

CSE 505 - Computing with Logic

Harsh Wardhan Agarwal - 111465389

Fall 2017

Final Project Report

1 Introduction

Logic programming is designed to focus mainly on knowledge representation and derive inferences from the given knowledge base. With the extensive research over the years, we have been able to build logic systems that are capable of breaking through very complex logic. But, as we progress more into the internet era, the data (or the knowledge base) that we accumulate is becoming huge. Our logic programs, that run on stand-alone systems do not hold the intrinsic capability to scale themselves to handle billions of facts and take huge amounts of time to generate conclusions. Addressing to this problem, I have made an attempt to study and implement the paper, “Efficient Computation of the Well-Founded Semantics over Big Data” which talks about one of the possible ways in which we can scale logic programming over distributed system architecture and parallelly compute billions of facts from the knowledge base for a given query.

2 Working Methodology

The core idea behind parallelising a logic program is to distribute the facts to multiple systems (referred as nodes in the language of Big Data) and run the same query on each system. This can be interpreted as multiple but same logic programs running over several machines handling different facts but the same query. Once the results are generated, we combine and collect the results in a single machine. This architecture of distributed code/data flow resembles the MapReduce architecture which is where we are moving with this discussion. MapReduce is a framework that helps us achieve scalability over a distributed architecture and provide parallel processing of our data. Let us understand how we can apply distributed computing to logic programs.

Let us consider the following logic program written in Prolog.

```
parent(John, Alice).
parent(John, Jill).
sibling(Alice, Edward).
sibling(Jill, Mary).
female(Mary).
son(X,Y) :- parent(Y,Z), sibling(Z,X), not female(X).
```

We will first have to segregate facts that represent positive sub-goals and negative sub-goals. In this case, the facts with predicate 'parent' and 'sibling' relate to positive sub-goals and the facts with the predicate 'female' relate to negative sub-goals. Hence, we create two lists I and J for the segregated positive and negative sub-goal facts respectively.

I = parent(John, Alice), parent(John, Jill), sibling(Alice, Edward), sibling(Jill, Mary)

J = female(Mary)

It is important to understand that, the entire process of execution of the above program will take two MapReduce cycles. The first MapReduce cycle will generate all the possible answers from the positive sub-goals. The second MapReduce cycle will remove all the answers that satisfy the negative sub-goals. Note that, if there are many such positive and negative sub goals, then the number of MapReduce jobs will increase accordingly. Suppose we have 'n' positive sub-goals and 'm' negative sub goals, we will have to run (n-1) MapReduce jobs to generate all possible results from the positive sub-goals and then use these results followed by 'm' MapReduce jobs to drop all the results that satisfy negative sub goals. Hence, we will have to execute a total of (n+m-1) number of MapReduce jobs which is not efficient.

A solution to this problem can be addressed by developing a complex network of MapReduce Jobs that can process all the positive sub goal tasks simultaneously and then all the negative sub goal tasks simultaneously. Care must be taken while assigning predicates to each MapReduce job. It is intuitive that two predicates that are reliable on each other cannot be executed simultaneously. That is, if the input of one predicate is dependent on the output of other predicate, the first predicate will have to wait until we get the results from the second predicate.

Now, since we have generated I and J lists for our Prolog code, we can now call the first MapReduce job. This job will be called on the set I where we have the predicates 'parent' and 'sibling'. In the output of this MapReduce job, we will try to generate a new predicate 'ParentOfSiblings(X, Y, Z)' (arity 3) that will represent all the parents X who have their siblings as Y and Z. We will also load all the generated facts associated with 'ParentOfSiblings' into a new list and can be treated new facts in our Knowledge Base.

Having these results in hand we are now ready to initiate the second MapReduce job. In this job we will append the newly generated facts and the facts in the set 'J' and pass it as the input in the second MapReduce cycle. Here we try to apply a sort of anti-join on the set of facts. Here, instead of collecting all the matching pairs, we will ignore the results that tend to satisfy our negative sub goals. As a result, we will end up generating all the possible solutions, that satisfies all the positive sub goals and do not satisfy the negative sub goals.

