Recursive SPARQL for Graph Analytics

Anonymous Author(s)

ABSTRACT

Work on knowledge graphs and graph-based data management often focus either on declarative graph query languages or on frameworks for graph analytics, where there has been little work in trying to combine both approaches. However, many real-world tasks conceptually involve combinations of these approaches: a graph query can be used to select the appropriate data, which is then enriched with analytics, and then possibly filtered or combined again with other data by means of a query language. In this paper we propose a declarative language that is well suited to perform graph querying and analytical tasks. We do this by proposing a minimalistic extension of SPARQL to allow for expressing analytical tasks; in particular, we propose to extend SPARQL with recursive features, and provide a formal syntax and semantics for our language. We show that this language can express key analytical tasks on graphs (in fact, it is Turing complete), offering a more declarative alternative to existing frameworks and languages. We show how procedures in our language can be implemented over an off-the-shelf SPARQL engine with a specialised client that allows parallelisation and batch-based processing when memory is limited. Results show that with such an implementation, procedures for popular analytics currently run in seconds or minutes for selective sub-graphs (our target use-case) but struggle at larger scales.

KEYWORDS

SPARQL, graph queries, graph analytics, recursion

ACM Reference Format:

Anonymous Author(s). 2019. Recursive SPARQL for Graph Analytics. In *Proceedings of The Web Conference '20 (The Web Conference 2020).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Recent years have seen a surge in interest in graph data management, learning and analytics spanning various academic communities. Much of this work has been conducted under the title of "knowledge graphs" [33], centred on the composition and extraction of knowledge from graph-structured data at large-scale, drawing together techniques from communities such as Databases, Graph Theory, Machine Learning, the Semantic Web, and more besides [4]. A variety of major commercial websites are now using proprietary knowledge graphs to support various applications [6, 15, 19, 26, 32, 33]. Non-proprietary knowledge graphs like Wikidata [10] – published on the Web using Semantic Web standards – have been widely adopted for numerous applications. Wikidata's SPARQL query service now receives millions of queries per day [23].

However, while works on knowledge graphs are currently being pursued by various communities, more work is needed to combine complementary techniques from different areas [4]. As a prominent example, while a variety of query languages have been proposed for graphs [2, 3, 11, 14, 28], and a variety of frameworks have been proposed for graph analytics [22, 34, 37], there are few works that aim to combine both querying and analytics for graphs: while some analytical frameworks support lightweight query features [28, 37], and some query languages support lightweight analytical features [11, 14], these solutions are limited to specific types of queries, or specific analytics, or require imperative "glue" code. We argue that a more general declarative alternative is needed.

Take, for example, the following seemingly simple task, which we wish to apply over Wikidata: *find stations from which one can still reach Palermo metro station in Buenos Aires if Line C is closed.* Although standard graph query languages – such as SPARQL [14], Cypher [11], G-CORE [2], etc. – support path expressions that capture reachability, they cannot express conditions on the nodes through which such paths pass, as is required by this task (i.e., that they are not on Line C). Consider a more complex example that again, in principle, can be answered over Wikidata: *find the top author of scientific articles about the Zika virus according to their p-index within the topic.* The *p*-index of authors is calculated by computing the PageRank of papers in the citation network, and then summing the scores of the papers for each respective author [30]. One way this could currently be achieved is to: (1) perform a SPARQL query to extract the citation graph of scientific articles about the Zika virus; (2) load the graph into an external tool to compute PageRank scores; (3) perform another query to extract the (bipartite) authorship graph for the articles; (4) load the authorship graph into the external tool to join authors with papers, aggregate the *p*-index score per author, sort by score, and output the top result. Here the user must ship data back and forth between different tools to solve the task. Another strategy might be to load the Wikidata dump into a graph-analytics framework, writing code to extract the required graphs, analyse them, and aggregate the results; in this case, we lose the convenience of a declarative query language and database optimisations for extracting the relevant data, performing joins and aggregations, etc., as the task requires.

In this paper, we instead propose a general, (mostly) declarative language that supports *graph queralytics*: tasks that combine querying and analytics on graphs, allowing to interleave both arbitrarily. We coin the term "*queralytics*" to highlight that these tasks raise new challenges and are not well-supported by existing languages and tools that focus only on querying or analytics. Rather than extending a graph query language with support for specific, built-in analytics, we rather propose to extend a graph query language to be able to express any form of (computable) analytical task of interest to the user: namely we add recursion to the query language. Specifically, we explore the addition of recursive features to the SPARQL

The Web Conference 2020, April 20–24, 2020, Taipei

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

query language, proposing a concrete syntax and semantics for our language, showing examples of how it can combine querying and analytics for graphs. We call our language the *SPARQL Protocol and RDF Query & Analtyics Language* (*SPARQAL*). We study the expressive power of SPARQAL with similar proposals found in the literature [8, 27, 35]. We then discuss the implementation of our language on top of a SPARQL query engine, introducing evaluation strategies that aim to find trade-offs between scalability and performance. We present experiments to compare our proposed strategies on real-world datasets, for which we devise a set of benchmark queralytics over Wikidata. Our results provide insights into the scale and performance with which an existing SPARQL engine can perform standard graph analytics, showing that for queralytics wherein a selective sub-graph is extracted for analysis, interactive performance is feasible; on the other hand, the current implementation struggles for an analytical benchmark on a larger-scale graph.

Example 1.1. To illustrate our proposal, we provide a queralytic in our language for the first example seen in the introduction (the Zika/*p*-index task will be seen later). Namely, suppose that there is a concert close to Palermo metro station in Buenos Aires; however, Line C of the metro is closed due to a strike. We would like to know from which metro stations we can still reach Palermo. We can express this queralytic in our SPARQL-based language as follows:

```
1  LET reachable = (
2    SELECT ?s WHERE {
3      wd:Q3296629 wdt:P197 ?s .
4      MINUS { ?s wdt:P81 wd:Q1157050 }
5    }
6  );
7  DO (
8    LET adjacent = (
9      SELECT (?adj AS ?s) WHERE {
10       ?s wdt:P197 ?adj .
11       MINUS { ?adj wdt:P81 wd:Q1157050 }
12       QVALUES(reachable)
13     }
14   );
15   LET reachable = (
16     SELECT DISTINCT ?s WHERE {
17       { QVALUES(adjacent) }
18       UNION
19       { QVALUES(reachable) }
20     }
21   );
22 ) WHILE( FIXPOINT(reachable) );
23 RETURN(reachable);
```

Here we work with the Wikidata dataset, where two adjacent stations are given by the property `wdt:P197` and the metro line by `wdt:P81`; the entities `wd:Q3296629` and `wd:Q1157050` refer to Palermo metro station and Line C, respectively. From lines 1 to 6, we first define a *solution variable* called `reachable` whose value is the result of computing all stations directly adjacent to Palermo that are not on Line C. From lines 7 to 22 we have a loop that executes two instructions: the first, starting at line 8, computes all stations directly adjacent to the current reachable stations not on Line C; the second, starting at line 15, adds the new adjacent stations to the list of known reachable stations with a union. The loop is finished when the set of solutions assigned to the variable `reachable` does not change from one iteration to another (a fixpoint is thus reached). Finally, on line 23, we return the reachable stations. □

2 RELATED WORK

In terms of related works, we first discuss frameworks and languages for applying graph analytics. We then discuss prior proposals for combining graph querying and graph analytics. We then introduce works on extending graph query languages with recursion. We end by highlighting the novelty of this work.

Frameworks for Graph Analytics. Given the growing need to perform graph analytics at large-scale – involving the Web, social networks, etc. – various frameworks have been proposed for such settings, including GraphStep [9], Pregel [22], HipG [18], PowerGraph [13], GraphX [37], Giraph [7], Signal/Collect [34], and more besides. All such frameworks operate on a computational model – sometimes called the systolic model [21], Gather/Apply/Scatter (GAS) model [13], graph-parallel framework [37], etc. – that involves each node in a graph recursively computing its state based on data available for its neighbouring nodes according to a given function. Although such frameworks allow for large-scale graph analytics to be applied in a distributed setting, implementing queries on such frameworks, selecting custom sub-graphs to be analysed, etc., is not straightforward. Similar computational models are used in the case of graph neural networks [29, 36], which have been shown to be as discriminative as the (incomplete) Weisfeiler–Lehman (WL) graph isomorphism test [39]: in other words, by basing computation only on local information in each node's neighbourhood, there are certain pairs of non-isomorphic graphs that will return "isomorphic results" for any algorithm implemented in the framework.

