Extending SPARQL for data analytic tasks

Julian Dolby[1] Achille Fokoue[1] Mariano Rodriguez Muro[1] Kavitha Srinivas[1]
Wen Sun[2]

[1] IBM Thomas J. Watson Research Center
{dolby,achille,mrodrig,ksrinivs}@us.ibm.com
[2] IBM Research China
sunwenbj@cn.ibm.com

Abstract. SPARQL has many nice features for accessing data integrated across different data sources, which is an important step in any data analysis task. We report on the use of SPARQL for two real data analytic use cases from the healthcare and life sciences domains, which exposed certain weaknesses in the current specification of SPARQL, specifically when the data being integrated is most conveniently accessed via RESTful services and in formats beyond RDF, such as XML. We therefore extended SPARQL with *generalized service*, constructs for accessing services beyond the SPARQL endpoints supported by `service`. For efficiency, our constructs support posting data, which is also not supported by `service`. We provide an open source implementation of this SPARQL endpoint in an RDF store called Quetzal, and evaluate its use in the two data analytic scenarios over real datasets.

1 Introduction

SPARQL, due to its minimal schema requirements and declarative graph queries, is ideally suited for the painful data integration steps that precedes any data analysis. However, when applying SPARQL to two data analytic use cases in the healthcare and life sciences domain, we uncovered roadblocks in its use: in our use cases, the data is most conveniently accessed via RESTful services and in formats beyond RDF, such as XML. The current SPARQL specification provides no construct to access such data; the existing federation construct, `service`, handles only static queries to other RDF endpoints, so we extend `service` to send data to and retrieve data from RESTful services. These services take and return solution bindings as do other SPARQL constructs, and so integrate seamlessly with SPARQL syntax, and preserve its declarative nature and opportunities for query optimization.[3]

The first use case is in the domain of drug safety, in particular Drug-Drug Interaction (DDI) predictions. Drug-Drug Interactions are a major cause of preventable adverse drug reactions (ADRs), causing a significant burden on the patients' health and the healthcare system [6]. It is widely known that clinical

[3] This is similar to what RDBMS do with foreign functions that are declared to be functional.

studies cannot sufficiently and accurately identify DDIs for new drugs before they are made available on the market. As a result, the only practical way to explore the large space of unknown DDIs is through in-silico prediction of drug-drug interactions. We have built a system called Tiresias [7] that takes in various sources of drug-related data and knowledge as inputs, and provides DDI predictions as outputs. After semantic integration of the input data, we computed several similarity measures between all the drugs, and used the resulting similarity metrics as features in a large-scale logistic regression model to predict potential DDIs. In order to gather comprehensive information about drugs and their relevant associated bio-entities, Tiresias originally relied on Drugbank for the data integration task of combining information from multiple sources such as Uniprot. Unfortunately, data integration performed by a data source such as Drugbank is static, and it becomes out-of-date after the release of a specific version. A second problem was that numerous tools (R, scala, Java) were used to create the pipeline for data integration in the system, making it difficult to manage the different pieces of the pipeline.

The second use case investigates the potential health consequences of rising air pollution in China, especially high PM2.5 levels in many major cities in China. A critical question is whether PM2.5 levels have any correlation with the onset of some particular diseases. To provide an answer to this question, we looked for a potential correlation between diagnosis for particular diseases in Electronic Medical Record (EMR) data of a hospital in Beijing and elevated PM2.5 levels available in public sources such as StateAir[4]. The second use case requires flexibility in being able to specify different time windows for the data, to examine possible correlations between air pollution and disease, which is not difficult to manage across different tabular data and statistical software, but it does tend to be static.

In the above two use cases, to address the staleness and management issues, we tried to leverage SPARQL to perform a more dynamic and complete integration of information coming directly from the most authoritative sources for the entities of interest (e.g., Uniprot for proteins and genes, Drugbank for direct properties of drugs and StateAir for PM2.5 levels). A difficulty using SPARQL is its lack of means to integrate computation, such as the similarity metrics computed for drug pairs in the first use case or the statistical correlations between pollution and diseases in the second case. Our extension allows these computation to be expressed as RESTful services and hence integrated cleanly into our overall queries.

