SYRql: A Dataflow Language for Large Scale Processing of RDF Data

Fadi Maali[1], Padmashree Ravindra[2], Kemafor Anyanwu[2], and Stefan Decker[1]

[1] Insight Centre for Data Analytics, National University of Ireland Galway
{fadi.maali, stefan.decker}@insight-centre.org
[2] Department of Computer Science, North Carolina State University, Raleigh, NC
{pravind2, kogan}@ncsu.edu

Abstract. The recent big data movement resulted in a surge of activity on layering declarative languages on top of distributed computation platforms. In the Semantic Web realm, this surge of analytics languages was not reflected despite the significant growth in the available RDF data. Consequently, when analysing large RDF datasets, users are left with two main options: using SPARQL or using an existing non-RDF-specific big data language, both with its own limitations. The pure declarative nature of SPARQL and the high cost of evaluation can be limiting in some scenarios. On the other hand, existing big data languages are designed mainly for tabular data and, therefore, applying them to RDF data results in verbose, unreadable, and sometimes inefficient scripts. In this paper, we introduce *SYRql*, a dataflow language designed to process RDF data at a large scale. SYRql blends concepts from both SPARQL and existing big data languages. We formally define a closed algebra that underlies SYRql and discuss its properties and some unique optimisation opportunities this algebra provides. Furthermore, we describe an implementation that translates SYRql scripts into a series of MapReduce jobs and compare the performance to other big data processing languages.

1 Introduction

Declarative query languages have been a corner stone of data management since the early days of relational databases. The initial proposal of relational algebra and relational calculus by Codd [8] was shortly followed by other languages such as SEQUEL [7] (predecessor of SQL) and QUEL [34]. Declarative languages simplified programming and reduced the cost of creation, maintenance, and modification of software. They also helped bringing the non-professional user into effective communication with a database. Database languages design continued to be an active area of research and innovation. In 2008, the Claremont Report on Database Research identified declarative programming as one of the main research opportunities in the data management field [2].

There is, indeed, a large number of examples of publications describing design and implementation of query languages that embed queries in general purpose programming languages [19, 37, 30], for semi-strucutred data [25, 1], for Semantic

Web data [12, 3], for graphs [6, 17] and for network analysis [10, 27, 26] to name a few. Furthermore, the recent big data movement resulted in a surge of activity on layering declarative languages on top of distributed computation platforms. Examples include PIG Latin from Yahoo [20], DryadLINQ from Microsoft [39], Jaql from IBM [4], HiveQL [35] from Facebook and Meteor/Sopremo [13]. This paper focuses on declarative languages for large Semantic Web data represented in RDF.

In fact, there has been a significant growth in the available RDF data and distributed systems have been utilised to support larg-scale processing of the RDF data [36, 15, 16]. Nevertheless, the surge of analytics languages was not reflected in the Semantic Web realm. To analyse large RDF datasets, users are left mainly with two options: using SPARQL [12] or using an existing non-RDF-specific big data language.

SPARQL is a graph pattern matching language that provides rich capabilities for slicing and dicing RDF data. The latest version, SPARQL 1.1, supports also aggregation and nested queries. Nevertheless, the pure declarative nature of SPARQL obligates a user to express their needs in a single query. This can be unnatural for some programmers and challenging for complex needs [18, 11]. Furthermore, SPARQL evaluation is known to be costly [22, 29].

The other alternative of using an existing big data language such as Pig Latin or HiveQL has also its own limitations. These languages were designed for tabular data mainly, and, consequently, using them with RDF data commonly results in verbose, unreadable, and sometimes inefficient scripts. For instance, listings 1.1 and 1.2 show a basic SPARQL graph pattern and an equivalent Pig Latin script, respectively. Listing 1.2 has double the number of lines compared to listing 1.1 and is, arguably, harder to read and understand.

Listing 1.1: SPARQL basic pattern

```
?prod a :PoductType .
?r :reviewFor ?prod .
?r :reviewer ?rev
```

Listing 1.2: Corresponding Pig Latin script

```
rdf = LOAD 'data' USING PigStorage(' ') AS (S,P,O);
SPLIT rdf INTO reviewers IF P = ':reviewer',
      reviews IF P = ':reviewFor',
      prods IF P = 'a' and O = 'ProductType';
tmp1 = JOIN prods BY S, reviews BY O;
tmp2 = JOIN tmp BY reviews::S, reviewers BY S;
```

In this paper we present SYRql, a declarative dataflow language that focuses on the analysis of large-scale RDF data. Similar to other big data processing languages, SYRql defines a small set of basic operators that are amenable to parallelisation and supports extensibility via user-defined custom code. On the other hand, SYRql adopts a graph-based data model and supports pattern matching as in SPARQL.

SYRql could not be based on SPARQL Algebra [22] as this algebra is not fully composable. The current SPARQL algebra transitions from graphs (i.e. the initial inputs) to sets of bindings (which are basically tables resulting from pattern matching). Subsequently, further operators such as Join, Filter, and Union are applied on sets of bindings. In other words, the flow is partly "hard-coded" in the SPARQL algebra and a user cannot, for instance, apply a pattern

matching on the results of another pattern matching or "join" two graphs. In a dataflow language, the dataflow is guided by the user and cannot be limited to the way SPARQL imposes. We, therefore, define a new algebra that underpins the SYRql language. In particular, this paper provides the following contributions:

- We define the syntax and semantics of a compositional RDF algebra (sections 2.2). The algebra achieves composability by always pairing graphs and bindings together. We relate the defined algebra to SPARQL algebra (section 2.3) and report some of its unique properties that can be useful for optimising evaluation (section 2.4).
- We describe SYRql, a dataflow language based on the defined algebra (section 3.1). An open source implementation of SYRql that translates scripts into a series of MapReduce jobs is also described (section 3.2).
- We present a translation of an existing SPARQL benchmark into a number of popular big data languages (namely Jaql, HiveQL, and Pig Latin). The performance of equivalent SYRql scripts is compared with the other big data languages (section 4). The comparable results that SYRql implementation showed are encouraging giving its recency relative to the compared languages. Nevertheless, a number of improvements are still needed to ensure a good competitiveness of SYRql.

