Improving Data Processing Efficiency in Distributed File Systems

Hanxiang Pan

ABSTRACT

Many organizations use Distributed File Systems (DFS) to store and manage large data sets, and the performance of these large scale data systems is detrimental to developing breakthroughs and unveiling insights. Current DFS and MapReduce frameworks exhibit huge emphasis for efficiency and scalability. However, overtime, we have identified that users as well as the developers of the system have been experiencing slow workflow. Taking into account the architectures and tradeoff decisions in the development of these DFS and frameworks, we focused on several factors that accounts for these symptoms including data locality issues causing extra computation time, imbalanced workload amongst worker nodes leading to underutilized processing power, and wasted duplicated computations in iterative computational environments that could be easily reused.

In this paper, we will discuss the factors that contribute to the data processing efficiency in DFS. Also, we will investigate the ways to mitigate the efficiency bottleneck by improving upon the prominent DFS and MapReduce frameworks.

Index Terms

distributed file systems, load balancing, iterative computing, data locality

1 INTRODUCTION

The world in which we live in is evolving rapidly across many industries. We have come to a point where humans can make decisions relying on data using patterns from the past to draw insights. In 2004, the advent of *Google File System* (GFS) diminished the traditional data processing bottleneck by allowing multiple cheap machines, located in various geographic locations, to work on the same job concurrently. (Mone, p. 22) At the same time, the inception of the *MapReduce* framework redefined how computational tasks can be coordinated and broken down. (Mone, p.23) These two breakthroughs rendered data processing to become more accessible to a wider range of data consumers.

However, with more data being aggregated everyday, people are looking for affordable solutions to analyze these enormous amounts of data within reasonable time. Personally, I have encountered the situation where a data task was submitted to the remote server (worker node) for processing and it took very long for a simple query to return with the result.

1.1 MapReduce Framework Overview

MapReduce is a framework that is designed to break down a certain data processing task into chunks and distributed each chunk to remote machines, aggregate the results into meaningful output. The process is depicted in Figure 1. The data

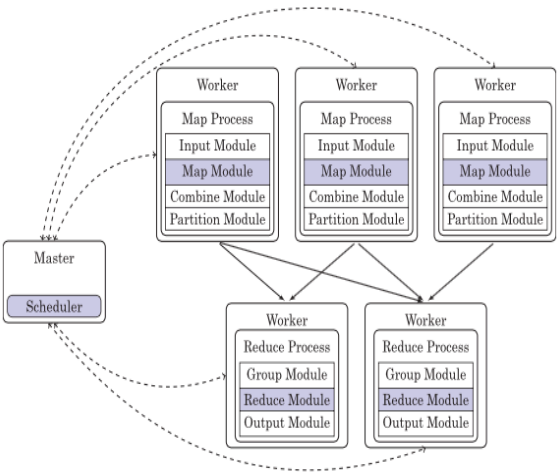

Fig. 1 – MapReduce (Li, Ooi, Özsu, Wu, p. 5) Framework

processing task is done in a two-step fashion consisting of a *Map* function and a *Reduce* function.

First, all the input data are broken down into equal-sized pieces, then the *scheduler* assigns these pieces to all the worker nodes. The *Map* function takes these pieces, converts it to key-value pairs (mapKey, mapValue) and maps them to different compute nodes. If there are some key-value pairs that resembles the same value, they are constructed into a list (list(mapKey, mapValue)) for preparation to the next step. On each compute node, these key-value pairs are then sorted by the keys. Then a *Reduce* function takes the sorted key-value pairs from each compute node as input and aggregates the values by grouping those with the same keys into lists (reduceKey, list(reduceValue)). (Li et al., p. 3-4) Finally, the *Reduce* function appends the final output to the rest of the output, if any.

A big bottleneck in the *MapReduce* workflow is the data locality issue. Among all the nodes (machines), there is a *master* node that schedules tasks to the *worker* nodes. When scheduling, the time spent on moving data from a remote node to the computation node is greater than the computation itself. This issue directly affects the performance of the whole data processing workflow. (Sakr, Liu, Fayoumi, p. 5)

Iterative data process is also common in data intensive applications, such as finding the shortest distance between two arbitrary points, also known as the Single-source Shortest Path algorithm, or the famous Google's PageRank algorithm. The native MapReduce framework doesn't allow for efficient handling of this type of data process, and thus potential for improvements in this area is present. (Zhang, Gao, Gao, Wang, p. 1884)

GFS is a DFS that is considered to be the fundamental stepping stone of modern data intensive applications. (Mone, p.22) It's scalability is linear due to the commodity hardware requirements to achieve massive data processing power.

