# An Experimental Comparison of Iterative MapReduce Frameworks

Haejoon Lee[†] Minseo Kang[‡] Sun-Bum Youn[§] Jae-Gil Lee[‡*] YongChul Kwon[¶]
[†]School of Computing, KAIST, Korea
[‡]Graduate School of Knowledge Service Engineering, KAIST, Korea
[§]Netmarble Games, Korea
[¶]Microsoft Corp., USA
{philjjoon, minseo, jaegil}@kaist.ac.kr sbyoun@netmarble.com yongchul.kwon@microsoft.com

## ABSTRACT

MapReduce has become a dominant framework in big data analysis, and thus there have been significant efforts to implement various data analysis algorithms in MapReduce. Many data analysis algorithms are inherently *iterative*, repeating the same set of tasks until a convergence. To efficiently support iterative algorithms at scale, a few variants of Hadoop and new platforms have been proposed and actively developed in both academia and industry. Representative systems include HaLoop, iMapReduce, Twister, and Spark. In this paper, we experimentally compare Hadoop and the aforementioned systems using various workloads and metrics. The *five* systems are compared through four iterative algorithms—PageRank, recursive query, $k$-means, and logistic regression—on 50 Amazon EC2 machines (200 cores in total). We thoroughly explore the effectiveness of their new caching, communication, and scheduling mechanisms in support of iterative computation. Our evaluation also shows the performance depending on data skewness and memory residency. Overall, we believe that our evaluation and interpretation will be useful for designing a new framework or improving the existing ones.

## Keywords

Iterative algorithms; MapReduce; Hadoop; Spark; Benchmark

## 1. INTRODUCTION

MapReduce [5] has become the most popular framework for large-scale processing and big data analysis. It allows us to easily develop distributed, parallel applications running in a cluster of commodity machines, without tedious detail of distributed execution such as scheduling and fault-tolerance. Its sweet spot is known to be batch processing of data-intensive tasks [6]. Thus, many data-intensive database and data mining operations, such as similarity join and decision tree classification, have been successfully translated to the MapReduce framework [15]. As a result, we have witnessed remarkable performance improvements compared to original implementations. Moreover, Hadoop [1], an open-source implementation of MapReduce, has contributed to the huge success of the framework.

Despite its popularity, MapReduce has faced criticisms for its performance limitations in several advanced applications [6, 12]. One of such criticisms is the inefficiency of handling *iterative* algorithms [2, 3, 4, 7, 17, 18]. Many data analysis algorithms, such as PageRank, recursive query, $k$-means, and logistic regression, require iterative computation, repeatedly executing the same set of tasks until a stopping condition is satisfied. MapReduce does not directly support such iterative computation, but users could handcraft a chain of MapReduce jobs by writing a driver program [14]. Although this naïve implementation can scale beyond capacity of a single machine, it may incur severe performance penalty because the framework is not aware of the iteration. For example, the data sets invariant while running an algorithm must be loaded and processed repeatedly for each iteration, thereby wasting disk I/O, network bandwidth, and CPU resources.

To alleviate the inefficiency for iterative algorithms, some variants of Hadoop and new platforms have been developed in research and open-source communities. Representative systems include HaLoop [2, 3], Twister [7], iMapReduce [18], MapReduce Online [4], and Spark [17]. HaLoop, iMapReduce, and MapReduce Online are implemented by modifying the MapReduce component of Hadoop, whereas Twister and Spark are implemented independently from Hadoop. Various techniques such as disk caches and distributed shared memory have been incorporated into those systems in support of efficient iterative computation.

To the best of our knowledge, those systems have not been systematically compared with one another, though each system was compared against Hadoop only. Therefore, to examine a broad range of techniques, we choose five systems—Hadoop, HaLoop, Twister, iMapReduce, and Spark—for our comparison. Hadoop is the baseline which lacks the capability of handling iterative computation. HaLoop, Twister, and iMapReduce are research prototypes, which all of them are being actively cited. MapReduce Online is not explicitly included since iMapReduce is built on top of it. Apache Spark is included mainly because of its popularity in recent days even though it does not strictly conform to the MapReduce paradigm. However, we do not include graph processing systems [8] such as Pregel and GraphLab in our comparison, because the scope of this paper is comparison of "MapReduce-like" general purpose systems rather than graph processing systems.

In this paper, we experimentally compare the *five* systems, seeking for in-depth understanding of the performance of iterative computation. Since three of them are just research prototypes and are not optimized to production level, we do *not* concentrate on the absolute performance of each system but on the intrinsic capabilities and the techniques to support iteration in each system.