2 RDF Algebra

The goal of this algebra is to define operators similar to those defined in SPARQL algebra but that are fully composable. To achieve such composability, the algebra operators input and output are always a pair of a graph and a corresponding table. We next provide the formal definitions and description.

2.1 Preliminaries

We use $\mathbb{N}$ to denote the set of all natural numbers. We assume the existence of two disjoint infinite sets: $\mathcal{U}$ (URIs) and $\mathcal{L}$ (literals). The set of all terms is denoted by $\mathcal{T}$ (i.e. $\mathcal{T} = \mathcal{U} \cup \mathcal{L}$). We also assume that both $\mathcal{U}$ and $\mathcal{L}$ are disjoint from $\mathbb{N}$. An RDF triple[3] is a triple $(s, p, o) \in \mathcal{U} \times \mathcal{U} \times \mathcal{T}$. In this triple, s is the subject, p is the predicate and o is the object. An RDF graph is a set of RDF triples. We use $\mathcal{G}$ to refer to the set of all RDF graphs. Furthermore, we assume the existence of the symbol '?' such that $? \notin \mathcal{T}$ and define a triple pattern as any triple in $(\mathcal{T} \cup \{?\}) \times (\mathcal{T} \cup \{?\}) \times (\mathcal{T} \cup \{?\})$.

Definition 1 *A binding is a sequence of RDF terms (URIs or literals).*

Bindings are used to represent results of some operator over RDF graphs. Notice that our notion of binding is similar to that of SPARQL. However, in SPARQL a binding is a function that maps variable names to values. Our definition of binding obviates the need for variable names by using an ordered

[3] We only consider *ground* RDF graphs and therefore we do not consider blank nodes.

sequence. A common way to represent a set of bindings is by using a table. We use subscript to access binding elements based on their position in the sequence (i.e. The *ith* element of a binding S is S_i). The length of a binding S is denoted as $|S|$ and the empty binding is denoted as $()$ (notice that $|()| = 0$).

The concatenation of two bindings $S = (a_1, a_2, ..., a_n)$ and $T = (b_1, b_2, ..., b_m)$ is the binding $(a_1, a_2, ..., a_n, b_1, b_2, ..., b_m)$. We denote concatenation by a dot (i.e. $S \,.\, T$).

2.2 RDF Algebra Expressions

Syntax An RDF expression e is defined inductively as follows:

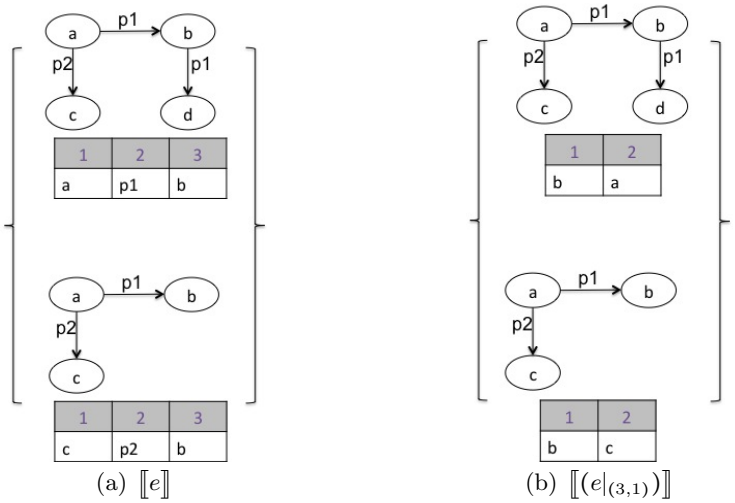
1. **Atomic**: if g is an RDF graph (i.e. $g \in \mathcal{G}$) then g is an RDF expression.
2. **Projection**: if e is an RDF expression and $(a_1, ..., a_n)$ is a sequence of natural numbers, then $(e|_{(a_1,...,a_n)})$ is an RDF expression. For example $(e|_{(4,2)})$ is a projection RDF expression.
3. **Extending bindings**: if e is an RDF expression, h is an n-ary function that takes n RDF terms and produces an RDF term (i.e. $h : \mathcal{T}^n \to \mathcal{T}$) and $a_1, ..., a_n$ is a sequence of natural numbers then
   $(e \ \oplus_{(a_1,...,a_n)} \ h)$ is also an RDF expression.
4. **Extending graphs**: if e is an RDF expression, a_1, a_2, a_3 are three natural numbers or RDF terms (i.e. $a_1, a_2, a_3 \in \mathcal{T} \bigcup \mathbb{N}$) then
   $(e \ \oplus \ (a_1, a_2, a_3))$ is also an RDF expression.
5. **Triple pattern matching**: If e is an RDF expression and t is a triple pattern then $(e[t])$ is also an RDF expression.
6. **Filtering**: if e is an RDF expression, $a, b \in \mathbb{N}$ and $u, v \in T$ then the following are valid RDF expressions:
   $(e[\, a \ \theta \ b \,])$, $(e[\, a \ \theta \ u \,])$ and $(e[\, u \ \theta \ v \,])$
   Where θ is $=$, $\neq$, $<$ or $\leq$. For example, $(e[\, 1 \leq 2])$ and $(e[\, 1 =$"label" $])$ are two filtering RDF Expressions.
7. **Cross product**: if e_1 and e_2 are RDF expressions, then so is $(e_1 \times e_2)$.
8. **Aggregation**: we define aggregate functions that take a set of terms and return a single value[4]. Therefore, the signature of an aggregate function is $f : 2^T \to \mathbb{N}$. If e is an RDF expression, a and b are two natural numbers and f is an aggregate function then $(e \, \langle a, f, b \rangle)$ is an RDF expression.