Graph Queries and Analytics. Our work aims to combine graph queries and analytics, focusing on RDF graphs. One such proposal along these lines is Trinity.RDF [41], which stores RDF in a native graph format where nodes store inward and outward adjacency lists, allowing to traverse from a node to its neighbours without the need for index lookup; the system is then implemented in a distributed in-memory index, with query processing and optimisation components provided for basic graph patterns. Although the authors discuss how Trinity.RDF's storage scheme can also be useful for graph algorithms based on random walks, reachability, etc., experiments focus on SPARQL query evaluation from standard benchmarks [41]. Later work used the same infrastructure in a system called Trinity [31] to implement and perform experiments with respect to PageRank and Breadth-First Search, this time rather focusing on graph analytics without performing queries. Though such an infrastructure could be adapted to apply graph queralytics at scale, the authors do not discuss the combination of queries and analytics, nor do they propose languages along these lines.

Most modern graph query languages directly support some built-in analytical features. SPARQL 1.1 [14] introduced *property paths* [17] that allow for specifying regular expressions on paths; these can then be used in the context of a SPARQL query to find pairs of nodes connected by some path matching the regular expression. The Cypher query language for property graphs [11] (used by the Neo4j graph database [24]) also allows for querying on paths; though limited in terms of the regular expressions it allows on paths when compared to SPARQL 1.1, it offers features that SPARQL 1.1 does not, including shortest paths, returning paths, etc. The G-CORE query language [2] also supports features relating to paths,

allowing to store and label paths, find weighted shortest paths, and more besides. In general, however, graph query languages tend to only support analytics relating to path finding and reachability [3].

The Gremlin language [28] is more imperative in style than the aforementioned query languages, allowing to express analytical tasks through graph traversals. Per the Trinity.RDF system [41], graph traversals, when combined with variables, can be used to express and evaluate, for example, basic graph patterns [2]. Gremlin [28] also supports some declarative query operators, such as union, projection, negation, path expressions, and so forth, along with recursion, which allows to capture general analytical tasks; in fact, the Gremlin language is Turing complete [28].

In the context of SQL, languages such as Shark [38] have been proposed that allow SQL queries to be embedded and executed in the context of distributed frameworks (in this case Spark [40]) within which analytics can also be imperatively coded. Aside from embedding SQL into imperative languages, a number of languages have recently been proposed to combine relational algebra with linear algebra – including LARA [16] and MATLANG [5] – based on the observation that although relational algebra is often used for declarative querying, and linear algebra for learning and analytics, many operations in relational algebra can be simulated with linear algebra, and vice-versa, where it is thus of interest to understand the expressive power of both and how they complement each other [12].

Recursive Graph Queries. Previous works have looked at adding recursive features to graph query languages. As aforementioned, most query languages support recursively matching path expressions in a graph; however, per Example 1.1, more powerful forms of recursion are needed in the context of graph query languages to support the general class of analytics that we target here.[1]

A number of authors have proposed more general recursion for graph query languages. Reutter et al. [27] propose to extend SPARQL with recursion based on `CONSTRUCT` queries; in particular, noting that `CONSTRUCT` transforms one RDF graph to another, they propose a syntax for recursively applying a `CONSTRUCT` template to the input graph up to a fixpoint, where a query can then be executed on the resulting fixpoint graph; they further propose a *linear recursive* fragment of their language, which assumes that in each iteration only the data from the original graph and the previous iteration are required, reducing the complexity of evaluation. In later work, Corby et al. [8] proposed the LDScript language, which supports the definition of functions using SPARQL expressions; local variables that can store individual values, lists or the results of queries; and iteration over lists of values using loops, as well as recursive function calls. Recently Urzua and Gutierrez [35] proposed an extension of the G-CORE language to support linear recursion, and show how the resulting language can be used in principle to express various graph algorithms, such as a topological sort, which cannot be expressed in G-CORE without recursion.

Novelty. Unlike graph analytics frameworks, we propose a language for combining queries and analytics on graphs. Compared with Gremlin, our language is more declarative, based on an extension of an existing query language (SPARQL) to allow for expressing and combining graph analytics and queries. The closest proposals to ours are those that extend graph query languages with recursive features [8, 27, 35]. In comparison with the proposal of Reutter et al. [27] and Urzua and Gutierrez [35], we allow recursion over `SELECT` queries, which adds flexibility by not requiring to maintain intermediate results as (RDF) graphs: for example, with `SELECT` we can maintain a table of four columns/variables representing a weighted RDF graph, where the first three columns denote an RDF graph and the fourth column denotes weights on individual triples; in the case of `CONSTRUCT`, we would rather require some form of reification to capture weighted triples. Furthermore, while we support fixpoint recursion, we also support other forms of recursion; in particular, we allow for terminating a loop based on a boolean condition (an `ASK` query), which offers greater flexibility for defining termination conditions in cases where, for example, an analytics task is infinitary and/or requires approximation in practice (e.g., PageRank). In comparison with LDScript [8] – which also supports recursion on `SELECT` queries – our focus is rather on supporting graph analytics with such a language, supporting features, such as fixpoint, that are useful in this setting.

[1] Though more complex forms of "navigational patterns" have been proposed in the literature, they are mostly limited to path-finding and reachability [3].

3 LANGUAGE

Recursion stands out in the literature as a key feature for supporting graph analytics. Our proposal – called SPARQAL – extends SPARQL (1.1) with recursion by allowing to iteratively evaluate queries (optionally) joined with solution sequences of prior queries until some condition is met. In order to support this form of iteration, we need two key operators. First, we extend SPARQL with *solution variables* to which the results of a SELECT query can be assigned, and which can then be used within other queries to join solutions. Second, we extend SPARQL with `do-while` *loops* to support iteratively repeating a sequence of SPARQL queries until some termination condition is met; this condition may satisfy a fixed number of iterations, a boolean ASK query, or a fixpoint on a solution variable (terminating when the set of solutions do not change).

We refer back to Example 1.1, which illustrates how our language can be used to address a relatively simple queralytic task. We now present the syntax of our language, and thereafter proceed to define the formal semantics. We finish the section with a second, more involved example for computing the *p*-index of authors in an area.

Preliminaries: To formally define our language and give our examples we assume familiarity with SPARQL and basic notions of graph analytics algorithms. We use the standard syntax and semantics of SPARQL in terms of mappings [14]. We recall the notion of a *solution sequence*, which is the result of a SPARQL query evaluated on a graph (or dataset), listing the ways in which the query matches the data. There may be zero, one or multiple solutions to a query.

3.1 Syntax

SPARQAL aims to be a minimalistic extension of the SPARQL language that allows to express queralytic tasks. Specifically, a task is defined as a *procedure*, which is a sequence of *statements*. A statement can be an *assignment*, *loop* or *return* statement, as follows.

Assignment: Assigns the solution sequence of a query to a solution variable. The syntax of an assignment statement is:

```
LET var = (Q);
```

where `var` is a variable name and `Q` is a SPARQL query that may use constructs of the form `QVALUES(var')` as subqueries, where `var'` names a solution variable.

Loop: Executes a sequence of statements until a termination condition holds. The syntax of a loop statement is:

```
DO (S) WHILE (condition);
```

where `S` is a sequence of statements and `condition` is one of the following three forms of termination condition:

- `TIMES t`, where `t` is an integer greater than 0.
- `FIXPOINT (var)`, where `var` is a solution variable.
- `AQ`, where `AQ` is an ASK query that may use constructs of the form `QVALUES(var)` as subqueries.

Return: Specifies the solution sequence to be returned by the procedure. The syntax of a return statement is:

```
RETURN (var);
```

where `var` is a solution variable.

Finally, a SPARQAL *procedure* is a sequence of statements satisfying the following two conditions:

- the last statement is a return statement and no other (nested) statement is a return statement;
- all solution variables used in `QVALUES`, `FIXPOINT` and `RETURN` have been assigned by `LET` in a previous statement (or a nested statement thereof).

Example 3.1. Example 1.1 illustrates a SPARQAL procedure with three statements, one of which contains two additional nested statements. The first statement is an assignment statement that goes from line 1 to 6. The second statement is a loop statement that goes from line 7 to 22; this statement has a `FIXPOINT` ending condition, and it contains a sequence of two nested assignment statements: the first goes from line 8 to 14 while the second goes from line 15 to 21. The last statement, on line 23, is a return statement. □

3.2 Semantics

We now give the semantics of statements that form procedures in SPARQAL. More formally, let $P = s_1; \ldots; s_n$ be a sequence of statements, and let `var_1`, ..., `var_k` be all variables mentioned in any statement in P (including in nested statements). For a tuple $\text{val}_0 = (r_1, \ldots, r_k)$ of initial assignments of (possibly empty) solution sequences to variables `var_1`, ..., `var_k`, we will construct a sequence $\text{val}_0, \ldots, \text{val}_n$ of k-tuples, where each val_i represents the value of all variables after executing statement s_i. (Note that for brevity, in what follows, we assume the SPARQL dataset upon which queries are evaluated to be fixed.)

