In-Database Graph Analytics with Recursive SPARQL

Aidan Hogan[1], Juan L. Reutter[2], and Adrián Soto[3]

[1] DCC, Universidad de Chile & IMFD, Chile
[2] Pontificia Universidad Católica de Chile & IMFD, Chile
[3] Faculty of Engineering and Sciences, Universidad Adolfo Ibáñez & Data Observatory Foundation & IMFD, Chile

Abstract. Works on knowledge graphs and graph-based data management often focus either on 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 language that is well-suited for both graph querying and analytical tasks. We propose a minimalistic extension of SPARQL to allow for expressing analytical tasks over existing SPARQL infrastructure; 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). Moreover, queries in this language can also be compiled into sequences of iterations of SPARQL update statements. We show how procedures in our language can be implemented over off-the-shelf SPARQL engines, with a specialised client that can leverage database operations to improve the performance of queries. Results for our implementation show that procedures for popular analytics currently run in seconds or minutes for selective sub-graphs (our target use-case).

1 Introduction

Recent years have seen a surge in interest in graph data management, learning and analytics within different sub-communities, particularly under the title of "knowledge graphs" [1]. However, more work is needed to combine complementary techniques from different areas [2]. As a prominent example, while numerous query languages have been proposed for graph databases, and numerous frameworks have been proposed for graph analytics, few works aim to combine both: while some analytical frameworks support lightweight query features [3,4], and some query languages support lightweight analytical features [5,6,7], only specific types of queries or analytics are addressed.

Take, for example, the following seemingly simple task, which we wish to apply over Wikidata[4]: *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 [5] or Cypher [6] support path expressions that capture reachability, they cannot express conditions on the nodes through which such paths pass, as is required by

[4] `https://www.wikidata.org/`, or see endpoint at `https://query.wikidata.org/`

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 [8]. One way this could currently be achieved is to: (1) perform a SPARQL query to extract the citation graph of articles about the Zika virus; (2) load the graph or connect the database with an external tool to compute PageRank scores; (3) perform another query to extract the (bipartite) authorship graph for the articles; (4) load or connect again 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 or languages to solve the task. Another strategy might be to load the Wikidata dump directly into a graph-analytics framework and address all tasks within it; in this case, we lose the convenience of a query language and database optimisations for extracting (only) the relevant data.

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. Specifically, we explore the addition of recursive features to the SPARQL 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 & Analytics Language* (*SPARQAL*). We study the expressive power of SPARQAL with similar proposals found in the literature [9,10,11,12]. We then discuss the implementation of our language on top of a SPARQL query engine, introducing different evaluation strategies for our procedures. 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 larger-scale graphs, opening avenues for future research.

Example 1. 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. As mentioned in the introduction, we would like to know from which metro stations we can still reach Palermo. The data to answer this query are available on Wikidata [13]. We can express this request in our SPARQL-based language, as shown in Figure 1. 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 5, 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 6 to 17 we have a loop that executes two instructions: the first, start-

```
 1 LET reachable = (    # stations directly adjacent to Palermo not on Line C
 2   SELECT ?s WHERE {
 3     wd:Q3296629 wdt:P197 ?s . MINUS { ?s wdt:P81 wd:Q1157050 }
 4   }
 5 );
 6 DO (
 7   LET adjacent = (   # stations adjacent to stations in variable reachable
 8     SELECT (?adj AS ?s) WHERE {
 9       ?s wdt:P197 ?adj . MINUS { ?adj wdt:P81 wd:Q1157050 } QVALUES(reachable)
10     }
11   );
12   LET reachable = (  # add stations in variable adjacent to variable reachable
13     SELECT DISTINCT ?s WHERE {
14       { QVALUES(adjacent) } UNION { QVALUES(reachable) }
15     }
16   );
17 ) UNTIL(FIXPOINT(reachable) );
18 RETURN(reachable);
```

Fig. 1. Procedure to find metro stations from which Palermo can be reached

ing at line 7, computes all stations directly adjacent to the current reachable stations not on Line C; here the `QVALUES(reachable)` clause is used to invoke all solutions stored in variable `reachable`. The second, starting at line 12, 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 18, we return reachable stations. □

2 Related Work

We now discuss frameworks for applying graph analytics, proposals for combining graph querying and graph analytics, and recursive extensions of graph query languages.