Semantics We now define the semantics of the previous expressions. For each expression e the value of it is denoted as $[\![e]\!]$. The value is always a set of pairs of a graph and a binding. The graph and binding components of $[\![e]\!]$ are, respectively, denoted as $[\![e]\!].g$ and $[\![e]\!].b$. To depict the values in this paper, we denote each pair by drawing a graph and a table close to each other. The table represents a binding and uses the order of elements in the binding as columns headers. The set of pairs that constitute an expression value are surrounded by curly brackets.

[4] For simplicity of the presentation here, we restrict aggregate functions to those that take a set of single values and return an integer. Generalising this is straightforward.

In the figures, sub-figure *(a)* is the input while sub-figure *(b)* shows the result of applying an operator to the input (see figure 1 for an example).

1. **Atomic**: $[\![g]\!] = \{(g, ())\}$
   The value of an atomic expression gives an empty binding.
2. **Projection**: $[\![(e|_{(a_1,...,a_n)})]\!] = \{(g, S) \,|(g, S') \in [\![e]\!], S = (S'_{a_1}, ..., S'_{a_n})\}$



(a) $[\![e]\!]$ (b) $[\![(e|_{(3,1)})]\!]$

Fig. 1: Projection example

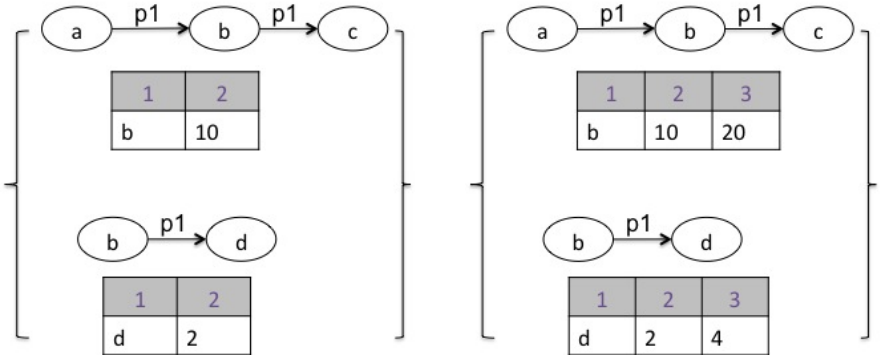
Projection allows choosing a sub-sequence of the bindings while leaves the graph component in each pair unaffected. Figure 1 provides an example of a projection expression value.

3. **Extending bindings**:

$$[\![(e \oplus_{(a_1,...,a_n)} h)]\!] = \{(g, S \,.\, (h(S_{a_1}, ..., S_{a_n}))) \,|(g, S) \in [\![e]\!]\}$$

These expressions allow extending the binding with a new value that is calculated based on existing values in the binding. See Figure 2 for an example. Notice that h can be viewed as a Skolem function arising from the quantification $\forall S_{a_1} \forall S_{a_2} ... \forall S_{a_n} \exists c : c = h(S_{a_1}, ..., S_{a_n})$

4. **Extending graphs**: We use the convention that $S_a = a$ for some binding S and a term $a \in \mathcal{T}$. Notice that $\mathcal{T}$ and $\mathbb{N}$ are disjoint and therefore the previous convention does not cause any ambiguity.
   $[\![(e \oplus (a_1, a_2, a_3))]\!] = \{(g \cup \{(S_{a_1}, S_{a_2}, S_{a_3})\}, S) \,|(g, S) \in [\![e]\!]\}$
   These expressions are similar to the extending bindings expressions defined before but they allow extending the graph. An example evaluation of such expression can be seen in Figure 3.

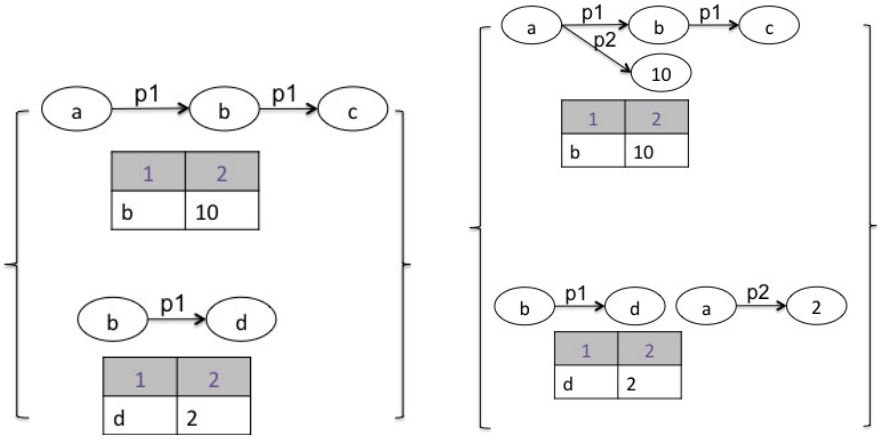


(a) $[\![e]\!]$

(b) $[\![(e \oplus_{(2)} h_{\times 2})]\!]$ Notice that $h_{\times 2}$ is a function that doubles its input (i.e. $h_{\times 2}(x) = x \times 2$

Fig. 2: Defining a new variable example (extending bindings)