[*]Jae-Gil Lee is the corresponding author.

*CIKM'16 , October 24-28, 2016, Indianapolis, IN, USA*
© 2016 ACM. ISBN 978-1-4503-4073-1/16/10. . . $15.00
DOI: http://dx.doi.org/10.1145/2983323.2983647

Specifically, the contributions of this paper include:

1. We investigate the impacts of such techniques on performance through systematic and fair comparison of the systems. To precisely explain the impacts, we use various evaluation metrics as well as various workloads generated by running diverse algorithms on both real-world and synthetic data sets (Section 3).
2. We discuss the advantages and disadvantages of the systems with regard to iterative computation such that users and developers can exploit our lessons in the use and development of such systems (Section 4).

We found that all systems are scalable but some of the key techniques such as input data caching may have diminishing return depending on data sets and algorithms. The primary factors on the performance are shown to be the size of input and intermediate data, invariability of the keys of intermediate data, efficiency of network communication, flexibility of task scheduling, and so on. Also, we observed that memory-based engines generally perform better than disk-based engines but the performance rapidly degrades when the data does not fit in memory.

The full (10 page) version of this paper is available at http://dm.kaist.ac.kr/lab/papers/cikm16_expr_full.pdf. The source code is available at https://github.com/IterativeExperimentsMapReduce.

## 2. EVALUATION METHODOLOGY

### 2.1 Cluster and System Setup

We conduct experiments on an Amazon EC2 cluster (Sections 3.1–3.3) or on a local cluster (Section 3.4), as described below.

- *Amazon EC2 cluster*: The cluster consists of 50 m1.xlarge Amazon EC2 spot instances, located in US West (Oregon). Each instance has four virtual CPUs, 4×420 GB of local storage, and 15 GB of main memory. All instances run on Ubuntu 14.04.1 LTS.
- *Local cluster*: The cluster consists of 4 commodity servers. A server has two Intel Core i7-4790 CPUs, each equipped with four cores, 1 TB of local storage, and 32 GB of main memory. Thus, this cluster has 32 cores in total. All servers run on Ubuntu 14.01 LTS.

We use Hadoop 1.2.1, HaLoop 0.20.2, Twister 0.9, iMapReduce 0.1, and Spark 1.2.0. HaLoop and iMapReduce are implemented using Hadoop 0.8 and 0.19.2 respectively. Twister is configured to use ActiveMQ [16] 5.4.3 for its message broker. Spark runs with Scala 2.10.0 and Hadoop 1.2.1. JDK 1.8.0 is used for running the systems. The maximum per-machine Java heap size is configured to be 12 GB.

### 2.2 Data Sets

#### 2.2.1 Real-World Data Sets

We use the real-world data sets in Table 1, which are characterized into graph data and row data. The two *graph* data sets are used for both PageRank and recursive query, and the two *row* data sets are used for $k$-means and logistic regression respectively.

As for graph data, *LiveJournal* is a social network data set available at the SNAP repository [13]. *ClueWeb* is a web graph data set, and we use TREC Category B[1]. ClueWeb has much more vertices and edges than LiveJournal. LiveJournal is more densely-connected than ClueWeb, in that the average degree (i.e., the ratio of the number of vertices to that of edges) of the former is about ten times higher than that of the latter. We substitute all vertex identifiers in LiveJournal with longer strings to increase its size without changing its network structure, just like Bu et al. [2, 3] did. As a result, the data set was extended from 1 GB to 10 GB.

[1]http://lemurproject.org/clueweb09/

As for row data, *Cosmo-Gas* and *Cosmo-All* are scientific simulation data sets, where each row represents the information of a particle. Cosmo-All has all particles of the *cosmo50* data set [10], whereas Cosmo-Gas contains only the gas particles. For $k$-means clustering, the three variables for $x$, $y$, and $z$ positions are kept with Cosmo-Gas. For logistic regression, a binary variable that indicates if a row is a star particle is added to Cosmo-All as a *dependent* variable, and other variables are used as *independent* variables.

Table 1: Real-world data sets used for all algorithms.

| Graph Data | Algorithms | # of Vertices | # of Edges |
|---|---|---|---|
| **LiveJournal** | PageRank, | 4,847,571 | 68,993,773 |
| **ClueWeb** | Recursive | 428,136,613 | 454,075,638 |
| Row Data | Algorithm | # of Points | # of Dims |
| **Cosmo-Gas** | $k$-means | 147,251,521 | 3 |
| **Cosmo-All** | Logistic | 315,086,245 | 10 |

With these base data sets in Table 1, we generate various sizes of data sets to test the scalability of the systems. In the next section, "1X"∼"5X" indicate the variations of a data set having proportional sizes. For LiveJournal, ClueWeb, and Cosmo-Gas, the base data set becomes "1X", and larger ones are generated by replicating it by 2, 3, 4, and 5 times. When replicating Cosmo-Gas, Gaussian noises are added in order to avoid the exactly same values. For Cosmo-All, the base data set becomes "5X" since it is sufficiently large, and smaller ones are obtained by randomly sampling 20%, 40%, 60%, and 80% of the base data set.