The construction is done inductively. Assume that $\text{val}_{i-1} = (r_1, \ldots, r_k)$. The value of val_i depends on the nature of s_i. First, if s_i is the assignment statement:

```
LET var_j = (Q);
```

then tuple val_i is constructed as follows. Define SPARQL query $Q[(\texttt{var\_1}, \ldots, \texttt{var\_k}) \mapsto (r_1, \ldots, r_k)]$ as the result of substituting each subquery `{QVALUES(var_i)}` in Q for the solution sequence r_i[2], and let r^* be the result of evaluating this extended query over the database. Tuple val_i is then defined as

$$\text{val}_i = (r_1, \ldots, r_{i-1}, r^*, r_{i+1}, r_k),$$

that is, the result of substituting r_i for r^* in the tuple val_{i-1}.

Next, if s_i is the return statement

```
RETURN(var_j)
```

Then the program terminates and returns the solution sequence r_j that is the j-th component of val_i.

Finally, if s_i is the loop statement

```
DO (S) WHILE (condition);
```

The tuple val_i is constructed as follows. Assume that S is the sequence $s'_1, \ldots, s'_\ell$ and notice that (by definition) S must use a subset of the k solution variables in P. Repeat the following steps until the terminating condition is met:

(1) Initialize $\text{val}'_0 := \text{val}_{i-1}$.
(2) Compute the tuple val'_ℓ that represents the result of executing statements $s'_1, \ldots, s'_\ell$.
(3) If val'_ℓ does not satisfy the condition, set $\text{val}'_0 := \text{val}'_\ell$ and repeat step 2 above.
(4) Otherwise finish, and set $\text{val}_i := \text{val}'_\ell$.

To define when a tuple val'_ℓ over k variables satisfies a condition, we cover all three cases:

- If the condition is `TIMES t`, then the condition is met once the loop above has repeated t times.
- If the condition is `FIXPOINT (var_j)`, then the condition is met when the j-th component of val'_ℓ contains the same set of solutions as the j-th component of val'_0.
- If the condition is `AQ`, then the condition is met when the ASK query $AQ[(\texttt{var\_1}, \ldots, \texttt{var\_k}) \mapsto \text{val}'_\ell]$ evaluates to true.

Note that we assume all variables to have a global scope as it makes the semantics simpler to define; one could define the semantics for variables with local scope in a similar way.

Example 3.2. We recall again Example 1.1, this time to illustrate the semantics of SPARQAL. In the first `LET` statement, we assign the solution sequence of the given SPARQL query to the variable `reachable`. Then the procedure enters a loop. We assign `adjacent` to the results of a SPARQL query that embeds the current solutions of `reachable` as a sub-query, leading to a join between current `reachable` stations and pairs of adjacent stations not on Line C. We then update the `reachable` solutions, adding `adjacent` solutions; here we can use `reachable` in the `LET` and `QVALUES` of the same statement since it was assigned previously (line 1). In each iteration the solutions for `reachable` will increase, discovering new stations adjacent to previous ones, until a fixpoint. Finally, the `RETURN` clause specifies the solutions to be given as a result of the procedure. □

3.3 Example with PageRank

We now illustrate a procedure for a more complex queralytic.

Example 3.3. Suppose that we have the citation network of a group of articles on a topic of interest. After obtaining such network, we want to compute a centrality algorithm in order to know which articles of the network are the most important. Thereafter we wish to use these scores to find the most prominent authors in the area. We can express this task using SPARQAL. In this case we will

[2] A syntactic way of doing this is to use a `VALUES` command in SPARQL.

```
LET zika = (              # directed graph of citations between Zika articles
  SELECT ?node ?cite WHERE {
    ?node wdt:P31 wd:Q13442814 ; wdt:P921 wd:Q202864 ; wdt:P2860 ?cite .
    ?cite wdt:P31 wd:Q13442814 ; wdt:P921 wd:Q202864 .
  }
);
LET nodes = (             # all nodes of Zika graph
  SELECT DISTINCT ?node WHERE {
    { QVALUES(zika) } UNION { SELECT (?cite AS ?node) WHERE { QVALUES(zika) } }
  }
);
LET n = (                 # number of nodes in Zika graph
  SELECT (COUNT(*) AS ?n) WHERE { QVALUES(nodes) }
);
LET degree = (            # out-degree (>1) of nodes in Zika graph
  SELECT ?node (COUNT(?cite) AS ?degree) WHERE { QVALUES(zika) } GROUP BY ?node
);
LET rank = (              # initial rank
  SELECT ?node (1.0/?n AS ?rank) WHERE { QVALUES(nodes) . QVALUES(n) }
);
DO (                      # begin 10 iterations of PageRank
  LET rank_edge = (         # spread rank to neighbours via edges
    SELECT (?cite AS ?node) (SUM(?rank*0.85/?degree) AS ?rankEdge) WHERE {
      QVALUES(degree) . QVALUES(rank) . QVALUES(zika)
    } GROUP BY ?cite
  );
  LET unshared = (          # compute total rank not shared via edges
    SELECT (1-SUM(?rankEdge) AS ?unshared) WHERE { QVALUES(rank_edge) }
  );
  LET rank = (              # split and add unshared rank to each node
    SELECT ?node (COALESCE(?rankEdge,0)+(?unshared/?n) AS ?rank) WHERE {
      QVALUES(nodes) . QVALUES(n) . QVALUES(unshared) . OPTIONAL { QVALUES(rank_edge) }
    }
  );
) WHILE (TIMES 10);
LET p_index_top = (       # compute p-index for authors, select top author
  SELECT ?author (SUM(?rank) AS ?p_index) WHERE {
    QVALUES(rank) . ?node wdt:P50 ?author .
  } GROUP BY ?author ORDER BY DESC(?p_index) LIMIT 1
);
RETURN(p_index_top);
```

Figure 1: Procedure to compute the top author in terms of p-index for articles about the Zika virus

consider the citation network of all the articles about the Zika virus, where we then run the PageRank algorithm to know which articles are more relevant in the network, using the resulting scores to compute p-indexes for the respective authors. We show a procedure in our language for solving this task in Figure 1.

In this procedure we start by defining a variable that contains a solution sequence with pairs (?node, ?cite) such that both ?node and ?cite are instances of (P31) scientific articles (Q13442814) about (P921) the Zika virus (Q202864) and ?node cites (P2860) ?cite. The solutions for this query are assigned to zika. We can consider this variable as the representation of a directed subgraph extracted from Wikidata. We also define the variables nodes with all nodes in the subgraph, n with the number of nodes, and degree with the out-degree of all nodes in the graph (with some out-edge).

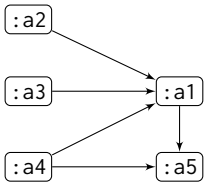
After extracting the graph and preparing some data structures for it, we then start the process of computing PageRank. First we assign the variable rank with initial ranks for all nodes of $\frac{1}{n}$. We then start a loop where we will execute 10 iterations of PageRank.[3] In each iteration we will first compute and assign to rank_edge the PageRank that each node shares with its neighbours; here we assume a damping factor $d = 0.85$ as typical for PageRank [**?**], denoting the ratio of rank that a node shares with its neighbours. Next we compute and assign to unshared the total rank not shared with neighbours in the previous step (this arises from nodes with no out-edges and the $1 - d$ factor not used previously for other nodes). We then conclude the iteration by splitting and adding the unshared rank to each node equally, updating the results for rank. The loop is applied 10 times, computing PageRank for each article.

Finally, we join the PageRank scores for articles with their authors, and use an aggregation to sum the scores for each author, applying ordering and a limit to select the top author, assigning the solution to p_index_top. Finally, the procedure returns the solution for p_index_top denoting the top author. □

[3]We select this termination condition for simplicity; we could also implement, for example, conditions based on residual norm, correlation coefficients, etc.

?node	?cite
:a2	:a1
:a3	:a1
:a4	:a1
:a4	:a5
:a1	:a5

(a) Solutions



(b) Graph

Figure 2: Example results for `zika` solution variable

?node	?cite
:a2	:a1
:a3	:a1
:a4	:a1

(a) `zika_a1`

?node	?cite
:a2	1
:a3	1
:a4	2

(b) `degree_a1`

?node	?rank
:a2	1/5
:a3	1/5
:a4	1/5

(c) `rank_a1`

Figure 3: Intermediate solution sequences for `:a1` batch

4 EVALUATION IN BATCHES

Procedures in SPARQAL use **QVALUES** clauses to coordinate solution sequences between statements, allowing to pass, extend and refine data throughout the procedure. A natural way to coordination solution sequences across statements is to store them in memory during the execution of the procedure; however, large solution sequences may not fit in memory. To alleviate this issue, we develop an alternative approach to perform the joins instigated by **QVALUES** clauses in batches, using a technique reminiscent of the Map-Reduce paradigm.[4] This approach allows to evaluate queries without assuming that intermediate solution sequences fit in memory and, moreover, allows to parallelise the evaluation of queries.