5. **Triple pattern matching**:
   We discuss each possible triple pattern separately assuming $s, p, o \in \mathcal{T}$
   $[\![(e[\,(s,p,o)\,])]\!] = \{(\,\{(s,p,o)\}, (\,)\,) \,|\exists (g,S) \in [\![e]\!] \wedge (s,p,o) \in g\}$
   $[\![(e[\,(s,p,?)\,])]\!] = \{(\,\{(s,p,o)\}, (o)\,) \,|\exists (g,S) \in [\![e]\!] \wedge (s,p,o) \in g\}$
   $[\![(e[\,(s,?,o)\,])]\!] = \{(\,\{(s,p,o)\}, (p)\,) \,|\exists (g,S) \in [\![e]\!] \wedge (s,p,o) \in g\}$
   $[\![(e[\,(s,?,?)\,])]\!] = \{(\,\{(s,p,o)\}, (p,o)\,) \,|\exists (g,S) \in [\![e]\!] \wedge (s,p,o) \in g\}$
   $[\![(e[\,(?,p,o)\,])]\!] = \{(\,\{(s,p,o)\}, (s)\,) \,|\exists (g,S) \in [\![e]\!] \wedge (s,p,o) \in g\}$
   $[\![(e[\,(?,p,?)\,])]\!] = \{(\,\{(s,p,o)\}, (s,o)\,) \,|\exists (g,S) \in [\![e]\!] \wedge (s,p,o) \in g\}$
   $[\![(e[\,(?,?,o)\,])]\!] = \{(\,\{(s,p,o)\}, (s,p)\,) \,|\exists (g,S) \in [\![e]\!] \wedge (s,p,o) \in g\}$
   $[\![(e[\,(?,?,?)\,])]\!] = \{(\,\{(s,p,o)\}, (s,p,o)\,) \,|\exists (g,S) \in [\![e]\!] \wedge (s,p,o) \in g\}$
   Triple pattern matching expressions filter graphs to only triples matching the provided pattern and introduces the corresponding bindings. A key difference from SPARQL pattern evaluation is retaining the matching triples in addition to the bindings. Figure 4 shows an example. Notice that a triple pattern matching expression yields a graph with only one triple and eliminates previous bindings. Notice also that one can still apply further pattern matching on the results, something that is not possible in SPARQL.
6. **Filtering**:
   $[\![(e[\,a\ \theta\ b\,])]\!] = \{(g,S) \in [\![e]\!] \,|S_a\ \theta\ S_b\}$
   $[\![(e[\,a\ \theta\ u\,])]\!] = \{(g,S) \in [\![e]\!] \,|S_a\ \theta\ u\}$
   $[\![(e[\,u\ \theta\ v\,])]\!] = \begin{cases} [\![e]\!] & \text{if } u\ \theta\ v \\ \phi & \text{Otherwise} \end{cases}$
7. **Cross product**:
   $[\![(e_1 \times e_2)]\!] = \{\ (g_1 \cup g_2, S\ .\ T) \,|(g_1,S) \in [\![e_1]\!] \wedge (g_2,T) \in [\![e_2]\!]\}$
   See figure 5 for an example.
8. **Aggregation:**
   the expression $(e\,\langle a, f, b\rangle)$ groups by the a^{th} element in the binding, then



(a) $[\![e]\!]$ (b) $[\![(e \oplus (a, p2, 2))]\!]$

Fig. 3: Defining a new triple (extending graphs)

apply the aggregate function f on the values of the b^{th} element in each group. $[\![(e\,\langle a, f, b\rangle)]\!] = \{(\phi, \{k, f(\{S_b \,|\exists (g, S) \in [\![e]\!] \wedge S_a = k\})\})\}$
See Figure 6 for an example and notice that the resulting RDF graphs are empty (absence of graph in the figure indicates empty graph component).

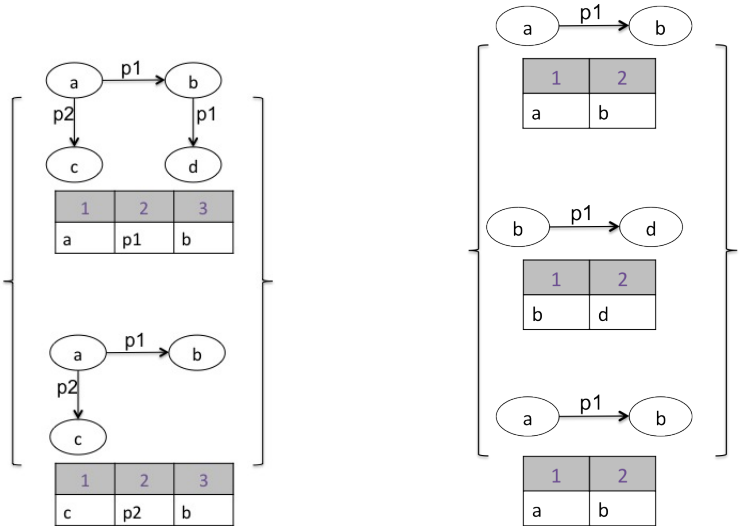
2.3 Relationship to SPARQL

Lemma 1. *RDF Algebra expressions can express SPARQL 1.1 basic graph patterns with filters, aggregations and assignments.*

Proof. SPARQL filters, aggregation and assignments can be directly mapped to "filtering", "aggregation" and "extending bindings" expressions in RDF Algebra. SPARQL individual triple patterns can be expressed by "triple pattern matching" expressions. Basic graph patterns in SPARQL imply a join on common variables among individual triple patterns. These expressions can be expressed by a sequence of "cross products" and filtering "expressions" in the same way that natural join is defined in relational algebra. □

