

RDF2Rule: An Implementation in Spark/GraphX

Kunal Jha^{1,2*} and Akhilesh Vyas^{2†}

¹ DICE Research Group, AKSW, Universität Leipzig, Germany
kunal.jha@uni-bonn.de

² Masters of Science, Universität Bonn, Germany
s6akvyas@uni-bonn.de

Abstract

Recently, RDF knowledge graphs have gained extreme popularity in every possible domain of application. In order to enrich the knowledge graph inference rules are applied to the existing facts. Meanwhile the amount of RDF data contained in the graphs have also increased exponentially. Therefore, the idea of handling this ever-growing data and deducing new facts from them in an efficient distributed manner has gained tremendous significance. RDF2Rules algorithm first mines frequent predicate cycles (FPCs), a kind of interesting frequent patterns in knowledge bases, and then generates rules from the mined FPCs. As a part of the lab, this report presents the re-implementation of the algorithm for a distributed dataset using the latest technologies like GraphX and Pregel.

The report is a submission to the Big Data Analytics Lab for Summer Semester 2017.

1 Introduction

The number of large scale Knowledge bases (KBs) created and published using RDF has grown significantly over the past few years. These KBs contain not only huge number of entities but also rich entity relations, which makes them successfully used in many applications such as Question Answering, Semantic Relatedness Computation and Entity Linking.

RDF2Rules algorithm [7], aims at mining rules from an RDF knowledge base based on the mining of Frequent Predicate Cycles (FPCs). This report outlines the implementation of this algorithm with certain improvisations to achieve better performance for distributed RDF Knowledge bases. We have used APIs like Spark [4], Graphx[8] and Pregel[6] to achieve this implementation.

2 Problem Statement

2.1 Definitions

The definitions for path and cycles follow from the standard definition of paths and cycles of graph. The definitions have been modified to fit into the context of RDF Knowledge Graph.

- A *path* in an RDF KB $G = (E, P, T)$, (where E is the set of vertexes representing all the vertexes (entities), P is the set of predicates, and $T \subseteq E \times P \times E$ are directed and labeled

*Responsible for Pregel approach

†Responsible for GraphX approach

edges between vertexes (entities)) is a sequence of consecutive entities and predicates $(v_1, p^{d_1}, v_2, p^{d_2}, \dots, p^{d_{k-1}}, v_k)$, where $v_i \in E$, $p^i \in P$ and $d_i \in \{1, -1\}$ denotes the direction of predicate p^i , if $d_i = 1$ then $\langle v_i, p^i, v_{i+1} \rangle \in T$; otherwise, $\langle v_{i+1}, p^i, v_i \rangle \in T$.

- A *predicate path* is a sequence of entity variables and predicates $(x_1, p^{d_1}, x_2, p^{d_2}, \dots, p^{d_{k-1}}, x_k)$, where $d_i \in \{1, -1\}$ denotes the direction of predicate edge p_i .
- *Frequent Predicate Path/Cycle* : A path (cycle) is called the instance of a predicate path (cycle) if it can be generated by instantiating variables with entities in the predicate path (cycle). For a predicate path (cycle), the number of its instances that exists in the given RDF KB is called the support of it. If the support of a predicate path (cycle) is not less than a specified threshold, it is called the frequent predicate path (cycle).

2.2 Goal

The goal of this work (as defined in the scope of the lab) is to be able to mine Frequent Predicate Cycles from large scale database and enrich them with type information in an efficient manner. Beyond the lab, these enriched FPCs can be fed as an input to SANSa Inferencer [3] (or equivalent) to generate new facts in the Knowledge Graph. The consideration behind this implementation for this algorithm is to overcome the limitation of the original algorithm which was implemented for a multiple core processor. The exponential increase in the amount of triples in a RDF KB has lead to the problem of extending the KB to multiple data source clusters , therefore, making the original implementation outmoded.

3 Approach & Implementation

We divided the whole algorithm into three major subtasks as follows:

1. Finding Frequent Predicate Cycles (FPCs)
2. Adding Type Information to the entities in FPCs
3. Rule Generation

We have completed the first two subtasks as a part of the lab¹. The members of the team followed two different approaches to implement these tasks ².

3.1 Approach 1 (Using Pregel)

3.1.1 Pre-processing

The input to our implementation is an N-triple files containing the triples of RDF data. We use a small implementation of RDF parser (as per lab exercises) and use it to create a GraphX graph which is used as input for all the subsequent stages.