However, there is still potential for improvement in terms of data processing efficiency. Several main factors that contribute to data processing performance in DFS include: data locality (Section 2), load balancing (Section 3) and iterative computations (Section 4). Also, we will be showing several methods to improve data processing efficiency with regards to each factor.

The study will be tested and evaluated in Section 5. Related works regarding data processing in distributed file systems can be found in Section 6. The study is summarized in Section 7.

2 DATA LOCALITY PROBLEM

The architecture of *GFS*, as depicted in Figure 2, is made up of a single master and multiple *chunkservers*. Files that are to be processed are broken down into fixed-sized chunks. The master controls and maintains all the metadata regarding these chunks, including the namespace, mapping from files to chunks and the location of chunks. (Ghemawat et al., p. 31)

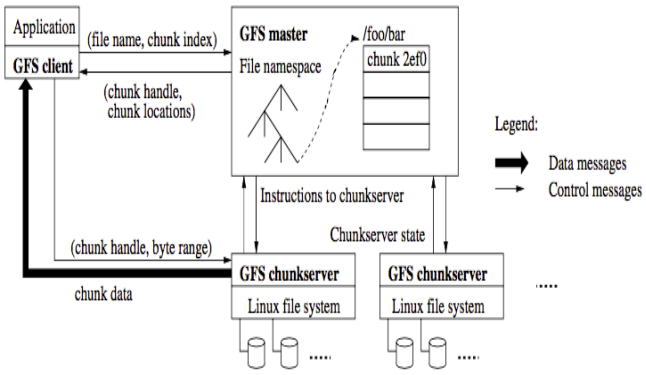

Fig. 2 - GFS Architecture (Ghemawat, Gobioff, Leung, p. 31)

2.1 Block vs File

The most typical operation in data processing is the read operation. The *GFS* client (application) requests the file chunk from the *GFS* master using the file name and the chunk index, the master responds with the chunk location (replica) and the chunk handle. This interaction could go back and forth as more than one file chunk may be requested. The chunk is then cached locally in the client to save extra overhead. Then the client can use the chunk handle and chunk locations to locate the closest replica. (Ghemawat et al., p. 38) However, the master doesn't retain the location of chunks, instead, it periodically requests data from the chunk server. In other words, the placement of files is entirely independent of *GFS* applications and therefore optimizing for data locality is crucial to improving the overall data processing efficiency. (Ghemawat et al., p. 31)

Performance improvement can be realized by keeping a file on a single node without breaking it into chunks, for two reasons. Firstly, for applications that don't know the semantics of the file, it would be difficult for those applications to quickly identify the boundaries of blocks for integration. Also, having chunks require another extra step of gathering and integration before the actual computation can be performed. On the other hand, eliminating chunks would limit the flexibility of load balancing, since it is a lot easier to switch around chunks, than to move around files, to achieve efficient load balancing.

2.2 Data Locality at the Job Level

In *DFS*, to ensure data reliability, multiple copies of the same data item are stored in the multiple nodes. In the context of the *MapReduce* framework, the scheduler selects the map tasks that are closest to the data item.

In a case where there are many long-term tasks running on a node, this node would be unavailable for local tasks, which takes much shorter time. Another case is where certain local data is used to support many jobs.

