Programmable Analytics for Linked Open Data

Bo Hu
Fujitsu Laboratories of Europe
Middlesex, UK
bo.hu@uk.fujitsu.com

Eduarda Mendes Rodrigues
Fujitsu Laboratories of Europe
Middlesex, UK

Emeric Viel
Fujitsu Laboratories Ltd
Kawasaki, Japan
emeric.viel@jp.fujitsu.com

ABSTRACT

LOD initiative has made a major impact on data provision. Thus far, more than 800 datasets have been published, containing tens of billions of RDF triples. The sheer size of data has not resulted in a significant increase of data consumption. We contend that a new programming paradigm is necessary to simplify LOD data utilisation. This paper reports an early phase development towards *programmable* web of LOD data. We propose to tap into a distributed computing environment underpinning the popular statistical toolkit `R`. Where possible, native `R` operators and functions are used in our approach so as to lower the learning curve. The crux of our future work lies in the full implementation and evaluation.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.12 [**Interoperability**]: Data mapping

Keywords

Linked Open Data, RDF, R, Programmability

1. INTRODUCTION

As of mid 2013, totally 870 datasets had been published as part of the Linked Open Data (LOD) cloud, exposing nearly 62 billion RDF triples in a computer-readable representation format[1]. These numbers are still rapidly growing largely attribute to open governmental data initiatives and "online" high-throughput scientific instruments. As greater amounts of data become available through LOD cloud, the expected virtuous cycle–more data leading to more consumption and thus encouraged data publication–has not been clearly witnessed. On the contrary, it is observed that, in many occasions, after the initial spark of interest and test applications, data use at many linked data hosting sites declined significantly [3]. Some critics believed that the massive amounts of semantic-rich data accumulated so far have actually driven away potential users. On the one hand, it adds extra layers of abstraction/conceptualisation to the data, making them not suitable for toolkits tuned against data represented in tabular format. On the other hand, the sheer volume of data renders many semantic web tools less productive. We contend that a major obstacle that prevents ordinary users from tapping into LOD cloud is the lack of a mechanism that allows people to make "sense" out of the overwhelming amount of data. More specifically, in order to facilitate the general uptake of LOD by research communities and practitioners, simply making the data available is not sufficient. It is essential to offer, along side the data, a means of utilising such resources in such a way that is comprehensible to users with a wide range of backgrounds and potentially limited knowledge of semantic technologies.

In this paper, we propose a solution that tightly integrates *linked data computing* with the popular statistic programming platform `R`. This brings together two well established efforts and thus two large user bases: `R` offers a declarative and well formed programming language for mining and analysing LOD datasets while the LOD Cloud paves the way to instantaneous access to a large amount of structured data on which existing `R` functions and packages can be applied.

1.1 Programmability of LOD

Thus far, data available through LOD Cloud are accessed primarily using SPARQL. Typically, this is conducted by submitting query scripts to a SPARQL endpoint and based on the query results, filtering/joining/aggregating (available from SPARQL 1.1) candidate results either on the server side or at the local clients. SPARQL is based on set algebra. This is both an advantage and a disadvantage. It resembles the prevailing SQL for RDB. People familiar with the latter can, therefore, enjoy a fast learning curve when making the paradigm shift. On the other hand, SPARQL is mainly a query language and thus does not stand-out for post-query data processing. In many cases, the results of SPARQL queries are extracted and converted into the native data structures of other programming languages (e.g. Java) for further manipulation.

Equipping and/or enhancing LOD with high programmability beyond SPARQL has been investigated previously. The (dis)similarity between RDF as the underlying data structure of LOD and the general object oriented methodology inspired ActiveRDF [5], where semantic data are exposed

[1]http://stats.lod2.eu. Accessed: January 2014.

LDOW2014 April 8, 2014, Seoul, Korea.

through a declarative script language. Along the same direction, RDFReactor [9] packs RDF resources as Java objects where instances are objects and properties are accessed through java methods.