¹<https://github.com/Kunal-Jha/RDF2RuleSansa>

²**Disclaimer** :Each member is responsible for his own implementation, report, results and claims since both approach has been implemented independently.

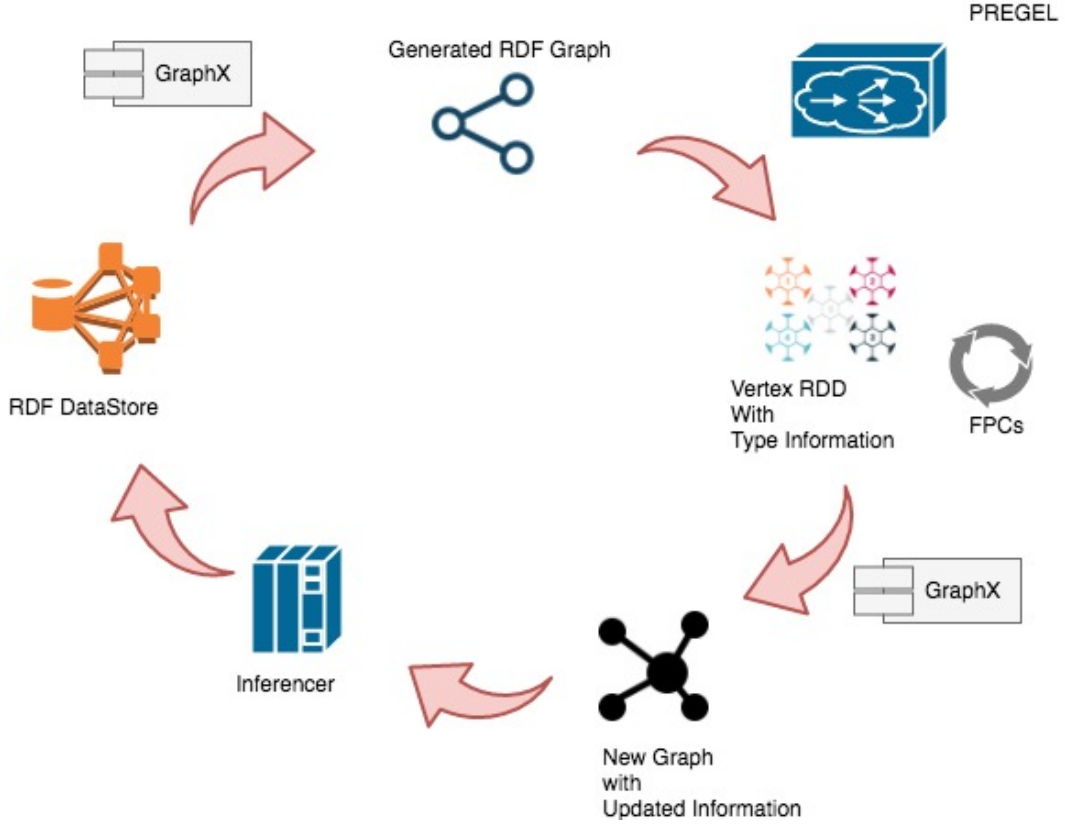


Figure 1: The Architecture Overview

3.1.2 Finding Frequent Predicate Cycles (FPCs)

Original Algorithm The original algorithm finding FPC is to find frequent 1-predicate path, and then repeatedly find frequent k-predicate paths from the frequent k-1 predicate paths. Algorithm starts a loop that discovers frequent k-predicate paths by extending k-1 predicate paths repeatedly in parallel. For each path we create an instance of a k-1 predicate path and then on the last element of the path we smartly add the new nodes to the path. (Shown in figure). The newly generated set of paths are checked for frequency threshold and in the process of extending predicate paths, the set of last entities in the instance paths are always kept, which facilitates adding new predicates and counting the support of predicate paths.

Implementation This approach [2] was implemented with two different variations using Pregel [1] (integrated in GraphX) which is a programming model specifically targeted to handle large scale graph problems. The reason behind using Pregel is its ease of programming as it makes vertices and edges first-class citizens and encourages programmers to think in those terms, rather than in terms of dataflows and transformation operators on parts of the graph. Moreover, the framework is designed to support iterative computations more efficiently than MapReduce, because it keeps the dataset in memory rather than writing it to disk after every iteration. This is useful since many graph algorithms are iterative. It also handles the fact that

graph algorithms generally have poor memory access locality, by locating different vertices on different machines and passing messages between machines as necessary.

