Memory or Time: Performance Evaluation for Iterative Operation on Hadoop and Spark

Lei Gu, Huan Li
State Key Laboratory of Software Development Environment
School of Computer Science and Engineering
Beihang University, Beijing, China
gulei@act.buaa.edu.cn, lihuan@buaa.edu.cn

***Abstract*—Hadoop is a very popular general purpose framework for many different classes of data-intensive applications. However, it is not good for iterative operations because of the cost paid for the data reloading from disk at each iteration. As an emerging framework, Spark, which is designed to have a global cache mechanism, can achieve better performance in response time since the in-memory access over the distributed machines of cluster will proceed during the entire iterative process. Although the performance on time has been evaluated for Spark over Hadoop [1], the memory consumption, another system performance criteria, is not deeply analyzed in the literature. In this work, we conducted extensive experiments for iterative operations to compare the performance in both time and memory cost between Hadoop and Spark. We found that although Spark is in general faster than Hadoop in iterative operations, it has to pay for more memory consumption. Also, its speed advantage is weakened at the moment when the memory is not sufficient enough to store newly created intermediate results.**

I. Introduction

The rapid development of the Internet has generated vast amount of data that poses big challenges to traditional data processing model. To deal with such challenges, a variety of cluster computing frameworks have been proposed to support large-scale data-intensive applications on commodity machines. MapReduce, introduced by Google [2] is one such successful framework for processing large data sets in a scalable, reliable and fault-tolerant manner.

Apache Hadoop [3] provides an open source implementation of MapReduce. However, Hadoop is not good for iterative operations which are very common in many applications. MapReduce cannot keep reused data and state information during execution. Thus, MapReduce reads the same data iteratively and materializes intermediate results in local disks in each iteration, requiring lots of disk accesses, I/Os and unnecessary computations. Spark is a MapReduce-like cluster computing framework [1] [4], but it is designed to overcome Hadoop's shortages in iterative operations. Spark introduces a data structure called resilient distributed datasets (RDDs) [5], through which reused data and intermediate results can be cached in memory across machines of cluster during the whole iterative process. This feature has been proved to effectively improve the performance of those iterative jobs that have low-latency requirements [5].

Although Spark can achieve tremendous speedup, we suspect the memory cost it has to pay for the introduction of RDDs. In addition, how to quantify the tradeoff in terms of time and memory for iterative operations is another interesting problem.

In this paper, we attempted to conduct exhaustive experiments to evaluate the system performance between Hadoop and Spark. We choose a typical iterative algorithm PageRank [6] to run for both real and synthetic data sets. Experimental results show: 1) Spark can outperform Hadoop when there is enough memory for Spark in the whole iterative process; 2) Spark is memory consuming. Although input datasets are not that large compared to the whole memory of the cluster, intermediate results generated at each iteration can easily fill up the whole memory of the cluster; 3) Hadoop has better performance than Spark when there is not enough memory to store newly created intermediate results.

This paper is organized as follows. Section II provides system overview of Hadoop and Spark. Section III describes our experimental settings. Section IV reviews the PageRank algorithm and shows our implementation of PageRank on Hadoop and Spark. Section V presents results of our experiment. Related Work is in Section VI. We give our conclusions and future work in Section VII.

II. System Overview

As a general framework for many different types of big data applications, such as log analysis, web crawler, index building and machine learning, Hadoop [3] provides an open source Java implementation of MapReduce. Hadoop is composed of two layers: a data storage layer called Hadoop distributed file system (HDFS) and a data processing layer called Hadoop MapReduce Framework. HDFS is a block-structured file system managed by a single master node. A processing job in Hadoop is broken down to as many Map tasks as input data blocks and one or more Reduce tasks. An iterative algorithm can be expressed as multiple Hadoop MapReduce jobs. Since different MapReduce jobs cannot keep and share data, frequent used data has to be read from and written back to HDFS many times. Those operations will obviously incur a lot of disk accesses, I/Os and unnecessary computations [7] [8] [9] during the execution of iterative algorithm.