Unfortunately, the above integrations have not lowered the threshold to fully exploring LOD cloud. Among other reasons, the most prominent ones include the follows. It will be very difficult for such approaches to deal with missing values and sparse structures, which abound in uncurated or automatically produced collections. The size and quality of LOD cloud lends itself to statistical data analysis. Performing such analysis using SPARQL queries can become cumbersome in many cases, requiring recursive SPARQL queries and multiple join operations. Moreover, neither SPARQL nor the integrated framework enjoys native support to matrix operations and solving linear equations, while such characteristics become increasingly critical in processing large amounts of data.

`R`, as a dynamic and functional language, offers good capacity to enhance the programmability of LOD and remedy the shortcoming of existing approaches.

1.2 Why R?

`R` is a programming language and a software toolkit for data science. Though not outspoken, `R` is designed for domain experts instead of conventional software programmers. It focuses on transactions that are more familiar to the former, e.g. organising data, manipulating spreadsheets and data visualisation. `R` is open source with over 2,000 packages/libraries for a wide variety of data analytics[2]. The most distinctive feature of `R` is its native support to vector arithmetics. In addition, versatile graphics and data visualisation packages as well as easy access to a large number of specialist machine learning and predictive algorithms make `R` a widely adopted computing environment in scientific communities (*c.f.* [2]). `R` is essentially single threaded. Scaling `R` for Big Data analysis can be achieved with RHadoop[3]. In this paper, we focus on adapting `R` for LOD data structure.

Integrating `R` and LOD has been inspected previously. The SPARQL `R` Library [8] aims to expose RDF data and wrap SPARQL endpoints with a black-box style connector library. Largely in the same vein, the most recent effort, `rrdf` library [10], allows loading and updating RDF files through manually crafted RDF-`R` mapping. The in-memory RDF models can then be queried using SPARQL. We see the following issues with SPARQL-based integration. Firstly, SPARQL queries and the target RDF data sets are not transparent to `R` users, making it difficult to validate and optimise the processes. Arbitrary SPARQL queries can incur global scans that drastically impede the system performance. Secondly, `R` environment loses the regulatory control over SPARQL queries. Such a blindness subjects the system to safety and security concerns. Finally, domain experts and statisticians are required to manually compose the SPARQL queries. This means learning the fundamentals of RDF and a new query language.

[2]http://www.r-project.org. Accessed: January 2014.
[3]https://github.com/RevolutionAnalytics/RHadoop/wiki

2. PROGRAMMABLE LOD

LOD Cloud provides a framework to access and navigate through apparently unrelated data, with conceptual models capable of explicating hidden knowledge. The logic based axioms (underpinning RDF) in many cases are not powerful enough to capture all the regularities in the data. We vision that a programming language, aiming to utilise and interact with LOD cloud (the datasets therein), is preferably to present the following characteristics.

Native support to LOD data structure. The underlying data structure of LOD is RDF triples which essentially compose a directed, labelled graph. SPARQL, the standard RDF querying language, transforms data into tabular form for better alignment with the RDB conventions. This extra formatting layer is not always necessary when the underlying data structure can be accessed with native graph operators.

Native support to data analysis. Better data accessibility inherent to LOD presents itself as both an opportunity and a challenge. With better access, an LOD data consumer is exposed to data linked in through semantic relations, most of which he or she may not be aware of. More data is not always necessarily a merit. In this case, the consumer is likely to be overwhelmed by data with different formats and different semantics, making analysis struggling. A programming platform capable of dynamically handling different format becomes desirable.

Ready for distributed processing. Applications accessing LOD Cloud can easily be exposed to billions of triples, tantamount to terabyte-grade data transactions. Single machine and single threaded statistical offerings will find themselves struggling in such situations. The programming platform should offer parallelisation capacity for good scalability.