- **Finding FPCs (traditional way)** : The first implementation is theoretically similar to the original algorithm where we take all possible pairs of nodes in the KB and find all possible paths between them using Pregel. Even though the usage of Pregel makes the path finding faster but taking all possible combination of path makes the algorithm relatively slower. Once all the possible paths are made available they are filtered based on existence of cycles and frequency count of edges occurrence.
- **Finding PCs (in one Traversal of Pregel)**: The second implementation of the algorithm takes one pass of pregel and is able to locate find all paths for each of the node and create a new VertexRDD which can be replaced in the original graph. We particularly believe that this approach is highly scalable and would result in an enriched knowledge graph in which the type information and path/cycles associated with a resource stored within the vertex and then used easily for frequency count. However we were unable to count frequency of edge types in the same traversal by pregel as well as facing bugs in mapping the source and destination in the paths.

The major challenge and learning from the project is to transform GraphX graph abstraction into a Vertex like communication for faster traversal of large scale graph. The pregel follows the following constructor

```
def pregel[A]
  (initialMsg: A,
   maxIter: Int = Int.MaxValue,
   activeDir: EdgeDirection = EdgeDirection.Out)
  (vprog: (VertexId, VD, A) => VD,
   sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
   mergeMsg: (A, A) => A)
  : Graph[VD, ED]
```

The first argument list contains configuration parameters:

initialMsg is the user defined message that will be sent to the vertices prior to superstep 0. **maxIter** is the maximum number of iterations we will perform before it terminates. **activeDir** is the edge direction in which to send messages

The second argument list contains the functions that handles the mutation of state and message passing: **vprog** is the user defined function for receiving messages. **sendMsg** is the user defined function to determine the messages to send out for the next iteration and where to send it to. **mergeMsg** is the user defined function to merge multiple messages arriving at the same vertex at the start of a superstep before applying the vertex program vprog.

3.1.3 Adding Type Information to the Entities in FPCs

The extraction of type information using pregel is trivial and uses one iteration of the graph and returns the most frequent type information. The algorithm is implemented similar to the pregel approach for finding PCs. The following code shows the way the implementation uses the modified KB graph (output of GraphX) and its attributes to pass the message to be communicated between vertices of the KB and added the type results to the original graph.

```
val pregelGraph = graph.mapVertices((id, label) => (id, label,
```

```

List.empty[String]))
  val messages = pregelGraph.pregel[List[String]](initialMsg,
    maxIterations, activeDirection)
  ((vid, oldMessage, newMessage) => (vid, oldMessage._2, newMessage
    ++ oldMessage._3),
    (triplet) => {
      if (triplet.attr.toString().contains("type"))
        Iterator((triplet.srcId, List(triplet.dstAttr._2)))
      else
        Iterator.empty
    },
    (msg1, msg2) => msg1 ++ msg2)

```

In conclusion, the final output of the implementation is an enriched graph in which each vertex (Resource) contains type information. Further, we also have a list of cycles, which are used for counting all possible edge patterns. Thus, we have successfully achieved the goal of the first two algorithms successfully but there is much more room for improvement.

3.1.4 Evaluation & Results

The system was tested over the implementation on 69,284 triples which was a subset of Semantic Web Dog Food Database. The testing was done based on a 2.9 Ghz i5 Processor with 8 GB 1867 MHz DDR3 memory.

Approach	Evaluation Time
Finding FPCs (traditional way)	\approx 18 minutes
Finding PCs (one Traversal of Pregel)	\approx 3.6 seconds
Finding Type Information (Using Pregel)	\approx 2.6 seconds

Table 1: Approach 1 Results

The results of the FPC algorithm for a Resource with vertex ID 687 is shown as follows. The format of the output (*< Lengthofpath >, < TypeofEdge(InboundorOutbound) >; < VertexId >; < Edgelabel >; VertexId*).

```

(Paths length between 687 7 is,2)
Outgoing ; 687 ; http://data.semanticweb.org/ns/swc/ontology#heldBy ; 7
Outgoing ; 7 ; http://data.semanticweb.org/ns/swc/ontology#holdsRole ; 687

```

The result of the type Information for a Resource with vertex ID 75 showing the most frequent resource type for this resource.

```

(75,List(http://data.semanticweb.org/ns/swc/ontology#TalkEvent))