Frameworks for Graph Analytics. Various frameworks have been proposed for performing graph analytics at large-scale, including GraphStep [14], Pregel [15], HipG [16], PowerGraph [17], GraphX [3], Giraph [18], Signal/Collect [19], etc. These frameworks operate on a computational model – sometimes called the systolic model [20], Gather/Apply/Scatter (GAS) model [17], graph-parallel framework [3], etc. – whereby each node in a graph recursively computes its state based on data available in its neighbourhood. However, implementing queries on such frameworks, selecting custom subgraphs to be analysed, etc., is not straightforward. Datalog variants also offer an interesting framework for graph analytics, especially when Datalog is extended with arithmetic features, as in e.g. [21,12,22,23]. As we discuss in Section 4, SPARQAL can be seen as bridging existing RDF databases and SPARQL services with such frameworks.

Graph Queries and Analytics. Our work aims to combine graph queries and analytics for RDF/SPARQL. Along these lines, Trinity.RDF [24] 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 [24]. Later work used the same infrastructure in a system called Trinity [25] 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, the authors do not discuss the combination of queries and analytics, nor do they propose languages.

Most modern graph query languages offer some built-in analytical features. SPARQL 1.1 [5] introduced *property paths* [26] that allow for finding pairs of nodes connected by some path matching a regular expression, and some extensions allow for invoking specific extra analytical features [7]. The Cypher query language [6] (used by Neo4j [27]) also allows for querying on paths with limited regular expressions; however, it also supports shortest paths, returning paths, etc. The G-CORE query language [28] 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 [29].

Gremlin [4] is an imperative scripting language that can express analytical tasks through graph traversals. Per the Trinity.RDF system [24], graph traversals, when combined with variables, can be used to express and evaluate, for example, basic graph patterns [28]. Gremlin [4] also supports some standard 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 [4]. However, Gremlin is specifically designed to work under a property graph data model, and more importantly is missing practical RDF-specific features of SPARQL such as datatype ordering, built-in functions (e.g., `langMatches`, `isIRI`, `year`), named graphs, federation, etc. Thus, using Gremlin in the context of RDF databases would require porting these features between both systems, which is precisely what we want to avoid.

Recursive Graph Queries. Most graph query languages support recursively matching path expressions; however, per Example 1, more powerful forms of recursion are needed in order to support a more general class of analytics.[5] Later we will compare the expressive power of our proposal to recursive graph query languages, such as those proposed by Reutter et al. [9] for SPARQL, and by Urzua and Gutierrez [11] for G-CORE. We also highlight the LDScript language as proposed by Corby et al. [10], which also relates to our proposal, supporting 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. We remark that LDScript does not include support for arbitrary do–until iteration, where applying a fixed number of iterations is insufficient for a broad range of analytical tasks.

Novelty. Unlike graph analytics frameworks, we propose a language for combining queries and analytics on graphs. Unlike Gremlin and Datalog variants, we propose a language designed to extend SPARQL, thus benefiting from its built-in support for RDF. The closest proposals to ours are those that extend graph query languages with recursive features [9,10,11]. In comparison with the proposal of Reutter et al. [9] and Urzua and Gutierrez [11], we allow recursion over `SELECT` queries, which adds flexibility by not

[5] Though more complex forms of "navigational patterns" have been proposed in the literature, they are mostly limited to path-finding and reachability [29].

requiring to maintain intermediate results as (RDF) graphs: for example, allowing us to maintain multiple intermediate relations of arbitrary arity (without requiring some form of reification); we further allow for terminating a loop based on a boolean condition (an `ASK` query), which can more easily express termination conditions in cases where an analytics task is infinitary and/or requires approximation (e.g., PageRank). Unlike LD-Script [10], our focus is on supporting graph analytics, adding features, such as fixpoint and do–until loops, that are essential for many forms of graph analytics.

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–until 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, 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 [5]. We recall the notion of a *solution sequence*, which is the result of a SPARQL query evaluated on a graph (or dataset), listing zero-or-more solutions for which the query matches the data. We assume use of the full SPARQL 1.1 query language as defined by the standard [5].

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.

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 `SELECT` query that may use constructs of the form `QVALUES(var)`.

Loop: Executes a sequence of statements until a termination condition holds. The syntax of a loop statement is `DO (S) UNTIL (condition);` where `S` is a sequence of statements and `condition` is one of the following three forms of termination condition: (1) `TIMES t`, where `t` is an integer greater than 0; (2) `FIXPOINT (var)`, where `var` is a solution variable; (3) `AQ`, an `ASK` query that may use `QVALUES`.

