# Enhanced Approach To Compute Entity Relationships over MapReduce

Harsh Wardhan Agarwal
*Graduate Student, CS*
*Stony Brook University*

Rishabh Srivastava
*Graduate Student, CS*
*Stony Brook University*

Ritu Agarwal
*Graduate Student, CE*
*Stony Brook University*

***Abstract*—Identifying patterns and discovering hidden relationships with the help of data analysis is a key factor that many technology giants leverage today to their profit and scale up their businesses. Not only in leveraging the business, such analysis is also crucial to many domains such as the health industry where it can be used to find patterns and understand relationships between genes of different species. The theory can potentially be extended to many such domains. But a major challenge our analysis systems face today are not because of unsophisticated algorithms, but their computation limits. The data that needs to be analysed is so huge that it is practically impossible to perform the entire computation without a distributed architecture. In our work, we propose an idea that not only leverages the advantages of MapReduce but also parallelises the MapReduce tasks themselves by breaking the bigger goal into smaller sub problems, identifying the tasks that can be run independently and perform multiple and distinct MapReduce tasks parallely to reduce the computation time drastically.**

***Index Terms*—Big Data, MapReduce, Parallel computation, Distributed computing**

## I. Introduction

Organizations today hold huge amounts of data and constantly attempt to extract insightful details out of it. Logic programming (such as Prolog) is one of the solution that assists us with finding knowledge out of raw data. 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, even though we have sophisticated algorithms that hold the capacity to serve our purpose, the computational limits often become a major issue. 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, we made an attempt to study 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. We could come up with a few improvement measures that if incorporated in the original model, we can more efficiently harness the power of parallel computing and make significant improvements in the overall running time.

## II. Motivation

Since we aim to solve problems which involve relations, it would be of a great benefit to companies who deal with huge amounts of data. For example, Facebook might have lots of users who are related to each other (father-son relationship for example) and hence huge amount of data corresponding to the same. These relations can be used for trend analysis and other predictions or data mining tasks. This will result in up-scaled revenue as observing trends and then working on what users exactly want has always proved to be beneficial. But before this, we first need to have user relations. Speedily finding user relations by parallelising MapReduce tasks is what we aim to do. Another example would be of an e-commerce like Amazon. Relations like, if buyer A buys soap A only after he has bought shampoo B, can be used with a great benefit to their sales as they might then consider placing that shampoo product before that soap product on their website. The world of personalized recommendations and predictions majorly depends on finding relationships between the objects in our data. Hence, to get these relationships in a speedy manner is the root to other data analysis task and hence it is important.

## III. Overview

Although we will be discussing the entire logic behind our idea in detail in the later sections of the paper, the basic overview of the proposed idea would help you understand the problem statement better and align you with the vision of the paper and can be understood as follows. Suppose we have a database containing details of the users of a particular service and we would like to analyze certain relationships or patterns between the users and certain other fields. For example, let us say we run an e-commerce website and we would like to understand the relationships between certain products and detect patterns among the sellers and the buyers related to these products. To be more specific and for simplicity, let us consider a simple query that gives us the relationship mapping of a particular product with all its sellers and all its buyers.

–[Product] :- [ListOfSellers, ListOfBuyers]

Traditionally, we would take the approach of selecting a product from the given list of products, search for all the buyers who bought that particular product, then search for all the sellers who bought this particular product, and finally join all the results indexed on that particular product. We would simply loop through the entire list of products and generate the results. Although the solution seems straight forward, when the size of the data is scaled up exponentially, this approach turns out to be very inefficient. However, going with our intuition we can parallelise the situation. The search of relationships between the product and the sellers does not need to wait for the results from the search for the relationships between the product and the buyers. Hence these two tasks can be parallelised. The basic idea of the paper is to identify such instances where the tasks can be broken based on their independent nature among each other and run them parallely without wasting time unnecessarily by waiting for other tasks to complete whose output is completely irrelevant to other tasks. To experiment with our idea, we take the help of logic programming and run our analysis.