Spark is a novel cluster computing framework that is designed to overcome Hadoop's shortages in iterative operations. Spark does not have its own distributed file system. Instead, it uses Hadoop supported storage systems (e.g, HDFS, HBase) as its input source and output destination. Spark introduces a data structure called resilient distributed datasets (RDDs) to cache data. The runtime of Spark consists of a driver and multiple workers. Before an iterative operation, the user's driver program will launch multiple workers. These workers will read data blocks from a distributed file system and cache them in their own memory as partitions of RDD. These partitions form a whole RDD from the view of the driver program. The RDD feature of Spark avoids data reloading from disk at each iteration, which dramatically speeds up the iterative operation. Users can explicitly cache reused data and intermediate results in memory in the form of RDDs and control their partitioning by parameter tuning to optimize data placement. Since RDDs are partitioned across the memory of machines in the cluster, they can be processed in parallel. RDDs are fault-tolerant which means they can be rebuilt if a partition of them is lost. RDDs are read-only, so changes to current RDDs will cause creation of new RDDs.

III. Experimental Environment

A. Cluster Architecture

The experimental cluster is composed of eight computers. One of them is designated as master, and the other seven as slaves. We use the operating system Ubuntu 12.04.2 (GNU/Linux 3.5.0-28-generic x86 64) for all the computers. Table 1 shows the hostname, machine modal, IP address, CPU and memory information of the eight computers mentioned above. All of the eight computers are in the same Local Area Network and connected by a H3C S5100 switch whose port rate is 100Mbps. All of the eight computers have 4 cores, but master and slaves have different frequencies.

Apache Mesos [10], [11] is a cluster manager that provides efficient resource isolation and sharing across distributed frameworks. It can run Hadoop, Spark, MPI, Hypertable and other frameworks. We use Mesos to manage the resources of our experimental cluster. Fig. 1 shows the architecture, where each slave contributes 4 CPU cores and 3GB memory to Mesos. So the total resources managed by Mesos are 28 CPU cores and 21 GB memory. When a job is submitted to Hadoop or Spark, Mesos will allocate resources to it according to its demanding needs.

We use Mesos 0.9.0, Hadoop 0.20.205.0 and Spark 0.6.1 for all the experiments shown below, since Mesos 0.9.0 and Spark 0.6.1 are relatively new stable release. Although Hadoop 1.1.2 [1] is the most recent stable release as of writing this paper, only Hadoop 0.20.205.0 is ported to run on mesos. We choose Hadoop 0.20.205.0 in order to let Spark and Hadoop share the same experimental conditions.

B. Dataset Description

Data size and data connectedness are two important properties of a specific dataset. In terms of graph dataset, the number of nodes reflects its size and the number of edges reflects its internal connectedness. These two properties have great impact on the running time and memroy usage of the PageRank algorithm. Also, PageRank run on directed graphs. So we will choose directed graphs with different node number and edge number as our experimental datasets.

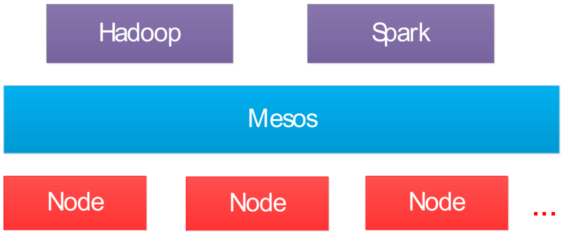


Figure 1: Cluster architecture.

We choose five real graph datasets and generate five synthetic graph datasets to do comparative experiments. Table 2 lists these ten graph datasets. They are all in the format of edge list. To be more specific, each line in the file is a [src ID] [target ID] pair separated by whitespace. The first four real graph datasets come from SNAP [12] and the fifth one is obtained from [13]. We choose them because they are directed, have significant differences in size and are in different application fields. The five synthetic graph datasets are generated by stochastic Kronecker graph generator. According to , the Kronecker generator can generate graphs with similar properties as real-world graphs, such as heavy tails for in- and out-degree distribution, small diameters, and densification and shrinking diameters over time. We utilize the Kronecker generator provided in SNAP [12]. We first create a radom seed graph with only two nodes, then pass it to the generator and make it iterate the specified number of times to get the expected size of synthetic graph we need. We will use the name attribute to reference the specific graph dataset in the following description.

IV. Implementation

A. PageRank Overview

The basic idea behind PageRank is that a node which is linked to by a large number of high quality nodes tends to be of high quality. In order to measure a node's quality, an authoritive value called PageRank value can be assigned to each node in the whole graph. Formally, the PageRank P of a node n is defined as follows:

$$P(n) = \sum_{m \in L(n)} \frac{P(m)}{C(m)} \tag{1}$$

where $L(n)$ is the set of nodes that link to n, $P(m)$ is the PageRank of node m, $C(m)$ is the out-degree of node m.