Inspecting `R` within the scope of the above requirements, we can make the following observations. Firstly, `R` is a functional language with lazy evaluation, wherein functions are lifted to become first class citizen. Also, `R` has a dynamic type system. These fit well with RDF's idiosyncrasy. Secondly, `R` is designed for statistical computing. Missing value support and sparse matrix handling permeates all `R` functions and operations. Finally, though `R` is single-threaded, for many machine learning tasks it is possible to distribute the underlying `R` data structures and facilitate process distribution over a layer of data abstraction.

3. SYSTEM ARCHITECTURE

The concept of programmable LOD is experimented on the *BigGraph* platform, denoted as BG^R. *BigGraph* aims at a generic distributed graph storage with RESTful interface. Figure 1 illustrates the main building blocks of BG^R. At the top, there is the user interface. An BG^R user programs using `R` primaries with dedicated functions that facilitate the RDF to `R` data type mapping. BG^R programs are submitted to a *master* node as the main entry point through which the user interacts with the system. The runtime at the master is responsible for the following tasks: 1) interpreting BG^R programs; 2) interacting with the in-memory graph model for graph transactions; and 3) deciding which data server/worker it should directly query.

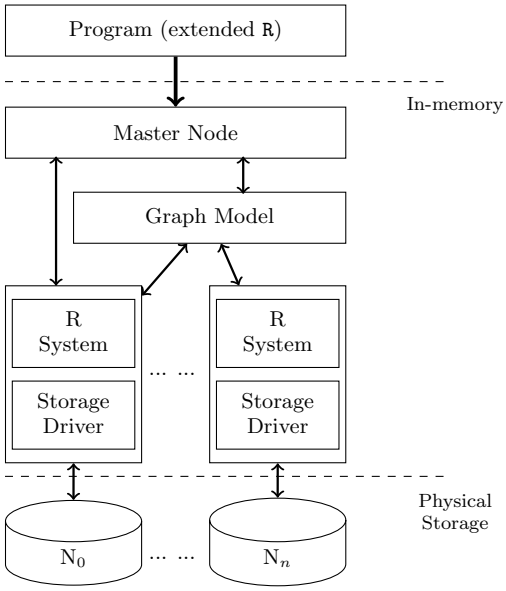

Figure 1: System architecture

The runtime on each data server mainly consists of two key components: R environment and storage driver. Each local R installation executes statistical analysis directly or exposes such analytical capacity through the in-memory graph model. A storage driver is responsible for I/O with the underlying storage unit.

3.1 Mapping RDF resources to R variables

The fundamental data structure for storing data in R is *vector*, where a single integer for example is seen as a vector of length one. Variations and extensions of vector data type include *matrices*, *arrays* and *data frames*. Though RDF graphs can be easily stored as adjacency matrices or adjacency lists, we would opt against a full conversion of LOD cloud, adding extra computing expenses. Rather, a direct one-to-one mapping between RDF resources (being classes and instances) and R variables can provide a seamless and smooth integration while at the same time ensures the integrity of the original data. For instance, an RDF instance becomes an R dataframe consisting of single-element vectors. Similarly, an RDF class can be assigned to a two dimensional dataframe with rows corresponding to instances and columns the properties. Instance values can be loaded either column wise or row wise depending on the analytical and performance requirements. In the following example, column-based initialisation is conducted.

```
> s <- data.frame(name=av, age=bv, email=cv)
> s
     age    name       homepage
P1   5      foo        foo@bar.com
P2   6      john       john@bar.com
...
```

Note that in this example, a class resource is extensionally represented by the set of its instances at the snapshot of data loading.

3.2 Mapping to the underlying storage

In order to accommodate the sheer size of LOD Cloud and leverage parallel data loading, a distributed storage is necessary. We opt for an edge-based storage solution that fits nicely with the principles of a Key-Value Store (KVS) [4]. KVS plays a key role in our approach to scale-out RDF graphs. RDF triples are, however, not KVS ready. The first and foremost step is therefore to define the key-value tuples that a standard KVS can conveniently consume. In BG^R, different components of a triple are concatenated together and encoded as UUID which is then treated as the key while the value parts of KVS are reserved for other purposes, e.g. named graph, provenance, and access control.