We provide next a couple of example SPARQL queries along with their equivalent RDF Algebra expressions:

- The SPARQL query: `SELECT ?s ?v WHERE { ?s :p ?o . ?o :p2 ?v }`
  evaluated on the RDF graph g is equivalent to the expression:
  $((((g[(?, :p, ?)]) \times (g[(?, :p2, ?)]))[2 = 3])|_{(1,4)})$
- The SPARQL query:
  `SELECT ?s ?z WHERE { ?s :p ?o . ?o :p2 ?v BIND(?v * 1.1) AS ?z }`
  evaluated on the RDF graph g is equivalent to the expression:
  $((((((g[(?, :p, ?)]) \times (g[(?, :p2, ?)]))[2 = 3]) \oplus_{(4)} \times_{1.1})|_{(1,5)})$
  Where $\times_{1.1}(x) = x \times 1.1$



(a) $[\![e]\!]$ (b) $[\![(\, e[\, (?, p1, ?)\,]\,)]\!]$

Fig. 4: Triple pattern matching example

2.4 Algebraic properties

Algebraic laws are important for query optimisation. RDF Algebra shares some operators with SPARQL algebra and therefore related properties and laws defined in SPARQL algebra carry along. We focus here on triple patterns properties that are unique to our algebra. First, we define a partial ordering relationship between triple patterns.

Definition 2 $\forall x, y \in \mathcal{T} \bigcup \{?\} : x \preceq y$ *iff one of the following holds:*

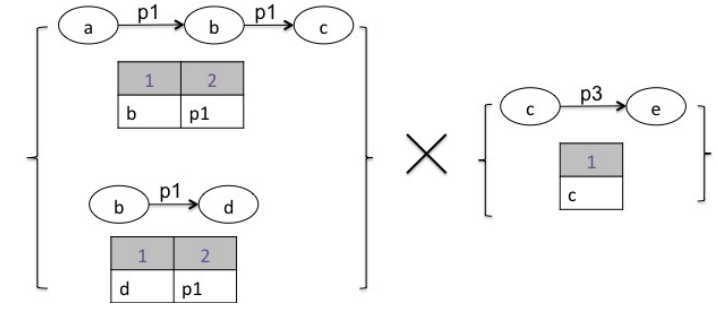
- *Both* x *and* y *are* $?$.
- x *and* y *are equal RDF terms (i.e.* $x, y \in \mathcal{T} \wedge x = y$*).*
- x *is a term and* y *is* $?$ *(i.e.* $x \in \mathcal{T} \wedge y = ?$*).*

We generalise $\preceq$ to triple patterns.

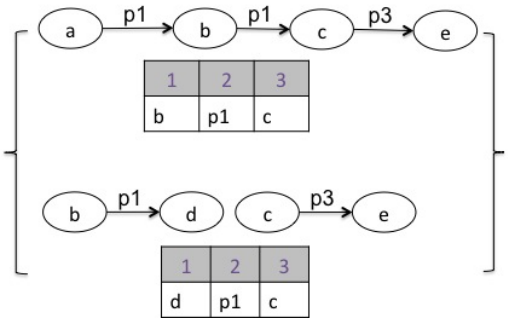
Definition 3 *For two triple patterns* (x_1, x_2, x_3) *and* (y_1, y_2, y_3) *we say that* $(x_1, x_2, x_3) \preceq (y_1, y_2, y_3)$ *iff* $x_1 \preceq y_1$, $x_2 \preceq y_2$, *and* $x_3 \preceq y_3$.

The defined partial ordering relationship between triple patterns ($\preceq$) can be thought of as a "more specific" relationship. The following list contains a number of algebraic properties that use this relationship. We also highlight potential optimisation opportunities of each of these algebraic property.

1. $[\![((e[t_1])[t_2])]\!].g = [\![(e[t_1])]\!].g$ if $t_1 \preceq t_2$
   Applying a less specific triple pattern does not change the resulting graph. It can nevertheless change the binding. For example,
   $[\![((e[(?, :p, :o)])[(?, ?, :o)])]\!].g = [\![(e[(?, :p, :o)])]\!].g$



(a) $[\![e_1]\!] \times [\![e_2]\!]$



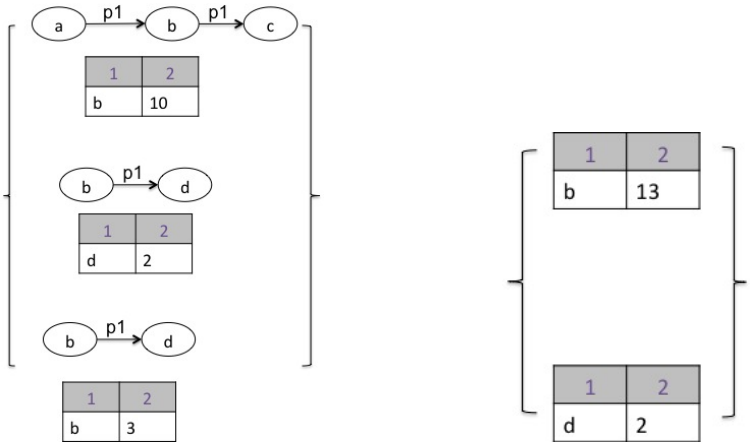
(b) $[\![(e_1 \times e_2)]\!]$

Fig. 5: Cross product example

2. $[\![(e[t_1])]\!] = [\![((e[t_2])[t_1])]\!]$ if $t_1 \preceq t_2$
   Therefore to calculate the results of matching expression e to the pattern t_1 one can instead try matching t_1 against the results of matching t_2 against e. This can be advantageous if the $e[t_2]$ is "cached".