4.1 Overall Strategy

The strategy for evaluation in batches is as follows. First, each SPARQL query in a (nested) statement of the procedure is associated with `Map` and `Reduce` functions. These functions replace a query Q working with one or more **QVALUES**`(var)` clauses – typically evaluated in full and passed to the query – to a sequence of queries $Q_1, \ldots, Q_n$ in which the instantiations of **QVALUES**`(var)` clauses only retrieves a subset of the tuples in variable `var`. These queries – representing batches – are generated by the `Map` function. The `Reduce` function then merges the results for each $Q_1, \ldots, Q_n$ into a single output. Because these queries are evaluated separately, and over smaller portions of the solution sequence, this approach reduces memory requirements and enables parallel evaluation. The downside is that we now execute a series of queries, instead of one.

Before formally defining the strategy, we provide an example.

Example 4.1. Recall Example 3.3 and the procedure to compute the top author in terms of p-index for Zika articles. Consider the example solution sequence for the variable `zika` shown in Figure 2 alongside the directed graph it represents (in practice, Wikidata returns over 3 thousand articles with over 38 thousand citations).

Next consider the assignment of the variable `rank_edge` on line 22 at the first iteration of the loop. Intuitively, this assignment computes how much PageRank score each article will receive from its citations. Instead of evaluating the query as usual, we will use a `Map` function in order to evaluate it in several batches. More specifically, let $Q_{\texttt{rank\_edge}}$ be the query that assigns the variable `rank_edge`. We associate $Q_{\texttt{rank\_edge}}$ with the following `Map` and `Reduce` functions. Our `Map` function receives two inputs: a SPARQL variable `?v` and a unary `SELECT` SPARQL query that mentions `?v`. This corresponds to the invocation `Map(?cite, [`$Q_{\texttt{node}}$`])` in our notation, where $Q_{\texttt{node}}$ is the following query that assigns to `?node` all articles that cite the article assigned to the query variable `?cite`.

```
SELECT ?node WHERE {
  ?node wdt:P31 wd:Q13442814 ;
    wdt:P921 wd:Q202864 ; wdt:P2860 ?cite .
}
```

This `Map` function will divide the query $Q_{\texttt{rank\_edge}}$ into a series of queries: one for each node of the citation network; this is done by splitting the solution sequence of the variable `zika` into a set of sequences where the binding of variable `?cite` is different. In our case, this corresponds to elements `:a1` and `:a5`. Thus, by splitting variable `zika` into two variables `zika_a1` and `zika_a5` – each of them instantiated with the respective solution sequence – we define two different queries for $Q_{\texttt{rank\_edge}}$: the first invokes **QVALUES**`(zika_a1)` and the second invokes **QVALUES**`(zika_a5)`.

We call these queries $Q_{\texttt{:a1}}$ and $Q_{\texttt{:a5}}$. Intuitively, they are meant to compute the result of $Q_{\texttt{rank\_edge}}$ in two different batches. Let us start with query $Q_{\texttt{:a1}}$. As mentioned, this query excludes all the tuples of the solution sequence stored in the variable `zika` where the value of `?cite` is not `:a1`. This implies that the **QVALUES**`(zika)` clause should be replaced by the solution sequence (batch) shown in Figure 3a labelled `zika_a1`; here, the mappings with the values `(:a4,:a5)` and `(:a1,:a5)` are not considered (they form `zika_a5`).

Now we need to assign solution sequences to the variables `rank` and `degree` corresponding to $Q_{\texttt{:a1}}$ (and later $Q_{\texttt{:a5}}$, respectively). While we could assign the full solution sequences to these variables, this would defeat the purposes of batching and is unnecessary: to compute the PageRank of (e.g.) the node `:a1` we only need information about the neighbours of `:a1`, not the entire graph. Instead, we again split `rank` and `degree`, making use of the query $Q_{\texttt{node}}$ in the definition of `Map`: we define one extra variable `?node`, and we evaluate a copy of query $Q_{\texttt{node}}$ in which the variable `?cite` is replaced by `:a1`, thus effectively storing in `?node` all papers that cite `:a1`. We use these values, and filter out any solution sequence of `degree` that is not binding `?node` to one of these values. In this case the **QVALUES**`(degree)` is replaced by the solution sequence shown in Figure 3b. We do the same for the **QVALUES**`(rank)` clause, which is replaced by the solution sequence shown in Figure 3c.

Now if we evaluate the query $Q_{\texttt{:a1}}$ replacing the **QVALUES** clauses with the respective batches of solution sequences, we would, in turn, obtain the solution sequence `{(:a1 0.425)}`. If we repeat the process for `:a5`, the query $Q_{\texttt{:a5}}$ results in the solution sequence `{(:a5 0.255)}`. Since we need to create a single solution sequence to assign to the variable `rank_next`, we now use a `Reduce` function; in this case, we will simply take the **UNION** of the batched solution sequences. The result is then the same as we would have obtained by evaluating the full solution sequences each time. □

[4] The approach is also similar to "shipping strategies" for federated queries [?].

This strategy of batching solution sequences thus reduces memory requirements. Note that a process like this could be continued for every query extended with **QVALUES** in the procedure of Example 3.3. We now formally define the strategy.

4.2 Formal Definition

The strategy we presented has two steps. The first one is the Map step, where we define how to split solution sequences, and the second is the Reduce step, where we group together the batches we evaluate. To formally define how these operators work, we will assume that we are writing Map and Reduce steps for a query Q that uses clauses **QVALUES**(var_1) , . . . , **QVALUES**(var_k).

Map: The Map operator has the following syntax:

$$\texttt{Map}(?v, [I_1, \ldots, I_m]))$$

where $?v$ is a SPARQL variable and $I_1, \ldots, I_m$ are unary standard SPARQL queries, that is, queries that project only one variable. We assume each query I_j projects the variable $?v_{I_j}$ for $j \in [1, \ldots, m]$.

Let us assume that upon calling query Q, each clause of the form **QVALUES**(var_i) is instantiated with a solution sequence r_i, for $i \in [1, \ldots, k]$, and define the set $QDom(Q, ?v)$ as the union of all values bound to the SPARQL variable $?v$ in any of the sequences $r_1, \ldots, r_n$; that is, if we use $r[?v]$ to denote the set of all elements that are bound to $?v$ in any mapping in r, then

$$QDom(Q, ?v) = r_1[?v] \cup \cdots \cup r_n[?v].$$

The output of the Map function is a set of tuples of solution variables of the form (var_c_1,…,var_c_k), for $c \in QDom(Q, ?v)$, each of which stores a solution sequence $r_{c,i}$ ($i \in [1, \ldots, k]$).

Let us use $I_j[?v \to c]$ to denote the SPARQL query I_j where all appearances of variable $?v$ are replaced with value c. For every value $c \in QDom(Q, ?v)$ and solution sequence r_i, $i \in [1, \ldots, k]$, we define $r_{c,i}$ as the subset of r_i satisfying the following conditions.

- If there is at least one mapping in r_i that binds variable $?v$, then $r_{i,c}$ contains exactly those mappings in r_i that bind $?v$ to value c.
- Otherwise $r_{i,c}$ contains all mappings that bind any of the variables $?v_{I_j}$ to the result of the query $I_j[?v \to c]$, respectively, for $j \in [1, \ldots, m]$
- If r_i does not contain a mapping that binds $?v$ or any of $?v_{I_1}, \ldots, ?v_{I_m}$, then $r_{i,c} = r_i$.

Note that we are defining the Map function in terms of a single variable $?v$, but it is possible to extend our definition to a set of variables $?v_1, \ldots, ?v_n$. In this case we should consider tuples $(c_1, \ldots, c_n) \in QDom(Q, ?v_1) \times \cdots \times QDom(Q, ?v_n)$.

Reduce: The Reduce function specifies how solution sequences are merged together; it can be their union, the sum of all bindings for a variable in their union, their intersection, etc. Each reducer receives one of the tuples (var_c_1,…,var_c_k), for $c \in QDom(Q, ?v)$, each of which stores a solution sequence $r_{c,i}$, $i \in [1, \ldots, k]$. With these variables, it evaluates the query Q_c, which results in replacing every instance of a construct **QVALUES**(var_i), with **QVALUES**(var_c_i), for $i \in [1, \ldots, k]$. Once all queries have been evaluated by each reducer, all intermediate results of queries $\{Q_c \mid c \in QDom(Q, ?v)\}$ are merged together per the Reduce function (in Example 4.1, the Reduce function just computes the union of all sequences).