First, as is evident in our two use cases, the data being integrated is often most conveniently accessed via RESTful services and in formats beyond RDF, such as XML, which the current SPARQL specification does not fully support. We therefore generalize SPARQL's existing federation construct, `service`, to refer to RESTful services, using HTTP POST or GET methods to post solution bindings to a service. We also specify general mechanisms to extract tuples from

[4] StateAir U.S. Department of State Air Quality Monitoring Program: http://www.stateair.net/web/mission/1/

data returned by the web service. This allows a SPARQL query to inject data from anywhere on the web directly into the query computation, which is a very useful ability in integrating disparate data. Our use of GET and POST allow RESTful services to be integrated unchanged. We define both service and table functions for web data ingestion, the distinction being that service functions send solution bindings singly and table functions post entire solutions. The latter is especially useful for more efficient federated computations, as we show in our use cases. Our service functions use RESTful services, as opposed to foreign function calls because these functions can provide language neutrality, statelessness, and insulation from operational details. Last, we added the construct of row numbers to SPARQL to allow batching of computations to external web services. We show how all these extensions operate in our use cases to dynamically inject data from a web service (e.g., from Uniprot) or include sophisticated computations on the data (e.g. computation of chemical similarity between drugs) into a query.

Overall, our core contributions in this paper are as follows:

1. We describe two use cases from the healthcare and life sciences domain, where we encountered problems initially in using SPARQL for data analytic tasks.
2. We introduce a minimal set of three extensions to SPARQL to broaden the scope of its use in data integration tasks.
3. We define the semantics of each extension formally, and describe extensions to the SPARQL algebra for them.
4. We provide a open source reference implementation of these extensions in a github project (Quetzal-RDF)[5] over relational and non-relational backends.
5. We discuss important lessons learnt that could be generalized to other applications.

The rest of this paper is organized into the following sections. Section 2 describes a simplified version of two real use cases for SPARQL as a data preparation tool, and we use them as examples for the extensions described in the paper. Section 3 presents our extensions to SPARQL, with a corresponding section 4 to define the algebra. Section 5 shows the use of these extensions in the 2 real use cases and discusses lessons learnt. Section 6 describes related work.

2 Motivating Examples

We present two motivating examples from health care and life sciences; the first involved drug-drug interactions and the second involved potential health effects of pollution.

2.1 Drug Drug Interactions

Our first motivating example is a simplified version of the DDI problem. For simplicity we describe the steps to determine drug similarity based on chemical structure and their biological functions in Figure 1:

[5] https://github.com/Quetzal-RDF/quetzal

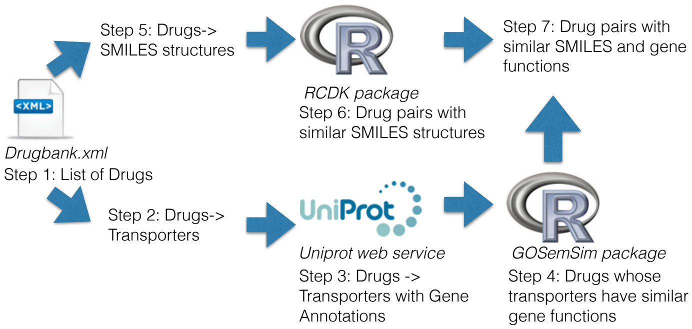

Fig. 1. Workflow for the DDI example

1. A list of drugs is obtained from Drugbank
2. Their transporter proteins are obtained from Drugbank
3. Transporter gene functions are obtained from Uniprot.[6]
4. Pairs with similar transport gene functions are found with a complex algorithm in an R package called GoSemSim [18].
5. Drug chemical structures in the standard SMILES format are obtained from Drugbank.
6. Pairs with similar structures are found with an R version of another complex algorithm implemented in the Chemistry Development Kit (CDK) [15].
7. These two sets of pairs are intersected to get similar drugs.

These steps map directly to the steps in our example query in Figure 2.[7] We divide the query into functions corresponding to each portion of the work flow. These functions are called with `bind`. Within the `select`, the functions are shown in the same order as the list above; however, the last step of intersecting the two sets of similar drugs is implicit in the join of `?drugName`, `?similarDrug` and `?similarityValue` triples obtained from steps 4 and 6.

The implementations of these steps illustrate aspects of our extensions,and will be given in Section 3.1. Each function is assumed to be published at the URL specified by the function. The `getDrug` function (step 1) illustrates a simple use of *service functions*, applying XPath to extract the list of drugs from the DrugBank XML file. The functions `getDrugChemicalStructure` (step 5) and `getDrugTransporters` (step 2) illustrate *table functions*, in which a service is called for all drugs at once using an HTTP POST to compute corresponding `?smiles` and `?transporter` values respectively. The function `getProteinGeneFunctions` (step 3) is once again a service, but we will show later how we batch requests to

[6] While gene functions for these proteins are specified in Drugbank as well, the most comprehensive and up-to-date annotation of gene functions is in UniProt.