Nodes that have local data for many jobs (hotspots), these jobs would would have to wait to be scheduled on the few nodes that have small input files, resulting in long delays for some of these jobs. (Tiwari, Sarkar, Bellur, Indrawan, p. 24)

One way of solving this issue is to constraint the scheduling algorithm by the amount of data transfer. When scheduling a task to a node, we can construct a bipartite graph in which the amount of data transfer is represented by an edge from a task to a node. And by finding the minimum cost path through the graph, we can ensure that data non-locality doesn't hinder the overall efficiency of the process. (Tiwari et al., p. 24)

3 LOAD BALANCING PROBLEM

Another issue with DFS for intensive data processing is related to imbalanced load among computing nodes. In a *DFS* that follows the *GFS* model and uses the *MapReduce* framework, files are broken into equal-sized chunks that are further allocated to nodes for the Map tasks. As the computational process continues, files in remote servers may be created, deleted, or modified, this leads to certain computing

nodes having less or more chunks than the average computing node. In essence, this creates the load balancing problem, since the network bandwidth is not fully utilized and additional movement costs might occur to increase the utilization. (Hsiao, Chung, Shen, Chao, p. 952)

Figure 3 shows a scenario of imbalanced chunks after some basic file operations. (a) shows an initial balanced state with three nodes N1, N2, N3. (b) Files f2 and f5 are deleted. (c) File f6 is appended to node N2. (d) Node N4 joins the cluster. In this case, subgraph (c) and (d) are imbalanced.

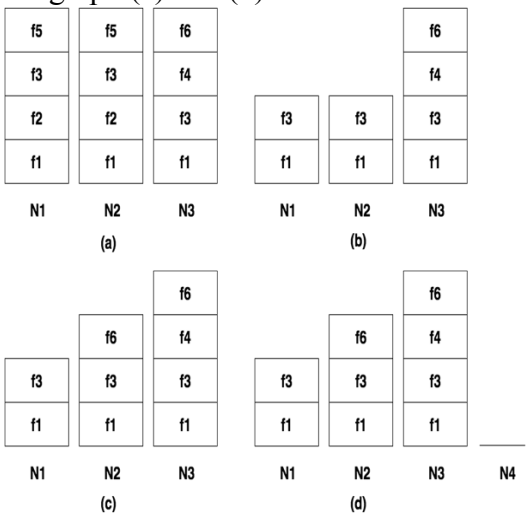

Fig. 3 – Imbalanced Load Example (Hsiao et al., p. 953)

Load balancing, in this context, can be handled through two cases, nodes that have global knowledge and without global knowledge. We will discuss each case separately. In both cases, we need to utilize the capability of *Distributed Hash Tables* (DHS) to speed up the process.

3.1 Load-balancing Nodes with Global Knowledge

The underlying assumption of this solution is that all chunk servers are homogeneous, with the same storage capacity and equal movement costs between any two nodes. In addition, each node has global knowledge of all other nodes. We will define the balanced threshold of chunks to be A, a light node to be $(1-\ell)\times A$, a heavy node to be $(1+\mu)\times A$, where ℓ and μ are between 0 and 1. Our goal is, by using the light nodes, to balance the load from the heavy nodes. Firstly, we take the least loaded node i in the system, and migrate its chunks to its successor node $(i+1)$, then release node i from the system. Secondly, we can use node i to relieve the heaviest node (node j) in the system, allowing node i to rejoin the system as the successor of node j, node $(j+1)$, and the original successor of node j will be pushed to become node $(j+2)$ and so on. The actual relieve operation involves taking $min\{\mathcal{L}_j - A, A\}$ amount of load from node j and load it on to node $(j+1)$, or formerly node i. However, this doesn't guarantee that node j is no longer the heaviest node in the system. We will repeat the process by choosing the least-loaded node i_2 and repeating the process until the heaviest node is entirely relieved. At this point, one iteration of load balancing is complete. The next iteration is to find the next heaviest node in the system and relieve it using the previous steps. (Hsiao et al., p. 954)

This method of load balancing renders two advantages. Minimizing the movement cost of balancing nodes, since the least-loaded node is already offloaded before rejoining, the only movement is between the successor and the predecessor. Also, it allows quick convergence rate for the whole system as the least-loaded node is the most effective node to balance the heaviest node. (Hsiao et al., p. 954)

3.2 Load-balancing Nodes without Global Knowledge

It is usually difficult and infeasible for all the nodes in the distributed system to have global knowledge due to dynamic environment and large amounts of data. This

section, we will investigate into the second case where nodes don't have global knowledge.

This method involves two algorithms, *Algorithm 1* performs the relieving operation from the heavy node j to the light node i, note that we no longer have the information of which node is heaviest and lightest. *Algorithm 2* collects a sample of nodes and extracts their node statuses to build a vector $\mathcal{V}$. This vector consists of their ID, network address and load status of each randomly selected node. Since each vector will have different number of nodes, some adjustments must take place - each vector exchanges its load with a neighbouring vector until all vectors have a certain number of nodes, denoted as s. The algorithm then calculates the average load of the s nodes.