5 EXPRESSIVE POWER

In this section we review the expressive power of procedures in SPARQAL. Our results come in two flavours: first we focus on what the language can do, showing Turing-completeness and complexity results, and then we turn to the comparison between our language and other related query languages extended with recursion.

5.1 Turing-completeness

Although do-while loops may appear to be just a mild extension to a query language, our first result states that this is actually enough to achieve Turing-completeness. Formally, we say that a query language $\mathcal{L}$ is Turing-complete if for every Turing machine M over an alphabet Σ one can construct a query Q in $\mathcal{L}$ and define a computable function f that takes a word in Σ^* and produces an RDF graph, and such that a word $w \in \Sigma^*$ is accepted by M if and only if the evaluation of Q over graph $f(w)$ produces a non-empty result. Along these lines, we prove the following result:

Theorem 5.1. *SPARQAL is Turing-complete*

The proof of this theorem (presented in the extended version of this paper [1]) relies on the combination of do-while loops and the ability to create new values in the base SPARQL language through **BIND** statements and algebraic functions [14]. Of course, for the proof one must assume that there is no limit on the memory used by the evaluation algorithm; however, the proof reveals a linear correspondence between the memory used by the query and the number of cells visited by the machine M.

Traditional theoretical results have tended to study languages assuming that the creation of new values is not possible, or, if possible, that there is a bound on the number of values that are created. But this is not the case with SPARQAL procedures; for starters, we can iterate and sum to create arbitrarily big numbers. However, for the purpose of comparing SPARQAL procedures against other traditional database languages, we ask, what would be its expressive power if one disallows the creation of new values? In fact, do-while loops have been studied previously in the literature, especially in the context of relational algebra (see e.g. [**?**]). In our context, we ask what happens if we disallow the invention of new values in the procedure: more formally, we say that a procedure P *does not invent new values* if for every graph G and every variable var defined in P, all mappings in any solution sequence associated to var always binds variables to values already present in G. In this case, there is a limit on the maximum number of mappings in the solution sequence of any variable at any point in time during evaluation of the procedure, and this limit depends polynomially on the size of the graph. This implies that the evaluation of this procedure can be performed in PSPACE (in data complexity), and we can also show that this bound is tight. To formally state this result, let P be a SPARQAL procedure. The evaluation problem for P receives a graph as an input, and asks whether the evaluation of P over G is not empty.[5] We can then state the following:

Proposition 5.2. *The evaluation problem for SPARQAL procedures that do not invent new values is PSPACE-complete.*

[5] This corresponds to boolean evaluation. This is without loss of generality because the standard evaluation problem where one considers a tuple of values as an input can be simulated by means of filters.

5.2 Comparison with Similar Languages

We now turn to the comparison between our language and similar proposals in the literature.

Recursive extensions to SPARQL: We base our comparison on the recursive extension proposed by Reutter et al. [27], but these results apply to similar languages, such as the (with) recursive operator in SQL. The first observation is that these languages only define semantics for monotone queries. For example, recursive SPARQL uses constructs of the form:

```
WITH RECURSIVE G AS {Q_CONSTRUCT}
Q_SELECT
```

where G is an IRI used to denote a temporal graph, $Q_{\text{CONSTRUCT}}$ is a CONSTRUCT SPARQL query and Q_{SELECT} is a SELECT SPARQL query. The idea of this form of recursion is that $Q_{\text{CONSTRUCT}}$ defines a query meant to compute G in an iterative fashion (there may also be reference to the graph G inside this same query). In other words, we can view $Q_{\text{CONSTRUCT}}$ as an operator $T_Q(G)$ that – as a single step – takes as input an RDF graph and produces as output an RDF graph. The final output graph then corresponds to the least fixed point of the sequence $T_Q(\emptyset)$, $T_Q(T_Q(\emptyset))$, ... Such a fixed point is only guaranteed when $Q_{\text{CONSTRUCT}}$ is *monotone*: where $G \subseteq G'$ implies that $T_Q(G) \subseteq T_Q(G')$. To guarantee having monotone queries, Reutter et al. [27] impose major syntactic restrictions on the operands available for the $Q_{\text{CONSTRUCT}}$ query, forbidding, for example, the use of **BIND**, **NOT EXISTS**, **MINUS**, as well as **OPTIONAL** patterns that are not *well designed* [25].

So how does our language compare with these recursive variants? The first observation is that all of these queries can actually be expressed as a SPARQAL procedure: a query in the form above can be straightforwardly simulated by the following procedure:

```
DO (
  LET graph = (
    SELECT ?s ?p ?o WHERE P'_CONSTRUCT
  )
) WHILE ( FIXPOINT (graph) )
LET result = Q'_SELECT;
RETURN result;
```

Here $P'_{\text{CONSTRUCT}}$ is the pattern corresponding to the **WHERE** part of $Q_{\text{CONSTRUCT}}$ from the recursive SPARQL query, but where instead of using temporal graph G we retrieve those triples from the subquery **QVALUES**(graph). Query Q'_{SELECT} corresponds to Q_{SELECT} from the recursive SPARQL query, but where again we use **QVALUES**(graph) instead of the temporal graph G.

In the other direction, can recursive SPARQL simulate SPARQAL procedures? This depends on what sort of queries we allow in $Q_{\text{CONSTRUCT}}$. If we take the language as originally defined by Reutter et al., so that queries $Q_{\text{CONSTRUCT}}$ must be monotone, then we know that the evaluation for recursive SPARQL queries is in PTIME [27]. Together with Proposition 5.2, this means that recursive SPARQL cannot simulate SPARQAL procedures unless PTIME = PSPACE, which is widely assumed to be false. We also remark that a similar result was shown for similar extensions to relational algebra: relational algebra equipped with fixed point cannot simulate do-while queries unless PTIME = PSPACE [**?**].

On the other hand, when one allows to use operands such as **BIND** clauses, the operator given by $Q_{\text{CONSTRUCT}}$ becomes non-monotone, and the semantics for this case is not defined. The standard solution for this case is to assign a partial fixed point semantics, which means that a query of the form above would retrieve a graph G which is the fixed point of the sequence $T_Q(\emptyset)$, $T_Q(T_Q(\emptyset))$, ..., if it exists, or an empty graph otherwise (when the operator runs into an infinite loop). In this context, and if we allow full SPARQL 1.1 in $Q_{\text{CONSTRUCT}}$, one can actually show that both languages coincide, because recursive SPARQL becomes Turing-complete as well.

Graph Neural Networks (GNNs): Another framework for graph analytics that has recently received considerable attention is that of GNNs (see e.g. [**?**]). Roughly speaking, the basic architecture for GNNs consists of a sequence of L *layers* that combine the feature vectors $\mathbf{x}_v$ of every node v of the graph with the multiset of feature vectors of its neighbours. Formally, let $\mathcal{N}_G(v)$ contain all neighbours of a node v in G. For each layer one defines sets of aggregation and combination functions $\{\text{AGG}^{(i)}\}_{i=1}^{L}$ and $\{\text{COM}^{(i)}\}_{i=1}^{L}$, and vectors $\mathbf{x}_v^{(i)}$ of graph labels are computed for every node v of a graph G via the following recursive formula, for $i = 1, \ldots, L$:

$$\mathbf{x}_v^{(i)} = \text{COM}^{(i)}\left(\mathbf{x}_v^{(i-1)}, \text{AGG}^{(i)}\left(\{\!\{\mathbf{x}_u^{(i-1)} \mid u \in \mathcal{N}_G(v)\}\!\}\right)\right) \quad (1)$$

where each $\mathbf{x}_v^{(0)}$ is the initial feature vector $\mathbf{x}_v$ of v. GNNs also assume a final classification or readout functions to compute a global vector for the graph, that is applied at the end of the computation.

Thus, in terms of graph analytics, GNNs can be seen as functions that receive a graph as an input, and output either a global value or another graph that has the same nodes and edges, but where the label of nodes (and, in full generality, edges) may have been modified. We remark that this framework is congruous with the systolic abstraction at the heart of various frameworks for graph analytics [7, 9, 13, 18, 22, 34, 37], as discussed previously.

It is thus of interest to compare GNNs to our SPARQAL language; for this, we assume that we deal with RDF graphs in which all nodes are assigned a label via a triple with the property `rdfs:label`. Of course, since SPARQAL procedures are Turing-complete, one can simulate any GNN with such a procedure. What is more interesting to study is to reverse the question: to understand how GNNs relate to the expressive power of restricted forms of SPARQAL.

