Lab Distributed Big Data Analytics.

RDF2Rule: An Implementation in Spark/GraphX

Kunal Jha [1,2] , Akhilesh Vyas [2]

[1]DICE Research Group,
AKSW,
University of Leipzig, Germany

[2] Masters of Science, Computer Science Department
University of Bonn, Germany

kunal.jha@uni-bonn.de

s6akvyas@uni-bonn.de

Introduction

Recently, RDF knowledge graphs have gained extreme popularity in every possible domain of application. 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. 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[1], 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 large scale data processing engine Spark and its embedded APIs like GraphX and Pregel to achieve this implementation.[1]

Problem Statement

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) is a sequence of consecutive entities and predicates (v_1 , p^{d1} , v_2 , p^{d2} ,...,p^{dk-1},v_k), where vi ∈ E, p^i ∈ P; di ∈ {1,−1} denotes the direction of predicate p^i, if di = 1 then $\langle v_i,p^i,v_{i+1}\rangle$ ∈ T; otherwise, $\langle v_{i+1}, p^i, v_i\rangle$ ∈ T .
- A **predicate path** is a sequence of entity variables and predicates (x_1, p^{d1} , x_2, ..., p^{dk} , x_{k+1}), where di ∈ {1, −1} denotes the direction of predicate edge pi.
- **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).

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,

[1] https://github.com/Kunal-Jha/RDF2RuleSansa

these enriched FPCs can be fed as an input to SANSA Inferencer (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.

Approach & Implementation

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

- Finding Frequent Predicate Cycles (FPCs)
- Adding Type Information to the entities in FPCs
- Rule Generation

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

Approach 1 (Using Pregel)

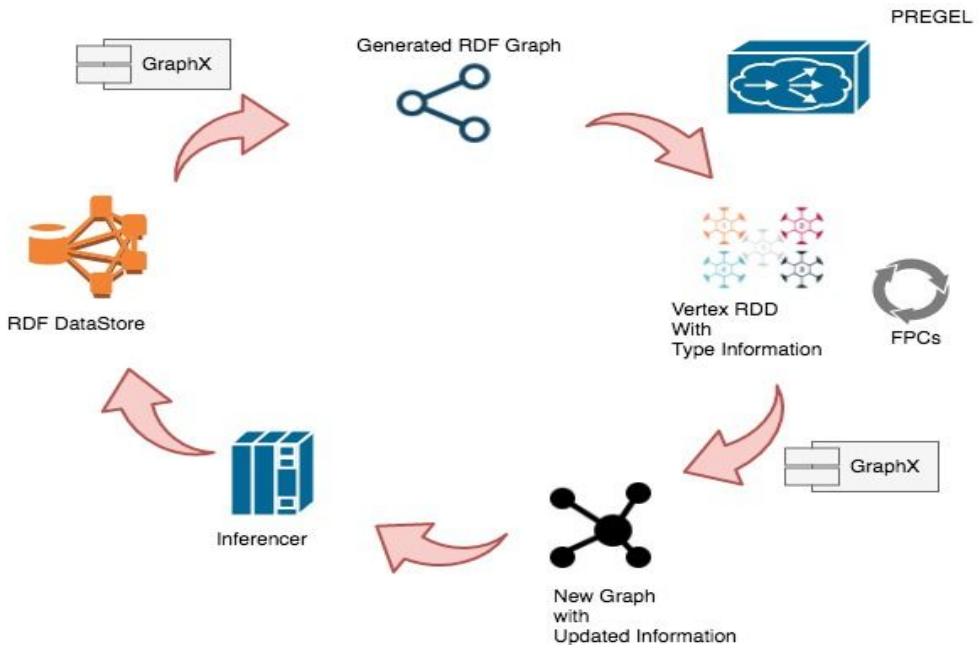


Figure 1: *The Architecture Overview*

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.

Finding Frequent Predicate Cycles (FPCs)

Original Algorithm

The original algorithm finding FPC is to find frequent 1-predicate path, and then iteratively 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 iteratively 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

We implemented the algorithm with two different variations using Pregel (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:

1. initialMsg is the user defined message that will be sent to the vertices prior to superstep 0.
2. maxIter is the maximum number of iterations we will perform before it terminates.
3. 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:

1. vprog is the user defined function for receiving messages.
2. sendMsg is the user defined function to determine the messages to send out for the next iteration and where to send it to.
3. 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.

Figure 3. shows the way we transformed 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

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.

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.

```
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);
```

Figure 3 : Pregel Code for TypeInfo.

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	Execution Time
Finding FPCs (traditional way)	≈ 18 minutes
Finding PCs (one Traversal of Pregel)	≈ 3.6 seconds
Finding Type Information (Using Pregel)	≈ 2.6 seconds

Table1 : Runtime Results for Different Implementations.

The results of the FPC algorithm for a Resource with vertex ID 687 looks like Figure 4. The format of the output (<Length of path>, <Type of Edge (Inbound or Outbound)> ; <VertexId> ; <Edge label>; 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
```

Figure 4 : Sample Output Showing a FPC.

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))
```

Figure 5 : Sample Output Showing Most common Typeinfo Using Pregel for Node 75.

Approach 2 (GraphX core Approach)

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) ⇒ (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 x2 becomes a country, the prediction will have a higher probability to be correct. Based on this observation, the following rule is more preferable.