[7] Note that translation of the workflow to the query can be done in multiple ways, we choose this one for illustration.

```
prefix drug: <http://www.drugbank.ca>
prefix xs: <http://www.w3.org/2001/XMLSchema>

function drug:getDrugNames ( -> ?drugName)
function drug:getDrugTransporters (?drugName -> ?drugName ?transporter)
function drug:getProteinGeneFunctions (?protein -> ?protein ?geneFunction)
function drug:getProteinGeneFunctionSimilarity
                          (?drugName ?geneFunction -> ?drugName ?similarDrug ?similarityValue)
function drug:getDrugChemicalStructure (?drugName -> ?drugName ?smile)
function drug:getDrugChemicalSimilarities
                                     (?drugName ?smile -> ?drugName ?similarDrug ?similarityValue)
select drugName, ?similarDrug where {
  BIND( drug:getDrugNames() AS (?drug))
  BIND( drug:getDrugTransporters(?drugName)
                                                   AS (?drugName ?transporter ) )
  BIND( drug:getProteinGeneFunctions(?transporter) AS (?transporter ?geneFunction) )
  BIND( drug:getProteinGeneFunctionSimilarity(?drugName ?geneFunction
                                                   AS (?drugName ?similarDrug ?similarityValue) )
  BIND( drug:getDrugChemicalStructure(?drugName) AS (?drugName ?smile) )
  BIND( drug:getDrugChemicalSimilarities(?drugName ?smile)
                                                   AS (?drugName ?similarDrug ?similarityValue) )
}
```

Fig. 2. Simplified Example Query

make it more efficient. The details are involved, and are described in Section 3.1. The other two steps, functions `getDrugChemicalSimilarities` (step 6) and `getProteinGeneFunctionSimilarities` (step 4) are also *table functions*: they are each invoked once on the entire set of tuples of drug names and SMILES (or gene functions) for a complex similarity computation.

2.2 Air Pollution and Health

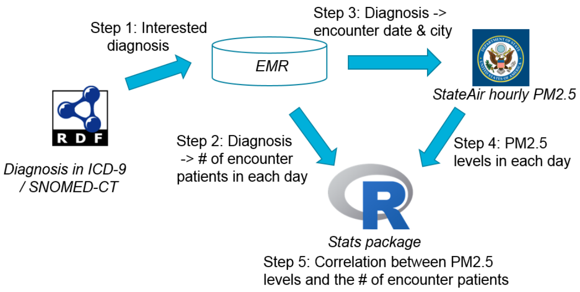

Fig. 3. Workflow for the air pollution and health example

Another example is due to rising air pollution levels, especially high PM2.5 levels in many major cities in China, where there is increasing attention to the impact of air pollution on people's health. A critical problem is to determine whether PM2.5 levels have any correlation with the onset of particular diseases. To study this problem, a possible workflow may contain the following steps as shown in Figure 3:

1. Get a set of diagnosis codes of interest.
2. The total number of the encountered patients with the diagnosis in each day is queried in the EMR data.
3. The date of the encounter and city of patients are obtained from the electronic medical record (EMR) data.
4. PM2.5 levels of the treated dates are found in public sources like StateAir
5. Correlations between PM2.5 levels and the number of encountered patients are calculated with the R statistics package.

Similar to the previous example, the above steps can be supported by using our proposed extensions. Particular diagnosis codes or disease codes can be selected from open data repositories. Then EMR datasets are queried to retrieve the target patients' historical encounter information and the PM2.5 levels for each encounter date. At step 5, R's statistics package can be used to calculate the Person's correlation between the PM2.5 levels and the patient encounter.

3 Extensions

In this section, we first motivate our language extensions based on our example use case. Then we present definitions of our functions.

3.1 Constructs

The first step of our DDI use case is to gather the drugs of interest, which we obtain from Drugbank, a standard drug information repository that is conveniently accessed as XML. That motivates our first extension, *service functions*, which provide the ability to get sets of solution bindings from an HTTP call; the result is expected to be in XML and is interpreted in a manner analogous to XMLTABLE in SQL using XPath. The functions are analogous to service calls in SPARQL; the function executes once for each solution binding and the URL can be computed as an expression.

```
function drug:getDrugNames
service drug:getDrugNames [] -> "//row" : "./drug"
```

In this case, the getDrugNames function invokes the service bound to the getDrugNames URL, and expects an XML result where the rows are obtained by applying the XPath expression `//row`, which gets every XML element with that name. Within each row, the `./drug` XPath expression is applied to get a column value. These functions are called using a BIND syntax, one call for each current solution binding; in this case, the result is a set of solution bindings, each having the name ?drug bound to a drug name returned from Drugbank.

```
bind(drug:getDrugNames() as (?drug))
```

