Beyond Simple Parallelism: Challenges for Scalable Complex Analysis over Social Data

Cong Yu
Google Research, New York, NY
congyu@google.com

ABSTRACT

With the increasing popularity of social platforms such as Facebook, twitter and Google+, applications built on those platforms are increasingly relying on the rich social activity data generated by their users to deliver personalized experiences. Those so-called social applications perform various analysis tasks to gather insights from the data and, to handle the data at large scale, adopt parallel programing paradigms for those tasks, MapReduce being the most notable model. In this position paper, we describe the challenges facing sophisticated analysis tasks where simple parallelization no longer works and pose a few questions for future research.

1. INTRODUCTION

Users' social activities on the Web are becoming a rich and critical information source for many applications that are pivoted around providing personalized user experience based on the social data. With the Facebook platform leading the pack with over 800 million users, and other social platforms such as twitter and Google+[1] also reaching hundreds of millions of users, those social applications are empowered with enormous amount of data[1] that allow them to perform various analysis tasks. Those tasks chew over hundreds of terabytes of data on a daily basis and therefore require scalable tools. Over the last few years, MapReduce[4] has emerged as the most popular computing paradigm for parallel, batch-style, analysis of large amount of data. Simplicity is one of the main reasons that MapReduce has become so popular, but it also introduces challenges for tasks that are not embarrassingly parallel[6]. For example, while counting the number of users with different profile features (e.g., age or location) is trivially parallelizable, a more sophisticated analysis of counting distinct number of users over all possible feature combinations can be very challenging. In this paper we discuss the challenges facing the MapReduce model for those non-trivial tasks.

The rest of the paper is organized as follows: Section 2 provides a brief introduction to the MapReduce model. Section 3 presents example tasks that are difficult to do using simple MapReduce programs and highlights the challenges. Section 4 poses a few questions for future research.

2. MAPREDUCE

MapReduce is a shared-nothing parallel data processing paradigm that consists of two main operations that the programmer can customize: *Map* and *Reduce*. Each **map operation** processes one record r from the input and produces zero or more $\langle key, val \rangle$ pairs using the customized map function. Each **reduce operation** processes a single key and all the vals associated with that key to produce zero or more output records using the customized reduce function. The critical aspect of the MapReduce model is that each map operation is independent from each other, and as a result, the entire input can be distributed across many machines (called *mappers*) and be processed in parallel. Similarly, the reduce operations can also be distributed across many machines (called *reducers*). The shuffling of vals into groups of the same key is accomplished by the *Shuffle* phase, which occurs between the *Map* and the *Reduce* phases, and is critical to the overall performance of the MapReduce system.

Table 1 illustrates a simple example of counting the occurrences of each character within an input set of strings.

Map Input	Map Output	Shuffle Phase	Reduce Input	Reduce Output
hello	$\langle h,1\rangle$, $\langle e,1\rangle$		$\langle h,\{1,1\}\rangle$	$\langle h,2\rangle$
	$\langle l,2\rangle$, $\langle o,1\rangle$	...	$\langle e,\{1,1\}\rangle$	$\langle e,2\rangle$
home	$\langle h,1\rangle$, $\langle o,1\rangle$	...	$\langle l,\{2\}\rangle$	$\langle l,2\rangle$
	$\langle m,1\rangle$, $\langle e,1\rangle$		...	...

Table 1: Counting Characters in MapReduce.

3. CHALLENGES

Not all analysis tasks can be easily accomplished in MapReduce. Mappers and reducers are commodity machines and therefore have limited memories and disk space. Furthermore, the overhead of launching mappers or reducers and the communication cost between jobs can be quite significant. As a result, for any given problem, there are three critical conditions that a solution must meet for a MapReduce program to solve it efficiently. The first condition is **single worker friendliness**, i.e., the workload of a single mapper or reducer needs to be within the capacity of a single machine. While each map operation is usually manageable by a mapper[2], a reduce operation can be troublesome if there are too many values associated with a single key, due to either data skew or inherent characteristics of the problem. In fact, this is often the main issue for many MapReduce programs dealing with real life data. In Section 3.1, we provide such a challenging problem.

The second condition is **limited critical operations**, i.e., the total number of required invocations of any user provided critical operations should be linear to the input size[3]. A critical operation is an expensive piece of user code that must be executed many times to solve the problem. As an example, to compute average user visit duration for each profile groups, *averaging two weighted numbers* is a critical operation in the straight forward solution. Since this operation is invoked once for each input user, this solution satisfies

[1]According its IPO filing on Feb 1, 2012: Facebook sees 2.7 billion likes and comments per day.