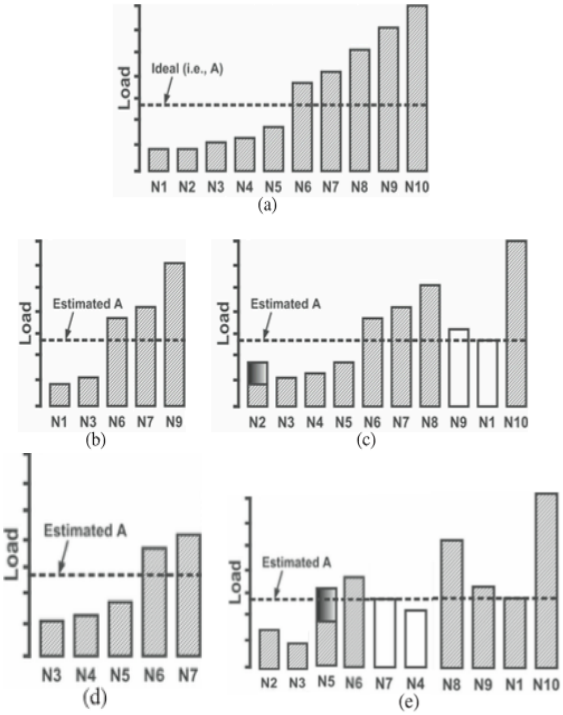

Fig. 4 - Load Balance Cases (Hsiao et al., p. 955)

The load-balancing step comes into action at this stage. Every node first determines if it is a light node, if so, then it will relieve the load of the heaviest node, measured by the largest standard deviation in the sample, by requesting some of its chunks.

To take a closer look at this algorithm, Figure 4 shows the algorithm for load-balancing without global knowledge. (a) depicts the initial state of different nodes N1 to N10. (b) to (c) shows the following: Let node N1 be the subject node for this example. N1 randomly selects some nodes in the system and builds the vector V. N1 estimates the average load A based on:

$$A = (\mathcal{L}\eta_1 + \mathcal{L}\eta_3 + \mathcal{L}\eta_6 + \mathcal{L}\eta_7 + \mathcal{L}\eta_9) \div 5$$

Then N1 compares itself with A, it realizes that it is a light node. N1 needs to find a heavy node to perform the relieve operation. In this case, the heaviest node within the sample is N9, N1 offloads it current load to its successor N2, and leaves the system. Similarly to the global knowledge algorithm, N1 rejoins the system as the successor of N9 and takes $min\{\mathcal{L}\eta_9 - \mathrm{A}, \mathrm{A}\} = \mathrm{A}$ chunks from N9. In (d), N4 becomes the subject of the example, it performs random sampling of N3, N4, N5, N6, N7, and N4 offloads its load to its successor, N5 and randomly rejoins as the successor of the heaviest node in the sample. (Hsiao et al., p. 955)

4 ITERATIVE COMPUTATIONS

In today's world, many use cases of data processing in distributed file system have an iterative component to it. *Iterative Computation* is defined as "An iterative algorithm typically performs the same computation on a data set in every iteration, generating a sequence of improving results." (Zhang & Chen, p. 1) A prominent example of *iterative computation* is Google's *PageRank*. The basic idea of this algorithm is to rank web pages based on how many links are directed to a certain page. (Franceschet, p. 92) In other words, "A Web page is important if it is pointed to by other important pages". (Franceschet, p. 94) Each page is assigned a score by the quantity and quality of

hyperlinks to that page. Since not all links are weighted equally, the quality of a link is determined by the source page score of the link, which is determined by the links pointing to the source page and so on. The iterative piece is involved since each page score is calculated using the output of the link score as inputs for the next iteration. If all the scores are computed from scratch every time a page rank is calculated, there would be many duplicated intermediary values (link weights), wasting precious resources.