```

3.2 Approach 2 (GraphX core Approach)

3.2.1 Find Entity Type Information and Update-Graph

Adding entity type information to rules can produce more accurate rules. For example, without entity type information, we may get rules like $(x1, bornIn, x2) \implies (x1, diedIn, x2)$. Based on this rule, if we already know that someone was born in someplace, then we can predict that this man also died in the same place. However, if the entity instantiating $x2$ is a small town, the prediction of this rule is prone to be incorrect, because most people would not stay in the same

town in their whole lives. If the entity instantiating x_2 becomes a country, the prediction will have a higher probability to be correct. Based on this observation, the following rule is more preferable. Example , $(x_1, bornIn, x_2) \wedge (x_1, typeOf, People)$
 $(x_2, typeOf, Country) \implies (x_1, die - In, x_2)$

3.2.2 How to Implement it using Spark in Steps:

Output: A set of FPCs with type information $\bar{\Theta}$

```

1 Set  $\bar{\Theta} = \emptyset$ ;
2 Get all the instance cycles of  $\theta$ ;
3 for  $i = 1, \dots, k$  do
4   Set  $C_i = \emptyset$ ;
5   Get  $\Pi_{x_i}(\theta)$ , the set of entities instantiating variable  $x_i$ 
6   for each entity  $e \in \Pi_{x_i}(\theta)$  do
7     Get the set of types  $Type(e)$  of  $e$ 
8     for each type  $t \in Type(e)$  do
9       if  $t \in C_i$  then
10         $t.count = t.count + 1$ ;
11      else
12         $t.count = 1$ ;
13         $C_i = C_i \cup \{t\}$ ;
14      end
15    end
16  end
17  Remove types from  $C_i$  whose counts are less than  $\tau$ ;
18  if  $|C_i| > k$  then
19    Only keep the top- $k$  frequent types in  $C_i$ ;
20  end
21  Set  $H_i = \{\langle x_i, typeOf, Thing \rangle\}$ ;
22  for each type  $t \in C_i$  do
23    Generate a triple  $h = \langle x_i, typeOf, t \rangle$ ;
24     $H_i = H_i \cup \{h\}$ ;
25  end
26 end
27 for each  $\bar{h} \in H_1 \times H_2 \times \dots \times H_k$  do
28   Generate a FPC with type information  $\bar{\theta} = \theta \oplus \bar{h}$ ;
29    $\bar{\Theta} = \bar{\Theta} \cup \{\bar{\theta}\}$ ;
30 end
31 return  $\bar{\Theta}$ 

```

1. Line 2: Generate the subgraph from RDD Graph based on filtering for the given FPCs.
2. Repeat below code for all predicates.
3. Line 5: Get set of entities for given a predicate from FPC
4. Line 6: For loop for each entity can be distributed under graphX RDD .
5. Line 6-16 For a given Type-of predicate filter operation we have list of all type list RDD.
6. Line 6-16 Sort All type list RDD according to count to filter for given support count.
7. 17-19 Take only top K frequent predicate .
8. 17-31 Generate edges from (entity, typeof, frequent-Type) and update graph.

3.2.3 Advantages(Spark over Algorithm)

Algorithm works very well on spark distribution wherever you want to do just some transformation and action on the processed graph. (eg. from above para step1, step2-7).

3.2.4 Disadvantages(Spark over Algorithm)

Algorithm for updating graph with new edges takes lots of time as it needs to be processed through many iterations. (please find reference in below flow-chart for loop- step 17-31).

3.2.5 Evaluation

This algorithm is totally based on frequent type count which is possibly not a better approach. Eg. (a, type-of, male) and (b, type-of, female), (male, subclassof, person), (female, subclassof, person). Ideally using frequent type list by this algorithm, you may get possibly a large frequent type for the female candidates. So later while updating graph you need to assign (a, type-of, female) which is already a typeof male . Due to this the rules like $(x1, givesBorn, x2) \wedge (x1, typeOf, female) \wedge (x2, typeOf, child) \implies (x1, motherOf, x2)$, will go wrong (depends on frequent support count for person). **Solution:** we should consider domain and range of each frequent type before processing it to graph solely based on support count of frequent type.

Implementation	Number of triples	Time
Find Frequent Type (full predicate list)	69284	1.7 seconds
Find Frequent Type (full predicate list)	1561	0.43 seconds
Updating Graph	69284	91.7 seconds
Updating Graph	1561	4.2 seconds

Frequent Predicate Type is not completed due to bad algorithm, taking too much time only for next edge processing. Sublinear time, Brute Force search for next edge. Spark only help while finding frequent type count from database.

