A Comparison of General-Purpose Distributed Systems for Data Processing

Jinfeng Li, James Cheng, Yunjian Zhao, Fan Yang, Yuzhen Huang, Haipeng Chen, Ruihao Zhao
Department of Computer Science and Engineering
The Chinese University of Hong Kong
{*jfli, jcheng, yjzhao, fyang, yzhuang, hpchen, rhzhao*}*@cse.cuhk.edu.hk*

***Abstract*—General-purpose distributed systems for data processing become popular in recent years due to the high demand from industry for big data analytics. However, there is a lack of comprehensive comparison among these systems and detailed analysis on their performance, which makes it difficult for users to choose the right systems for their applications and hard for system developers to identify which aspects of a system can be improved. In this paper, we conduct an extensive performance study on four state-of-the-art general-purpose distributed computing systems. We evaluate the performance of these systems on three types of workloads that are very common for big data analytics in industry today, namely non-iterative bulk workloads, iterative graph workloads, and iterative machine learning workloads. Through the study, we identify the strengths and limitations of each system. We also test the scalability and analyze the programming complexity of using each system. Our results reveal useful insights on the design and implementation of general-purpose distributed computing systems, which help the development of better new systems in the future.**

I. Introduction

In recent years, we have seen a surge of distributed computing systems for large-scale data analytics, which have created high impact on both industry and academia. MapReduce [8] and Hadoop [2] are among the first such systems. However, MapReduce and Hadoop are not designed for processing iterative workloads and are hence inefficient for jobs such as machine learning and graph analytics. More recent solutions [25], [1], [17], [12] take advantage of the large aggregate memory in a modern computer cluster, and design systems that focus on in-memory processing. Due to much better performance as well as user-friendly interfaces, these recent systems have become more popular and have attracted much attention from industry. We chose four recent systems for detailed performance comparison, which are Spark [25], Flink [1], Naiad [17], and Husky [12]. Both Spark and Flink are Apache projects, and they have been popularly used for big data analytics in industry. Naiad is known for its impressive performance and was shown to outperform earlier general-purpose systems such as DryadLINQ [24]. Husky is a new system reported to have remarkable performance in many types of workloads, and even outperforms domain-specific systems such as GraphLab [9] and Parameter Servers [11], [14], [21].

This study is mainly motivated by the following reasons. First, while systems such as Spark and Flink are well-known and popularly used, whether these popular systems are really suitable for a given application is in fact unclear to most users and researchers. Second, it would be good for users to understand the limitations of these systems, so that they can make a better choice of which system to use. Third, the design differences in these systems, their strengths and limitations, are valuable information that can help researchers design better new systems, improve existing systems, and develop better applications on these systems.

We study the performance of these systems using three types of workloads: (1) non-iterative bulk workloads, (2) iterative graph analytics workloads, and (3) iterative machine learning workloads. These three types of workloads cover a wide range of important applications of big data analytics in industry. We analyze how various techniques used in each system affect the system performance in terms of both computation and communication for each workload. We discuss how the implementation and the programming language of a system may result in different performance for different tasks. We identify the strengths and limitations of each system, and provide insights how a system can be improved. We also show how user-friendly it is to program in each of these systems for different types of data analytics workloads. The results we obtained and the insights we drew from this study are particularly useful for the development of a new user-friendly system for efficient large-scale data analytics.

We organize our study as follows. We first give a brief introduction of various systems in Section II. Then, we discuss the setup of our performance study in Section III. We analyze in details the performance of each system in Sections IV to VI for the three types of workloads. We test the scalability of the four systems in Section VII. Finally, we summarize our findings in Section VIII.

II. System Overview

We first give an overview on the systems we studied, as well as other systems and related work.

A. Spark

Spark [25] abstracts distributed datasets as *resilient distributed datasets* (*RDDs*). An RDD is a collection of immutable records. Users can easily develop distributed algorithms by defining *coarse-grained* transformations on

RDDs, such as `map`, `flatMap`, `join`, `reduceByKey`, and so on. To support more efficient graph analytics, a graph library, GraphX [10], was built on Spark, which implements optimization techniques specifically for graph processing, such as distributed join and materialized view maintenance. Thus, in our experiments, we use GraphX to implement graph algorithms in our performance evaluation.