#### 2.2.2 Synthetic Data Sets

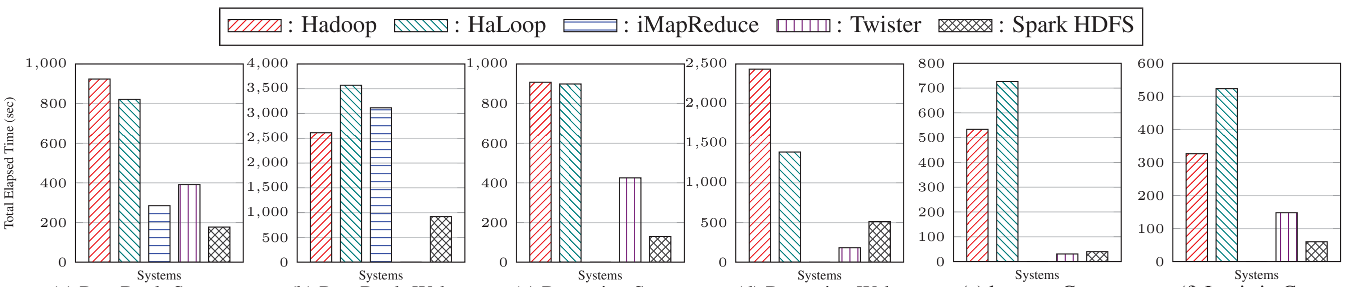
We use also synthetic data sets in order to investigate the response of each system to the *skewness* of the input data. The performance of PageRank is sensitive to skewness since the algorithm sends PageRank contributions to more neighbors on higher-degree vertices. The LFR benchmark [11] generates graph data sets for use in PageRank, which are summarized in Table 2. They are generated according to the following parameters: $N$ is the number of vertices, $\langle k \rangle$ is the average degree, $max\_k$ is the maximum degree, and $t1$ is the exponent part of the power-law distribution of degrees. Here, $max\_k$ exponentially increases while $N$ and $\langle k \rangle$ are fixed. As $max\_k$ gets larger, the degree values should disperse to a larger range to keep $\langle k \rangle$ the same; that is, the skewness of degrees gets higher. Consequently, some tasks will transfer much larger number of PageRank contributions than others. In this way, we control the skewness between the tasks.

Table 2: Synthetic graph data sets used for PageRank.

| Data Set | $N$ | $\langle k \rangle$ | $max\_k$ | $t1$ |
|---|---|---|---|---|
| **LFR1** | 200,000 | 1,000 | $1,000 \times 1.6^0$ | 2 |
| **LFR2** | | | $1,000 \times 1.6^1$ | |
| **LFR3** | | | $1,000 \times 1.6^2$ | |
| **LFR4** | | | $1,000 \times 1.6^3$ | |
| **LFR5** | | | $1,000 \times 1.6^4$ | |

### 2.3 Evaluation Metrics

*Total elapsed time* is the time spent for running an algorithm against a data set on a specific system. Total elapsed time is determined to be the interval between the job finish time and the job start time included in system log files. On the other hand, some systems require reformatting or full duplication of input data before run-



(a) PageRank-Soc. (b) PageRank-Web. (c) Recursive-Soc. (d) Recursive-Web. (e) kmeans-Cosmo. (f) Logistic-Cosmo.

Figure 1: Total elapsed time of the five systems (50 machines).

Table 3: Total HDFS read in Figure 1 (Unit: **GB**).

| Systems | 1a | 1b | 1c | 1d | 1e | 1f |
|---|---|---|---|---|---|---|
| Hadoop | 134.052 | 213.045 | 100.732 | 82.037 | 65.433 | 82.027 |
| HaLoop | 44.173 | 143.668 | 11.300 | 7.179 | 51.934 | 74.526 |
| iMapReduce | 6.240 | 30.534 | - | - | - | - |
| Spark | 9.900 | 7.600 | 9.900 | 7.600 | 6.500 | 8.200 |

Table 4: Reduce shuffle I/O in Figure 1 (Unit: **MB**).

| Systems | 1a | 1b | 1c | 1d | 1e | 1f |
|---|---|---|---|---|---|---|
| Hadoop | 30,593 | 241,205 | 14,943 | 98,227 | 0.847 | 0.241 |
| HaLoop | 16,912 | 141,951 | 2,979 | 9,642 | 0.818 | 0.739 |
| iMapReduce | 970 | 16,842 | - | - | - | - |
| Spark | 8,500 | 34,600 | 2,883 | 2.601 | 1.988 | 0 |

ning a main job, and this preprocessing time is *not* included in total elapsed time since it is a one-time cost.