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: (1) the last statement, and only the last statement, is a return statement; (2) all solution variables used in `QVALUES`, `FIXPOINT` and `RETURN` have been assigned by `LET` in a previous statement (or a nested statement thereof).

Example 2. Figure 1 illustrated a SPARQAL procedure with three statements: an assignment statement (lines 1–5); a loop statement with a fixpoint termination condition and two nested assignments (lines 6–17); and a final return statement (line 18). □

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 $\texttt{var\_1}, \ldots, \texttt{var\_k}$ be all variables mentioned in any statement in P (including in nested statements). For a tuple $\mathrm{val}_0 = (r_1, \ldots, r_k)$ of initial assignments of (possibly empty) solution sequences to variables $\texttt{var\_1}, \ldots, \texttt{var\_k}$, we will construct a sequence $\mathrm{val}_0, \ldots, \mathrm{val}_n$ of k-tuples, where each val_i represents the value of all variables after executing statement s_i.

The construction is done inductively. Assume that $\mathrm{val}_{i-1} = (r_1, \ldots, r_k)$. The value of val_i depends on whether s_i is an assignment, loop or return statement.

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[6], and let r^* be the result of evaluating this extended query over the database. Then, substituting r_i for r^* in the tuple val_{i-1}, we define $\mathrm{val}_i = (r_1, \ldots, r_{i-1}, r^*, r_{i+1}, r_k)$.

Next, if s_i is the loop statement `DO (S) UNTIL (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 $\mathrm{val}'_0 :- \mathrm{val}_{i-1}$.
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 $\mathrm{val}'_0 :- \mathrm{val}'_\ell$ and repeat step 2 above.
4. Otherwise finish, and set $\mathrm{val}_i :- \mathrm{val}'_\ell$.

To define when a tuple val'_ℓ over k variables satisfies a condition, we have 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 $AQ[(\texttt{var\_1}, \ldots, \texttt{var\_k}) \mapsto \mathrm{val}'_\ell]$ evaluates to true.

[6] 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) }
    }
  );
) UNTIL (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);
```

Fig. 2. Procedure to compute the top author in terms of *p*-index for articles about the Zika virus

Finally, 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.

Note that we assume all solution variables to have a global scope as it makes the semantics simpler to define; one could define local solution variables analogously. Moreover, some SPARQAL statements may incur infinite loops; later we will discuss fragments for which every program can be shown to terminate (as in, e.g., Datalog or recursive SPARQL). Currently we do not consider blank nodes when checking `FIXEDPOINT` conditions; these could be supported in a future version using the labelling of [30], which has been shown to be efficient for a wide variety of graphs.

Example 3. We recall Example 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 before (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 4. Suppose we have the citation network of articles on a topic of interest and, 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 consider the citation network of all the articles about the Zika virus on Wikidata, where we then encode and apply the PageRank algorithm over the citation network, using the resulting article scores to compute p-indexes for the respective authors. We show a procedure in our language for solving this task in Figure 2.

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 think of 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. 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 [31], 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 conclude the iteration by allocating the unshared rank to each node equally, updating the results for `rank`. The loop is applied 10 times.

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

3.4 Graph Updates

Although there is a straightforward way to implement our language on top of any engine using the `VALUES` clause, this can generate long query strings that current engines struggle to process. Hence we define a recursive algebra for graphs that can also express queralytics. As a motivating example, consider the declaration of variables `zika` and `degree`, in lines 1 and 15 respectively of Figure 2. These statements initialise these variables, but we can view them as queries constructing two graphs. More precisely, we use the graph `ex:zika` to store the result of the query:

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

Thus, instead of storing pairs of values for <`node`> <`cite`> in a SPARQAL solution variable `zika`, we store them as triples of the form <`node`> `ex:zikacites` <`cite`> in a graph named `ex:zika`. Using this graph we can now store the result of `degree` in graph `ex:degree` by means of the following query:

```
CONSTRUCT { ?node ex:zikadegree ?degree } WHERE {
  SELECT ?node (COUNT(?cite) AS ?degree) WHERE {
    GRAPH ex:zika {?node ?p ? cite} }
   GROUP BY ?node }
```

We remark that a general solution would involve reifying any SPARQAL variable using more than two SPARQL variables, possible generating new nodes.

Algebra of updates. Let $\mathcal{G} = \{(n_1, G_1), \ldots, (n_k, G_k)\}$ be a set of named graphs with IRIs $\{n_1, \ldots, n_k\}$ and RDF graphs $\{G_1, \ldots, G_k\}$ such that $n_i = n_j$ if and only if $i = j$. Let Q be a `CONSTRUCT` query. Given an IRI n, we use $n \leftarrow Q$ to express the action of storing the result G of $Q(\mathcal{G})$ as the named graph (n, G) in $\mathcal{G}$, overwriting the graph previously named n if necessary. Our algebra of updates consists of (1) update expressions of the form $n \leftarrow Q$, for n an IRI and Q a `CONSTRUCT` query that may reference any of the existing graphs in $\mathcal{G}$, (2) loop expressions of the form `DO A UNTIL (condition)` where `A` is a sequence of expressions and `(condition)` is again one of `TIMES t`; `FIXPOINT n`, where `n` is a graph name in $\mathcal{G}$; or `AQ`, an `ASK` query that may reference graphs in $\mathcal{G}$.

With respect to the semantics of this algebra, starting with the initial set $\mathcal{G}$, an expression modifies graphs in $\mathcal{G}$ as follows. An assignment expression $n \leftarrow Q$ removes the graph (n, G) from $\mathcal{G}$ (if it exists), and adds $(n, Q(\mathcal{G}))$, where $Q(\mathcal{G})$ denotes the evaluation of Q over $\mathcal{G}$. A loop expression `DO A UNTIL (condition)` applies iteration, evaluating the sequence `A`: t times if `condition` is `TIMES t`, or until the named graph $(n, G) \in \mathcal{G}$ did not change at the end of two subsequent iterations if `condition` is `FIXPOINT n`, or until the evaluation of query `AQ` over $\mathcal{G}$ returns true if `condition` is `AQ`.

Given an expression A dealing with graphs in $\mathcal{G}$, we use $A(\mathcal{G})$ to denote the result of evaluating A over $\mathcal{G}$. Looking at our motivating example, one sees that transforming our procedural language into the graph algebra is not difficult, and neither is transforming graph algebra expressions into our procedural language. The following proposition, proven in an extended version of this paper available online [32], summarises the claim that both languages have the same expressive power.

Proposition 1. *Let P be a SPARQAL procedure, with v the solution variable returned by P. Then one can construct an expression A in the algebra of updates mentioning a set $\mathcal{G}$ of graphs, and a* `SELECT` *query Q, such that evaluating Q over $A(\mathcal{G})$ yields the same solutions as those stored by v after evaluating P over $\mathcal{G}$. Likewise, for an algebra expression A mentioning graphs $\mathcal{G}$, and any named graph $(n, G) \in \mathcal{G}$, one can construct a SPARQAL procedure P returning a solution variable v over $\mathcal{G}$, and a* `CONSTRUCT` *query Q, such that evaluating Q over the solutions stored by v yields the graph G.*

Thus, we now have two strategies for implementing SPARQAL procedures: we can implement them directly by translating `QVALUES` clauses as `VALUES` statements while running the procedures, or we can compile the procedure into an expression in our algebra of updates and implement this directly. We will analyse these two possibilities in Section 5.2, but first we study the expressive power of these formalisms.

4 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.

4.1 Turing-completeness

Although do–until 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 the graph $f(w)$ produces a non-empty result. Along these lines, we prove the following result:

Theorem 1. *SPARQAL is Turing-complete*

The proof of this theorem (presented in the extended version of this paper [32]) relies on the combination of do–until loops and the ability to create new values in the base SPARQL language through `BIND` statements and algebraic functions [5]. 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–until loops have been studied previously in the literature, especially in the context of relational algebra (see e.g. [33]). 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.[7] We can then state the following (the proposition is proven in the extended version of this paper [32]):

Proposition 2. *The evaluation problem for SPARQAL procedures that do not invent new values is PSPACE-complete.*

4.2 Comparison with other recursive extensions to SPARQL

We base our comparison on the recursive extension proposed by Reutter et al. [9], 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 `CONSTRUCT` queries of the form:

```
WITH RECURSIVE G AS {$Q_{\text{CONSTRUCT}}$}
$Q_{\text{SELECT}}$
```

where G is an IRI used to denote a temporary 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 references 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 monotonicity, Reutter et al. [9] 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* [34].

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 $P'_{\text{CONSTRUCT}}$ ) ) UNTIL ( FIXPOINT (graph) );
LET result = $Q'_{\text{SELECT}}$;
RETURN result;
```

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

In the other direction, can recursive SPARQL simulate SPARQAL procedures? This depends on what sorts 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 [9]. Together with Proposition 2, this means that recursive SPARQL cannot simulate SPARQAL procedures unless PTIME = PSPACE, which is widely assumed to be false. A similar result

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

was shown for similar extensions to relational algebra: relational algebra equipped with fixed point cannot simulate do–until queries unless PTIME = PSPACE [33]

Conversely, the semantics for recursive SPARQL is not defined when one allows to use operands such as **BIND** clauses. 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 $T_Q(\emptyset)$, $T_Q(T_Q(\emptyset))$, $\ldots$, 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 show that both languages coincide, because recursive SPARQL becomes Turing-complete as well.