## IV. Core Idea

In order to device a working solution to our problem and to give life to our idea, we use Logic Programming for the purpose of demonstration as it is the simplest form of programming language to define and understand relationships between different entities. In our literature survey and background study, we majorly took insights from the ”Efficient Computation of the Well Founded Semantics over Big Data” paper and our idea is to improve the efficiency of the core idea presented in the paper. To briefly understand the basics of logic programming, let us use this simple example to witness how certain relationships can be expressed using logic programming. Recall the simple query example we discussed in the overview section above. The same query can be written in logic programming as:

ProductSales(X,Y,Z) :- buyer(X), purchased(X,Z), seller(Y), sold(Y,Z).

Let us say we have a fact set as:

buyer(Bob).
purchased(Bob,Book).
seller(Jim).
sold(Jim,Book).
seller(John).
sold(John,Book).

Now lets understand how logic programming works. Given the goal ’ProductSales(X,Y,Z)’, where X denotes the buyer, Y denotes the seller and Z denotes the product, we would unify Z with Book as it is our only product in our facts. Now the goal would change to ProductSales(X,Y,Book). Following the query, the compiler will iterate through all the clauses one by one.

1) For buyer(X) we have buyer(Bob) in our fact set.
2) For purchased(X, Book) we have purchased(Bob,Book) in our fact set.
3) For seller(Y) we have selller(Jim), seller(John) in our fact set.
4) For sold(Y, Z) we have sold(Jim, Book), sold(John, Book) in our fact set.

Unifying all the variables with the data from the facts set, we get,
X = [Bob]
Y = [Jim, John]
Z = [Book]

Hence, the output would be of the form:
–[Book] :- [Bob],[Jim,John],
meaning that there is an item Book whose buyer is a customer called Bob and whose sellers are vendors called Jim and John.
The above example was pretty straightforward. But what if

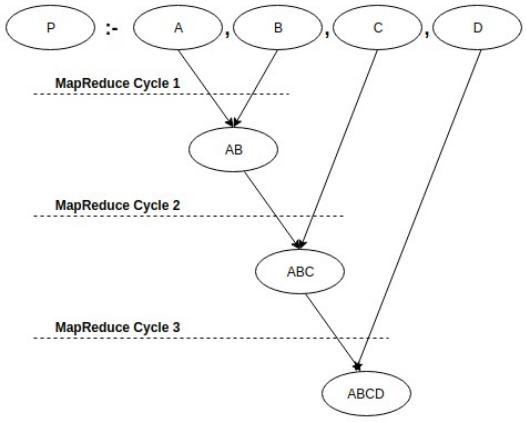


Fig. 1. Example of a Normal MapReduce cycle.

we have millions of products, buyers and sellers in our fact set. Our simple system would not be able to handle such a huge processing load. Having understood the working of logic programming, the previous workings suggest that we can parallelise this problem of computation using MapReduce when the size of the fact set becomes huge. The previous works suggest that in the given query, we can apply one MapReduce cycle to solve the two clauses : buyer(X), purchased(X,Z) joining on the common variable X, which would generate a temporary sub-goal BuyersWhoPurchased(X’,Z) where X’ is the list of buyers who purchased the product Z. We can now use this temporary sub-goal and apply another MapReduce cycle between : BuyersWhoPurchased(X’,Z), seller(Y). This would yield a new sub-goal BuyersWhoPurchasedPlusAllSellers(X’,Z,Y’) where Y’ is the list of all the sellers in our factset irrespective of the products they sold. Now using this newly formed sub-goal, weapply a final MapReduce cycle on : BuyersWhoPurchasedPlusAllSellers(X’,Z,Y’), sold(Y,Z) joining on the common key Z, which would filter out all the sellers who sold product Z. This would return our final result as

ProductSales(X',Y",Z) where X' is the list of buyers who bought product Z, Y" is the list of sellers who sold product Z.