An RDF triple is indexed three times each. Even though presenting a replication factor of at least three, our approach is under the consideration of query performance and fault recovery. Loading RDF data into R variables is normally taking the form of localised *range queries*, fixing either the subject or object of the triples and replacing the rest with wildcards. For instance `graph.find`$(s, null, null)$ retrieves all the triples of a resource while `graph.find`$(null, p, o)$ presents an inverse traverse from object o. By replicating triples, data can be sorted according to not only subjects but also predicates and objects. This improves query execution.

3.3 Loading graph

For performance, LOD datasets are treated in the following ways. For datasets with RESTful API (e.g. DBpedia), the RDF resource to R variable mapping can be realised straightforwardly. Some datasets expose only SPARQL endpoints. SPARQL queries become necessary with the restriction that only local scans (e.g. $\langle s, *, * \rangle$ or $\langle *, *, o \rangle$) are permitted. Ideally, results of scan are used to construct local data graph. In the long run, on-demand data crawling can maintain local copies of frequently used datasets, helping to ensure data quality and manage mappings through local data curation.

3.4 Processing data

R is inherently a single threaded application, though parallelisation has been implemented using `snow` and `snowfall` packages [6]. The use of LOD Cloud falls into the following categories for which we proposed solutions to achieving good scalability.

3.4.1 Bulky processing

This OLAP-like data processing aims to emerge patterns (such as hidden semantic relationships and semantic data clusters) out of data held in LOD Cloud. Such a process normally is performed on preloaded data and is not time critical. While a plethora of R packages can be leveraged for data mining, the main difficulty lies in populating R dataframes with LOD data that can facilitate R functions. By encoding each RDF resource as one R variable, it is easy to construct matrices that fit with special purposes. For many predictive machine learning tasks, voting based aggregation (e.g. bagging [1]) can distribute the overall learning tasks to carefully sampled subsets of the target datasets. This can be easily achieved and managed by traversing the graph to the selected subsects of concept instances.

Example. Given a dataset with patients data, the following code fragment splits the set of patient instances into 10

subsets[4]. Traversal with named vertices and edges can be carried out along both inbound and outbound directions.

```
1: patient_v <- graph_get_vertex("Patient")
2: all_patients <- graph_get(patient_v, edge="rdf:type")
3: for(i in 1:10) {
4:   vname <- paste("p_set", i, sep="");
5:   assign(vname,
        sample(all_patients,length(all_patients)/10))
6:   saveRDS(vname, file="...")
7: }
```

Here, we assume the entire set of patient instances will be loaded into memory. Alternatively, a partial loading can be executed to lower the demand for computing resources and latency. In the following example, edges of `patient` resource is indexed. Sampling is conducted against the index. Only selected instances are loaded.

```
1: patient_v <- graph_get_vertex("Patient")
2: patient_size <- graph_get_edge_count(patient_v,
                       edge="rdf:type")
3: patient_index <- graph_edge_index(patient_v,
                        edge="rdf:type")
4: n <- c(1:patient_size)
5: ns <- sample(n, size/10)
6: for (i in ns) {
7:   ins<-graph_traverse(patient_v,edge=index[i])
8:   saveRDS(ins, file="...")
9: }
```

The following code fragment intends to construct a random-forest-based prognosis model (line 11) for a certain disease based on a patient's gender and age. The patient data are loaded with a graph traverse transaction over the given patient instance vertices and the given outgoing edges (line 3-6, where wildcard indicates all the outgoing edges). Missing values are set to a default one (i.e. `age` = 75) for simplicity (line 9).

```
 1: p_partition<-readRDS(file="...")
 2: patients <- data.frame()
 3: for(i in p_partition) {
 4:   p_data <- graph_traverse(vertex=i, out_edge="*");
 5:   patients <-rbind(patients, p_data)
 6: }
 7: size <- length(patients)
 8: training_set <- data.frame(
                       age=patients$has_age,
                       gender=patients$has_gender, ...)
 9: training_set$age[is.na(training_set$age)] <- 75
10: labels <- as.factor(patients$status);
11: rfp <- randomForest(training_set, labels)
```

In this example, we assume that the patient data partitions are passed using data file residing on the disk (line 1). This is for illustrative purposes only and does not exclude shared memory or message passing based solutions.