4.3 Comparison with the Datalog framework

Our algebra of graph updates also gives us a way of comparing with Datalog variants for analytics tasks that have been proposed in the literature (for this discussion we assume familiarity with the Datalog language). Indeed, consider a set of named graphs $\mathcal{G} = \{(n_1, G_1), \ldots, (n_k, G_k)\}$, a sequence A of graph updates of the form $n \leftarrow Q$, for n one of $n_1, \ldots, n_k$ and Q a construct query over $\mathcal{G}$. If we assume that each Q is monotone, then an algebra expression **DO** A **UNTIL FIXPOINT** n_i can be understood as a Datalog program over k ternary predicates $T_1, \ldots, T_k$, each interpreted as the triples in graphs $n_1, \ldots, n_k$, given by the rules $\leftarrow T_1, \ldots, \leftarrow T_k$ and a rule $T_j \leftarrow Q$ for each update $n_j \leftarrow Q$ in A. We evaluate this program until the data for predicate T_i does not change.

Thus, for example, if we restrict queries in SPARQL so that they match the expressive power of the SociaLite language by Seo et al. [12], then we end up precisely with SociaLite. What SPARQAL adds on top of these Datalog variants is (1) native support for SPARQL, since the right-hand side of rules are actually stated in SPARQL, and (2) not having to depend on particular fixed point semantics[8]. As we commented when comparing to recursive SPARQL, this does come with an increase in expressive power.

5 Experiments

In this section we present our prototypical implementation of a queralytics engine based on the SPARQAL language, along with experiments over different datasets to ascertain its performance and limitations. The goals of this prototype are to demonstrate that the language can be used, in practice, to express in-database analytics, and to ascertain the performance achievable when operating over an off-the-shelf SPARQL query engine. The target use-case for our prototype is – per the scenarios outlined in Examples 1 and 4 – to run queralytics (near-)interactively on small-to-medium graphs projected from a larger graph using a query. Along these lines, the prototype was developed on top of the Apache Jena Framework, version 3.10 (for our second set of tests we also provide a version of the prototype mounted on top of Virtuoso). The implementation provides the following core functionalities: (1) it parses a SPARQAL procedure into a sequence of statements, which are evaluated according to their semantics by: (2a) maintaining a map of solution variables to solution sequences; (2b) replacing variables used

[8] Here we are not interested in languages with decidable containment, in part because we are not addressing how to do reasoning within SPARQAL, but this is a fertile area for future work.

Table 1. Number of nodes and edges in graphs considered

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

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, keeping the previous solution sequence of the respective variable in-memory to track changes. We also provide an initial prototype for the algebraic strategy defined in Section 3.4; this prototype creates the new graphs using CONSTRUCT statements, and deletes/adds new graphs using the native functionalities provided by SPARQL systems.

Experiments were tested on a MacBook Pro with a 3.1 GHz Intel I5 processor and 16 GB of RAM. Regarding our motivating scenarios, Example 1 took just 1.3 seconds to return 16 stations from which Palermo can be reached without using Line C, and Example 4 – 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.[9]

To further test our implementation, we design a benchmark based on Wikidata for running analytical tasks on sub-graphs extracted through queries. Finally, we stress-test our prototype for a graph analytics benchmark at a larger scale. In particular, we show that the algebraic approach may be better suited for handling large datasets.[10]

5.1 Wikidata: Queralytics Benchmark

To the best of our knowledge, there is no existing benchmark for queralytics along the lines discussed in this paper. This led us to design a novel benchmark for queralytics over the Wikidata knowledge graph. We took the "truthy" RDF dump of Wikidata as our benchmark graph [35]. 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[11]. 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

[9] For reference, the top such author is George Dick, with a *p*-index of 0.124.
[10] All sources and datasets are available at `https://adriansoto.cl/files/SPARQAL.zip`.
[11] `https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples`