Once the drug names have been obtained, the next steps for determining DDIs are to compute the *transporters* for each drug and its drug structure, called SMILES. These illustrate the need for the next extension: the *service function* interface used above is not suitable, since it would result in one call for each drug, which becomes expensive for large numbers of drugs. What we need is an interface that efficiently posts a set of solution bindings as a table, which we call *table functions.*

```
function drug:getDrugTransporters
table drug:getDrugTransporters
[?drug->?drug ?transporter] -> "//row" : "./drug" "./transporter"

function fn:getSMILES ( ?drug -> ?smiles )
table fn:getSMILES [ "funcData" -> post data ]
 -> "//x:row" :: "./x:drug" "xs:string" "./x:smiles" "xs:string"
```

When these functions are called, the ?drug element of each binding in the current solution is used to construct an XML document that is posted to the given URL. The result is a solution in which each binding has ?drug and ?transporter (or ?smiles respectively) elements. The key distinction with service functions is that the call is made once per solution rather than once per solution binding.

The motivation for our last extension comes from the fact that many services provide the capability to *batch* service requests. Continuing with our example, once we have the transporters for each drug, one way of looking for potential DDI is based on genes, using the gene annotations that have been associated with each transporter. These annotations are maintained in UniProt, where each transporter is an entity, and we query these annotations for each transporter. The standard UniProt website provides a query interface in which a UniProt identifier can be specified in the URL, and data about that entity returned in a variety of formats. The original service function from the example was defined as follows:

```
function drug:getProteinGeneFunctions (?protein -> ?gene ?id ?type)
service CONCAT("http://www.uniprot.org/uniprot/", CONCAT(?protein, ".xml"))
  [ ] -> "/up:uniprot/up:entry/up:dbReference" ::
    "./@id" "xs:string"
     "../accession[1]" "xs:string"
    "./@type" "xs:string"
```

But this interface is not suited to querying a large number of identifiers at once, and issuing hundreds of queries to a website will likely result in requests being simply denied. Fortunately, the European Bioinformatics Institute (EBI), part of the European Molecular Biology Laboratory (EBML), provides a flexible interface to freely-available life sciences data, including UniProt, that does allow querying identifiers in a batch[8]. The interface for calling this service is below; note that the URL itself is an expression in which the parameter ?ids becomes part of the URL.

```
function drug:getProteinGeneFunctionsGroup ( ?ids -> ?gene ?id ?type)
service CONCAT("http://www.ebi.ac.uk/...&id=", ?ids)
  [ ] xs-> "/uniprot/entry/dbReference" ::
     "./@id" "xs:string"
     "../accession[1]" "xs:string"
     "./@type" "xs:string"
```

To use this interface, we construct URLs that encode a collection of identifiers; the identifiers are encoded as a +-separated string in the query URL. We can easily express this with existing constructs in SPARQL; specifically the `group_concat` function and a group by clause. However, to group the transporters into chunks, we added one additional extension: the function `rowNumber` which is analogous to the SQL function. The `rowNumber` function simplifies grouping URLs into chunks to batch to a service request. To maintain the interface we had before, we encapsulate this mechanism in a SPARQL function as follows. Functions will be defined later, but essentially they work as in programming languages: ?transporter bindings are the parameter, and ?gene and ?id are results. Now the original query does not have to change.

```
function drug:getProteinGeneFunctions (?transporter -> ?gene ?id)
{ select (group_concat(?transporter; separator='+') AS ?ids) where
  {
    { select distinct ?transporter where
      {
        BIND( fn:getDrugBankNames() AS ( ?drug ) )

        BIND( fn:getTransporters( ?drug )
          AS ( ?drug ?transporter ) )
      }
    group by (xsd:int(rowNumber() / 30)) }
  }

  BIND( drug:getProteinGeneFunctionsGroup( ?ids )
    AS ( ?geneFunction ?t2 ?type ) )

}
```

As we show in the query fragment above, since the length of URLs is limited, we need a series of `ids` strings, each of which has up to 30 transporter ids. This

[8] http://www.ebi.ac.uk/Tools/dbfetch/dbfetch

set of `ids` strings then are used in a series of requests of the EBI Web service. The call to getProteinGeneFunctions needs to be a get operation due to the nature of the the dbfetch URL, but it uses the groups of transporters computed above to minimize the number of get transactions it makes.

Thus, our extensions allow us to flexibly access services using both GET and POST mechanisms, and give us the power of SPARQL to construct complex URLs.