3.4.2 Incremental processing

OLTP-like realtime data processing is supported through an event-driven mechanism that applies classifiers (obtained as in the previous section) to data in an incremental fashion. This incremental characteristic is two-fold. Firstly, the system should detect the difference between existing classified data and inputs so as to isolate the changes and restrain reclassification only against the differences. Secondly, the system should update only those classifiers whose input data have changed since the most recent retraining. BG^R accommodates both requirements through distributed logging of graph structural changes and localised event propagation observing graph structures. For instance, "*OutEdgeCreatedEvent*" is issued by the storage *listener* if an edge is inserted. This event instance carries information such as the edge (in triple form) and on which vertex (v_s) this edge is created. Events propagate along paths that originate from v_s to avoid global scans. As a result, affected classifiers along the propagation routes are scheduled for update. Note that some machine learning algorithms can be easily adapted to fulfill the requirements (c.f. random forest [**?**]).

Versioning resources. An RDF resource normally consists of multiple triples jointly stating the constrains on the resource. Therefore, the event-driven incremental processing, which only has visibility of individual triples, requires a mechanism to obtain complete statements of the resource. We use versioning to ensure consistency when data are classified and when classifiers are retrained. Version information is stored at the value part of the key-value tuples and version updates are treated as atomic operations.

Multiple threads. Multi-threaded `R` is not likely to be available in the near future. As spawning threads is not possible, BG^R runs multiple processes communicating through socket. For instance, one `R` process listens to the underlying storage driver for fetching graph structural events through a dedicate socket address. The events are then parsed to extract event types, triples that raise the events, and versions of the triples. Other `R` processes handle the events and dispatch them for further actions when necessary, again by writing to a socket address. Socket-based communication may not provide ideal performance; in many cases it becomes the main bottle neck of performance. It, however, offers the most cost-effective solution to increase parallelism without dismantling `R`.

3.5 Resource local processors

We advocate and practice a declarative and resource-centric approach in BG^R. More specifically, expected analytics are constructed at the resource level and are associated with the target resource through RDF property declarations. For instance the following RDF triples assign an `R` random-forest classifier (defined in section 3.4.1) to a resource (i.e. the "Patient" class).

```
:Patient a  owl:Class ;
  rdfs:subClassOf
    [ a  owl:Restriction ;
         owl:onProperty :has_behaviour ;
         owl:someValuesFrom
           [ a  owl:Class ;
                owl:oneOf (:new_patient_behaviour
                           :update_patient_behaviour)]].
...
:new_patient_behaviour
  a            :Behaviour , owl:NamedIndividual ;
```

[4]Based on the literature, bagging should take a fraction between 1/2 to 1/50 depending on the size of the sample data.

```
:event       :onNewInstanceAdded ;
:has_handler "R:rfp" .
```

This essentially defines how a resource (e.g. Patient) reacts to (or behaves against) events (e.g. onNewInstanceAsserted event), realised using the attached process (e.g. R:rfp). At the ontology class level, *enumeration* (`owl:oneOf`) is used to establish conceptual relationship between the `Patient` class and the desired functionalities w.r.t. the corresponding events. The actual implementation of *behaviour* instances can be realised, for example, in `R`. Depending on the size of the compiled code, the implementation can be stored either entirely at the value part of the KV tuple of ⟨`:new_patient_behaviour, :has_handler, "R:rft"`⟩ or separately with a pointer from the value part of the tuple. When a new patient instance is asserted, an event is raised which will trigger the embedded `R` function to react to such a change in the storage.