If we can reuse the output of the last iteration for the current iteration, we would be able to save many repeated calculations/operations. Figure 5 shows the data stream partitioned into two flows, a static flow (i.e. graph structure used to feed the map function) and a state flow (i.e. *PageRank* score). Prior to start of the *MapReduce* framework, both static and state data are allocated to the respective nodes

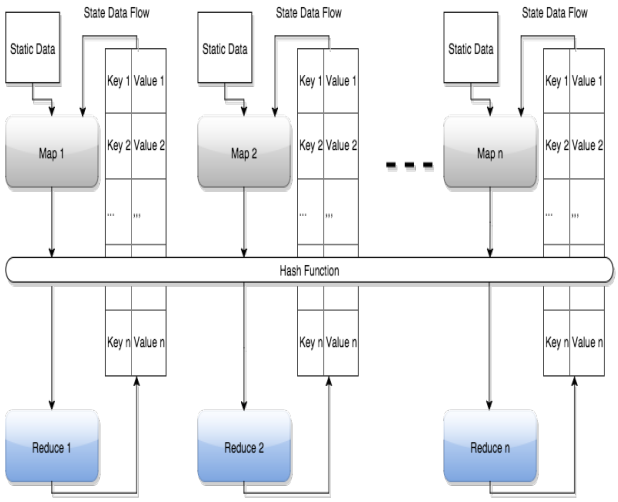

Fig.5 – Improved MapReduce process with iterative processing structure[1]

(workers), the algorithm will join the two streams before the map operation. The state data provides the input for the map function, which includes a *key*, representing *node ID* (nid), and the *value* of the node. The most important component is the value of the node, which represents the node state (i.e *PageRank* score). The static data includes node linkage info, indexed by *nid*. Another step is to evenly partition the two streams of data using a hash function F. Each partition is assigned to a working node by the master, and each working node can work with more than one partition. The output of the hash function is a partition id (pid), $pid = \mathrm{F}(nid, n)$, where n is the total number of desired partitions. (Zhang et al., p. 1887)

The *MapReduce* framework will be used at this stage. The *map* function will process the partition with the pid equal to the map task id. The output of the *map* function is a pair <key, value>/<node, state>. By using the same hash function F, this output is sent to the *reduce* function with the same pid throughout. The *reduce* function aggregates and updates the results from the state data and outputs a pair of <node, state>, which is further sent to the next *map* function iteration. To ensure that the dynamic state data and the static data is always joined on to the same node, the hash function F is consistent throughout the entire process for both partitioning and *MapReduce*. (Zhang et al., p. 1887)

The key component to improving efficiency in iterative computations is maintaining the state values. During the iterative process, a node's state is updated only after every iteration. When the state value needs to updated, its access is maintained through "a *StateTable* implemented with an in-memory hash table". (Zhang et al., p. 1887) Two types of states should be taken in consideration when dealing with iterative computations, *iterative state* and *cumulative state*. The iterative state is solely used for iterative computations, and the cumulative state is specifically used to aggregate previous iterations. (Zhang et al., p. 1887)

4.1 Page Rank Use Case

The *PageRank* algorithm utilizes this mechanism to achieve higher performance

and efficiency. The iterative state in this case contains the incremental *PageRank* score that is calculated for an additional web page pointing to the subject web page. The cumulative state contains the aggregated *PageRank* score for all the links pointing to the subject web page so far. Thus, in every iteration, the *PageRank* algorithm must perform the necessary incremental computation of the added link to update the iterative state, and maintain the accumulated state to reflect the most recent scores of each page. The benefit of this method for *PageRank* is that the *reduce* function updates the corresponding entry in the *StateTable* according to the received value, rather than performing a reduce function on all the received values associated with the same key. (Zhang et al., p. 1887)

5 EVALUATION AND RESULTS

5.1 Load Balancing Simulation

Our simulations involve 1000 nodes with 10,000 file chunks. To simulate the distribution of file chunks on a certain node, we assume the file chunks follow a geometric distribution. To measure the performance of the load balancing algorithm, we will compare it with the typical distributed matching mechanism as well as the centralized matching mechanism that has been prevalent in the industry. The main factor for comparison is movement costs, we have defined a criterion, weighted movement cost (WMC). (Hsiao et al., p. 959)

$$WMC = \sum_{chunk\ i \in \mathcal{M}} size_i \times link_i$$

$size_i$ represents the size of the file chunk being balanced, and $link_i$ represents the number of physical links the chunk *i* needs to travel to balance the load. For the purpose of the simulation, we assumed that $size_i = 1$ for all chunks, without the loss of generality. Thus, we can infer that the greater the number of links each chunk has to go through results in a greater WMC. (Hsiao et al., p. 959)

