & Correct**	Error	Incompl. & Correct	Complete & Incor.	Incompl. & Incor.	**Complete & Correct**	Error	Incompl. & Correct	Complete & Incor.	Incompl. & Incor.	**Complete & Correct**	Error	Total
Inverse	0	0	0	20	0	0	0	0	20	0	0	0	0	20	0	20
Sequence	0	0	0	24	0	0	0	0	24	0	0	0	0	24	0	24
Alternative	0	0	0	23	0	0	0	0	23	0	0	0	0	23	0	23
Existential	0	2	0	19	3	0	0	0	24	0	0	0	0	24	0	24
Transitive Reflexive-Closure	11	0	0	10	22	0	0	0	43	0	0	0	0	43	0	43
Reflexive-Closure	11	0	0	14	10	0	0	0	35	0	0	0	0	35	0	35
Negated Property Set	0	0	2	19	0	0	0	0	21	0	0	0	0	21	0	21
Inverse Negated Property Set	0	0	2	19	0	0	0	0	21	0	0	0	0	21	0	21
Negated and Inverse Property Set	0	0	5	26	0	0	0	0	31	0	0	0	0	31	0	31
Total	22	2	9	**174**	35	0	0	0	**242**	0	0	0	0	**242**	0	**242**

time between starting query execution and the production of the final results by the client.

5.2 Experimental results

Presented results correspond to the average obtained of three successive execution of the queries workloads. We fixed a time out of 30 minutes.

Does compressed automata approach follow the W3C semantics? We run the BeSEPPI benchmark with SaGe-AC, Comunica and SaGe-Jena clients. Table 1 presents the results for the different approaches.

SaGe-AC follows the W3C semantics of SPARQL property paths, it returns complete and correct results for the 242 queries. SaGe-Jena is just as compliant as Jena, i.e. it follows the semantics. However, Comunica is unable to compute the transitive path expressions when paths have longer more than one, or reflexive closure must be computed.

Does compressed automata approach outperform mono-predicate automata? We run the 30 queries of gMark workload with the SaGe-A and SaGe-AC approaches. Figure 5 shows the execution time, the number of HTTP calls and the data transfer for each query in the workload for both approaches. Dashed lines represent incomplete queries after an execution time of 30 minutes. As expected, when it is possible to improve the decomposition of property path

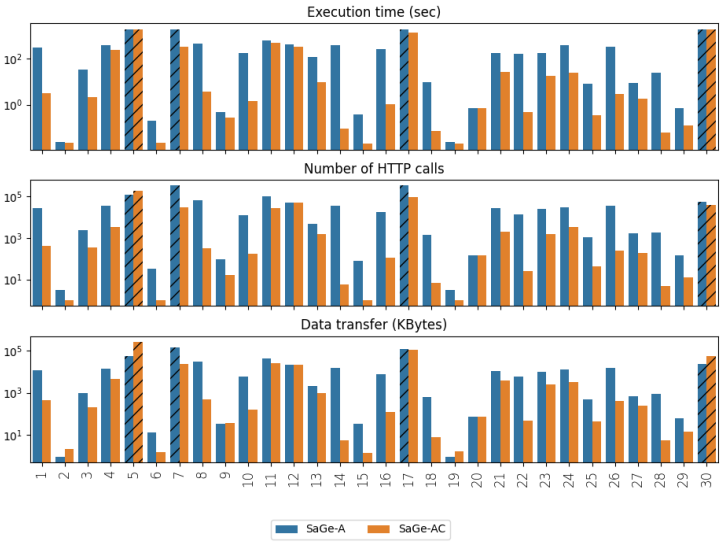


Fig. 5: gMark queries using SaGe-A and SaGe-AC (logarithmic scale).

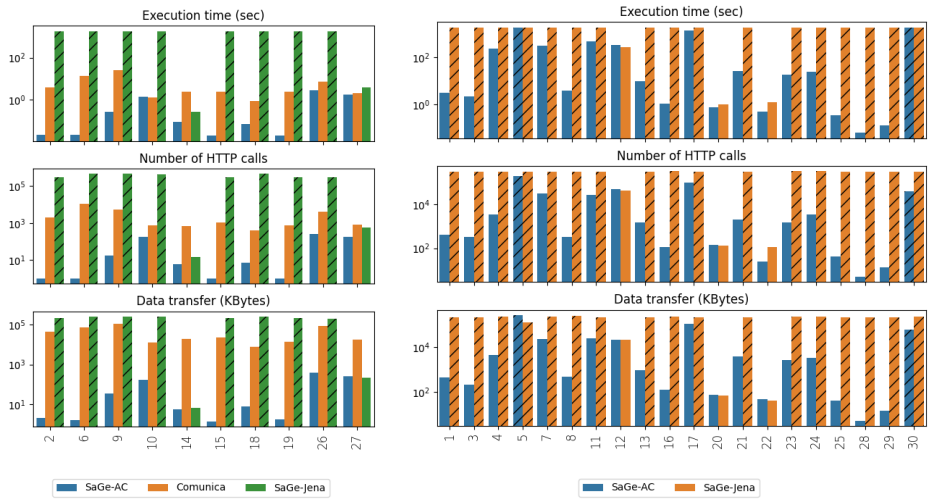
queries, SaGe-AC outperforms SaGe-A in terms of HTTP calls, data transfer and execution time. However, when mono-predicate automata cannot be compressed, then both approaches are equivalent. Queries 12 and 20 are examples of property path queries for which mono-predicate automata and multi-predicate automat have similar performance.