3. $[\![((e_1 \times e_2)[t])]\!] = [\![(((e_1[t]) \times (e_2[t]))[t])]\!]$
   More generally $[\![((e_1 \times e_2)[t])]\!] = [\![(((e_1[t_1]) \times (e_2[t_2]))[t])]\!]$ for all t_1, t_2 such that $t \preceq t_1$ and $t \preceq t_2$. This can cut down the cost of a cross product between two expressions by substituting them with further matched expressions.

The list above is not comprehensive by any means. Further study of other algebraic properties of triple patterns is one of our current research focus. We believe that studying this "triple algebra" can yield fruitful results that can further be applied in tasks like caching RDF query results, views management and query results re-use.



(a) $[\![e]\!]$ (b) $[\![(e\,\langle 1, SUM, 2\rangle)]\!]$

Fig. 6: Aggregate expression example

3 A Data flow Language for RDF

3.1 SYRql Language

SYRql is a dataflow language that is grounded in the algebra defined before. A SYRql script is a sequence of statements and each statement is either an assignment or an expression. The core set of operators in SYRql are those defined by the algebra in sections 2.2.

The syntax of SYRql borrows the use of "– >" syntax to explicitly show the data flow. According to the designers of Jaql, the "– >" syntax, inspired by the Unix pipes, makes scripts easier to read and debug [4]. It allows eliminating the need for defining variables (as in PIG) or for a WITH clause (as in SQL) in each computational step. It is worth mentioning that Meteor [13] language dropped the pipe notation of Jaql to support operators with multiple inputs and outputs. In SYRql, operators with multiple inputs or outputs are not common and therefore we decided to adopt the pipe syntax. However, SYRql does support multi-input operators such as multi-way joins.

Pattern matching in SYRql uses identical syntax to basic graph patterns of SPARQL. SPARQL syntax for patterns is intuitive, concise and well-known to many users in the Semantic Web field. We hope that this facilitates learning SYRql for many users.

Listing 1.3 shows an example SYRql script that performs pattern matching, filtering, and aggregation. Notably, line 10 in the script provides an example of composability that is not directly available in SPARQL. In line 10, a pattern matching is applied to the results of another pattern matching. We believe that such capabilities are useful for complicated scripts, specifically for exploratory

tasks, and for reusing previous scripts as well as previously computed results. Further description and examples of the SYRql language is available online[5]. The BNF grammar defining the syntax can also be found on SYRql website.

Listing 1.3: Example SYRql script

```
$rdf = load('hdfs://master:9001/bsbm20k');

$janReviewers = $rdf -> pattern('?review rev:reviewer ?reviewer .
                                 ?review dc:date ?date .
                                 ?reviewer bsbm:country ?cntry .')
              -> filter(?date >= '2008-01-01' &&  ?date < '2008-02-01');

$janReviewers -> group by ?cntry into janReviewersCount:count(?review);

$irelandJanReviewers = $janReviewers -> pattern('?rev bsbm:country :IE');
```

3.2 SYRql Implementation

The current implementation[6] translates SYRql scripts into a series of MapReduce [9] jobs. We use Java and Apache Hadoop 2 API[7] in our implementation.

Data representation: JSON[8] is used for internal representation of the data. Particularly, we use JSON arrays for bindings and JSON-LD [31] to represent graphs. JSON-LD is a recent W3C recommended serialisation of RDF. It has attracted good adoption so far and this can be expected to grow. Consequently, by using JSON-LD a large amount of RDF data can be directly processed using SYRql. Furthermore, existing works such as NTGA [24] have demonstrated the benefit of manipulating RDF graphs as "groups of triples" that share the same subject. In this work, we utilize JSON-LD's ability to represent star subgraphs as single JSON objects, thus eliminating the need for joins when evaluating star-join queries. This particular way of encoding RDF in JSON-LD is referred to as the flattened document form[9] and it is the format used in SYRql implementation. Moreover, we provide a MapReduce implementation that converts RDF data serialised as N-Triple format[10] into flattened JSON-LD.

Parsing, compiling and evaluation: We use ANTLR [11] to parse SYRql scripts and build the abstract syntax tree. Each node in the tree represents an expression and the children of the node are its inputs. For triple matching expressions, triple patterns are grouped by subject to utilise the data stored as star-structured subgraphs, thus reducing the number of required joins. The tree is then translated into a directed acyclic graph (DAG) of MapReduce jobs. Sequences of expressions that can be evaluated together are grouped into a single MapReduce job. Finally, the graph is topologically sorted and the MapReduce

[5] https://gitlab.insight-centre.org/Maali/syrql-jsonld-imp/wikis/home
[6] https://gitlab.insight-centre.org/Maali/syrql-jsonld-imp
[7] http://hadoop.apache.org/
[8] http://json.org
[9] http://www.w3.org/TR/json-ld/#flattened-document-form
[10] http://www.w3.org/TR/2014/REC-n-triples-20140225/
[11] http://www.antlr.org/

jobs are scheduled to execute on the cluster. It is worth mentioning that for join expressions we implemented the optimised repartition join [14].

4 Evaluation

We conducted a performance evaluation of SYRql. Our goal of this evaluation is two-fold:

- Compare performance of SYRql to other popular alternatives, namely Jaql, Pig Latin, and HiveQL. Our thesis is that SYRql's features and syntax can improve user productivity when processing RDF data and help generating scripts that are easier to understand and debug. Therefore, we want to measure the loss in performance, if any, that an early adopter of the language might have to tolerate.
- In the same spirit of Pig Mix[12] that is developed as part of Pig, we want this benchmark to measure performance on a regular basis so that the effects of individual code changes on performance could be understood.