<pre>[Stage 29:===== (1 + 1) / 2] -----FreqSetOfTypeListWithVertexId----- 4581 6668 18962 18982 Done--Duration for FreqSetOfTypeList:1.7838758349999998 -----FreqSetOfTypeListWithArgument----- http://xmlns.com/foaf/0.1/Organization http://xmlns.com/foaf/0.1/Person http://data.semanticweb.org/ns/swc/ontology#TalkEvent http://swc.ontoware.org/ontology#InProceedings -----New Updated Edges----- [Stage 29474:===== (1 + 1) / 3] 63847 -----Old Updated Edges----- 63847 Done--Duration for Updating Graph:91.663364229</pre>	<pre>Done--Duration for FreqSetOfTypeList:-0.42560479399999995 -----FreqSetOfTypeListWithArgument----- http://xmlns.com/foaf/0.1/Organization http://www.w3.org/ns/org#Organization http://xmlns.com/foaf/0.1/Person http://data.semanticweb.org/ns/swc/ontology#Paper -----New Updated Edges----- 1560 -----Old Updated Edges----- 1560 Done--Duration for Updating Graph:4.197613652</pre>
---	--

4 Project Timeline

• May 29, 2017 - June 5, 2017

- Kunal
 - * Read the paper.
 - * Understood the Algorithm and Divided tasks
- Akhilesh
 - * Read the paper.
 - * Understood the Algorithm and Divided tasks

• June 6, 2017- June 12, 2017

- Kunal
 - * Explored all possible features of Spark and GraphX
 - * Decided the final technology stack after discussion with mentors

- * Analyzed the portions of algorithm to be improvised / omitted according to the technology stack used.
- Akhilesh
 - * Study of scala-spark language
 - * Understanding of Algorithm in the context of Spark
- **June 13, 2017- June 20, 2017**
 - Kunal
 - * Learned the usage of required API.
 - * Implemented Shortest path algorithm using Pregel.
 - * Started working on the FPCs.
 - * Figured out the type Information can be done using Pregel itself.
 - * Prepared the presentation for mid evaluation.
 - * Explored improvements in the algorithm for distributed data processing.
 - Akhilesh
 - * Learned core graphx API and mapReduce functionality.
 - * Implemented entity-Type-Info using graphX APIs
 - * Implemented unit test code for entityType Information
- **June 21, 2017- July 17, 2017**
 - Kunal
 - * Finished First Implementation of FPC
 - * Evaluated the Implementation
 - * Redesigned the algorithm
 - * Implemented second implementation of PC.
 - * Fixed Bugs in type Information
 - * Worked on the Report (drew the figures for architecture and explained the Pregel methodologies).
 - * Discussed the ideas to incorporate frequency count.
 - * Studied GraphFrames.
 - Akhilesh
 - * Added Type Information to Original Graph Using Graphx.
 - * Worked on FPC - frequent subgraph algorithms using core GraphX API (Not Completed due to its sublinear time or brute force search)
 - * Evaluated entity type-Information algorithm advantages and disadvantages with respect to spark.
- **July 18, 2017 July 25, 2017**
 - Kunal
 - * Prepared the Presentation
 - * Made Final improvements on the report
 - * Wrote the final report in latex.

- * Had small meetings with various people in trying to look for a solution for the algorithm implementation.
- Akhilesh
 - * Added implementation content to the doc report.

5 Future Work And Improvements

There are multiple improvisation and extension of the project which we are curious to explore and incorporate into the project. These are listed as follows:

1. The FPC generation can be implemented using the RochaThatte Algorithm which is a faster implementation of distributed graph traversal algorithm and much less computational overhead.
2. The type information and the FPC generation (second implementation with correct mapping) can be integrated into one traversal of Graph via Pregel which would result in approximately 2 times the current traversal speeds.
3. Integrate the output into an Inferencing engine.
4. Comparison between the algorithms proposed in 1 and 2 points would also be an interesting insight.
5. Based on the results, in future, the project can be easily exported to work with DataFrames instead of RDDs and GraphX can be replaced with GraphFrames[5].

References

- [1] Nous : Construction and querying of dynamic knowledge graphs.
- [2] Tutorial graphx with pregel.
- [3] Smart Data Analytics. Semantic analytics stack (sansa).
- [4] Apache Software Foundation. Apache spark.
- [5] Apache Software Foundation. Graphframes.
- [6] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.
- [7] Zhichun Wang and Juanzi Li. Rdf2rules: Learning rules from rdf knowledge bases by mining frequent predicate cycles. *arXiv preprint arXiv:1512.07734*, 2015.
- [8] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.