Figure 6 shows that the cost of the proposed load balancing algorithm is 0.37 times the cost of distributed matching. The reason is because for distributed matching, the heaviest-loaded node is not relieved by

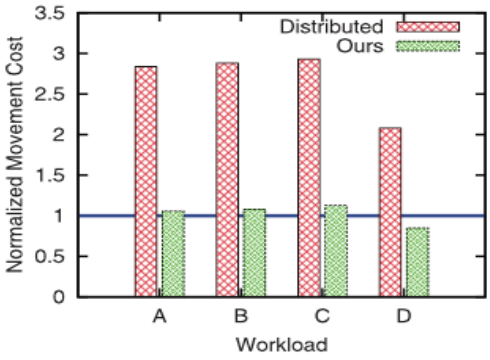

Fig. 6 (Hsiao et al., p. 959)

the least-loaded node, rather by a relatively less heavy node. (Hsiao et al., p. 959)

5.2 Iterative Computing Simulation

To evaluate the performance of the iterative computing algorithm, we will use a

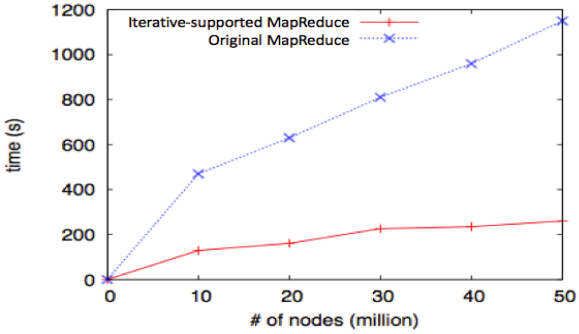

Fig. 7 (Zhang & Chen, p. 3)

20-node cluster, and test it to the PageRank use case. By randomly changing 10,000 edges in a simulated web graph, we identify that the time taken to perform MapReduce operation is significantly lower in an iterative-supported system. Fig. 7 shows that MapReduce with iterative processing support outperforms the original MapReduce

framework by almost 5 times. With many use cases such as the PageRank algorithm and Single-Source Shortest Path Algorithm (SSSP), handling iterative computing is crucial in scaling up data intensive applications.

6 RELATED WORK

6.1 HadoopDB Hybrid

Considering Hadoop being an advancement from the Parallel Database Management Systems (DBMS), researchers have identified advantages from both systems and developed a hybrid architecture called HadoopDB. This architecture allows the parallel DBMS to retain its superior performance over Hadoop, mainly due to having no load and modelling procedure. Also, since Parallel DBMS assumes that failure is rare, pairing up with Hadoop with MapReduce's exceptional fault tolerance, scalability (up to 1000+ nodes), flexibility to handle unstructured data and low cost, this issue can be alleviated. With more data, specifically structured data, being aggregated overtime, the demand for large node systems and structured data processing capability is higher than ever before. With the hybrid system, these problems can be efficiently solved. Since the hybrid system uses both architectures, developers can write queries in SQL and the system will be able to translate it into MapReduce code. The MapReduce is used as the communication layer above the worker nodes to be communicated with the single node databases. (Abouzeid, Bajda-Pawlikowski, Abadi, Silberschatz, Rasin, p. 2)
Whereas, our approach is to identify the bottlenecks of DFS and try to improve on them.

6.2 Data Access Prediction and Optimization

Rather than improving the factors that contribute to performance in distributed file systems, some researchers have taken a different approach - data access prediction and optimization. This method allows for the system to predict application behavior through analyzing past data and creating a time series model of those behaviors.