As previously mentioned, it was recently shown [39**?**] that the power of GNNs in terms of computing vectors of nodes is bounded by, and captures, the Weisfeiler–Lehman (WL) graph isomorphism test [**?**]. The WL test can be understood as a procedure that starts from a labelled graph, and iteratively assigns, for a certain number of *rounds*, a new label to every node in the graph; this is done in such a way that the label of a node in each round has a one-to-one correspondence with its own label and the multiset of labels of its neighbours in the previous round. If the WL test on a given graph G assigns the same label to two nodes a and b of G, then every GNN must also assign the same label to both of these nodes [39**?**]: this is because GNNs can only aggregate local information for nodes.

In what follows we will define a restricted form of procedure in SPARQAL whose expressive power is comparable to that of GNNs, i.e., that it is bounded by, and captures, the WL test. Formally, we define a *local SPARQAL procedure* as a procedure of the form:

```
LET var_1 = (Q1);
  ⋮
LET var_k = (Qk);
LET vector = (
  SELECT ?v ?lab WHERE { ?v rdfs:label ?lab });
DO (
  ⋮
) WHILE ( condition );
RETURN(vector);
```

such that (i) each query $Q_1, \ldots, Q_k$ is a basic graph pattern of the form $\{?v\ p_j\ ?v_j\}$ or $\{?v_j\ p_j\ ?v\}$, for variables $?v, ?v_1, \ldots, ?v_k$ and properties $p_1, \ldots, p_k$; (ii) all statements in the `do-while` loop only use variables $?lab, ?v, ?v_1, \ldots, ?v_k$ in their queries, and no constants (that is, they cannot retrieve any further information from the graph), and (iii) queries in the `DO-WHILE` loop are evaluated in the `Map/Reduce` framework, but where the `Map` function is just `Map`($?v$). The intuition behind this is as follows. Solution variables `var_1`,...,`var_k` are restricted so that all they can store are tuple of values describing parts of the neighbourhood of a node. Then, the iteration can only look at this neighbourhood, and update the label according to this information. We now state our result.

Theorem 5.3. *The power of local SPARQAL procedures is bounded by, and captures, the WL-test; specifically:*

- *When running any local SPARQAL procedure over a graph G, if there are nodes a and b that are assigned the same label by the WL-test, then the returned sequence r for variable* `vector` *must be such that for any two mappings μ_1 and μ_2 in s where $\mu_1[?v] = a$ and $\mu_2[?v] = b$, it holds that $\mu_1[?lab] = \mu_2[?lab]$.*
- *There is a local SPARQAL procedure P that can reproduce the WL test: for every graph G, the output of P over G is the same as the output of the WL test over G.*

Together with the result that GNN are also bounded, and capture, the WL-test [39**?**], we have that local SPARQAL procedures are comparable in term of expressivity to GNNs.

6 EXPERIMENTS

In this section we present our implementation of a queralytics engine based on the SPARQAL language. This implementation was developed on top of the Apache Jena Framework, version 3.10. The core implementation provides the following core functionalities: (1) it parses the SPARQAL procedure into a sequence of statements, which are evaluated according to their semantics by: (2a) maintaining a map where the key is the variable name and the value is the solution sequence; (2b) replacing variables used within a **`QVALUES`** clause with a **`VALUES`** string with the respective solution sequence; (2c) evaluating SPARQL queries, and (2d) in order to handle **`FIXPOINT`** conditions, maintaining the previous solution sequence of the respective variable in-memory to monitor changes. We further implement the Map/Reduce strategy defined in Section 4.

We adopt a query engine for the current implementation as our target use-case is – per the scenarios outlined in Examples 1.1 and 3.3 – to run queralytics (near-)interactively on small-to-medium size graphs that have been projected from a larger graph using a query. We first report results for our two motivating scenarios. We then devise a benchmark based on Wikidata for running popular analytical tasks on selective sub-graphs that are similarly extracted through queries. Finally, though not part of our target use-case, we stress-test our implementation for a graph analytics benchmark at a larger scale, including results for the Map/Reduce framework designed to reduce memory requirements by using batches.

Experiments were tested on a MacBook Pro with a 3.1 GHz Intel I5 processor and 16 GB of RAM. The source code, procedures and datasets used are available online in an anonymous appendix [1].

Table 1: Top-5 authors according to their Zika p-index

?author	?p_index	?name
wd:Q18876341	0.124	George Dick
wd:Q24696365	0.084	Ademola H. Fagbami
wd:Q21165078	0.083	Alexander John Haddow
wd:Q24515005	0.078	Stuart Fordyce Kitchen
wd:Q24727761	0.046	Robert S. Lanciotti

6.1 Wikidata: Motivating Examples

Our first experiment is to anecdotally evaluate the procedures described in Examples 1.1 and 3.3, evaluating the Buenos Aires metro and Zika p-index queralytics. Example 1.1 took just 1.3 seconds to return 16 stations from which Palermo can be reached without using Line C. Example 3.3 – running 10 iterations of PageRank on a graph of 38,738 edges (citations) and 3,057 nodes (articles) – took 53.1 seconds to find the top author (from 2,214 authors) according to their p-index in the citation network; for reference, Table 1 shows the results for the top 5 authors, ordered by their p-index.

6.2 Wikidata: Queralytics Benchmark

To the best of our knowledge, there is no existing benchmark for queralytics along the lines discussed in this paper (and exemplified by the previous motivating examples). This led us to devise a novel benchmark for queralytics on the Wikidata knowledge graph. We took the "truthy" RDF dump of Wikidata as our benchmark graph [23]. Designing the queralytic tasks required collecting and combining two elements: queries that return results corresponding to graphs, and graph algorithms to apply analytics on these graphs.

In terms of the queries returning graphs, we revised the list of use-case queries for the Wikidata Query Service[6]. From this list, we identified the following six queries returning graphs:

- **Q1** A graph of adjacent metro stations in Buenos Aires
- **Q2** A graph of citations for articles about the Zika virus
- **Q3** A graph of characters in the Marvel universe and the groups they belong to
- **Q4** A graph of firearm cartridges and the cartridges they are based on
- **Q5** A graph of horses and their lineage
- **Q6** A graph of drug–disease interactions on infectious diseases

These queries provide a mix of connected graphs, disconnected graphs, bipartite graphs, trees, DAGs, near-DAGs, and so forth. We provide the sizes of these graphs in Table 2, where we see that the smallest graph is indeed the Buenos Aires metro graph, while the largest is the citation graph for Zika articles.

[6] https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples

Table 2: Number of nodes and edges in graphs considered

Id	Nodes	Edges
Q1	93	172
Q2	3,057	38,738
Q3	480	766
Q4	266	211
Q5	7,194	8,719
Q6	627	996

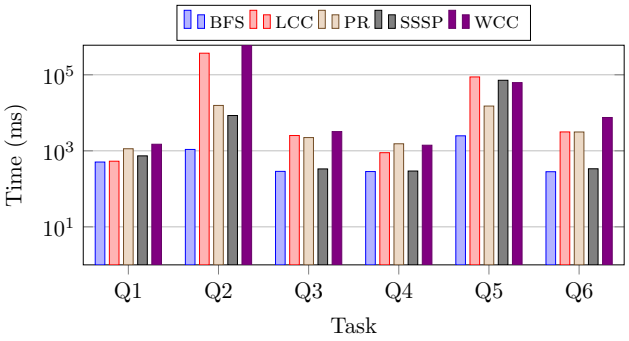


Figure 4: Result for Wikidata queralytic benchmark

Next we must define the analytics that we would like to apply on these graphs. For this, we adopted five of the six algorithm sproposed for the Graphalytics Benchmark [20] defined by the Linked Data Benchmark Council (LDBC); namely:

BFS Breadth-First Search
LCC Local Clustering Coefficient
PR PageRank
SSSP Single-Source Shortest Path
WCC Weakly Connected Components

We do not include the *Community Detection through Label Propagation* algorithm as not all our graphs have natural categorical labels upon which this analytical task depends (we will test this algorithm in the experiment that follows, however). We implement these five algorithms as procedures in the SPARQAL language, prefixing each with the six different Wikidata graph queries, stored as solution variables. The result is a benchmark of $6 \times 5 = 30$ queralytic tasks.

In Figure 4, we show the results for these 30 tasks using our in-memory implementation. First we remark that the Weakly Connected Components (WCC) algorithm timed-out in the case of the Zika graph after 10 minutes; furthermore, the LCC algorithm failed with memory errors on the Zika graph, where the time shown is thus for the Map/Reduce implementation. While the cheapest algorithm in general was BFS, the most expensive was WCC. Although some of these tasks took over a minute in the case of graphs with thousands or tens of thousands of nodes (Zika/Q1 and Horses/Q5), those with fewer than a thousand nodes/edges ran in under a second, compatible with interactive use.