3 Implementation

In this section, we will try to understand the step by step implementation of the MapReduce tasks to get the desired output. I have made use of Python2.7 to implement the program. I have generated a sample test input file randomly using the baby names data set from kaggle. The data set contains about 50,000 facts and can be found with the filename randomsample.txt. Now, let us understand the steps by using the same example as above.

Step1: Load the text file where we have the Prolog code.

Step2: Read through the file and generate the sets I and J. This can be achieved using simple string processing using regular expressions if required.

Step3: Call the first MapReduce job. Here, we pass the set I as the input get the corresponding results using the mapper and the reducer functions. We will have predetermined number of mappers and reducers required.

```
def map(self, k, v):
    predicate = get_predicate_string(v) #List
    arg = get_parameters_of_the_predicate(v) #List
    dictionary = {}
    if predicate == 'parent':
        dictionary[arg[1]] = {predicate, arg[0]}
    elif predicate == 'sibling':
        dictionary[arg[0]] = {predicate, arg[1]}
    elif predicate == 'ParentOfSiblings':
        dictionary[arg[1]] = {predicate, arg[0], arg[2]}
    elif predicate == 'female':
        dictionary[arg[0]] = {predicate}
    return dictionary.items()
```

[The above pseudo code is implemented in reference to the Algorithm 4 (Single Join) stated in the Appendix of the paper under discussion.]

```
def reduce(self, k, vs):
    dictionary = {}
    vs.sort()
    value = ''
    for i in range(len(vs)):
        if "female" in vs[i]:
            return
            value+=vs[i]
            value+=','
    value = value[:-1]
    dictionary[k] = value
    return dictionary.items()
```

[The above pseudo code is implemented in reference to the Algorithm 6 (Anti-Join) stated in the Appendix of the paper under discussion.]

By executing the above MapReduce jobs, we get a new predicate 'ParentOfSiblings'. And the new set of facts we get are ParentOfSiblings(John,Mary,Jill), ParentOfSiblings(John,Edward,Alice).

Step4: Append the newly generated facts to the Set J.

Step5: Call the second Map-Reduce job. This job will run on the same mapper and reducer, as the Mapper and Reducer functions are designed to handle both the MapReduce cycles. There is no need to create completely separate Mapper and Reducer functions to achieve this goal. Hence, you can refer the same map and reduce functions as discussed above for this task too. After, the second MapReduce cycle, we get our final answer as:

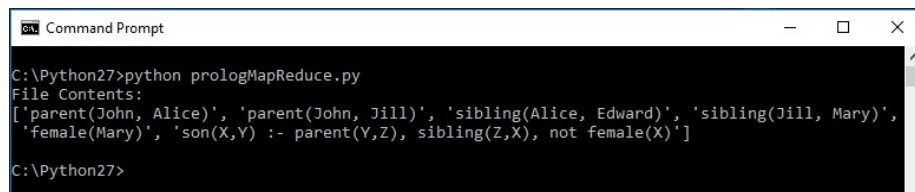
Son(Edward,John).

This means that from the given set of facts, we could use distributed computing to generate an inference as Edward is the son of John.

4 Results

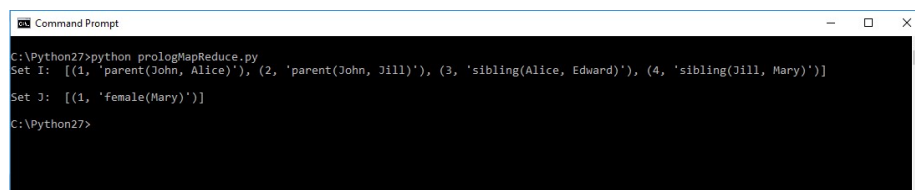
The outputs at each were obtained as below:

Step1:



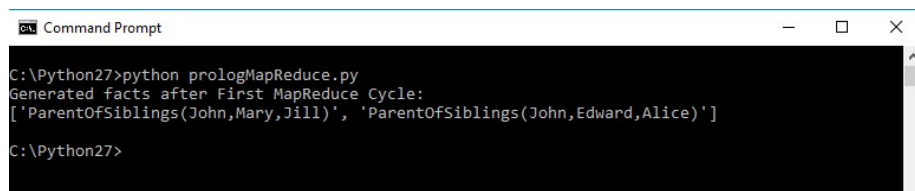
```
Command Prompt
C:\Python27>python prologMapReduce.py
File Contents:
['parent(John, Alice)', 'parent(John, Jill)', 'sibling(Alice, Edward)', 'sibling(Jill, Mary)',
'female(Mary)', 'son(X,Y) :- parent(Y,Z), sibling(Z,X), not female(X)']
C:\Python27>
```