3.2 Functions

We introduce a *function* and we add a form of BIND to represent calls.

Definition 1 (Function call) *We extend BIND in SPARQL to express function calls: we allow BIND to specify the function uri and the input and output variables:*

$$\text{BIND } \{\textit{output var}\}^* \text{ AS uri}(\{\textit{input var}\}^*)$$

Definition 2 (Function) *A function encapsulates an operation; it is identified by a uri and specifies sets of input and output variables. They are specified by a function declaration as follows:*

`function` *uri* `(` *input var** `->` *output var** `)`

There are multiple types of function within this interface; each type of function extends the definition above with an implementation specification. For function declarations as above that do not have a definition, the body of the function must be obtainable using the uri that defines the function. This extension to SPARQL is analogous to 'include' in languages such as C or 'import' in Java, but it is more powerful because it incorporates URIs as a mechanism for providing function bodies rather than specifying files. This allows SPARQL service providers to provide, in addition to a general SPARQL endpoint, specific functions that provide building blocks from which users can construct more complex queries of their data.

Definition 2.1 (SPARQL Function) *A SPARQL function encapsulates a SPARQL Pattern. When these functions are called, the input and output variables of the function definition are substituted by the formal parameters. Other variables in the pattern are private to each call, so there can be no name collisions between them and the surrounding query. Its definition simply specifies a pattern in addition to the declaration above.*

{ *pattern* }

While the notion of input and output variables from the function interface is kept in SPARQL functions, there is in fact no semantic distinction between them. Both input and output variables are substituted with their formal parameters at call sites, other pattern variables are renamed to unique temporaries, and the resulting SPARQL included in the overall query.

Definition 2.2 (Service Function) *A Service function denotes calls to an HTTP service. The semantics is that the service is invoked for each distinct set of bindings for all the variables in the input vars; each solution mapping is extended by adding all variables in the output vars corresponding to its bindings of the input vars. This means that a service with no input vars is invoked once and its output extends all solution mappings. These functions extend the declaration above as follows:*

`service` *uri* `(get | post) [(`*param* `->` *expr*`)*] -> (xml | json)` *row* `: (`*column type*`)*`

This uri *is what is used to invoke the HTTP service; it is not the uri that defines the function. This* uri *can be a uri literal or an expression that returns a string. The parameters are bound to the specified expressions, and passed to the service call using a GET or POST request as specified. Only input variables may be used in these expressions.*

'xml or 'json' must be specified. If 'xml' is specified, the results is expected to be an XML document which is parsed into result rows using the row *XQuery or XPath expression; each row is parsed for each* column *using the corresponding XQuery or XPath expression, and each such column is given the type denoted by the corresponding* type *XQuery/XPath expression. If 'json' is specified, the same process expects JSONPath [1] expressions* [9]*.*

In general, each column expression may return multiple values. In this case, we define the result as a set of rows corresponding the cross product of all of the column results. Each solution mapping is thus replicated and extended with each binding in the cross product. Note that this cross-product behavior is also exhibited by `XMLTABLE` *as defined by SQL/XML.*

Definition 2.3 (Table Function) *A Table function denotes calls to an HTTP service. The semantics is that the service is invoked for each group with a table consisting of all sets of bindings for the input parameters. Any variables in the scope of any table call that are not used as input parameters must be in a group by clause. Note that query parameters can be defined using grouping functions like sum over the input variables. Every solution binding in each group is extended with the bindings of output variables generated by the query. The output is processed either with XQuery/XPath or JSONPath, just as for service functions. These functions extend the declaration above as follows:*

`table` *uri* `[(`*param* `->` *expr*`)*] -> (xml | json)` *row* `: (`*column type*`)*`

4 Translation

We define our extensions by augmenting the algebraic translation of SPARQL given in §18 of the specification [9]. We then present the algebra for service and

[9] Systems that fully support SQL XML such as Oracle or DB2 can support XQuery, but we realize that in other implementations such as PostgreSQL, only XPath is supported. Allowing XPath accommodates more backend engines.

table functions as an extension to the existing alebra. We then define the basic support for invoking HTTP services.

Algebra Extensions. We define our extensions by extending the translation of Graph Patterns, which is what §18 calls all the patterns of SPARQL. The overall algorithm is in §18.2.2.6, and we extend it to handle service and table functions. In this algorithm Ω represents the entire current solutionm, i.e. a collection of tuples; μ denotes a single tuple, called a solution binding. The new algorithm is Algorithm 1, and we present just the additional pieces. Service functions (lines 5-11) call the service for each solution binding μ and combine the results. Table functions (lines 11-13) call the service once with current solution Ω. Join must be used rather than extend for service functions because the call might return zero or more results, and following SQL/XML, we define that using join semantics. We use $\{\mu\}$ to denote the singleton multiset containing solely the binding μ.