Does compressed automata approach existing client-side approaches We run the 30 queries of gMark workload with the SaGe-AC, communica and SaGe-Jena clients. Figure 6 shows the execution time, the number of HTTP calls and the data transfer for each query in the workload for three approaches.

As Comunica does not support gMark transitive queries, we split the query workloads into two groups: transitive queries and non-transitive queries. The transitive queries regroupes queries that have at least one transitive closure expression. The non-transitive queries regroupes other queries.

Figure 6b presents the results of transitive queries with only SaGe-AC and SaGe-Jena, as these queries cannot be executed by Communica. As we can see, SaGe-AC outperforms SaGe-Jena in terms of execution time, number of HTTP calls and data transfer for all queries. Moreover, SaGe-AC provides complete result for all queries (20 queries) except the query Q30, while SaGe-Jena provides complete results for only three queries (Q12, Q20 and Q22).

Figure 6a presents results for the non-transitive queries with SaGe-AC, SaGe-Jena and Comunica. SaGe-AC outperforms SaGe-Jena and Comunica in term of the execution time, the number of HTTP calls and the data transfer. These results demonstrate empirically the performance of automata compression approach compared to Communica and SaGe-Jena decomposition of property path



(a) Queries without transitive closure expressions

(b) Queries with at least one transitive closure expression

Fig. 6: gMark queries using SaGe-A, SaGe-AC, SaGe-Jena (logarithmic scale).

queries. These results demonstrate also the advantage of using the web preemption instead of TPF. The web preemption ensures queries completeness, as TPF, while providing a more expressive interface. Therefore, operations that could be costly to compute on the client-side are supported directly by the server-side. Consequently, communication costs are significantly decreased and only final results are transferred to the client. Obviously, the more operators supported by the server, the better the performance will be.

Of course, it is possible to optimize queries evaluation with TPF [23]. Using a better join ordering could also improve queries performance [22]. This explains why Comunica offers better performance than SaGe-Jena on the non-transitive queries. However, even if these optimizations could decrease the number of HTTP calls sent to the server and execution times, they do not change the data transfer for client-side operators.

6 Conclusion

In this paper, we proposed an algorithm to decompose property path queries into BGPs subqueries. As BGP subqueries are guaranteed to terminate under the web preemption model, this ensures that property path queries are processed online and return complete answers. Compared to the state of art, decomposing into BGPs subqueries is much more efficient than decomposing into Triple Pattern queries. It finally achieves better execution time. We modeled property path expressions as an automaton, and we demonstrated that generating BGPs subqueries instead of triple pattern subqueries can be seen as compressing of

the automaton. We implemented our approach in smart client for a preemptable SPARQL server. We demonstrated that our approach outperforms existing approach in term of generated subqueries, data transfer and execution time while supporting full SPARQL 1.1 property path expressions.

The current approach has several limitations. First, in case of simple transitive closure such as `?x sameas* ?y`, then there is no room for BGP optimisation. Second, when property path expressions are included inside a BGP, i.e. `?x rdf:type Person . ?x sameas* ?y`, then joins have to processed in the smart client, generating high data transfer. Such limitations are the consequences of the lack of support for property path expressions on restricted SPARQL servers. Improving performances for property path expressions requires to find a way to process fairly property path expressions on server-side. This is clearly challenging as property paths may explore a large part of the knowledge graph while remembering visited nodes.