[2]A single record exceeding the capacity of a machine is rare.
[3]This implies that the output size is also linear to the input size, which is almost always true for practical analysis tasks.

the critical operation condition. In Section 3.2, we provide a concrete problem whose naive solution does not satisfy this condition and is therefore difficult to solve in MapReduce.

The third condition is **bounded iterations**, i.e., the total number of required map-shuffle-reduce iterations should be constant[4]. This issue is of particular importance in graph analysis where many graph problems require iterative solutions that might take a non-trivial number of MapReduce iterations, such as personalized PageRank computation[3]. Many researchers in the theory community have started to look into this—interested readers can start with those studies that aim to model the MapReduce paradigm[5, 7].

3.1 Challenging Aggregations

While most aggregation analyses fit the MapReduce model, many aggregations also turn out to be rather challenging. One prominent example is large scale cube computation with non-algebraic measures (e.g., distinct counts), which was first presented in [8]. Specifically, given a user activity log, a location hierarchy and a topic hierarchy (the latter two being derived from the user and his/her activity in the log), we want to compute `volume` and `reach` of all cube groups whose `reach` is above a certain threshold, and identify those with unusually high `reach` compared with their `volume`. Here, a cube group is defined as any combination of (location, topic) pairs, such as (DC, politics) or (all, all), the latter denoting the group of all users and topics. The measure `volume` is defined as the number of tuples in the group, while `reach` is defined as the number of unique users performing those activities. This analysis is inspired by the need to identify activities that are performed infrequently by the users, but cover a large number of users: these are often missed by traditional frequency based analyses because of their relative low volume, even though they have impact on a disproportionally large user population.

Naive solutions to the problem turn out to be very challenging for MapReduce due to the violation of the first condition: single worker friendliness. Specifically, without careful optimization, a single reduce operation for computing reach can be inundated with a large number of user IDs (the entire input in the worse case) and drag down the whole job.

3.2 Challenging Joins

Traditionally a hard problem in MapReduce, join processing using MapReduce has seen significant progresses made by a few recent studies. In particular, [9] proposed a general framework for processing database-style theta-joins using statistics gathered from a join matrix to guide the distribution of the map and reduce operations, while [10] solved a particularly important problem of Jaccard distance similarity join using a multi-iteration MapReduce pipeline with optimizations highly tuned for the problem.

However, there are many real world complex joining problems where the above techniques don't apply and as such they remain challenging to solve by the MapReduce model. One prominent example is k-nearest-neighbor joining over billions of objects with complex similarity functions, a problem we are currently investigating[2] for social networks. Those similarity functions are either too difficult to reason about because they are user provided C++ functions or simply impossible to know because they are computed using a machine learning model. As a result, naive solutions can not avoid performing $O(N^2)$ number of distance computations, violating the second condition: limited critical operations.

[4]This is a loose condition—the number of iterations can be, for example, logarithmic to the input size as long as the base factor is very large such that, in any reasonable practical settings, it is a small number.

4. QUESTIONS

We believe the above two examples are only the tip of the iceberg of many more problems that are challenging for the MapReduce model. We ask the following questions and hope other researchers can join us in seeking the answers. First, what are the core characteristics of those problems that make them so hard for MapReduce, i.e., is there a "complexity theory" for MapReduce? Second, for problems whose naive solutions violate one or more conditions discussed above, are there principled ways for identifying "good" solutions? If such principles exist, a declarative language on top of MapReduce can potentially be used to produce such MapReduce solutions, an approach us database researchers are quite familiar with. Finally, are there alternative computing models that are more suitable than MapReduce for those large scale analysis tasks?

5. CONCLUSION

In this position paper, we described challenges involved in solving large scale data analysis problems that are not straight-forwardly suited for the MapReduce paradigm. We posed some questions whose answers can get us closer to solving those challenging problems with or without the MapReduce model.

6. ACKNOWLEDGEMENT

We thank Sihem Amer-Yahia (QCRI), Boulos Harb (Google Research), Herald Kllapi (University of Athens), Arnab Nandi (The Ohio State University) for collaborations and discussions.

7. REFERENCES

[1] http://plus.google.com/.
[2] manuscript under preparation.
[3] B. Bahmani, K. Chakrabarti, and D. Xin. Fast personalized pagerank on mapreduce. In *SIGMOD*, 2011.
[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.
[5] J. Feldman, S. Muthukrishnan, A. Sidiropoulos, C. Stein, and Z. Svitkina. On the complexity of processing massive, unordered, distributed data. *CoRR*, abs/cs/0611108, 2006.
[6] I. T. Foster. *Designing and building parallel programs - concepts and tools for parallel software engineering.* Addison-Wesley, 1995.
[7] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *SODA*, 2010.
[8] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan. Distributed cube materialization on holistic measures. In *ICDE*, 2011.
[9] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *SIGMOD*, 2011.
[10] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, 2010.