Several advantages are evident by assigning `behaviour` and storing its implementation close to a resource. Firstly, for a distributed data storage, this implies a close proximity of data and process localities. Secondly, behaviour enhances the reactive programming principle by packing small process units against very specific data units. Thirdly, data behaviours and their implementations are conceptualised with well-formed RDFS constructs. This facilitates ontological inferences when necessary, though with caveats: i) increased inference complexity and ii) anonymous resources complicating RDF query handling.

4. PRELIMINARY RESULTS

BG^R is still under development. This section reports the system design that has been considered so far and lists out potential future work.

The underlying graph storage is a distributed KVS based on HBase. HBase also handles data partition, locality, replication and fault tolerance. Jena graph introduces the necessary abstraction layer for indexing and retrieving triples in the KVS. A simple graph programming interface is responsible for graph traversal and scan operations. It follows the Tinkerpop Blueprint convention[5] and currently talks to Jena graph so as to construct resource subgraph from the edge based storage data structure. The use of Jena is mainly for the convenience of leveraging Jena models when in-memory ontology inference becomes necessary. In the future, direct communication between storage and graph API is expected to improve the overall system performance. This is at the price of reduced ontological inference capacity.

Both storage and graph modules are implemented in Java. `R` communicates with the storage driver through an `R`-Java interfacing library, `rJava` package [7]. Calling Java methods are straightforward as illustrated in the following example:

```
1: .jinit()
2: # do something before loading the graph
3: g.obj<- .jnew("Graph")
4: # do something else
5: graph.find <- function(x, y) {
6:   .jcall(g.obj, "S", "find", x, y)
7: }
```

[5]https://github.com/tinkerpop/blueprints/wiki

We intend to minimise the effort of extending `R`, i.e. avoiding introducing compiled `R` packages. This is under mainly practical considerations. It lowers the learning curves for people already familiar with `R`, as basically no extra operators need to learn. Also, it increases the visibility of data management with respect to the underlying data structure.

5. CONCLUSIONS

This paper calls for user-friendly and programmable LOD by leveraging and enhancing `R`, a free software toolkit for statistical computing and graphics.

Note that there are a few `R` packages (e.g. `bigmemoRy`) that aims in particular at Big Data computing. There are also `R` packages (e.g. `foreach`, `ff`, etc.) for strengthening `R` parallelism. Our proposal is not to compete with such existing solutions but to advocate a collaboration of two independent efforts and provide solutions that fit the visions and requirements of linked data paradigm.

We also do not see competition with the RESTful movement, such as Linked Data Platform (LDP, [**?**]) which already gained momentum in the LOD community. LDP works at a layer lower than the proposed LOD/`R` integration, assisting data exposure so that the data can be consumed by the BG^R functions and operators.

6. REFERENCES

[1] E. Bauer and R. Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine Learning*, 36(1-2):105–139, July 1999.

[2] B. Everitt and T. Hothorn. *A handbook of statistical analyses using R*. CRC Press, Boca Raton, Fla, 2010.

[3] N. C. Helbig, A. M. Cresswell, B. Burke, and L. Luna-Reyes. The dynamics of opening government data. Technical report, Nov. 2012.

[4] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[5] E. Oren, B. Heitmann, and S. Decker. Activerdf: Embedding semantic web data into object-oriented languages. *Web Semant.*, 6(3):191–202, Sept. 2008.

[6] L. Tierney, A. J. Rossini, and N. Li. Snow : A parallel computing framework for the r system. *International Journal of Parallel Programming*, 37(1):78–90, 2009.

[7] S. Urbanek. *rJava: Low-Level R to Java Interface*, 2009. R package version 0.8-1.

[8] W. R. van Hage and T. Kauppinen. SPARQL package for R, 2011. available at `http://linkedscience.org/tools/sparql-package-for-r`.

[9] M. Völkel. Rdfreactor – from ontologies to programatic data access. In *Proc. of the Jena User Conference 2006*. HP Bristol, May 2006.

[10] E. Willighagen. Accessing biological data with semantic web technologies. http://dx.doi.org/10.7287/peerj.preprints.185v1, 2013.