There are two anomalies we need to prevent. The first one involves the structure called *spider trap* which are group of nodes that have no links out of the group. This will cause the PageRank calculation to place all the PageRank within the spider trap. The second one is related to *dangling nodes* which are nodes in the graph that have no outlinks. The total PageRank mass will not be conserved due to dangling nodes.

To deal with the first problem, a random jump factor α can be added to the Formula 1. According to [14], the second

[1] We also install Hadoop 1.1.2 separately to repeat the experiment, and find that the new version Hadoop does have performance improvement and the improvement is a constant under different graph datasets. Due to space limitations, we will not show the results here.

Hostname	Machine Modal	IP	CPU Info	Total Memory	Free Memory
master	Dell Optiplex 790	192.168.2.99	Intel i5 3.10GHz	3910MB	˜3400MB
slave0	Dell Optiplex 980	192.168.2.100	Intel i5 3.20GHz	3816MB	˜3400MB
slave1-slave6	Dell Optiplex 960	192.168.2.101-192.168.2.106	Intel 2 Quad 2.83GHz	3825MB	˜3400MB

Table 1: Information of machines in the cluster.

Name	File Size	Nodes	Edges	Description
wiki-Vote	1.0 MB	7,115	103,689	Wikipedia who-votes-on-whom network
soc-Slashdot0902	10.8 MB	82,168	948,464	Slashdot social network from February 2009
web-Google	71.9 MB	875,713	5,105,039	Web graph from Google
cit-Patents	267.5 MB	3,774,768	16,518,948	Citation network among US Patents
Twitter	1.3 GB	11,316,811	85,331,845	Twitter social network on who follows whom network
kronecker19	40 MB	416,962	3,206,497	Synthetic graph generated by Kronecker generator with 19 iterations.
kronecker20	89MB	833,566	7,054,294	Synthetic graph generated by Kronecker generator with 20 iterations.
kronecker21	209MB	1,665,554	15,519,448	Synthetic graph generated by Kronecker generator with 21 iterations.
kronecker22	479MB	3,330,326	34,142,787	Synthetic graph generated by Kronecker generator with 22 iterations.
kronecker23	1.1GB	6,654,956	75,114,133	Synthetic graph generated by Kronecker generator with 23 iterations.

Table 2: Graph Datasets.

problem can be solved by redistributing PageRank mass "lost" at dangling nodes across all nodes in the graph evenly. That is, for node n, its current PageRank value $P(n)$ is updated to the final PageRank value $P(n)'$ according to the following formula:

$$P(n)' = \alpha\ (\frac{1}{|G|}) + (1-\alpha)\ (\frac{l}{|G|} + P(n)) \qquad (2)$$

where $|G|$ is the total number of nodes in the graph, α is the random jump factor, l is the total PageRank mass "lost" at dangling nodes.

B. PageRank on Hadoop

Our PageRank implementation on Hadoop references the idea in [15]. The basic process of each PageRank iteration can be divided into two Hadoop MapReduce jobs.

In the first job, each node m first divides its current PageRank value $P(m)$ evenly by $C(m)$, the number of nodes m links to and passes each share to them separately. This distribution process is implemented by a map function. Then each node n sums up all PageRank contributions that have been passed to it and gets its new PageRank value $P(n)$. This aggregation process is implemented by a reduce funtion. The first job implementation of PageRank iteration on Hadoop is shown in Algorithm 1.

Before the second job, the total PageRank loss l at dangling nodes needs to be calculated. We get the total PageRank loss l by first summing up the PageRank value each node has after the first job and then using one minus this sum.

In the second job, each node n updates its current PageRank value $P(n)$ to $P(n)'$ according to Formula 2. The PageRank loss l computed before can then be evenly distributed to all the nodes in the graph. The random jump factor α can also be added at each node. The second job is implemented by a map function only job. Algorithm 2 gives the second job implementation of PageRank iteration on Hadoop.