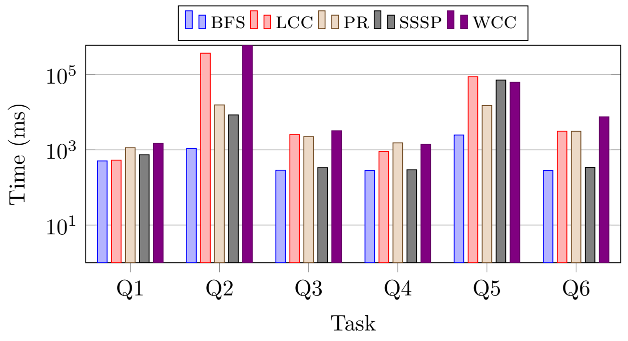

Fig. 3. Results for Wikidata queralytic benchmark

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 1.

Next we must define the analytics that we would like to apply on these graphs. For this, we adopted five of the six algorithm proposed for the Graphalytics Benchmark [36] defined by the Linked Data Benchmark Council (LDBC); namely:

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

We do not include the *Community Detection through Label Propagation* **CDLP** as it assumes data with initial labels. 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 3, 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. 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, and thus would be compatible with interactive use.

5.2 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 [36] to perform stress tests for our prototype at larger scale with the goal of identifying the choke points of the current implementation. We adopt the `cit-Patents` dataset: a directed graph with 3,774,768 vertices and 16,518,947 edges. We implement both alternatives for evaluating SPARQAL procedures: using **`VALUES`** and using Graph Updates. In order to try a different backend, we also implemented the Graph Updates alternative on top of Virtuoso, using Python to access the database with the `SPARQLWrapper` library.

The results of the Graphalytics benchmark are shown in Table 2. For the **`VALUES`** implementation, we identify two key choke points. An obvious choke-point is presented

Table 2. Execution time (min) for Graphalytics benchmark. Here OOM is for out-of-memory error.

Algorithm	**BFS**	**LCC**	**PR**	**SSSP**	**WCC**
SPARQAL/Jena–Values	11	OOM	250	300	OOM
SPARQAL/Jena–Updates	2	26	112	127	13
SPARQAL/Virtuoso–Updates	1	TIMEOUT	244	5	310

by the fact that solution sequences are stored in memory: this puts an upper-bound on scalability, leading to OOM errors for complex queralytics on larger graphs (more specifically, in this case, running **LCC** and **WCC** on a graph of millions of nodes and tens of millions of edges). The other choke-point is the handling of `QVALUES` clauses using a `VALUES` clause with large solution sequences, yielding queries that are inefficient for Apache Jena. We view a number of possibilities for addressing these choke points in future work. Keeping with the in-database analytics scenario, the first choke point could be alleviated with compression and indexing techniques, while both choke points could be addressed by batch-at-a-time processing of `QVALUES` clauses.

The performance issues of the `VALUES` implementation are alleviated, to some extent, when we switch to the implementation based on graph updates. Intermediate graphs are stored in memory, but their sizes tend to be smaller than the size of solution sequences, as one avoids replication. Here, the main choke-point is the fact that constructed graphs are not currently indexed, and thus queries over them run slower. When comparing the Jena/Updates implementation against the one using Virtuoso, we see several differences. Both implementations handle BFS much better. We speculate that Virtuoso is better at `SSSP` because it is more efficient when dealing with strings representing paths. On the other hand, all of `LCC`, `WCC` and `PR` require large update operations on temporary graphs, something that transactional databases like Virtuoso are not designed for.

Looking to the future, we speculate that implementing lightweight indexes in constructed graphs would provide even faster times for our Updates implementation. Another in-database alternative would be using GPU-acceleration for parallelising batches. In general, however, in order to process larger graphs, an in-database solution may not be feasible, but rather SPARQAL procedures would need to be translated to tasks that can run on distributed graph processing frameworks, as discussed in Section 2.

6 Conclusion

We believe that the combination of graph queries and analytics is a natural one, in the sense that tasks of interest to users often involve interleaving both paradigms. The SPARQAL language provides a way to express such tasks, and makes initial steps towards a system to support them. We see this language as being useful for combining querying and analytical tasks specifically in an RDF/SPARQL setting.

We hope that our proposal ignites the discussion on different ways for enriching SPARQL with graph analytics, and the best architecture to support them (see [37] for a related discussion). A key research challenge relates to the optimisation of SPARQAL procedures. We have investigated batch-at-a-time and also compilation into algebraic-like statements for evaluation within the database, but we still need support for indexing temporal graphs (perhaps as in [38]), and looking whether or not traditional database optimisation tasks are suitable for optimising SPARQAL procedures.