Notice that the above approach took use three seperate

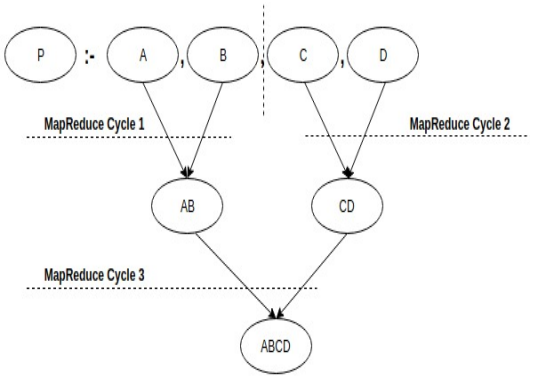


Fig. 2. Example of a Parallel MapReduce cycle.

MapReduce cycles which run in a serial order one after the other. Now imagine a generallised case where we have many clauses in our query such as:

N(a,b,c,d,e,..,z) :- N1(a,b), N2(d,e), N3(f,g),..., Nn(a,b,c,d,..,z).

We would end up applying a N-1 MapReduce cycles such as:
MapReduce Cycle 1 between N1 and N2 —- MR1
MapReduce Cycle 2 between result of MR1 and N3 —- MR2
MapReduce Cycle 3 between result of MR2 and N4 —- MR3
.
.
MapReduce Cycle N-1 between result of Mrn-2 and Nn —- MRn-1

Hence, even though MapReduce uses parallelism within itself, what do you do when there are N number of MapReduce cycles occurring in serial order one after the other. This would take a huge amount of time until we get the final results. Thus, we suggest an idea that can help us solve this problem and reduce the computation time drastically. Our idea is to identify such MapReduce jobs whose input is completely independent of any other MapReduce Jobs output. For example, in the above case if N1, N2, N3 where independent of each other by the parameters passe inside them, they can be computed parallely starting at the same time stamp and not waiting to execute one after the other.

Let us go back to our old query:
ProductSales(X,Y,Z) :- buyer(X), purchased(X,Z), seller(Y), sold(Y,Z).

Now, we know that the MapReduce cycles : buyer(X), purchased(X,Z) and seller(Y), sold(Y,Z) are nowhere related to each other. The first MapReduce cycle is computing all the buyers of a product and the second MapReduce cycle is computing all the sellers of a product. Hence the input of either of the jobs is not dependant of on the output of the latter and they can be run parallely. Thus this approach will compute two MapReduce cycle parallely at the same level and then move to compute one last cycle at the second level. The approach suggested in the previous works would take three MapReduce cycles performed over three levels executed one after the other while our method would simply take three MapReduce cycles performed over two levels. Note that the number of MapReduce cycles executed still remains the same but the number of levels are reduced by one. This will have a drastic improvement in the computation time.

Imagine a general case where there are n MapReduce cycles to be executed and out of n, k cycles are independent of each other. In such a case, those k MapReduce jobs can be initiated all at one and the remaining (n-k)+1 MapReduce jobs can be executed serially one by one. Intuitively we can say that the larger the k value, that is, the more number of independent MapReduce tasks, the more computation time we save over the traditional approach.

## V. Design And Implementation

**1. Original Work** For the actual work, we are going to generate a set of facts we described in the above section. For example, we have a set of inputs for buyers and sellers. Also, we have a set of products which a buyer has purchased and a seller has sold. All the data facts are generated randomly in a way in which the registered buyers and sellers are present in an organization. Random products are associated with these buyers and sellers. Our goal is to find a product to which a list of buyers who have bought and a list of sellers who have sold it.