6.3 Graphalytics: Stress Test

The scale of the previous graphs is quite low and uses (mostly) the in-memory algorithm. Hence we use the Graphalytics Benchmark [20] to perform stress tests for our implementation at larger scale. We adopt the `cit-Patents` dataset: a directed graph with 3,774,768 vertices and 16,518,947 edges. We implement SPARQAL procedures to run six graph algorithms on the full graph; in particular, we run the aforementioned five algorithms, as well as:

CDLP Community Detection through Label Propagation

Table 3: Execution time (min) for Graphalytics

Algorithm	SPARQAL/Jena	Python
BFS	11	1
CDLP	out of mem	15
LCC	out of mem	2
PR	250	5
SSSP	300	1
WCC	out of mem	1

The results of the Graphalytics benchmark are shown in Table 3 using the in-memory algorithm; for comparison, we also offer the times using an in-memory Python implementation. We see that the results are overwhelmingly negative, with poor performance due in particular to our handling of `QVALUES` clauses, which leads to unwieldy query strings when replaced by `VALUES` for large solution sequences. Switching to the Map/Reduce approach only solved half of our problems: although the procedures did not fail, they took even longer than the in-memory cases, where in other cases we estimated that the procedure would take months to finish due to the number of queries generated.

These results clearly demonstrate the limitations of our Jena-based implementation for large-scale graphs. While this is not currently our focus – which is rather achieving interactive performance on small-to-medium graphs – we identify this as an interesting challenge: can procedures in SPARQAL be optimised enough to be competitive with the imperative Python times shown?

7 CONCLUSION

We propose a declarative language called SPARQAL that allows for interleaving queries and analytics on graphs. We see this language as being useful in applications where analytical tasks require complex pre–and post–processing of the graph and results. In this context, we have proven some formal properties for our language, and discussed its formal relation to similar languages and abstractions. We have also implemented an initial system to support our language based on an off-the-shelf SPARQL query engine, showing that it offers interactive runtimes for typical analytics on graphs of fewer than one-thousand nodes (generated by means of a query). On the other hand, there is still much work to do if one wants a system supporting a declarative language that is competitive with standard frameworks for graph analytics. In particular, we need to look at the problem of how to compile and optimise SPARQAL procedures, ideally into smaller, lower-level components that can be implemented within database engines or analytical frameworks, depending on the scale. More generally, we believe that the combination of graph queries and analytics is a natural one, and one that raises interesting questions regarding languages and optimisations.

REFERENCES

[1] 2019. Online Appendix. https://github.com/VHDG88FKL/SPARQL-Analytics.

[2] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *SIGMOD*. 1421–1432.

[3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40.

[4] Piero Andrea Bonatti, Stefan Decker, Axel Polleres, and Valentina Presutti. 2018. Knowledge Graphs: New Directions for Knowledge Representation on the Semantic Web. *Dagstuhl Reports* 8, 9 (2018), 29–111.

[5] Robert Brijder, Floris Geerts, Jan Van den Bussche, and Timmy Weerwag. 2018. On the Expressive Power of Query Languages for Matrices. In *International Conference on Database Theory (ICDT)*. Schloss Dagstuhl, 10:1–10:17.

[6] Spencer Chang. 2018. Scaling Knowledge Access and Retrieval at Airbnb. AirBnB Medium Blog. https://medium.com/airbnb-engineering/scaling-knowledge-access-and-retrieval-at-airbnb-665b6ba21e95.

[7] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One Trillion Edges: Graph Processing at Facebook-Scale. *PVLDB* 8, 12 (2015), 1804–1815.

[8] Olivier Corby, Catherine Faron-Zucker, and Fabien Gandon. 2017. LDScript: A Linked Data Script Language. In *International Semantic Web Conference (ISWC)*. Springer, 208–224.

[9] Michael DeLorimier, Nachiket Kapre, Nikil Mehta, Dominic Rizzo, Ian Eslick, Raphael Rubin, Tomás E. Uribe, Thomas F. Knight Jr., and André DeHon. 2006. GraphStep: A System Architecture for Sparse-Graph Algorithms. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE Computer Society, 143–151.

[10] journal = Commun. ACM volume = 57 number = 10 pages = 78–85 year = 2014 Denny Vrandečić and Markus Krötzsch, title = Wikidata: a free collaborative knowledgebase. [n.d.]. ([n. d.]).

[11] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *International Conference on Management of Data (SIGMOD)*. ACM, 1433–1445.

[12] Floris Geerts. 2019. On the Expressive Power of Linear Algebra on Graphs. In *International Conference on Database Theory (ICDT)*. Schloss Dagstuhl, 7:1–7:19.

[13] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. USENIX Association, 17–30.

[14] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. 2013. SPARQL 1.1 Query Language. W3C Recommendation. https://www.w3.org/TR/sparql11-query/.

[15] Qi He, Bee-Chung Chen, and Deepak Agarwal. 2016. Building The LinkedIn Knowledge Graph. LinkedIn Blog. https://engineering.linkedin.com/blog/2016/10/building-the-linkedin-knowledge-graph.

[16] Dylan Hutchison, Bill Howe, and Dan Suciu. 2017. LaraDB: A Minimalist Kernel for Linear and Relational Algebra Computation. In *ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond (BeyondMR@SIGMOD)*. ACM, 2:1–2:10.

[17] Egor V. Kostylev, Juan L. Reutter, Miguel Romero, and Domagoj Vrgoc. 2015. SPARQL with Property Paths. In *International Semantic Web Conference (ISWC)*. Springer, 3–18.

[18] Elzbieta Krepska, Thilo Kielmann, Wan Fokkink, and Henri E. Bal. 2011. HipG: parallel processing of large-scale graphs. *Operating Systems Review* 45, 2 (2011), 3–13.

[19] Arun Krishnan. 2018. Making search easier: How Amazon's Product Graph is helping customers find products more easily. Amazon Blog. https://blog.aboutamazon.com/innovation/making-search-easier.

[20] LDBC. 2019. Graphalytics Benchmark Suite. https://graphalytics.org/.

[21] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2014. GraphLab: A New Framework For Parallel Machine Learning. *CoRR* abs/1408.2041 (2014). http://arxiv.org/abs/1408.2041

[22] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM Press, 135–146.

[23] 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)*. Springer, 376–394.

[24] Justin J. Miller. 2013. Graph Database Applications and Concepts with Neo4j. In *Southern Association for Information Systems Conference (SAIS)*. AIS eLibrary.

[25] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)* 34, 3 (2009), 16.

[26] RJ Pittman, Amit Srivastava, Sanjika Hewavitharana, Ajinkya Kale, and Saab Mansour. 2017. Cracking the Code on Conversational Commerce. eBay Blog. https://www.ebayinc.com/stories/news/cracking-the-code-on-conversational-commerce/.

[27] Juan L. Reutter, Adrián Soto, and Domagoj Vrgoc. 2015. Recursion in SPARQL. In *International Semantic Web Conference (ISWC)*. Springer, 19–35.

[28] Marko A. Rodriguez. 2015. The Gremlin graph traversal machine and language. In *Symposium on Database Programming Languages (DBPL)*. ACM, 1–10.

[29] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Trans. Neural Networks* 20, 1 (2009), 61–80.

[30] Upul Senanayake, Mahendra Piraveenan, and Albert Zomaya. 2015. The Pagerank-Index: Going beyond Citation Counts in Quantifying Scientific Impact of Researchers. *PLOS ONE* 10, 8 (08 2015), 1–34.

[31] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: a distributed graph engine on a memory cloud. In *SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, 505–516.

[32] Saurabh Shrivastava. 2017. Bring rich knowledge of people, places, things and local businesses to your apps. Bing Blogs. https://blogs.bing.com/search-quality-insights/2017-07/bring-rich-knowledge-of-people-places-things-and-local-businesses-to-your-apps.

[33] Amit Singhal. 2012. Introducing the Knowledge Graph: things, not strings. Google Blog. https://www.blog.google/products/search/introducing-knowledge-graph-things-not/.

[34] Philip Stutz, Daniel Strebel, and Abraham Bernstein. 2016. Signal/Collect12. *Semantic Web Journal* 7, 2 (2016), 139–166.

[35] Valentina Urzua and Claudio Gutiérrez. 2019. Linear Recursion in G-CORE. In *Alberto Mendelzon International Workshop on Foundations of Data Management (AMW)*, Vol. 2369. CEUR-WS.org.

[36] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2019. A Comprehensive Survey on Graph Neural Networks. *CoRR* abs/1901.00596 (2019).

[37] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. 2013. GraphX: a resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-loated with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*. ACM Press.

[38] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2013. Shark: SQL and rich analytics at scale. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. ACM Press, 13–24.

[39] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations (ICLR)*. OpenReview.net.

[40] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[41] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. 2013. A Distributed Graph Engine for Web Scale RDF Data. *PVLDB* 6, 4 (2013), 265–276.