Algorithm 1 First job of Hadoop PageRank iteration

```
function MAP ($m, < curPR, NeighborList >$)
    $contrib \leftarrow \frac{curPR}{|NeighborList|}$
    for all $n \in NeighborList$ do
        $EMIT\ (n, contrib)$ /*PageRank distribution*/
    end for
    $EMIT\ (m, NeighborList)$
end function

function REDUCE ($n$,$< ContribList, NeighborList >$)
    $p \leftarrow 0$
    for all $c \in ContribList$ do
        $p = p + c$ /*PageRank aggregation*/
    end for
    $EMIT\ (m, < p, NeighborList >)$
end function
```

Algorithm 2 Second job of Hadoop PageRank iteration

```
function MAP ($n, < p, NeighborList >$)
    $l \leftarrow$ PageRank loss
    $N \leftarrow$ number of nodes in the graph
    $\alpha \leftarrow$ random jump factor
    $p' = \alpha \times \frac{1}{N} + (1-\alpha) \times (\frac{l}{N} + p)$ /*PageRank loss distribution and random jump factor treatment*/
    $curPR = p'$
    $EMIT\ (n, < curPR, NeighborList >)$
end function
```

C. PageRank on Spark

The authors who invent Spark provide a simple implementation of PageRank on Spark. But their implementation did not consider the dangling nodes in the graph. There are actually a large number of dangling nodes in both real and synthetic graphs. Dealing with dangling nodes is a necessary step, so we reimplement PageRank on Spark by taking into account the dangling nodes in order to make the PageRank implementation on Hadoop and Spark be consistent and practical.

All the iterative operations can be implemented in a Spark driver. Each iteration includes four steps. The first step distributes the PageRank value of each node to its neighbors using the join and flatMap transformation. The second step does PageRank aggregation for each node using reduceByKey transformation. In the third step, PageRank loss at dangling nodes is computed. The fourth step deals with PageRank loss distribution and random jump factor.

Spark revolves around the concept of *resilient distributed dataset* **(RDD), which is a fault-tolerant collection of elements that can be operated on in parallel. There are currently two types of RDDs:** *parallelized collections*, **which take an existing Scala collection and run functions on it in parallel, and** *Hadoop datasets*, **which run functions on each record of a file in Hadoop distributed file system or any other storage system supported by Hadoop.**

Iterative operations can be implemented by Spark jobs. Before each Spark job, dataset of interest can first be loaded from disk into the memory and represented as RDD across machines of the whole cluster. Then the job can be run with data (in RDD form) kept in memory for the whole iterative process. After job running, results can be written to disk. Fewer disk accesses and memory based process contribute to its performance improvements.

RDDs support two types of operations: *transformations*, **which create a new dataset from an existing one, and** *actions*, **which return a value to the driver program after running a computation on the dataset. Some common transformations and actions supported by RDD and their meanings are listed in Table 3 and Table 4 separately.**

To implement PageRank algorithm on Spark platform, two steps are also required. The first one is to represent the graph with adjacency list which is similar to that of Hadoop but without PageRank initialization. This step can be implemented by making small modifications to that of Hadoop. The second one is PageRank iterations. we mainly focus on the implementation of the second step.

Before iteration, we need to compute the number of nodes in the graph and construct two RDDs. The first RDD is a pair list of (node, neighbors) with node standing for the id of a specific node in the graph and neighbors standing for a list of ids this node links to. The second RDD is a pair list of (node, rank) with node also standing for the id of a specific node in the graph and rank standing for the PageRank value of this node and being set to 1/N at the beginning.

Then we can proceed to do PageRank iterations. Each iteration includes four steps. The first step distributes the PageRank value of each node to its neighbors using the join and flatMap transformation. The second step does PageRank aggregation for each node using reduceByKey transformation. In the third step, PageRank loss at dangling nodes is computed. The fourth step deals with PageRank loss distribution and random jump factor.

Algorithm 3 Spark PageRank iteration

```
val N = ... /*number of nodes in the graph*/
val links = ... /*RDD of (node, neighbors) pairs*/
var ranks = ... /*RDD of (node, rank) pairs*/
for $i = 0$ to ITERATIONS do
    val contribs = links .join ( ranks ) .flatMap ({
            case ( node, (neighbors, rank) ) $=>$
                neighbors .map (
                    neighbor $=>$
                        ( neighbor, rank/neighbors.size) )
      }) /*PageRank distribution*/
    ranks = contribs .reduceByKey ( _ + _ )
      /*PageRank aggregation*/
    dMass = ranks .reduce (
            ( $p1$, $p2$ ) $=>$ ( ”dm”, $p1.\_2 + p2.\_2$ )
      ) ._2
    lMass = 1 - dMass /*PageRank loss computation*/
    ranks = ranks .mapValues (
            $v => \alpha \times \frac{1}{N} + (1-\alpha) \times (\frac{lMass}{N} + v)$
      ) /*PageRank loss distribution and random jump factor treatment*/
end for
ranks.saveAsTextFile (...)
```

After PageRank iterations, the rank for each node can be saved using saveAsTextFile action.