*Normalized time* is the ratio of the total elapsed time of a certain configuration to that of the base configuration. For example, in scalability tests, normalized time is the ratio of the total elapsed time for a certain data set to that for the smallest data set. Consequently, normalized time always starts from 1. It shows sensitivity of the total elapsed time to the change of a parameter value. The intention of this normalization is to mitigate the difference in maturity of each system.

*Total HDFS read* is the total amount of the data read *from the HDFS*—not from the local file system—during the entire execution of an algorithm. This metric assesses how effectively each system reduces *remote* disk I/O. It is the sum of the outputs of the counter "HDFS_BYTES_READ" for Hadoop, HaLoop, and iMapReduce; and it is the first output of the counter "Input" for Spark since the subsequent outputs of that counter indicate the accesses to the RDD which is typically stored in main memory.

*Reduce shuffle I/O* is the total amount of the data shuffled from mappers to reducers during the entire execution of an algorithm. This metric assesses how effectively each system reduces network communication for the shuffle. It is the sum of the outputs of the counter "Reduce Shuffle Bytes" for Hadoop and HaLoop, a custom counter added by the authors for iMapReduce, and the counter "Shuffle Read" for Spark.

We measure those metrics mostly twice and choose one measurement after confirming that the two measurements do not vary significantly, since it is too expensive to repeat many times on Amazon EC2. If an execution does not finish within two hours, we terminate the execution and give up obtaining the corresponding result.

## 3. EVALUATION RESULTS

### 3.1 Overall Comparison

Figure 1 shows the total elapsed time of the five systems for the six combinations of algorithms and data sets. More importantly, we would like to figure out the overall performance characteristics of the five systems. Overall, Hadoop showed the worst performance as expected; HaLoop showed a little improvements over Hadoop; iMapReduce and Twister showed more improvements than HaLoop; and Spark in general achieved the best performance. Since Spark is maintained and tuned by a vibrant open-source community, this commercial-quality tuning in addition to the advantage of its framework seems to contribute to the highest performance.

To facilitate this comparison of the overall performances, we present the results for *total HDFS read* and *reduce shuffle I/O*[2] of Figure 1 since these two metrics impact greatly on the total elapsed time. As for the former in Table 3, because the cache of each system converts remote disk I/O to local disk I/O, a value represents how much such a caching mechanism effectively reduces remote disk I/O. Roughly speaking, the difference between Hadoop and another system indicates the amount of disk I/O saved by the caching mechanism. As for the latter in Table 4, a value represents the amount of the data transferred from mappers to reducers, which incurs network communication. PageRank has high values owing to the PageRank contributions of *all* vertices, recursive query has moderate values since it carries only reachable vertices, and both $k$-means and logistic regression have low values since they need only the centroid or parameter information. In both tables, the smaller a value is, the higher the performance is. These values are highly correlated to the total elapsed time except HaLoop (owing to a configuration issue which will be explained later).

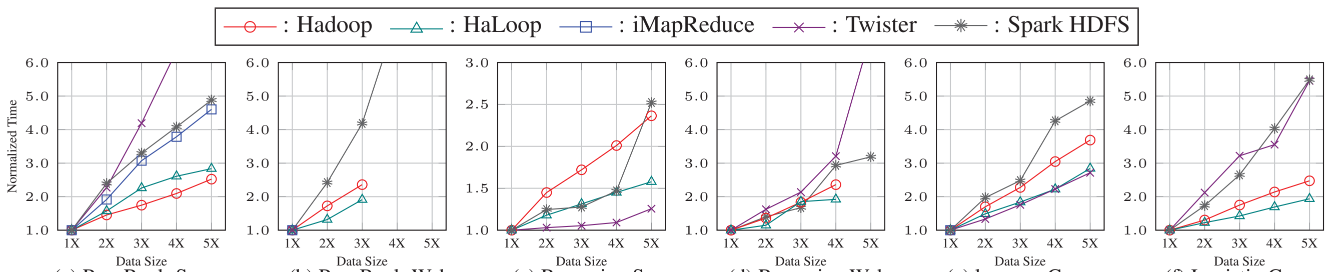
Before going into the details, we would like to address a few points that will be referenced for the interpretation of the results.