Acknowledgements This work was supported by the Millennium Institute for Foundational Research on Data (IMFD) and by Fondecyt Grant No. 1181896.

References

1. Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutierrez, C., Gayo, J.E.L., Kirrane, S., Neumaier, S., Polleres, A., Navigli, R., Ngomo, A.N., Rashid, S.M., Rula, A., Schmelzeisen, L., Sequeda, J.F., Staab, S., Zimmermann, A.: Knowledge Graphs. CoRR **abs/2003.02320** (2020)
2. Bonatti, P.A., Decker, S., Polleres, A., Presutti, V.: Knowledge Graphs: New Directions for Knowledge Representation on the Semantic Web. Dagstuhl Reports **8**(9) (2018) 29–111
3. Xin, R.S., Gonzalez, J.E., Franklin, M.J., Stoica, I.: GraphX: a resilient distributed graph system on Spark. In: International Workshop on Graph Data Management Experiences and Systems (GRADES), ACM Press (2013)
4. Rodriguez, M.A.: The Gremlin graph traversal machine and language. In: Symposium on Database Programming Languages (DBPL), ACM (2015) 1–10
5. Harris, S., Seaborne, A., Prud'hommeaux, E.: SPARQL 1.1 Query Language. W3C Recommendation (March 2013) `https://www.w3.org/TR/sparql11-query/`.
6. Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., Taylor, A.: Cypher: An Evolving Query Language for Property Graphs. In: International Conference on Management of Data (SIGMOD), ACM (2018) 1433–1445
7. Song, X., Chen, S., Zhang, X., Feng, Z.: A construct-based query for weighted rdf graph analytics. In: ISWC Satellites. (2019) 25–28
8. Senanayake, U., Piraveenan, M., Zomaya, A.: The Pagerank-Index: Going beyond Citation Counts in Quantifying Scientific Impact of Researchers. PLOS ONE **10**(8) (08 2015) 1–34
9. Reutter, J.L., Soto, A., Vrgoc, D.: Recursion in SPARQL. In: International Semantic Web Conference (ISWC), Springer (2015) 19–35
10. Corby, O., Faron-Zucker, C., Gandon, F.: LDScript: A Linked Data Script Language. In: International Semantic Web Conference (ISWC), Springer (2017) 208–224
11. Urzua, V., Gutiérrez, C.: Linear Recursion in G-CORE. In: Alberto Mendelzon International Workshop on Foundations of Data Management (AMW). Volume 2369., CEUR-WS.org (2019)
12. Seo, J., Guo, S., Lam, M.S.: Socialite: Datalog extensions for efficient social network analysis. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE), IEEE (2013) 278–289
13. Vrandečić, D., Krötzsch, M.: Wikidata: a free collaborative knowledgebase. Commun. ACM **57**(10) (2014) 78–85
14. DeLorimier, M., Kapre, N., Mehta, N., Rizzo, D., Eslick, I., Rubin, R., Uribe, T.E., Jr., T.F.K., DeHon, A.: GraphStep: A System Architecture for Sparse-Graph Algorithms. In: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), IEEE Computer Society (2006) 143–151
15. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: ACM SIGMOD International Conference on Management of Data (SIGMOD), ACM Press (2010) 135–146