The detailed PageRank implementation on Spark is shown in Algorithm 3.

V. Experiments and Results

A. Measurement Metrics

For each dataset we make PageRank run five iterations on Hadoop and Spark respectively, and record the total running time each dataset spends. Meanwhile, when PageRank is running on either Hadoop or Spark, we query and record each slave's system memory usage every second. When it ends, we can plot the temporal change of memory usage for each slave according to the records we have kept.

The purpose of choosing the same iteration number for all the datasets is for fair comparison. The reason why we only show five iterations for all the datasets instead of their convergence iteration time is that five iterations is long enough to help us quantify the time differences between Hadoop and Spark and observe their memory usage patterns.

B. PageRank Scalability

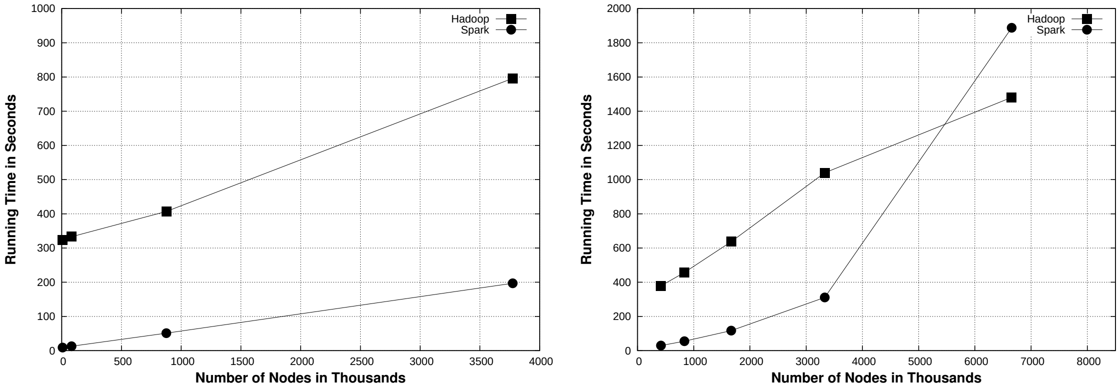
Fig. 2 shows running time versus number of nodes for real and synthetic graph datasets. We have two observations: 1) when datasets are smaller than kronecker22, the PageRank implementations exhibit near-linear scalability on both Hadoop and Spark whether the graphs are real or synthetic; 2) when datasets are larger than kronecker22, slope of Spark line is greater than that of Hadoop line which means that the performance advantage of Spark becomes smaller as the graph size increases.

Transformation	Meaning
map($func$)	Return a new distributed dataset formed by passing each element of the source through a function $func$.
flatMap($func$)	Similar to map, but each input item can be mapped to 0 or more output items (so $func$ should return a Seq rather than a single item).
reduceByKey($func$)	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function.
join($otherDataset$)	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.

Table 3: Common transformations supported by RDD.

Action	Meaning
reduce($func$)	Aggregate the elements of the dataset using a function $func$ (which takes two arguments and returns one). The function should be associative so that it can be computed correctly in parallel.
count()	Return the number of elements in the dataset.
saveAsTextFile($path$)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.

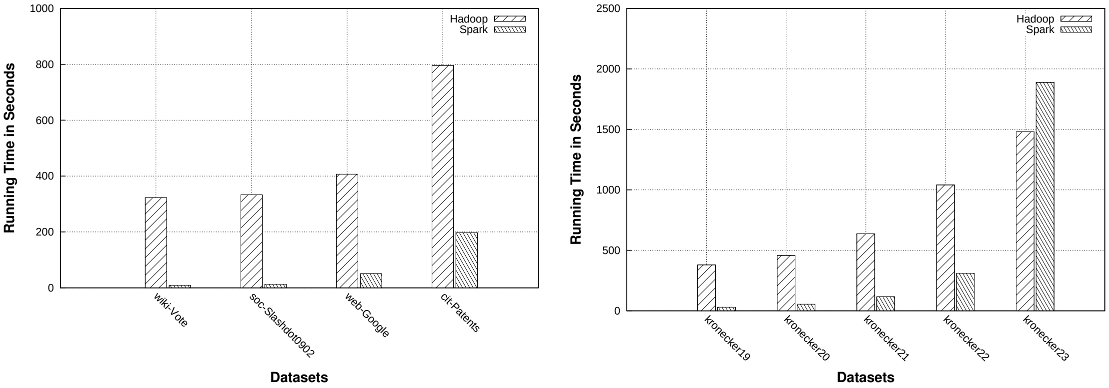
Table 4: Common actions supported by RDD.