- **Remark 1**: The communication efficiency (network utilization) is higher in Spark than in Twister since Twister performs broadcasting via a broker network. In other words, Twister requires significant effort in tuning the broker network to achieve high performance.
- **Remark 2**: Removing duplicates in recursive query tends to be more efficient in Twister than in Spark. Spark processes it at once globally according to *lazy evaluation* [9], whereas Twister is *originally designed* to do it in two steps—locally on each worker and then globally on the combiner.
- **Remark 3**: The amount of the data shuffled tends to be larger in PageRank than in recursive query because of the characteristics of the algorithms, as shown in Table 4.
- **Remark 4**: The amount of the data shuffled in recursive query is larger for LiveJournal than for ClueWeb since the former is much more densely-connected than the latter. See the entries of Spark in Table 4 where the static data is completely cached.

**Twister vs. Spark**

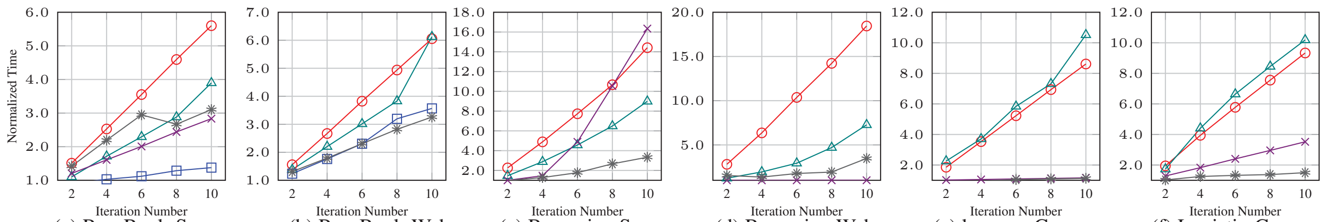
Spark and Twister are similar in that both are memory-based engines. Since the results of recursive query were available for two

[2]The results are not available for Twister since it does not provide counters.



(a) PageRank-Soc. (b) PageRank-Web. (c) Recursive-Soc. (d) Recursive-Web. (e) kmeans-Cosmo. (f) Logistic-Cosmo.

Figure 2: Normalized time of the five systems with varying the data size (50 machines).



(a) PageRank-Soc. (b) PageRank-Web. (c) Recursive-Soc. (d) Recursive-Web. (e) kmeans-Cosmo. (f) Logistic-Cosmo.

Figure 3: Normalized time of the five systems with varying the iteration number (50 machines).

different data sets, we compared the relative performance between these systems using Figures 1c and 1d. Twister showed lower performance than Spark in Figure 1c by **Remark 1** and **Remark 4** since the higher shuffle cost was managed well in Spark but not in Twister. On the other hand, Spark showed lower performance than Twister in Figure 1d by **Remark 2**. In Figure 1c, duplicate removal was done entirely on main memory in both systems. In Figure 1d, however, Spark spilled intermediate results to local disk owing a larger size of ClueWeb whereas Twister did not. Twister needed less memory than Spark because of its two-step approach.

We note that this difference is attributed to the *architectural choices* made in Twister and Spark: two-step and lazy evaluations. Spark adopts lazy evaluation to avoid unnecessary computation, but its benefit was hidden because Spark had to spill intermediate results. We did *not* inject the two-step approach into Spark even though we could do it. Again, please recall that our strategy is to follow the conventional style for each system.

**Hadoop vs. HaLoop**

Hadoop and HaLoop are similar in that HaLoop is a direct extension of Hadoop. The performance improvement in HaLoop is mostly achieved by the reducer input cache which converts HDFS I/O to local disk I/O [2]. The size of static data that can benefit from the reducer input cache is much larger for ClueWeb than for LiveJournal. Thus, the difference between Hadoop and HaLoop became more prominent in Figures 1b and 1d with ClueWeb than in Figures 1a and 1c with LiveJournal.

The effect of the reducer input cache can be marginal depending on the algorithms. In PageRank, recursive query, and logistic regression, the keys of the intermediate results that determine the reducer to be sent do *not* change across iterations. In contrast, in $k$-means, the keys of the intermediate results are determined by the current set of centroids, changing per iteration. Thus, the hit ratio of the reducer input cache cannot be very high. On the other hand, the mapper input cache is more important for $k$-means than the reducer input cache. Not surprisingly, Hadoop manages this issue by observing data locality, as shown by Figure 1e.