We give this as the input to our MapReduce cycle. A MapReduce consists of a Map phase and a Reduce phase. It splits the inputs into fixed size chunks and start up programs on a cluster of machines. Inputs to the Map phase are key-value pairs present in those chunks. It prepares a list in the form of ¡sentence number, Buyer(Name) and ¡sentence number, Purchased (Name, Product). These splits helpful as the time taken by smaller splits is smaller than the time taken by the whole input. After Map phase it produces intermediate results which is to be processed by Reduce phase. We do not need to store these intermediate results as soon as we forward it to the next step. In Reduce phase, we shuffle and aggregate those results. This is our Map Reduce Cycle-1. From the output of those Reduce phase, we get new facts with predicate BuyerWhoPurchased (Name, Product). We are going to store the results to a persistent media or can be forwarded to the next MapReduce cycle.

The same process is repeated with Seller (Name) and Sold (Name, Product) to be sent as key value pair as ¡sentence number, Buyer(Name) to the Map phase. The output of Map

phase is then fed to Reduce phase. The result is going to be in the format of SellerWhoSold (Name, Product). This is our Map Reduce Cycle-2.

For the third round of MapReduce Cycle, we will take the result of both MapReduce cycles as input. They are going to be split in the form of key- value pair. It will undergo the Map and Reduce phase to produce our final result. This will be stored in a file or some persistent media. The result is going to be in the format of ProduceSales (Product, Set of Buyers, Set of Sellers).

We are calculating the time taken by the whole process

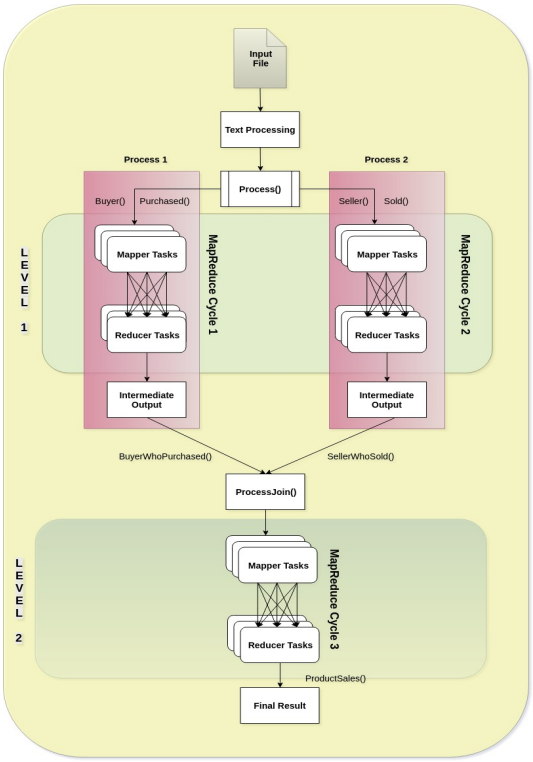


Fig. 3. Proposed Work for paralleling Map Reduce.

along with the time taken by each MapReduce cycle. This is a 3-level MapReduce cycle.

**2. Proposed Work**

We now know the procedure of how to serially work in MapReduce cycle with our problem statement of Buyers, Sellers and Products. As per our problem statement, we can segregate the facts in two halves. The two halves are independent of each other. In that case we do not have to follow a 3-level procedure for 3 different MapReduce cycles. Our job is to reduce the number of levels with the same of amount of Map Reduce Cycles required by parallelly implementing the first two MapReduce Cycles.

We observe that the set of facts that are present as input are Buyer (Name), Purchased (Name, Product), Seller(Name) and Sold (Name, Product) can be segregated into two different MapReduce Cycles. These can be executed in parallel to each other. To achieve the same, we have executed the two MapReduce cycles in two different Multiprocessing environments with two different process executing on two different CPU cores. Its implementation is similar as of Multithreaded systems. The result from the two MapReduce cycles are in the form of BuyerWhoPurchased (Name, Product) and SellerWhoSold (Name, Product). In the second MapReduce Cycle, we will take the result of both MapReduce cycles as input. Resultant will be in the form of ProduceSales (Product, Set of Buyers, Set of Sellers). We are computing the time taken by the whole process along with the time taken by each MapReduce cycle. This is a 2-level procedure for 3 MapReduce cycle.