Eg. (x1, bornIn, x2) ∧ (x1, typeOf, People)

(x2, typeOf,Country) ⇒ (x1, die-In, x2)

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

```
1  Set Θ̄ = ∅;
2  Get all the instance cycles of θ;
3  for i = 1, ..., k do
4      Set C_i = ∅;
5      Get Π_{x_i}(θ), the set of entities instantiating variable x_i
6      for each entity e ∈ Π_{x_i}(θ) do
7          Get the set of types Type(e) of e
8          for each type t ∈ Type(e) do
9              if t ∈ C_i then
10                 t.count = t.count + 1;
11             else
12                 t.count = 1;
13                 C_i = C_i ∪ {t};
14             end
15         end
16     end
17     Remove types from C_i whose counts are less than τ;
18     if |C_i| > k then
19         Only keep the top-k frequent types in C_i;
20     end
21     Set H_i = {⟨x_i, typeOf, Thing⟩};
22     for each type t ∈ C_i do
23         Genenrate a triple h = ⟨x_i, typeOf, t⟩;
24         H_i = H_i ∪ {h};
25     end
26 end
27 for each h̄ ∈ H_1 × H_2 × ... × H_k do
28     Generate a FPC with type information θ̄ = θ ⊕ h̄;
29     Θ̄ = Θ̄ ∪ {θ̄};
30 end
31 return Θ̄
```

How to Implement it using Spark in Steps:

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.

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).

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)

Evaluation

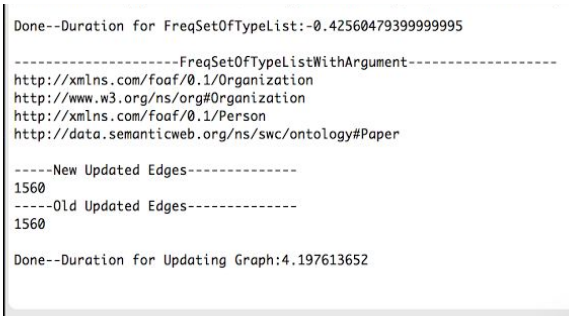
This algorithm is totally based on frequent type count which is possibly not a better approach. Eg. (a, type-of, male) and (b, type-of, female), (male, subclassof, person), (female, subclassof, person). Ideally using frequent type list by this algorithm, you may get possibly a large frequent type for 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) ∧ (x1, typeOf, female)∧ (x2, typeOf, child) ⇒ (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	**KnowledgeBase**	**Triplets**	**Time**
Find Frequent Type (full predicate list)	11.5 MB	69284	1.7 seconds
	222 KB	1561	0.43 seconds
Updating Graph	11.5 MB	69284	91.7 seconds
	222 KB	1561	4.2 seconds
Frequent Predicate Type (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.

Table : *Result-Table(GraphX core APIs- CPU- 2.0Ghz)*



Project Timeline

Duration (Dates)	**Project Tasks**	**Person**
May 29, 2017 - June 5, 2017	• Read the paper • Understood the Algorithm and Divided tasks	Kunal Akhilesh

June 6, 2017- June 12, 2017	• Explored all possible features of Spark and GraphX • Decided the final technology stack after discussion with mentors • Analysed the portions of algorithm to be improvised / omitted according to the technology stack used. • Study of scala-spark language • Understanding of Algorithm in the context of Spark	Kunal Akhilesh
June 13, 2017- June 20, 2017	• 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. • Learned core graphx API and mapReduce functionality. • Implemented entity-Type-Info using graphX APIs • Implemented unit test code for entityType Information	Kunal Akhilesh
June 21, 2017- July 17, 2017	• 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 • 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.	Kunal Akhilesh Vyas
July 18, 2017 – July 25, 2017	• Prepared the Presentation • Made Final improvements on the report • Had small meetings with various people in trying to look for a solution for the algorithm implementation. • Added contributions to the report.	Kunal Akhilesh Vyas

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 **Rocha–Thatte 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 Data-Frames instead of RDDs and GraphX can be replaced with GraphFrames.

References

1. *Wang, Zhichun, and Juanzi Li. "RDF2Rules: Learning rules from RDF knowledge bases by mining frequent predicate cycles." arXiv preprint arXiv:1512.07734 (2015).*
2. *Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N. and Czajkowski, G., 2010, June. Pregel: a system for large-scale graph processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (pp. 135-146). ACM.*
3. *Xin, R.S., Gonzalez, J.E., Franklin, M.J. and Stoica, I., 2013, June. Graphx: A resilient distributed graph system on spark. In First International Workshop on Graph Data Management Experiences and Systems (p. 2). ACM.*
4. *Spark Website: https://spark.apache.org/*
5. *Sansa Website : https://github.com/SANSA-Stack*
6. *GraphX Website: https://spark.apache.org/graphx/*
7. *Spark Summit: https://youtu.be/lFBPdL9XpA4?t=709*
8. *GraphFrames :* https://graphframes.github.io/
9. *Tutorial GraphX with Pregel : http://www.cakesolutions.net/teamblogs/graphx-pregel-api-an-example*
10. *NOUS : Construction and Querying of Dynamic Knowledge Graphs Repo (for seeing the extent to which Pregel can be used in RDF Knowledge base handling) : https://github.com/Geldren1/NOUS-KG*

Appendix

FlowChart of Entity Type Information and Update Graph