The reducer input cache is also less effective when intermediate data is small. With regard to logistic regression in Figure 1f, the size of the intermediate results to be shuffled is tiny as in Table 4, because they are partial sums of the parameter values. Also, they do not need to be joined with the static data.

**iMapReduce**

iMapReduce showed the best performance among the disk-based engines but showed worse performance than the memory-based engines. The performance gain is mostly achieved by the persistent connections between each pair of a map task and a reduce task. Its effect was verified by the smaller amounts of the total HDFS read and the reduce shuffle I/O in Tables 3 and 4. However, it was difficult to draw meaningful conclusions about iMapReduce since we could not complete many experiments due to its instability.

## 3.2 Effect of Data Size

Figure 2 shows the normalized time of the five systems for the same six combinations of algorithms and data sets as the data size increases from the data set **1X** to the data set **5X**. (1) In general, the disk-based engines showed sub-linear scalability as the data size increased. This high scalability is a little misleading because the elapsed time of a disk-based engine with 1X was much longer than that of a memory-based engine. Thus, the *normalized* elapsed time even for the same elapsed time was shown to be lower in Hadoop or HaLoop. (2) On the other hand, we observed that the elapsed time of a memory-based engine sometimes rapidly increased as the data size increased when the systems start using disk as well because a data set did not fit in main memory.

A few interesting observations in Figure 2 with regard to the memory-based engines are as follows.

- In Figures 2a and 2d, the elapsed time of Twister started to increase *rapidly* at 3X or 4X. ActiveMQ, the broker service, was configured to use 5 GB of main memory. When we measured the size of the data transferred from the master to each worker in Figure 2a, it amounted to 2.32 GB for 1X, 4.66 GB for 2X, and 7.86 GB for 3X. Thus, ActiveMQ used disk from 3X, causing the slow down in broadcasting.
- Since all the implementations are in Java, garbage collection has significant impact on the performance of the memory-based engines. A *major (or full) garbage collection* is triggered when the amount of available memory falls below a certain threshold. It is costly but occurs quite infrequently. We found that major garbage collection occurred for Twister with 4X and 5X in Figure 2d, contributed to the increase of the execution time.
- In comparison between Figure 2a and Figure 2b, the elapsed time of Spark increased more rapidly as the data size increased

with ClueWeb than with LiveJournal. The main difference between the two cases is whether disk was used for the shuffle. While no disk was used with LiveJournal, 34 GB (1X) to 484 GB (5X) of disk space was used with ClueWeb.

## 3.3 Effect of Iteration Number

Figure 3 shows the normalized time of the five systems for the same six combinations of algorithms and data sets as the number of iterations increases from 2 to 10. All five systems showed reasonable increases as the number of iterations increased. That is, the elapsed time for each iteration was nearly constant or even became a little smaller as an algorithm proceeded.

The disk-based engines in general showed constant elapsed time per iteration as shown in the close-to linear slope of Hadoop or HaLoop except in Figure 3d. The memory-based engines typically showed gentle slopes, meaning that the time added after the first iteration was relatively small. This is because after Twister or Spark load static data from the local disk or HDFS, as long as main memory is sufficient to hold static and variable data, they do not have to access disk anymore.

A few noticeable observations in Figure 3 are as follows.

- In Figure 3d, the slope of Hadoop was much steeper than those of the other systems, partially because of generally higher slopes for recursive query than for the other algorithms. Since there is no previous result to check duplicates at the beginning, the proportion of the *first* iteration is not very significant in recursive query compared with the other algorithms.
- The slope of Twister was lower in Figures 3d, 3e, and 3f than in Figures 3a and 3c. This trend can be explained by the amounts of reducer shuffle I/O, i.e., by **Remark 3** and **Remark 4**. The size of the static data mainly affects the elapsed time of the first iteration, whereas the size of the shuffled data mainly affects that of a subsequent iteration. Nevertheless, in Figure 3c, Spark showed higher efficiency than Twister by virtue of **Remark 1**.
- The slope of Spark was very low in Figures 3e and 3f for the same reason as above. While the slope of Spark was kept to be low in other figures, it became steeper in Figures 3c and 3d owing to **Remark 2** as well as in Figures 3a and 3b owing to **Remark 3**.

## 3.4 Effect of Data Skew

This experiment was conducted by running three reducers on our local cluster. Since the vertices in a LFR network were sorted in the order of their degrees, we divided the network into input splits by *range partitioning* to maximize data skew across input splits.