Step2:



```
Command Prompt
C:\Python27>python prologMapReduce.py
Set I: [(1, 'parent(John, Alice)'), (2, 'parent(John, Jill)'), (3, 'sibling(Alice, Edward)'), (4, 'sibling(Jill, Mary)')]
Set J: [(1, 'female(Mary)')]
C:\Python27>
```

Step3:



```
Command Prompt
C:\Python27>python prologMapReduce.py
Generated Facts after First MapReduce Cycle:
['ParentOfSiblings(John,Mary,Jill)', 'ParentOfSiblings(John,Edward,Alice)']
C:\Python27>
```

Step4:

```
Command Prompt
C:\Python27>python prologMapReduce.py
Appended newly generated facts to set J:
[(1, 'ParentOfSiblings(John,Mary,Jill)'), (2, 'ParentOfSiblings(John,Edward,Alice)'), (3, 'female(Mary)')]
C:\Python27>
```

Step5:

```
Command Prompt
C:\Python27>python prologMapReduce.py
Output after Second MapReduce Cycle (final answer):
Son(Edward,John)
C:\Python27>
```

Output Screen depicting entire execution process:

```
Command Prompt
Set I: [(1, 'parent(John, Alice)'), (2, 'parent(John, Jill)'), (3, 'sibling(Alice, Edward)'), (4, 'sibling(Jill, Mary)')]
Set J: [(1, 'female(Mary)')]
-----First MapReduce Cycle-----
Mapper Output:
[(2, ('Alice', '{parent,John}')),
 (2, ('Alice', '{sibling,Edward}')),
 (2, ('Jill', '{parent,John}')),
 (2, ('Jill', '{sibling,Mary}'))]
Reducer Output:
[[('Alice', '{parent,John},{sibling,Edward}')),
 (('Jill', '{parent,John},{sibling,Mary}')]]
Generated facts after First MapReduce Cycle:
['ParentOfSiblings(John,Mary,Jill)', 'ParentOfSiblings(John,Edward,Alice)']
Appended newly generated facts to set J:
[(1, 'ParentOfSiblings(John,Mary,Jill)'), (2, 'ParentOfSiblings(John,Edward,Alice)'), (3, 'female(Mary)')]
-----Second MapReduce Cycle-----
Mapper Output:
[(1, ('Edward', '{ParentOfSiblings,John,Alice}')),
 (1, ('Mary', '{ParentOfSiblings,John,Jill}')),
 (1, ('Mary', '{female}'))]
Reducer Output:
[[('Edward', '{ParentOfSiblings,John,Alice}')]
Output after Second MapReduce Cycle (final answer):
Son(Edward,John)
C:\Python27>
```

5 Conclusion

I have understood the idea behind distributed computing and how it can be extended to logic programming to extend its implementations to Big Data applications. To explore my understanding, I have implemented the algorithms discussed in the paper using simple examples in Python2.7. The implementation is dependent on the problem under consideration. That is, the mapper and the reducer code will change with the query in the prolog. For a different query, a customized mapper and reducer function will have to be designed. From the implementation perspective, it can also be observed that as the number of positive and negative sub goals increases, the number of MapReduce jobs will increase. Hence, the approach is efficient for answering queries that have a limited number of positive and negative sub-goals. For large number of positive and negative sub goals, either we have to intricately design the Map and Reduce

tasks or have a new approach addressing this problem.

References

- [1] <https://www.ime.usp.br/~ontobras/wp-content/uploads/.../ONTOBRAS-2015.pptx>
- [2] <https://www.overleaf.com/latex/templates/quantum-hall-effect-report-template/hqsgsyxsvphs.WjkoaN-nHIU>
- [3] https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html
- [4] <https://blog.matthewrathbone.com/2013/01/05/a-quick-guide-to-hadoop-map-reduce-frameworks.html>
- [5] http://www3.cs.stonybrook.edu/~has/CSE545/a1p1_lastname.py