(a) Real Graphs

(b) Synthetic Graphs

Figure 2: Scalability comparison.



(a) Real Graphs

(b) Synthetic Graphs

Figure 3: Running time comparison.

C. Running Time Comparison

Fig. 3 shows running time comparison between Hadoop and Spark using real and synthetic graph datasets. We have the following five observations.

1) When dataset is too small, e.g., wiki-Vote, soc-

Slashdot0902, Spark can outperform Hadoop by 25x-40x. In this case, the computing resources of all the slaves are underutilized. For Spark, it is the iterative computation time that dominates the total running time. But for Hadoop, it is the job initialization time, disk access time, I/O time and network communication time that contribute the major cost.

2) When dataset is relatively small, e.g., web-Google, kronecker19, kronecker20, Spark can outperform Hadoop by 10x-15x. In this case, the computing resources of all the slaves are moderately utilized. The iterative computation time begins to dominate the total running time of Hadoop and Hadoop starts to show its true performance.

3) When dataset is relatively large, e.g., cit-Patents, kronecker22, the advantage of Spark becomes small and Spark can only outperform Hadoop by 3x-5x. In this case, for Hadoop the memory of each slave in the cluster is still moderately utilized while for Spark the memory of each slave in the cluster is full (i.e. reaching the memory limit of each slave in the cluster) which result in Spark's reduced performance advantage.

4) When dataset is too large, e.g., kronecker23, Hadoop beats Spark. In this case, for Hadoop, the memory of each slave in the cluster is fully utilized while for Spark the memory of each slave in the cluster is not enough for PageRank running. There is no memory space for the newly created RDDs, so Spark starts its RDD replacement policy. Frequent RDD replacements cause Spark's performance to degrade to that of Hadoop, or even worse.

5) When dataset is extremely large, e.g., Twitter, PageRank on Spark crashes halfway with JVM heap exceptions while PageRank on Hadoop can still run.

D. Memory Usage Comparison

By comparing memory usage plots of all the seven slaves, we find that their plots are roughly the same. This is because of the uniform distribution of the input data on the seven slaves and the load balancing strategy adopted by both Hadoop and Spark. So we will only illustrate the results for one slave (slave1), shown in Fig. 4 and Fig. 5. In each plot, memory occupancy over time for different graph datasets are shown by different curves in different colors.

For Hadoop platform, when dataset is fixed, memory usage exhibits periodically increase and decrease with the increase in the number of iterations. There is also a gradual increment between two consecutive iterations. The periodicity is due to the iterative nature of PageRank algorithm. Each cycle corresponds to one iteration of PageRank. In each cycle, there are three peaks, which corresponds to the Map and Reduce stages in the first job and the Map stage in the second job respectively. All these three stages consume memory and lead to a peak of memory usage. The Map stage in the first job comsumes a little bit more memory than its corresponding Reduce stage in the first job and Map stage in second job. Reduce stage in the first job lasts longer than the other two stages.

For Hadoop platform, when iteration number is fixed, the shapes of memory usage plots are similar with the increase in the size of dataset. Differences exist in two aspects. When dataset size is larger, the gradual increment between two consecutive iterations is bigger and the three peaks in each iteration are higher. When the increment reaches memory limit of a slave, the increase stops and the height of the three peaks does not change.

For Spark platform, when dataset is fixed, memory usage only exhibits periodically gradual increase with the increase in the number of iterations. The reason for periodicity is the same as that for Hadoop. However, there are no peaks in each cycle. The gradual increment between two consecutive iterations is because of the creation of new auxiliary RDDs. Spark uses RDD for data cache and RDD is a read-only data structure. There are two main RDDs in Spark's PageRank implementation. One is used for caching the graph structure, which will not change during iterative process. We call this RDD the main RDD. The other one is used for caching temporary PageRank value for each node in the graph, which will change in each iteration. We call this RDD the auxiliary RDD. Because of the read-only nature of RDD, a new auxiliary RDD will be created when changes happen to it in each iteration.

For Spark platform, when iteration number is fixed, the shapes of memory usage plots are similar with the increase in the size of dataset. When dataset size is larger, the gradual increment between two consecutive iterations is bigger. When the increment reaches memory limit of a slave, the increase stops. When the dataset size is too large, the program cannot run without enough memory.