Figure 4 shows the elapsed time of the five systems. The elapsed time of Hadoop, HaLoop, and iMapReduce all increased as the skewness became higher. Comparing the increase rate, Hadoop and HaLoop did not show a significant difference—in both systems by 1.5 times from LFR1 to LFR5. iMapReduce showed a little higher rate (1.7 times) than the two systems. This higher rate was mainly caused by the one-to-one correspondence between mappers and reducers, which hindered the flexibility of scheduling.

Interestingly, Twister and Spark showed almost constant execution time regardless of the skewness. (1) Twister, because of its characteristics of relying on the combiner, typically does not use the vertex id as the key unlike other systems, but uses the index of the map task and pads the PageRank contributions to the value part of a key-value pair. In this way, each reducer will receive almost the same number of PageRank contributions since the keys are evenly distributed irrelevant to the skewness. Partial sums are obtained in each reducer, and these partial sums are aggregated in

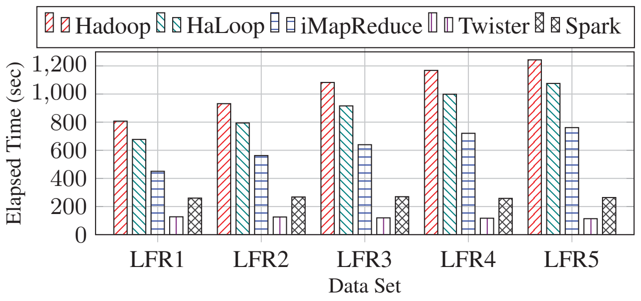


Figure 4: Elapsed time with varying skewness (4 machines).

the combiner. (2) Spark calculates partial sums by using the function `reduceByKey` [9]. Thus, even though a partition contains the vertices of high degrees, the PageRank contributions emitted from that partition to the same destination can be summed up locally. As a result, this function manages the amount of the data shuffled to other workers, just like the combiner of Hadoop. On the other hand, the implementations provided by the authors for Hadoop, HaLoop, and iMapReduce did not use the combiner for PageRank.

## 4. LESSONS LEARNED

In this section, we discuss the lessons learned from the experiments and identify missing opportunities.

**Caching Mechanism, Memory Management**

- The reducer input cache of HaLoop is effective especially when the static data is large. The cache is also sensitive to the characteristics of an algorithm as we observed in the $k$-means experiments. In a nut shell, memory-based engines are also effectively implementing the same technique by caching the static data in memory. To maximize its benefit, the algorithm developers should be aware of such static data partitioning. From the system perspective, an efficient indexing mechanism to quickly retrieve a cached entry would be useful for the developers.
- Memory-based engines are much more efficient than disk-based engines. Thus, the system and algorithm developers should pay more attention to memory management as it is a more critical resource in memory-based engines than in disk-based engines. Selectively storing static or variable data on main memory, just like the function `persist` of the RDD in Spark, is helpful to reduce the overall usage of main memory. Also, the performance of memory-based engines significantly degrades under low-memory condition. Smooth transition from memory to disk using faster storage (e.g., SSD) and efficient spilling would be challenging but interesting.

**Programming Model, High-Level Interface**

- An iteration may consist of complex data flows, and MapReduce may not be the best way to express such algorithms. For example, the map phase is always required before repartition even though no transformation is necessary. A flexible programming model in Spark allows users to express more generic data flows beyond the strict MapReduce API.
- It is promising to have a declarative language like SQL with *iterative construct* that can be translated into a well-understood set of operators and data structures so that let users focus on "what" rather than "how" as well. Spark is the closest in this direction among the systems evaluated in this paper by providing a set of well-understood operations such as map, filter, and joins. Optimizing the iterative data flow with these oprations is another interesting area of research.

**Data Transfer, Network Communication**

- It is *not* always beneficial to force all reduce outputs to be collected by the combiner in Twister. Checking a convergence condition sometimes can be omitted since one may want to terminate an iterative algorithm after a fixed number of iterations. The combiner necessarily becomes a bottleneck and at the same time a single point of failure.
- Efficient data transfer is crucial to achieve high performance. Twister delegates data transfer to the broker network, which requires significant expertise in tuning. Although the feature-rich communication layer may greatly simplify a distributed system, it adds more cost and complexity to the setup and maintenance of the whole system.
- We found that a broadcast is a frequent communication pattern in iterative algorithms. Also, hierarchical aggregation, e.g., which aggregates locally in the reducers and then again in the combiner as in recursive query of Twister, is beneficial for scalability and robust to data skew. Optimizing such common communication patterns is critical to achieve high performance.