References

1. Adrian Skubella, Daniel Janke, S.S.: Beseppi: Semantic-based benchmarking of property path implementations. European Semantic Web Conference (2019)
2. Aranda, C.B., Polleres, A., Umbrich, J.: Strategies for executing federated queries in SPARQL1.1. In: 13th International Semantic Web Conference, (2014)
3. Arenas, M., Conca, S., Pérez, J.: Counting beyond a yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In: 21st World Wide Web Conference 2012, WWW. pp. 629–638. ACM (2012)
4. Bagan, G., Bonifati, A., Ciucanu, R., Fletcher, G.H., Lemay, A., Advokaat, N.: gmark: Schema-driven generation of graphs and queries. IEEE Transactions on Knowledge and Data Engineering **29**(4), 856–869 (2016)
5. Baier, J., Daroch, D., Reutter, J.L., Vrgoč, D.: Evaluating navigational rdf queries over the web. In: Proceedings of the 28th ACM Conference on Hypertext and Social Media. pp. 165–174 (2017)
6. Bonifati, A., Fletcher, G.H.L., Voigt, H., Yakovets, N.: Querying Graphs. Synthesis Lectures on Data Management, Morgan & Claypool Publishers (2018)
7. Bonifati, A., Martens, W., Timm, T.: Navigating the maze of wikidata query logs. In: The World Wide Web Conference. pp. 127–138 (2019)
8. Brüggemann-Klein, A.: Regular expressions into finite automata. Theoretical Computer Science **120**(2), 197–213 (1993)
9. Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P.Y.: Sparql web-querying infrastructure: Ready for action? In: International Semantic Web Conference. pp. 277–293. Springer (2013)
10. Hartig, O., Letter, I., Pérez, J.: A formal framework for comparing linked data fragments. In: 16th International Semantic Web Conference, ISWC. Lecture Notes in Computer Science, vol. 10587, pp. 364–382. Springer (2017)
11. Hasnain, A., Mehmood, Q., e Zainab ang Aidan Hogan, S.S.: SPORTAL: profiling the content of public SPARQL endpoints. Int. J. Semantic Web Inf. Syst. **12**(3), 134–163 (2016)
12. Jachiet, L., Genevès, P., Gesbert, N., Layaïda, N.: On the optimization of recursive relational queries: Application to graph queries. In: SIGMOD 2020-ACM International Conference on Management of Data. pp. 1–23 (2020)

13. Kameda, T., Weiner, P.: On the state minimization of nondeterministic finite automata. IEEE Transactions on Computers **100**(7), 617–627 (1970)
14. Kostylev, E.V., Reutter, J.L., Romero, M., Vrgoč, D.: Sparql with property paths. In: International Semantic Web Conference. pp. 3–18. Springer (2015)
15. Losemann, K., Martens, W.: The complexity of evaluating path expressions in sparql. In: Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems. pp. 101–112 (2012)
16. Minier, T., Skaf-Molli, H., Molli, P.: Sage: Web preemption for public SPARQL query services. In: The World Wide Web Conference, The WebConf 2019. pp. 1268–1278 (2019)
17. Pérez, J., Arenas, M., Gutiérrez, C.: Semantics and complexity of SPARQL. ACM Transations on Database Systems **34**(3), 16:1–16:45 (2009)
18. Polleres, A., Kamdar, M.R., Fernández, J.D., Tudorache, T., Musen, M.A.: A more decentralized vision for linked data. In: 2nd Workshop on Decentralizing the Semantic Web (DeSemWeb 2018) co-located with ISWC 2018 (2018)
19. Reutter, J.L., Soto, A., Vrgoč, D.: Recursion in sparql. In: International Semantic Web Conference. pp. 19–35. Springer (2015)
20. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL query optimization. In: Database Theory - ICDT 2010. pp. 4–33 (2010)
21. Steve, H., Andy, S.: SPARQL 1.1 query language. In: Recommendation W3C (2013)
22. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: Sparql basic graph pattern optimization using selectivity estimation. In: Proceedings of the 17th international conference on World Wide Web. pp. 595–604 (2008)
23. Van Herwegen, J., Verborgh, R., Mannens, E., Van de Walle, R.: Query execution optimization for clients of triple pattern fragments. In: European Semantic Web Conference. pp. 302–318. Springer (2015)
24. Verborgh, R., Sande, M.V., Hartig, O., Herwegen, J.V., Vocht, L.D., Meester, B.D., Haesendonck, G., Colpaert, P.: Triple pattern fragments: A low-cost knowledge graph interface for the web. J. Web Sem. **37-38**, 184–206 (2016)
25. Yakovets, N., Godfrey, P., Gryz, J.: Waveguide: Evaluating SPARQL property path queries. In: 18th International Conference on Extending Database Technology, EDBT (2015)
26. Yamagaki, N., Sidhu, R., Kamiya, S.: High-speed regular expression matching engine using multi-character nfa. In: 2008 International Conference on Field Programmable Logic and Applications. pp. 131–136. IEEE (2008)