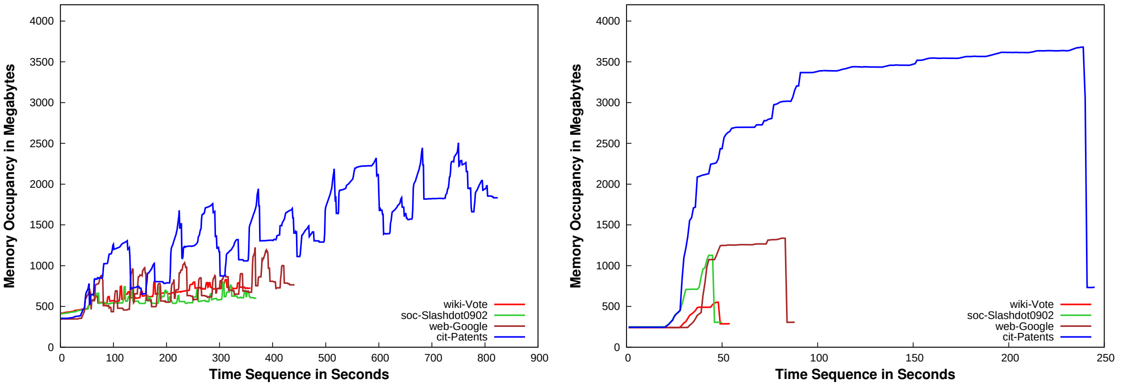
In Hadoop platform, iterative operations are implemented through multiple MapReduce computations. Each iteration corresponds to one or more MapReduce computations. When a Map or Reduce computation ends, the memory used by it will be released. Whereas, in Spark platform, iterative operations are implemented in a single Spark program. So there is no memory release phenomenon in Spark implementation.

From the experimental memory plots, we can also find that Spark is extremely memory consuming. For a graph dataset whose size is 300M-500M, the total 21GB memory of the cluster will be occupied.

VI. Related Work

As a new programming model, MapReduce [2] is suitable for processing and generating large data sets in a scalable, reliable and fault-tolerant manner. Apache Hadoop [3] provides an open source implementation of MapReduce. Performance of MapReduce has been deeply studied in [16] [17]. The authors show that with proper implementation, performance of MapReduce can approach traditional parallel databases while achieving scalability, flexibility and fault-tolerance.

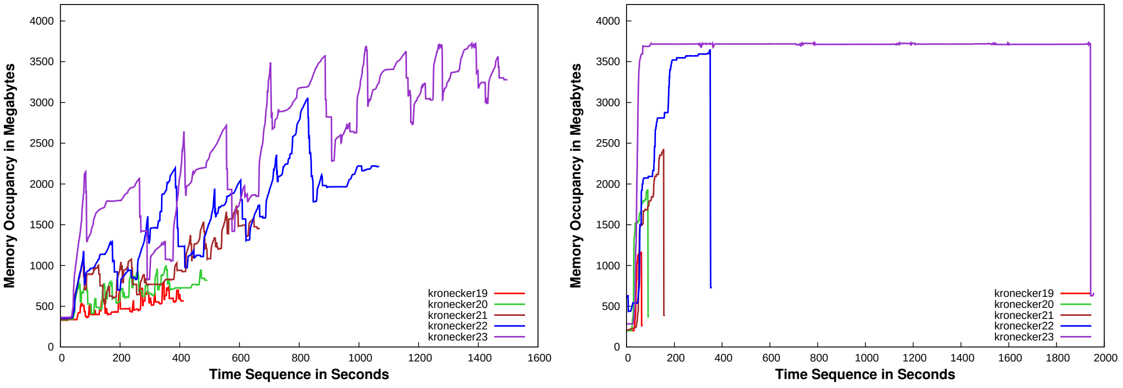
Spark [1] was developed recently to optimize iterative and interactive computation. It uses caching techniques to dramatically improve the performance for repeated operations. The main abstraction in Spark is called $resilient\ distributed\ dataset$ (RDD), which is maintained in memory across iterations and fault tolerant. Spark can outperform Hadoop by 10x in iterative machine learning jobs, and can be used to interactively query a 39 GB dataset with sub-second response time. Other similar works include Twister [7] and HaLoop [8]. Compared with Hadoop, Twister



(a) Hadoop

(b) Spark

Figure 4: Memory usage comparison using real graph datasets.