We observed that the time taken by the parallel approach is lesser than serial approach. Also if we increase the data input size, the more the difference in time taken is going to be.

## VI. Evaluations & Observations

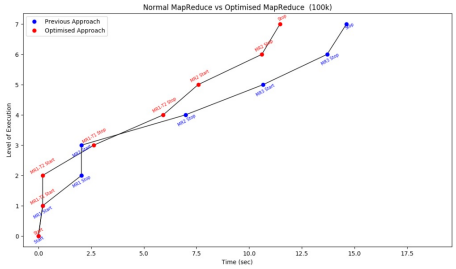


Fig. 4. Graphical Representation of both normal and parallel MapReduce cycle approach for sample 100k input.

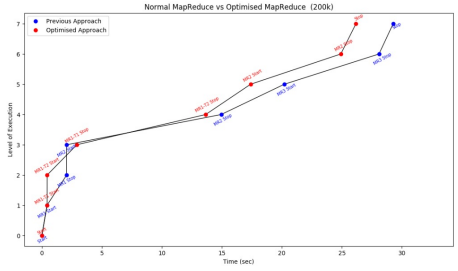


Fig. 5. Graphical Representation of both normal and parallel MapReduce cycle approach for sample 200k input.

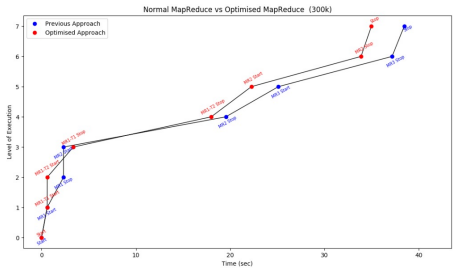


Fig. 6. Graphical Representation of both normal and parallel MapReduce cycle approach for sample 300k input.

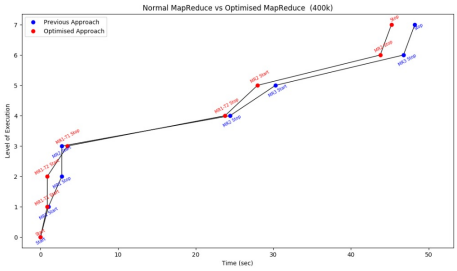


Fig. 7. Graphical Representation of both normal and parallel MapReduce cycle approach for sample 400k input.

| Data File | Start | MR1 Start | MR1 Stop | MR2 Start | MR2 Stop | MR3 Start | MR3 Stop | Stop |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |
| 100k | 0.0 | 0.322 | 2.150 | 2.150 | 7.475 | 10.432 | 13.437 | 14.602 |
| 200k | 0.0 | 0.427 | 2.065 | 2.065 | 20.943 | 20.206 | 28.094 | 29.272 |
| 300k | 0.0 | 0.597 | 2.337 | 2.337 | 19.542 | 25.120 | 37.165 | 38.456 |
| 400k | 0.0 | 1.028 | 2.754 | 2.754 | 24.419 | 30.283 | 46.765 | 48.215 |

Fig. 8. Time-stamps of stages for normal MapReduce cycle approach (in seconds).

| Data File | Start | MR1 - T1 Start | MR1 - T2 Start | MR1 - T1 Stop | MR1 - T2 Stop | MR2 Start | MR2 Stop | Stop |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |
| 100k | 0.0 | 0.232 | 0.233 | 2.598 | 5.894 | 7.728 | 10.927 | 11.815 |
| 200k | 0.0 | 0.438 | 0.440 | 2.910 | 13.651 | 17.396 | 24.911 | 26.174 |
| 300k | 0.0 | 0.600 | 0.603 | 3.376 | 17.991 | 22.258 | 33.875 | 34.973 |
| 400k | 0.0 | 0.879 | 0.882 | 3.478 | 23.773 | 27.961 | 43.768 | 45.208 |

Fig. 9. Time-stamps of stages for parallel MapReduce cycle approach (in seconds).