We based our benchmark on the Berlin SPARQL Benchmark (BSBM) [5] that defines an e-commerce use case. Specifically, we translated a number of queries in the BSBM Business Intelligence usecase (BSBM BI)[13] into equivalent programs in a number of popular big data languages. In particular, we provide programs in the following languages:

Jaql A scripting language designed for Javascript Object Notation (JSON).
Pig Latin A dataflow language that provides high-level data manipulation constructs that are similar to relational algebra operators.
HiveQL A declarative language that uses a syntax similar to SQL.

The programs were written by the authors of this paper who have intermediate to high expertise in those languages. We believe that they reflect what an interested user would write given a reasonable amount of time. We evaluated four queries from BSBM BI that cover all core operators i.e., filters, patterns, joins and aggregation. We plan to evaluate other benchmark queries as part of the near future work.

4.1 Setup

Environment: The experiments were conducted on VCL[14], an on-demand computing and service-oriented technology that provides remote access to virtualised resources. Nodes in the clusters had minimum specifications of single or duo core Intel X86 machines with 2.33 GHz processor speed, 4G memory and running

[12] `https://cwiki.apache.org/confluence/display/PIG/PigMix`
[13] `http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/BusinessIntelligenceUseCase/index.html`
[14] `https://vcl.ncsu.edu/`

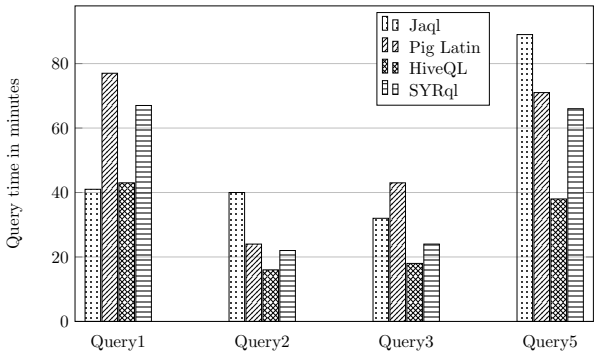


Fig. 7: Query processing times

Red Hat Linux. We used a 10-node cluster and the following software versions: Apache Hadoop 2.3.0, Jaql 0.5.1, Pig 0.12.1, and Hive 0.12.0.

Dataset and queries: We generated BSBM data for 400K products in N-triple format. The size of the data was about 35GB containing approximately 140 million triples. As mentioned before, the queries are the scripts corresponding to BSBM BI queries.

4.2 Results & Discussion

Figure 7 shows corresponding response time for each of the scripts. Jaql and SYRql required pre-processing of the data to convert the N-Triple RDF data into JSON-LD. The conversion, which took 40 minutes, is only needed once and then the data can be used by all the queries. In general, our SYRql implementation shows encouraging results. It is comparable to the times that Jaql and Pig Latin showed. However, Hive outperformed all the other four systems significantly. The superior performance of Hive was also reported in [33].

Both SYRql and Jaql can evaluate triple patterns that share the same subject together due to their underlying data model and their use of JSON-LD. Pig, on the other hand, evaluates each triple pattern individually and then joins the results. We believe that this is the main reason for the better performance that Jaql and SYRql generally achieved in comparison to Pig despite the maturity and the larger developers community that Pig enjoys.

Examining the generated MapReduce jobs, it was observed that Jaql and SYRql generated similar sequences of jobs. However, SYRql computes results for both graphs and bindings as specified in the underlying algebra. This results in more computation to be done. Nevertheless, separating bindings and graphs helped speeding up some operators through reading and processing less data. For example, filters operate only on the bindings and do not need to process the graphs. Similarly, joins are calculated based on the bindings and then joining the corresponding graphs is a simple union of the matched graphs (see the semantics of the cross product operator in Section 2.2).

We speculate that the superior performance of Hive is mostly due to its efficient join performance. Hive join optimisations such as conversion to map-joins can be applicable when the joining relations are small in size. Additionally, for grouping queries, Hive computes map-side partial aggregations using a Combiner, an optimisation we plan to integrate in our next version.

In summary, SYRql implementation showed a good performance that will hopefully encourage users to try it. Moreover, SYRql scripts contained 50% less lines than Pig scripts and 42% less than Jaql scripts. Evaluating the language ease-of-use and readability is planned for future work.

5 Related Work

A large number of declarative languages were introduced recently as part of the big data movement. These languages vary in their programming paradigm, and in their underlying data model. Pig Latin [20] is a dataflow language with a tabular data model that also supports nesting. Jaql [4] is a declarative scripting languages that blends in a number of constructs from functional programming languages and uses JSON for its data model. HiveQL [35] adopts a declarative syntax similar to SQL and its underlying data model is a set of tables. Other examples of languages include Impala[15], Cascalog[16], Meteor [13] and DryadLINQ [39]. [33] presented a performance as well as a language comparison of HiveQL, Pig Latin and Jaql. [28] also compared a number of big data languages but focuses on their compilation into a series of MapReduce jobs.

In the semantic web field, SPARQL is the W3C recommended querying language for RDF. A number of extensions to SPARQL were proposed in the literature to support search for semantic associations [3], and to add nested regular expressions [23] for instances. However, these extensions do not change the pure declarative nature of SPARQL. There are also a number of non-declarative languages that can be integrated in common programming languages to provide support for RDF data manipulation [21, 32]. In the more general context of graph processing languages, [38] provides a good survey.

6 Conclusions & Future Work

RDF Algebra, a fully composable algebra that is similar to SPARQL algebra, was presented in this paper. The composabilty of RDF Algebra is obtained by pairing graphs and bindings together. A number of unique algebraic properties were presented. Further study of these properties is at the top of our research agenda. We believe that this is a fruitful direction that can have impact in a number of related research problems.