**Data Skew, Persistent & Asynchronous Execution**

- Data skew in iterative algorithms can be mitigated by key reassignment and/or hierarchical (partial) aggregation, as shown in Twister and Spark.
- The benefit of persistent socket connection in iMapReduce is *not* verified in our experiments. Although the reduce shuffle I/O is reduced dramatically, the gain in total elapsed time is not as significant. In addition, persistent connection hinders flexible task scheduling, being susceptible to data skew.

Based on our evaluation and experience, the rule-of-thumb choice for a system is suggested in Figure 5.

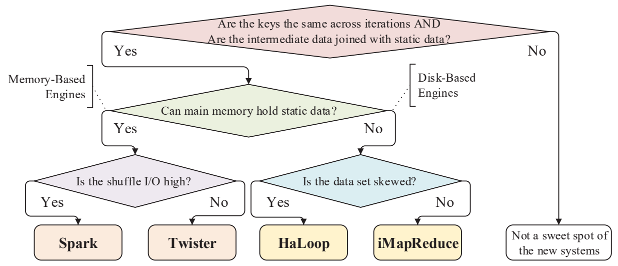

Are the keys the same across iterations AND
Are the intermediate data joined with static data?
Yes
No
Memory-Based Engines
Disk-Based Engines
Can main memory hold static data?
Yes
No
Is the shuffle I/O high?
Yes
No
Is the data set skewed?
Yes
No
Spark
Twister
HaLoop
iMapReduce
Not a sweet spot of the new systems

Figure 5: Choice of the systems depending on the environments.

## 5. CONCLUSION

We conducted extensive experiments on the five MapReduce-like systems, Hadoop, HaLoop, iMapReduce, Twister, and Spark, by running four different algorithms against various real-world and synthetic data sets. We ran these systems on 50 Amazon EC2 x-large instances (200 cores), measuring elapsed time, remote disk I/O, and reduce shuffle I/O, in order to capture their performance characteristics. The new systems were shown to improve the performance of iterative algorithms mainly by reducing the disk I/O for static data. We found that Spark in general achieved the best performance when the operations were executed entirely in main memory. Through our evaluation and interpretation, we discovered the primary factors that impact on the performance: invariability of the keys of variable data, availability of main memory especially in memory-based engines, amount of variable (intermediate) data transferred together with network utilization, and flexibility of task scheduling. The sweet spots of each system were identified using these factors. Since the resources (CPU cores and main memory) of a cluster node become more powerful, we expect that memory-based engines such as Spark will become more prevalent in the near future. Automatic tuning of granularity of tasks, graceful transition between memory and disk, and optimization of common operations are yet to be solved and still have room for improvement.

## Acknowledgments

This research, "Geospatial Big Data Management, Analysis and Service Platform Technology Development," was supported by the MOLIT (The Ministry of Land, Infrastructure and Transport), Korea, under the national spatial information research program supervised by the KAIA (Korea Agency for Infrastructure Technology Advancement) (16NSIP-B081011-03).

## 6. REFERENCES

[1] Apache. Apache Hadoop. https://hadoop.apache.org/.

[2] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[3] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. The HaLoop approach to large-scale iterative data analysis. *VLDB J.*, 21(2):169–190, 2012.

[4] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *NSDI*, pages 313–328, 2010.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[6] C. Doulkeridis and K. Nørvåg. A survey of large-scale analytical query processing in MapReduce. *VLDB J.*, 23(3):355–380, 2014.

[7] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative MapReduce. In *HPDC*, pages 810–818, 2010.

[8] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An experimental comparison of Pregel-like graph processing systems. *PVLDB*, 7(12):1047–1058, 2014.

[9] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia. *Learning Spark*. O'Reilly Media, 2015.

[10] Y. Kwon, D. Nunley, J. P. Gardner, M. Balazinska, B. Howe, and S. Loebman. Scalable clustering algorithm for n-body simulations in a shared-nothing cluster. In *SSDBM*, pages 132–150, 2010.

[11] A. Lancichinetti and S. Fortunato. Community detection algorithms: A comparative analysis. *Phys. Rev. E*, 80(5):056117, 2009.

[12] K. Lee, Y. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with MapReduce: A survey. *SIGMOD Record*, 40(4):11–20, 2011.

[13] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[14] D. Miner and A. Shook. *MapReduce Design Patterns*. O'Reilly Media, 2012.

[15] K. Shim. MapReduce algorithms for big data analysis. *PVLDB*, 5(12):2016–2017, 2012. Tutorial.

[16] B. Snyder and D. Bosanac. *ActiveMQ in Action*. Manning Publications, 2011.

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

[18] Y. Zhang, Q. Gao, L. Gao, and C. Wang. iMapReduce: A distributed computing framework for iterative computation. *J. Grid Comput.*, 10(1):47–68, 2012.