## VII. Discussion

### A. Other Scenarios

Example 1: In this Query, we are trying to find a simple relation like ”cousin brother” as Y. We are checking whether the parents of both X and Y are siblings and Y should be a male.

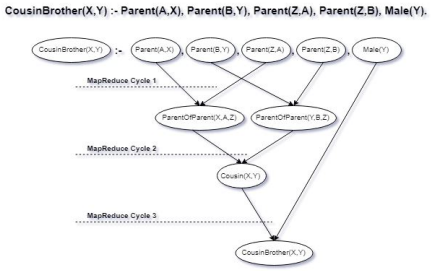


Fig. 10. There are 3 levels of parallel execution of the Relation(Cousin Brother as an example.)

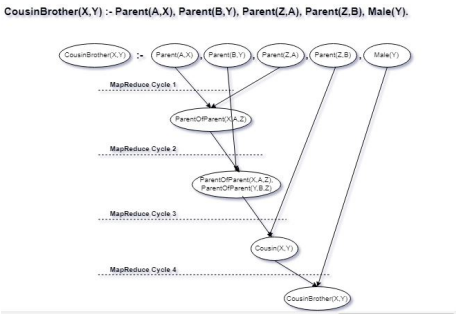


Fig. 11. There are 4 levels of normal execution of the Relation(Cousin Brother as an example.)

Example 2: In this Query, we are trying to validate the facts like whether X is a verified account and there exists a relationship like ”Father-in-law”. Then we are looking for the parent of the spouse of X.

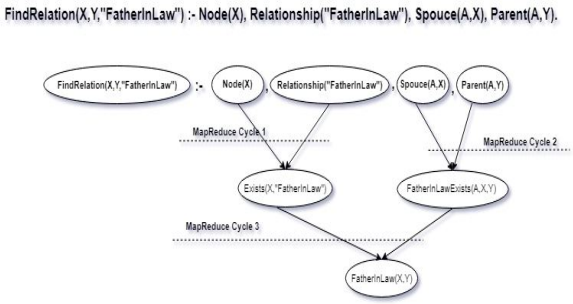


Fig. 12. There are 2 levels of parallel execution of the Finding Relationship pattern like in Facebook.

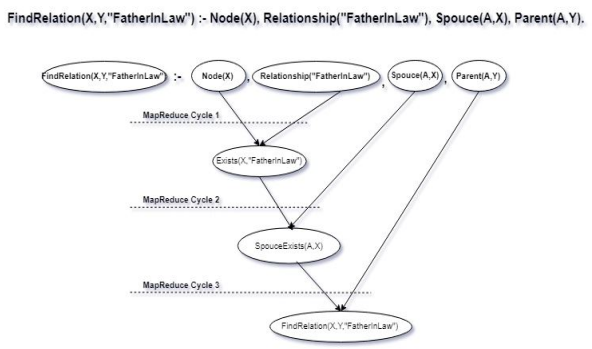


Fig. 13. There are 3 levels of normal execution of the Finding Relationship pattern like in Facebook.

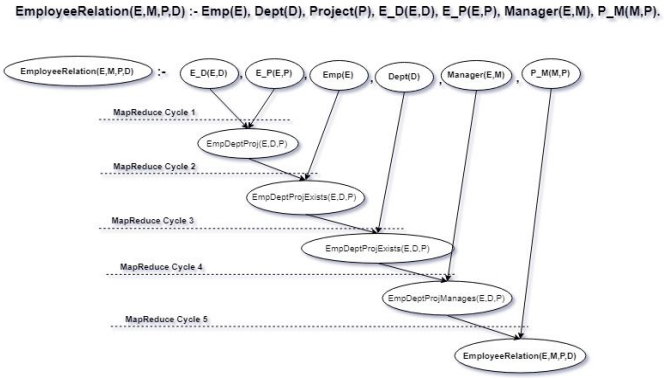


Fig. 15. We can extract Employee Information in an organization in 5 levels of normal execution.