Algorithm 1 if the form is *Group Graph Patterns*

```
 1: $FS \leftarrow \emptyset$
 2: $G \leftarrow$ the empty pattern
 3: for all $E \in GroupGraphPattern$ do
 4:     if E is BIND( $f(i_1, \ldots, i_n)$ AS $(o_1, \ldots o_m)$) then
 5:         if f is a service function then
 6:             $G' \leftarrow \emptyset$
 7:             for all $\mu \in G$ do
 8:                 $G' \leftarrow G' \cup JOIN(\{\mu\}, CALL(uri))$
 9:             end for
10:             $G \leftarrow G'$
11:         else if f is a table function then
12:             $G \leftarrow JOIN(G, CALL(uri))$
13:         end if
14:     end if
15:     Cases from §18.2.2.6
16: end for
17: return $G$
```

HTTP calls. Service and table functions are based on HTTP services, as is the existing service pattern in SPARQL. For service functions, our HTTP operation allows computing query parameters based on the current solution binding; in the case of table functions, the expression must be a grouping function because the entire solution Ω is passed to the HTTP query. When the call returns data, the specified row definition is used to partition the result into rows; each such row is then parsed for each column value and type. We use the follow state of function f:

$uri(f)$ ≡ the URI expression of f
$params(f)$ ≡ the *input var, expression* pairs of f
$out(f)$ ≡ the *output vars* of f
$row(f)$ ≡ the row XQuery/XPath/JSONPath of f
$cols(f)$ ≡ the columns XQuery/XPath/JSONPath of f
$types(f)$ ≡ the column type XQuery/XPath/JSONPath of f

The function `process` is used to abstract over whether XQuery, XPath or JSONPath is used to handle the result.

Algorithm 2 HTTP calls

```
procedure CALL($f$)
    $req \leftarrow new\ HttpRequest$
    $req.uri \leftarrow uri(f)$
    for all $\langle param, expr\rangle \in params(f)$ do
        $req[param] \leftarrow expr$
    end for
    $\Omega \leftarrow \emptyset$
    $\Omega_{raw} \leftarrow req.send()$
    for all $row_{raw} \in process(row(f), \Omega_{raw})$ do
        $\mu \leftarrow \emptyset$
        for all $var, colExpr, typeExpr \in out(f), col(f), type(f)$ do
            $type \leftarrow process(row, typeExpr)$
            $val \leftarrow process(row, colExpr)$
            $\mu[var] \leftarrow type(val)$
        end for
        $\Omega \leftarrow \Omega \cup \{\mu\}$
    end for
    return $\Omega$
end procedure
```

5 Evaluation and Lessons Learnt

This section presents our evaluation and a summary of lessons learnt.

5.1 Evaluation

In the overall drug-drug interaction use case, evaluations with real data reported in [8] show that Tiresias achieves very good DDI prediction quality with an average F-Score of 0.74 (vs. 0.65 for the baseline) and area under PR curve of 0.82 (vs. 0.78 for the baseline) using standard 10-fold cross validation. Furthermore, a retrospective analysis demonstrated the effectiveness of Tiresias to predict correct, but yet unknown DDIs. Up to 68% of all DDIs found after 2011 were correctly predicted using only DDIs known in 2011 as positive examples in training. For the simplified DDI prediction use case described in section 2, all our experiments were conducted on MacBook laptops with 8G memory. The SPARQL extensions were implemented in Quetzal-RDF and our tests were conducted on the storage backends of Apache Spark and DB2. Our first use case had the following characteristics:

- There are 541 approved drugs in Drugbank with 113 unique transporters in Uniprot (drugs to transporters is a many to many relationship, with 941 drug-transporter pairs), and 541 unique chemical fingerprints.
- Getting data from UniProt for 113 unique transporters means 113 separate GET requests. For efficiency, we batched the transporter IDs into batches of 10 or 30 depending on the storage layer (10 for Spark, 30 for DB2). There is a tradeoff between the size of the data we get back from Uniprot when transporters are batched into a single request (each transporter returns about 33K) which can place significant burden on the storage layer to parse the data and the efficiency that is gained by batching the requests. After processing the Uniprot data, we had 1916 Gene Ontology (GO) functions for 113 transporters, for a total of 17384 rows, and we finished processing the drugs to transporters and transporters to GO functions computations in 51 s in DB2, and 2.4 minutes in Spark.
- Drug-drug chemical fingerprint similarity computation is $O(n^2)$, so when results are streamed back as XML from the REST service, it can place significant demands on resources needed to parse the 12.5 Mb data file in storage layers, and join it with existing data from DrugBank. For the Apache Spark backend, the query to fetch the drugs from Drugbank, fetch the appropriate chemical fingerprints and compute the chemical similarity for each drug pair took 9.5 minutes, with a significant amount of time being used by the join code that joined the pairs coming back with the drugs computed in prior steps. In DB2, the bottleneck was in parsing the 12.5 Mb file and not in the joins, so we needed to be more careful in the query to reduce the amount of data we shipped back. We did so by altering the SPARQL query to add *rowNumber* to the rows we send to the service, which reduces the amount of data we need to ship back, and that resulted in DB2 handling the query in 9 minutes. In general, the limitations of the storage layer can become a bottleneck for the sorts of queries that can be processed using the approach advocated in the paper (e.g., this will not work if terabytes of data need to be shipped between machines), but clearly the approach can work for a number of real workloads like this one.
- Drug similarity computation for GO annotations is another bottleneck similar to the one discussed above for chemical similarity, but we faced a hurdle here in terms of the ability of R code to handle this computation efficiently. The drug pair similarity computation required an average of 4-5 s in the GOSemSim R library, and there was no mechanism to batch the computation in the R code as we did for the chemical similarity computations.

Performing the drug-pairs similarity for 541 drugs is therefore not feasible without parallelization of the computation on a cluster. Being able to extend our approach to farm out a specific workload to a cluster would be useful in future work.

In addition, we evaluated the air pollution use case with a real EMR dataset from a tertiary hospital in Beijing. The dataset contains about 0.7 million outpatient encounters of 6506 patients during Feb. 2015 to Nov. 2015. The PM2.5 levels of the same period were also collected from the StateAir website. By testing a number of different diagnosis codes and time windows for aggregation, we found that there is a correlation between the weekly average PM2.5 levels and the total number of the encounter patients with three specific diagnosis codes related to the respiratory system. The Pearson's correlation coefficient is 0.303, and the p-value is 0.043.

5.2 Comparison with alternative solutions

We compared the proposed approach in this paper to alternative solutions. In the air pollution use case, according to the data scientists working on the clinical data analysis of the EMR dataset, two alternative solutions are typically used to handle the problem. Some data scientists perform the steps in Section 2.2 separately with different tools, and rely on glue code or manual steps to connect the different steps. A problem with this approach, which was also encountered in the DDI prediction use case (where some similarity computations happen in R, java and scala, for example), is that users have to leverage multiple tools and programming languages to perform the analysis, which ends up being difficult to manage. Alternatively, data scientists tend to use existing software to manage pipelines (e.g., in Spark or SAS or Python ML libraries). While this solution solves the management problem, it is a procedural approach to managing pipelines and ties one to a specific set of tools, compared to a more declarative one proposed in this paper. In many cases, the declarative one as advocated in this paper can allow more flexibility, and quick experimentation needed in the data preparation and curation steps of the data analytic process.

5.3 Lessons learnt

We have learnt important lessons on applying SPARQL for data preparation in data analytic use cases, which can be translated into the following key requirements:

- We need flexibility in SPARQL in being able to connect to different RESTful services, which we provided here by generalizing `service`.
- We need flexibility in SPARQL to batch requests from a set of query parameters specified in a table into a single URL, which we provided using `rowNumber`.

- We need the ability in SPARQL to compute over a table of data for at least 2 reasons. First, REST is by definition stateless, and the computation is often a type of aggregation over the data, which requires posting an entire table to a service. Second, it is simply more efficient to send data from a table when one is shipping data that does not fit within a URL. We propose `table functions` to handle this requirement.
- We need to be able to farm out workloads to external data sources when the computation requires parallelization as in the use case computing drug pair GO similarities, which is something we plan for future work.

6 Related Work

Mechanisms for integrating non-RDF data and SPARQL break into two broad categories. R2RML [4] and Direct Mapping [3] map relational data into a virtual RDF, providing the ability to ask SPARQL queries directly on the data, and hence integrate with RDF data using a SPARQL endpoint. We extend this approach to other forms of data. Furthermore, if these endpoints implemented our extended version of service, they could be posted data, opening up a wider variety of federated computing. The second category is tools to create RDF data, such as CSV2RDF [16, 5] for CSV and spreadsheet data, and XSPARQL [2] for XML. These tools create RDF data that then can be queried as a separate step; hence it is not integrated into the language as is our approach. Finally, our approach opens up data available only as web services (e.g., the Google Maps API within a SPARQL query).

