Elliot Smith
Research 3 - MapReduce

1. The key need for the MapReduce programming model was that there were a rapidly growing number of tasks that required the processing of large quantities of data. These tasks, in order to be completed within a reasonable timeframe, required the computations to be done in parallel so as to expedite the total time spent on the task overall. At the time, each of these computations were being handled on a per-task basis, meaning that a solutions were being developed for each large-scale data task; each project was developed independently. The associated shortcoming is clear: in order to improve development time, if the parallelization aspect of these tasks involving large amounts of data could be abstracted out and applied broadly, a lot of development time would be saved. Essentially, if a piece of software could handle the parallelization component, the common thread of these potentially heterogeneous projects, software development time could be reduced for each task!

2. The key model that is the solution to this need is the MapReduce framework. MapReduce allows developers to focus on the computations they wish to perform that require multiple computing machines, while hiding the cumbersome details of parallelization, fault tolerance, data distribution and load balancing into a library they can use. The MapReduce model generalizes large-scale data computations into the follow operations: Map, which establishes a set of intermediary key/value pairs to be iterated on and a Reduce, which applies some computation to all of the key/value pairs that share a particular key (the Shuffle operation is ignored here as it is not explicitly mentioned in the Paper). By abstracting the details common to all large-scale data computation tasks, MapReduce was able to help software development teams focus on the particular data derivations and not on how to deal with the size and scope of the data.

3. The authors have not explicitly made note of any limitations/weaknesses or next steps, however I do think there maybe an area for improvement. The authors have clearly defined a system that handles fault tolerance very well; if a worker goes down, its task will be reassigned as other workers are available to handle it. The master is a special program compared to the workers as it helps distribute work and reassign tasks if a worker fails, among other pertinent functions. However, there is only one master assigned according to the Paper, and as a result, I could see there being an issue if the machine that is supporting the master were to fail. If the master fails then workers can no longer be assigned tasks and tasks that fail cannot be reassigned; this would paralyze the system. A solution to this could be that perhaps several “passive” masters could be assigned as a safety check. In this regard, all of the “passive” masters could periodically ping each other and the “active” master, while the “active” master will constantly share its decisions with the “passive” masters. If the “active” master goes down then one of the

"passive" masters will become active, perform the master operations and share its decisions with the other "passive" masters.

4. Something that I learned from this paper that I found very interesting is how incredible abstraction can be in terms of software engineering progress. The problem that MapReduce is solving, abstracting out the parallelization of large-scale data computation, is certainly non-trivial. I think of all the countless hours spent, but more importantly, the countless hours saved by the authors to implement an abstraction of this common thread issue. Those hours saved will be able to go towards further advancement in the field and allow others the opportunity to explore even greater improvements to the software development process. In my mind, a fact that has been reinforced by this Paper, the key to improving how we write software is by looking for the things that people are doing over and over and abstract them out in such a way that people never have to concern themselves with them ever again.

5. Lahmer, Ibrahim; Zhang, Ning. MapReduce: A critical analysis of existing authentication methods. 10th International Conference for Internet Technology and Secured Transactions. December 2015.

   This paper relates to our paper because they both are focused on the MapReduce framework. However, this paper by Lahmer and Zhang is highly critical of the security and authentication standards of MapReduce. The authors contend that security concerns were never considered by the original creators of MapReduce, and as such, MapReduce models existing on public clouds, particularly, the jobs being executed on them, are susceptible to impersonating or spoofing attacks. While the major IT players of the world, such as Facebook for example, have implemented MapReduce models in their private clouds, any public-facing cloud is vulnerable. The authors outline the weaknesses and limitations of the state-of-the-art MapReduce authentication methods against a set of specified conditions in order to illustrate their relative strengths and weaknesses.

   From this paper I learned that with any new and novel technology, there is always inherent risk, particularly as a product has had time in the market to gestate and those with nefarious intent learn to exploit its design weaknesses. It is of the utmost importance to be as forward thinking as possible in an attempt to anticipate issues that may arise, for example, that security may be an issue when there are multiple, potentially only loosely dependant, machines operating on one dataset where the results need to be carefully protected. In the event where it is unreasonable to anticipate issues, we must rely on pioneers in software engineering to find ways to mitigate or eliminate these risks.