Example 3: In this Query, we are trying to extract employee information such as the department in which he works, the project to which he was assigned and the manager under whom he is working.

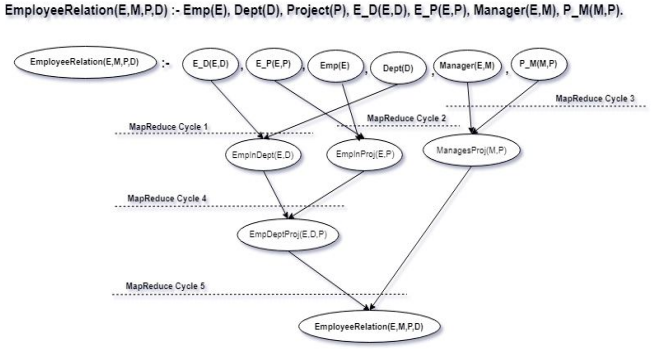


Fig. 14. We can extract Employee Information in an organization in 3 levels of parallel execution.

### *B. Work Load Distribution*

All the three team members have always remained active and wore together attending all the team meetings and sharing their thoughts. The initial workload was related to literature surveys where we had to read about the past related research works. Once we had acquired the knowledge about the past research work, we had several team meetings brainstorming several ideas and validating their feasibility. Having finalized the approach, we moved on to the implementation stage. Here, we divided the tasks based on the skill sets each member had. Ritu was assigned the task of generating the data set that we would be using for the evaluation of our scripts. She designed a python script that can take some text files as input and can flexibly generate data sets up to any specified size. Simultaneously, Harsh and Rishabh started working on the main MapReduce implementation that was discussed in the core paper which we were following. This phase of implementation too considerate amount of time and we got several inputs from Ritu during the implementation. Once this script was ready, Ritu started to test the script on various types of generated data sets (different kinds and sizes). Meanwhile, we started working on the extension of the paper where we were required to parallelize MapReduce phases. All three members before implementation, started exploring and discussing possible solutions to solve the problem. We tried multi-threading, multi-processing, used the pool resources of CPU as well but we achieved our goal by the implementation of multi-processing. Harsh and Rishabh started using multi-processing for the implementation phase and the team achieved its goal with a cumulative effort put on thoughts and sharing their coding sills. At the end, Ritu picked up the task of evaluating the results by generating graphs using the generated data sets on both the scripts that involved implementation using serial MapReduce and parallel MapReduce.

## VIII. Conclusion

Here, we have just parallelized few map reduce cycles based on the number of independent operations. With these few operations which have been parallelized, we have seen reduction in the time duration in the order of a few seconds. In a practical scenario, where we have zeta bytes of data being processed on a daily basis, we will be able to see drastic reduction in the processing times when operated in a parallel fashion and where we have lots of independent relations which can be parallelized.

We chose prolog queries to prove that our implementation of parallezing map reduce cycles is successful. Similarly, this concept can be extended to SQL and the like. Huge SQL queries can be processed to extract independent operations and then these operations can be parallelized in a similar fashion.

## References

[1] https://arxiv.org/pdf/1405.2590.pdf
[2] https://docs.python.org/2/library/multiprocessing.html
[3] http://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/
[4] https://docs.google.com/document/u/1/d/e/2PACX-1vTEhcyiTr-ANuO6sScz74OcPjZuOfwtIpvyUnDLmMkLzRL-n4Hd2zNCotxwmKW0PiFKgjCVXRg_TkFO_/pub
[5] https://github.com/karolmajek/hadoop-mapreduce-python-example
[6] https://github.com/Harsh-Wardhan-Agarwal/Projects/tree/master/AMS-_560_Project_Group_8