Based on RDF Algebra, we presented SYRql, a dataflow language for large scale processing of RDF data. An implementation of SYRql on top of MapReduce platform was described. This paper also reported some initial results on a

[15] `https://github.com/cloudera/impala`

[16] `http://cascalog.org/`

performance comparison between SYRql implementation and other existing big data languages. Our future work includes refining SYRql syntax and improving its performance. In particular, we plan to provide an implementation that runs SYRql scripts on top of Apache Spark[17] and to use binary representation of the JSON-LD RDF data instead of the textual one currently used.

Acknowledgements. This publication has emanated from research supported in part by a research grant from Science Foundation Ireland (SFI) under Grant Number SFI/12/RC/2289. Fadi Maali is funded by the Irish Research Council, Embark Postgraduate Scholarship Scheme. We thank Aidan Hogan, Marcel Karnstedt and Richard Cyganiak for valuable discussions.

References

1. Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L Wiener. The lorel query language for semistructured data. *International journal on digital libraries*, 1997.
2. Rakesh Agrawal et al. The Claremont Report on Database Research. *SIGMOD Rec.*, 2008.
3. Kemafor Anyanwu and Amit Sheth. P-queries: enabling querying for semantic associations on the semantic web. In *WWW*, 2003.
4. Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Y. Eltabakh, Carl-Christian Kanne, Fatma Özcan, and Eugene J. Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *PVLDB*, 2011.
5. Christian Bizer and Andreas Schultz. The Berlin SPARQL Benchmark. *IJSWIS*, 2009.
6. Peter Buneman, Mary Fernandez, and Dan Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *VLDB*, 2000.
7. Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A Structured English Query Language. In *SIGFIDET*, 1974.
8. E. F. Codd. A Data Base Sublanguage Founded on the Relational Calculus. In *SIGFIDET*, 1971.
9. Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.
10. Anton Dries, Siegfried Nijssen, and Luc De Raedt. A Query Language for Analyzing Networks. In *CIKM*, 2009.
11. Stefan Hagedorn and Kai-Uwe Sattler. Efficient Parallel Processing of Analytical Queries on Linked Data. In *OTM*, 2013.
12. Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language. W3C Recommendation 21 March 2013. http://www.w3.org/TR/sparql11-query/.
13. Arvid Heise, Astrid Rheinländer, Marcus Leich, Ulf Leser, and Felix Naumann. Meteor/Sopremo: An Extensible Query Language and Operator Model. In *BigData*, 2012.
14. Alex Holmes. *Hadoop In Practice*, chapter 4. Manning Publications Co., 2012.
15. Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 2011.

[17] `https://spark.apache.org/`

16. Ren Li, Dan Yang, Haibo Hu, Juan Xie, and Li Fu. Scalable RDF Graph Querying Using Cloud Computing. *J. Web Eng.*, 2013.
17. Yanhong A. Liu and Scott D. Stoller. Querying Complex Graphs. In *PADL*, 2006.
18. Fadi Maali and Stefan Decker. Towards an RDF Analytics Language: Learning from Successful Experiences. In *COLD*, 2013.
19. Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *SIGMOD*, 2006.
20. Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a Not-so-foreign Language for Data Processing. In *SIGMOD*, 2008.
21. Eyal Oren, Renaud Delbru, Sebastian Gerke, Armin Haller, and Stefan Decker. Activerdf: Object-oriented semantic web programming. In *WWW*, 2007.
22. Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. ISWC, 2006.
23. Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. nSPARQL: A navigational language for RDF. *Web Semantics: Science, Services and Agents on the World Wide Web*, 2010.
24. Padmashree Ravindra, HyeongSik Kim, and Kemafor Anyanwu. An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce. In *ESWC*, 2011.
25. J Robie, D Chamberlin, M Dyck, and J Snelson. Xquery 3.0: An xml query language, 2014. *Availab at:http://www.w3.org/TR/xquery-30/*, 2014.
26. Royi Ronen and Oded Shmueli. SoQL: A Language for Querying and Creating Data in Social Networks. In *ICDE, year = 2009,*.
27. Mauro San Martın, Claudio Gutierrez, and Peter T Wood. SNQL: A social networks query and transformation language. *AMW*, 2011.
28. Caetano Sauer and Theo Haerder. Compilation of query languages into mapreduce. *Datenbank-Spektrum*, 2013.
29. Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of sparql query optimization. In *ICDT*, 2010.
30. Daniel Spiewak and Tian Zhao. ScalaQL: language-integrated database queries for scala. In *Software Language Engineering*, pages 154–163. 2010.
31. Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, and Niklas Lindström. JSON-LD 1.0. W3C Recommendation 16 January 2014.
32. Steffen Staab. Liteq: Language integrated types, extensions and queries for rdf graphs. *Interoperation in Complex Information Ecosystems*, 2013.
33. Robert J Stewart, Phil W Trinder, and Hans-Wolfgang Loidl. Comparing High Level MapReduce Query Languages. In *APPT*. 2011.
34. Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. The Design and Implementation of INGRES. *ACM Trans. Database Syst.*, 1976.
35. Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang 0002, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a Petabyte Scale Data Warehouse Using Hadoop. In *ICDE*, 2010.
36. Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and Frank Harmelen. Scalable Distributed Reasoning Using MapReduce. In *ISWC*, 2009.
37. Limsoon Wong. Kleisli, a functional query system. *Journal of Functional Programming*, 2000.
38. Peter T. Wood. Query Languages for Graph Databases. *SIGMOD*, 2012.
39. Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: a System for General-purpose Distributed Data-parallel Computing Using a High-level Language. In *OSDI*, 2008.