(a) Hadoop

(b) Spark

Figure 5: Memory usage comparison using synthetic graph datasets.

outperforms Hadoop in parallel efficiency; HaLoop averagely reduces query runtimes by 1.85, and shuffles only 4% of the data between mappers and reducers.

Some efforts focus on iterative graph algorithms, an important class of iterative operations. Pegsus [18] unifies many iterative graph algorithms as Generalized Iterated Matrix-Vector multiplication (GIM-V). By exploiting special properties of matrix, it can achieve more than 5x faster performance over the regular version. Pregel [19] chooses a pure message passing model to process graphs. Distributed GraphLab [20] is similar to Pregel. The key difference between Pregel and GraphLab is that Pregel has a barrier at the end of every iteration, whereas GraphLab is completely asynchronous. Experiments show that applications created using Distributed GraphLab outperform equivalent Hadoop/MapReduce implementations by 20-60x.

VII. Conclusions and Future Work

For different experiment settings, we find that although Spark is in general faster than Hadoop, it is at the cost of significant memory consumption. If speed is not a demanding requirement and we do not have abundant memory, it's better not choose Spark. In this case, as long as we have enough disk space to accommodate the original dataset and intermediate results, Hadoop is a good choice.

For a specific iterative operation, if the application is time sensitive, Spark should be considered. But enough memory is a necessary condition for the operation in order to secure Spark's performance advantage. The problem is that it is hard to determine the accurate amount of memory for iterative operations running on Spark. Exactly how much memory is enough depends on the particular iterative algorithm and the size of the dataset it processes. Intermediate results during iterations are stored in memory as RDDs. Because of the read-only nature of RDD, new RDDs will always be created in each iteration. If the size of intermediate results is at constant level, the increase of memory usage between two consecutive iterations is not significant. If the size of the intermediate results is proportional to the size of the input dataset (like PageRank), the increase of memory usage between two consecutive iterations is significant.

As part of our future work, we plan to find a prediction model for the tradeoff of response time and memory usage for iterative operations. Also we will explore the influence of graph structures on response time of iterative operations.

ACKNOWLEDGEMENT

We would like to thank the Stanford Network Analysis Platform [12] for providing us the wiki-Vote, soc-Slashdot0902, web-Google, cit-Patents graph dataset and the Kronecker synthetic graph generator. We also want to thank the Social Computing Data Repository at ASU [21] for providing us the Twitter graph dataset [13].

This work is supported by the National Nature Science Foundation of China, NSFC (61170293); the National High Technology Research and Development Program of China (2012AA0011203); and State Key Laboratory of Software Development Environment (SKLSDE-2012ZX-03).

REFERENCES

[1] M. Zaharia, M. Chowdhury, S. S. Michael J. Franklin, and I. Stoica, "Spark: Cluster computing with working sets," *In HotCloud*, June 2010.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *In OSDI*, 2004.

[3] Apache Hadoop. [Online]. Available: http://hadoop.apache.org

[4] Spark. [Online]. Available: http://spark-project.org/

[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing," *In NSDI*, April 2012.

[6] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the Web," Technical Report, Stanford InfoLab, 1999.

[7] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for iterative MapReduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010, pp. 810–818.

[8] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient iterative data processing on large clusters," in *Proceedings of the VLDB Endowment*, September 2010, pp. 285–296.

[9] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "iMapReduce: A distributed computing framework for iterative computation," in *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, 2011, pp. 1112–1121.

[10] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*. USENIX Association, 2011, pp. 22–22.

[11] Apache Mesos. [Online]. Available: http://incubator.apache.org/mesos/

[12] Stanford SNAP. [Online]. Available: http://snap.stanford.edu/

[13] Twitter graph dataset. [Online]. Available: http://socialcomputing.asu.edu/datasets/Twitter

[14] M. Bianchini, M. Gori, and F. Scarselli, "Inside PageRank," in *ACM Transactions on Intenet Technology*, vol. 5, 2005, pp. 92–128.

[15] J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool, 2010.

[16] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, "The performance of MapReduce: An in-depth study," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 472–483, 2010.

[17] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel data processing with MapReduce: a survey," *ACM SIGMOD Record*, vol. 40, no. 4, pp. 11–20, 2012.

[18] "Pegasus: A peta-scale graph mining system implementation and observations," in *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 2009, pp. 229–238.

[19] "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[20] "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[21] R. Zafarani and H. Liu, "Social computing data repository at ASU," 2009. [Online]. Available: http://socialcomputing.asu.edu