16. Krepska, E., Kielmann, T., Fokkink, W., Bal, H.E.: HipG: parallel processing of large-scale graphs. Operating Systems Review **45**(2) (2011) 3–13
17. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: 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 (2012) 17–30
18. Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., Muthukrishnan, S.: One trillion edges: Graph processing at facebook-scale. PVLDB **8**(12) (2015) 1804–1815
19. Stutz, P., Strebel, D., Bernstein, A.: Signal/Collect12. Semantic Web Journal **7**(2) (2016) 139–166
20. Low, Y., Gonzalez, J.E., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Graphlab: A new framework for parallel machine learning. CoRR **abs/1408.2041** (2014)
21. Kaminski, M., Grau, B.C., Kostylev, E.V., Motik, B., Horrocks, I.: Foundations of declarative data analysis using limit datalog programs. arXiv preprint arXiv:1705.06927 (2017)
22. Bellomarini, L., Gottlob, G., Pieris, A., Sallinger, E.: Vadalog: A language and system for knowledge graphs. In: International Joint Conference on Rules and Reasoning, Springer (2018) 3–8
23. Eisner, J., Filardo, N.W.: Dyna: Extending datalog for modern ai. In: International Datalog 2.0 Workshop, Springer (2010) 181–220
24. Zeng, K., Yang, J., Wang, H., Shao, B., Wang, Z.: A Distributed Graph Engine for Web Scale RDF Data. PVLDB **6**(4) (2013) 265–276
25. Shao, B., Wang, H., Li, Y.: Trinity: a distributed graph engine on a memory cloud. In: SIGMOD International Conference on Management of Data (SIGMOD). (2013) 505–516
26. Kostylev, E.V., Reutter, J.L., Romero, M., Vrgoc, D.: SPARQL with Property Paths. In: International Semantic Web Conference (ISWC), Springer (2015) 3–18
27. Miller, J.J.: Graph Database Applications and Concepts with Neo4j. In: Southern Association for Information Systems Conference (SAIS), AIS eLibrary (2013)
28. Angles, R., Arenas, M., Barceló, P., Boncz, P.A., Fletcher, G.H.L., Gutierrez, C., Lindaaker, T., Paradies, M., Plantikow, S., Sequeda, J.F., van Rest, O., Voigt, H.: G-CORE: A Core for Future Graph Query Languages. In: SIGMOD. (2018) 1421–1432
29. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J.L., Vrgoc, D.: Foundations of Modern Query Languages for Graph Databases. ACM C. Surv. **50**(5) (2017) 68:1–68:40
30. Hogan, A.: Canonical forms for isomorphic and equivalent RDF graphs: algorithms for leaning and labelling blank nodes. ACM Transactions on the Web (TWEB) **11**(4) (2017) 1–62
31. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: Bringing order to the Web. Technical report, Stanford InfoLab (1999)
32. Hogan, A., Reutter, J.L., Soto, A.: In-Database Graph Anaytics with Recursive SPARQL [Extended Version]
33. Abiteboul, S., Hull, R., Vianu, V.: Foundations of databases. Volume 8. Addison-Wesley Reading (1995)
34. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Transactions on Database Systems (TODS) **34**(3) (2009) 16
35. Malyshev, S., Krötzsch, M., González, L., Gonsior, J., Bielefeldt, A.: Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph. In: International Semantic Web Conference (ISWC), Springer (2018) 376–394
36. LDBC: Graphalytics Benchmark Suite (2019) `https://graphalytics.org/`.
37. Raasveldt, M., Mühleisen, H.: Data management for data science-towards embedded analytics. In: CIDR. (2020)
38. Holanda, P., Raasveldt, M., Manegold, S., Mühleisen, H.: Progressive indexes: indexing for interactive data analysis. Proceedings of the VLDB Endowment **12**(13) (2019) 2366–2378

A Appendix: Proofs

A.1 Proof of Proposition 1

We have shown an example of how to go from the declaration of a variable to a `CONSTRUCT` query. For a general construction, for each declaration `LET var = (Q);` we construct an expression $A_{\text{var}} = G \leftarrow C$, where G is a graph named after var, and C is the construct query `CONSTRUCT H WHERE Q'`. Here `Q'` is obtained by performing the following operations on `Q` (for simplicity we assume solution variables in SPARQAL are IRIs, to name graphs after them):

- Every instance of `QVALUES(u)` is replaced by `GRAPH u {Hu}`, where `Hu` is the construct part of the query after the initialization of variable u.
- An additional `SELECT` query may be added if there is aggregation (as we saw in the examples).

Furthermore, `H` is a suitable reification of the variables in the `SELECT` part of `Q`. The remaining parts of SPARQAL are translated following these same principles.

To go from Graph Updates to SPARQAL, we build one variable for each named graph, and each time this graph is queried we replace it with its `QVALUES` command.

It is now trivial to check the properties for both directions of the translation.

A.2 Proof of Theorem 1

Let $M = (Q, F, \Sigma \cup \{B\}, q_0, \delta)$ be a deterministic Turing machine, where $Q = \{q_o, \dots, q_m\}$ 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 $w = a_0, \dots, a_n$ 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 initialise 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.

Initialisation:

The first group of statements is 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) {$T_\delta$}
  }
);
```

```
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
) UNTIL ( 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 temporary variable that will store the cell with the pointer and the 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.3 Proof of Proposition 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, giving us the upper bound.

For the lower bound we can use the construction in Theorem 1. Because we know 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–until 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.