A APPENDIX: PROOFS

A.1 Proof of Theorem 5.1

Let $M = (Q, F, \Sigma \cup \{B\}, q_0, \delta$ be a deterministic Turing machine, where $Q = \{q_o, \ldots, q_m\}$ is the set of states, there is a single final state $F = \{q_m\}$, Σ is the alphabet, B is the blank node covering all cells and δ is the transition function. Without loss of generality, and for readability, we assume that $\Sigma = \{0, 1\}$ and that δ does not define transitions for q_m. Let also $w = a_0, \ldots, a_n$ be a binary string. We construct a graph G and a SPARQAL procedure P such that M accepts w if and only if P returns a non-empty mapping.

Let us first assume that all states in Q and characters 0, 1, B are represented by IRIs, and that we use IRIs :right and :left. Define T_δ as a set of tuples of arity 5 containing one tuple (q, a, q', b, d) for each transition in δ of the form $\delta(q, a) = (q', b, d)$, for $d \in$:right, :left.

For readability we will not make the distinction between graph and program, and rather initialize everything in the program. But the construction can be easily adapted so that the input is not coded directly in the program but is queried from a graph. The procedure P consists of the following groups of statements.

Initialization:

First group of statements are in charge of initialising some of the solution variables. The idea of variable `transition` is to store the transitions of M. Solution variable `current` stores the content of the current cell that M is pointing on, and the current state of the run. Solution variables `positive_cells` and `negative_cells` store, respectively, all cells to the right of the head of M and all cells to the left of the head of M. Of course, the tape is infinite, but we only need to store cells we have already visited.

```
LET transition = (
  SELECT ?oldstate ?oldsymbol ?newstate ?newsymbol ?direction WHERE {
    VALUES (?oldstate ?oldsymbol ?newstate ?newsymbol ?direction) {T_δ}
  }
);
```

```
LET current = (
  SELECT  ?c_symbol ?c_state WHERE {
    VALUES ( ?c_symbol ?c_state) {(a0, q0}
  }
);
```

```
LET positive_cells = (
  SELECT ?p_pos ?p_symbol WHERE {
    VALUES (?p_pos ?p_symbol ) {(1, a1),...,(n,an)}
  }
);
```

Loop: The loop phase of the procedure is as follows:

```
DO (
  S1
  S2
  S3
  S4
) WHILE ( C );
```

Where all inner statements and conditions are defined next. The idea is that queries are used to check when the transition demands moving to the right or to the left, and depending on these values we update the cells accordingly. We use `new_current` as a temporal variable that will store the pointed cell and state of the machine in the next step of the run.

Statement S1:

```
LET new_current = (
  SELECT ?c_symbol ?c_state WHERE {
    SELECT (?newstate AS ?c_state) WHERE {
      QVALUES(transition)
      QVALUES(current)
      FILTER(?oldstate=?c_state && ?oldsymbol=?c_symbol)
    } .
    SELECT (?symbol AS ?c_symbol) WHERE {
      QVALUES(positive_cells)
      FILTER(?p_pos = 1)
      BIND(IF(!bound(?p_pos),"B",?p_symbol) AS ?symbol)
    }
  }
);
```

Statement S2:

```
LET positive_cells = (
  SELECT ?p_pos ?p_symbol WHERE {
```

```
    {
      SELECT (?p_pos -1 AS ?p_pos) ?p_symbol WHERE {
        QVALUES(positive_cells)
        QVALUES(transition)
      QVALUES(current)
        FILTER(?oldstate=?c_state && ?oldsymbol=?c_symbol)
        FILTER(?direction=:right)
      FILTER(?p_pos>1)
      }
    } UNION
    {
      SELECT (?p_pos + 1 AS ?p_pos) ?p_symbol WHERE {
        QVALUES(positive_cells)
        QVALUES(transition)
        QVALUES(current)
        FILTER(?oldstate=?c_state && ?oldsymbol=?c_symbol)
        FILTER(?direction=:left)
      }
    } UNION
    {
      SELECT (1 AS ?p_pos) (?newsymbol as ?p_symbol) WHERE {
        QVALUES(transition)
        QVALUES(current)
        FILTER(?oldstate=?c_state && ?oldsymbol=?c_symbol)
        FILTER(?direction=:left)
      }
    }
  }
);
```

Statement S3:

```
LET negative_cells = (
  SELECT ?n_pos ?n_symbol WHERE {
    {
      SELECT (?n_pos + 1 AS ?n_pos) ?n_symbol WHERE {
        QVALUES(negative_cells)
        QVALUES(transition)
        QVALUES(current)
        FILTER(?oldstate=?c_state && ?oldsymbol=?c_symbol)
        FILTER(?direction=:left)
        FILTER(?n_pos<-1)
      }
    } UNION
    {
      SELECT (?n_pos - 1 AS ?n_pos) ?n_symbol WHERE {
        QVALUES(negative_cells)
        QVALUES(transition)
        QVALUES(current)
        FILTER(?oldstate=?c_state && ?oldsymbol=?c_symbol)
        FILTER(?direction =:right)
      }
    } UNION
    {
      SELECT (-1 AS ?n_pos) (?newsymbol AS ?n_symbol) WHERE {
        QVALUES(transition)
        QVALUES(current)
        FILTER(?oldstate=?c_state && ?oldsymbol=?c_symbol)
        FILTER(?direction =:right)
      }
    }
  }
);
```

Statement S4:

```
LET current = (
  SELECT ?c_pos ?c_symbol ?c_state WHERE { QVALUES(new_current) }
);
```

Condition C:

```
ASK {
  QVALUES(transition)
  QVALUES(current)
  FILTER(?oldstate=?c_stat && ?oldsymbol=?c_symbol)
}
```

Return: Finally, below the loop, we return the state.

```
LET state = (
  SELECT ?state WHERE { QVALUES(current) FILTER(?c_state = :qm) }
);
RETURN(state);
```

One can check that this program effectively returns a non-empty mapping if and only if the procedure P terminates and variable `current` stores the state q_m. In turn, this happens if and only if M accepts on the input. This finishes the proof.

A.2 Proof of Proposition 5.2

We have already discussed how SPARQAL programs can be evaluated in PSPACE when they do not invent new values: all we need to store is (1) the current state of all variables, (2) the previous state of variables in fixed-point clauses, and (3) the current number of iterations for the case of loops with a max number (which is bounded by the query, as we do not need more iterations that the number stated. Additionally, SPARQL queries can themselves be computed in PSPACE, which gives us the upper bound.

For the lower bound we can use the construction in Theorem 5.1. Because we now that the machine M runs in PSPACE, the number of cells visited is bounded by a number which depends on the elements on the graph. Let then $|G|$ be the size of the graph, and assume that $n = |G|^k$ is the number of maximum cells visited in any computation of M over a graph with size $|G|$. The first thing we need is to construct a linear order from the elements of the graph, which we will store in a solution variable `order`. We can do this with a do-while iteration that keeps adding elements until there are no more to add. We can then extend this linear order into an order of 2k tuples, which will be stored in a solution variable `full-order`. With this full order we can now pre-compute all possible n cells that may be visited by M in solution variables `positive_cells` and `negative_cells`. We cannot use a numeric position anymore, but we can use our tuples in full order as the position. With these cells precomputed, we need to invoke the rest of the procedure. However, the last modification we make is that all arithmetic is replaced by the appropriate operation that uses our linear order.

A.3 Proof of Theorem 5.3

The first item is shown by induction. On the first step, the labels of the WL test and the ones stored in variable `vector` coincide, and thus the first bullet is clearly satisfied. Now assume that on the i-th iteration of the program, the same label in `vector` is assigned to nodes with the same label in the WL test. Going from iteration i to iteration $i+1$, if there are nodes in which a and b have the same label, it must be because (i) they had the same label in iteration i, (ii) their neighbours define an isomorphism, and (iii) their neighbours had the same label as well. Now if a pair a and b of nodes have different label in `vector`, it must be because queries Q_a and Q_b computed by Map draw different values. But this contradicts the fact that their neighbours are isomorphic and each of them have the same label.

For the second bullet, all we need is to find an injective function so that a neighbourhood is mapped to this value. We can do this using group concatenation in SPARQL as follows (for readability we assume that the graph has just one type of property : p, apart from the label, but this can of course be extended).

```
LET var_1 = ( SELECT ?v ?neighbour WHERE {?v :p ?neighbour});
LET vector = (
  SELECT ?v ?lab WHERE { ?v rdfs:label ?lab });
DO (
 LET vector = (
     SELECT (?node AS ?v) (GROUP_CONCAT(?n_lab; SEPARATOR=", ") AS ?lab)
     WHERE {SELECT (?v AS ?node) (?neighbour AS ?v) WHERE {QVALUES(var\_1)} }.
         {QVALUES(vector)} .
         FILTER (?neighbour = ?v)
 );
) WHILE ( condition );
RETURN(vector);
```