SPARQL federation resembles traditional federated SQL [14, 10]. SPARQL 1.1 Service Descriptors [17] describe datasets, including simple statistics. Systems like DARQ [12], SIHJoin [11] and FedX [13] use queries to compute statistics at runtime. FedX also minimizes data transmission by restricting queries sent to end-points using embedded bindings using the VALUES keyword, to pass pre-computed results. However, the ability to federate is currently severely restricted since one can only transmit a limited amount of data, since it must be encoded in the URL.

7 Conclusions And Future Work

We have shown a few simple extensions to SPARQL that can greatly enhance its utility for query diverse sources of data: *functions* that allow modularizing queries, and *service* and *table* functions for integrating data from RESTful services. We define these constructs, and show how they were useful in providing a declarative approach to the problem of data integration that precedes any data analysis.

While we provide a general mechanism for accessing services that use XML and JSON formats, the same constructs can natually handle other formats such as Protobuf, which can be more efficient. We intend to support such formats in the future. And service calls taking data from a large table are a natural basis for concurrent calls to the service; we intend to explore constructs for managing such parallelism.

References

1. JSONPath. https://www.npmjs.com/package/JSONPath.
2. XSPARQL specification. http://www.w3.org/Submission/xsparql-language-specification/.
3. M. Arenas, J. Sequeda, E. Prud'hommeaux, and A. Bertails. A direct mapping of relational data to RDF. W3C recommendation, W3C, Sept. 2012.
4. S. Das, R. Cyganiak, and S. Sundara. R2RML: RDB to RDF mapping language. W3C recommendation, W3C, Sept. 2012.
5. I. Ermilov, S. Auer, and C. Stadler. CSV2RDF: User-Driven CSV to RDF Mass Conversion Framework. In *Proc. of the ISEM '13*, Sept. 2013.
6. D. A. Flockhart, P. Honig, S. U. Yasuda, and C. Rosebraugh. Preventable adverse drug reactions: A focus on drug interactions. *Centers for Education & Research on Therapeutics*.
7. A. Fokoue, O. Hassanzadeh, M. Sadoghi, and P. Zhang. Predicting drug-drug interactions through similarity-based link prediction over web data. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16. ACM, 2016.
8. A. Fokoue, M. Sadoghi, O. Hassanzadeh, and P. Zhang. Predicting drug-drug interactions through large-scale similarity-based link prediction. http://researcher.watson.ibm.com/researcher/files/us-achille/adrTechreport.pdf.
9. S. Harris and A. Seaborne. SPARQL 1.1 query language. W3C recommendation, W3C, Mar. 2013.
10. D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Comput. Surv. (CSUR)*, 32(4):422–469, Dec. 2000.
11. G. Ladwig and T. Tran. SIHJoin: Querying Remote and Local Linked Data. In *Proc. of the 8th European Semantic Web Conference, (ESWC 2011)*, volume 6644 of *Lecture Notes in Computer Science*, pages 139–153. Springer, 2011.
12. B. Quilitz and U. Leser. Querying Distributed RDF Data Sources with SPARQL. In *Proc. of the 5th European Semantic Web Conference, (ESWC 2008)*, volume 5021 of *Lecture Notes in Computer Science*, pages 524–538. Springer, 2008.
13. A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: A Federation Layer for Distributed Query Processing on Linked Open Data. In *Proc. of the 8th European Semantic Web Conference, (ESWC 2011)*, volume 6644 of *Lecture Notes in Computer Science*, pages 481–486. Springer, 2011.
14. A. P. Sheth and J. A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Comput. Surv. (CSUR)*, 22(3):183–236, Sept. 1990.
15. C. Steinbeck, Y. Han, S. Kuhn, O. Horlacher, E. Luttmann, and E. Willighagen. The Chemistry Development Kit (CDK): An Open-Source Java Library for Chemo- and Bioinformatics. *J. of Chemical Information and Computer Sciences*, 43(2):493–500, 2003.
16. J. Tandy, I. Herman, and G. Kellogg. Generating RDF from Tabular Data on the Web. W3C proposed recommendation, W3C, Nov. 2015.
17. G. Williams. SPARQL 1.1 service description. W3C recommendation, W3C, 2013.
18. G. Yu, F. Li, Y. Qin, X. Bo, Y. Wu, and S. Wang. GOSemSim: an R package for measuring semantic similarity among GO terms and gene products. *Bioinformatics*, 26(7):976–978, 2010.