Throughout the process, the system spends 80% of the data sets training the model, and 20% evaluating the prediction results. The main algorithm is to execute an optimization heuristic to perform the data operations. With more data sets collected, the algorithm will become more accurate in reducing the time spent in data access operations. (Ishii & Mello, p. 1018)

7 CONCLUSION AND FUTURE STEPS

7.1 File vs Block

A crucial tradeoff proposed is whether to keep files as a whole on a node or break them down into blocks. Considering many different types of file operations can break files into smaller chunks, keeping blocks on nodes allows for greater flexibility in load balancing as well as improves efficiency when performing computations that require data to be copied from a remote node to the computation node. Having the whole file being copied greatly increases the overhead of each computation, leading to a data locality issue. In addition, load-balancing requires file chunks since not all nodes have the capacity to accommodate the entire file size.

7.2 Load Balancing

With file chunks implemented, file operations can create holes in nodes leading to an imbalanced state. Load-balancing allows DFS to fully utilize all the infrastructure by minimizing the amount of idle working nodes/capacity. In the worst case, load-balancing without global knowledge of loads can still be balanced through taking random samples. Within each sample, the system

calculates the local average and relieves the relatively heavy-loaded nodes using the relatively light-loaded nodes. The performance approximately converges to the case with global knowledge overtime.

7.3 Data Locality

It is difficult to ensure that a selected node for job execution is the most ideal. Especially in the case where several long-term jobs all request data from a specific node, this node becomes a hotspot. Fortunately, this issue can be mitigated by limiting the amount of data transfers between any two nodes through the implementation of a minimum flow graph.

7.4 Iterative Computations

In cases where iterative computations are common, saving as much duplicated steps is essential for improving data processing efficiency. This can be realized through an implementation of an in-memory hash table to store the state data

7.5 Future Steps

One area with regards to load balancing, that should be considered in the future, is simultaneous execution. Currently, there are only sequential execution when balancing loads among different nodes. As more data consumers utilize DFS to help them perform data-related computations, efficient data processing will be the key success factor towards the digital age.

8 REFERENCES

Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., & Rasin, A. (2009). HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. Proc. VLDB Endow. Proceedings of the VLDB Endowment, 922-933.

Franceschet, M. (2011). PageRank: Standing on the Shoulders of Giants. Communications of the ACM Commun. ACM, 54, 92-101. doi:10.1145/1953122.1953146

Ghemawat, S., Gobioff, H., & Leung, S. (2003). The Google file system. Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles - SOSP '03, 37(5), 29-43. doi:10.1145/1165389.945450

Hsiao, H., Chung, H., Shen, H., & Chao, Y. (2013). Load Rebalancing for Distributed File Systems in Clouds. IEEE Trans. Parallel Distrib. Syst. IEEE Transactions on Parallel and Distributed Systems, 24, 951-962. doi:10.1109/TPDS.2012.196

Ishii, R., & Mello, R. (2012). An Online Data Access Prediction and Optimization Approach for Distributed Systems. IEEE Trans. Parallel Distrib. Syst. IEEE Transactions on Parallel and Distributed Systems, 23, 1017-1029. doi:10.1109/TPDS.2011.256

Mone, G. (2013). Beyond Hadoop. *Communications of the ACM Commun. ACM, 56*, 22-22. doi:10.1145/2398356.2398364

Li, F., Ooi, B., Özsu, M., & Wu, S. (2014). Distributed data management using MapReduce. CSUR ACM Comput. Surv. ACM Computing Surveys, 46(3), 1-42. http://dx.doi.org/10.1145/2503009

Sakr, S., Liu, A., & Fayoumi, A. (2013). The family of mapreduce and large-scale data processing systems. *CSUR ACM Comput. Surv. ACM Computing Surveys, 46*(1), 1-44. http://dx.doi.org/10.1145/2522968.2522979

Tiwari, N., Sarkar, S., Bellur, U., & Indrawan, M. (2015). Classification Framework of MapReduce Scheduling

Algorithms. CSUR ACM Comput. Surv. ACM Computing Surveys, 47, 1-38. http://dx.doi.org/10.1145/2693315

Zhang, Y., & Chen, S. (2013). I^2 MapReduce. Proceedings of the 2nd International Workshop on Cloud Intelligence - Cloud-I '13. doi:10.1145/2501928.2501930

Zhang, Y., Gao, Q., Gao, L., & Wang, C. (2012). PrIter: A Distributed Framework for Prioritizing Iterative Computations. IEEE Trans. Parallel Distrib. Syst. IEEE Transactions on Parallel and Distributed Systems, 24, 1884-1893. doi:10.1109/TPDS.2012.272