B. Flink

Flink [1], also known as Stratosphere, combines low-latency stream processing with high-throughput batch processing. Flink provides three sets of APIs: *DataSet* API, *DataStream* API, and *Table* API, which are for batch processing, streaming processing, and SQL-like data analysis, respectively. Since our performance study focuses on general-purpose batch processing, we only use DataSet API in this paper. DataSet is the data abstraction in Flink, and transformations such as `map`, `flatMap`, `reduce`, `union`, etc., can be applied on DataSets, which is similar to applying transformations on RDDs in Spark. User defines DataSets and specifies transformations on DataSets, while the Flink system does the optimization and parallel execution of the program. Flink includes a sub-system called Gelly, which offers a set of graph operators to allow easier implementation of distributed graph algorithms.

C. Naiad

Naiad [17] proposes the *timely dataflow* computing model, which allows low-latency stream processing as well as high-throughput batch processing. Naiad implements timely dataflow in C# and provides a lightweight mechanism to manage the timestamps. Naiad's API provides four key primitives described as follows. `SendBy()` and `OnRecv()` are for asynchronous message processing. `NotifyAt()` and `OnNotify()` are for synchronous processing. For each message, Naiad will assign a timestamp to it and hence different streams from different batches can be distinguished. Users can cache the stream with a certain timestamp on `OnRecv()`, and process all the data with this timestamp together when `OnNotify()` is triggered. Naiad also proposes distributed progress tracking protocol to guarantee the correct invocation of `OnNotify()`. Naiad implements LINQ to provide an API with declarative programming, and also offers an API, called *GraphLINQ* for distributed graph algorithms.

D. Husky

Husky [12] adopts a flexible computation model with which users can define objects that can be both fine-grained (e.g., vertices in a graph, records in a table) and coarse-grained (e.g., subgraphs, tables), and specify the actions of the objects and the interactions among objects through messaging primitives such as `push` and `pull`. Husky supports popular computing frameworks, e.g., MapReduce [8], Pregel [15] (for graph processing), and Parameter Servers [11], [14], [21] (for machine learning), in one unified framework.

E. Other Systems and Related Work

Many systems have been proposed to generalize MapReduce and/or build high-level abstraction upon MapReduce and Hadoop, e.g., Hive [3], Pig [18], Dryad [13], DryadLINQ [24], and FlumeJava [6]. Other systems such as Haloop [4], iMapReduce [26], and DynMR [23], aim to improve the efficiency of Hadoop MapReduce (e.g., for iterative computing).

Pavlo et al. [19] compared Hadoop with parallel DBMS Vertica and a commercial DBMS, but iterative workloads are not included in their benchmarks. Cai et al. [5] evaluated the development and execution of machine learning algorithms on various distributed platforms, but they did not consider iterative graph algorithms and non-iterative bulk workloads. Shi et al. [20] compared Spark with Hadoop, but did not consider other general-purpose systems that support much more efficient iterative computation than Hadoop.

III. Experimental Setup

In the following four sections, we study the performance of Spark, Flink, Naiad, and Husky on (1) non-iterative bulk workloads, (2) iterative graph analytics workloads, (3) iterative machine learning workloads, and (4) scalability. In addition, we also compare the usability of the systems by showing how easy it is to program the different tasks in each system. All codes will be made available for repeatability.

We ran our experiments on a cluster with 18 machines, each is equipped two 2.0GHz Intel(R) Xeon(R) E5-2620 CPU (with total 24 cores), 48GB RAM, a SATA disk (6Gb/s, 10k rpm, 64MB cache) and a Broadcom's BCM5720 Gigabit Ethernet Controller. CentOS 6.5 with 2.6.32 linux kernel is installed on each machine. The JAVA version is 1.7.0 and Scala version is 2.10.4.

We used Spark 1.5.1 and ran in standalone mode, and disabled shuffling to disk to maximize its performance. We used kryo serializer because it can save memory and accelerate many algorithms. We used Flink 0.9.1 and followed settings advised by configuration guide. We used Naiad release_0.5m. Naiad officially supports mono and thus we built Naiad for mono and ran on our linux clusters. The mono version is 3.12.1. We used Husky 0.1 and used its default settings. We used HDFS of Hadoop 2.6.0.

IV. Non-Iterative Bulk Workloads

We first analyze the four systems on two typical non-iterative bulk workloads, WordCount and TeraSort.

```
val linesRDD = textFile(inputPath)
val resultRDD = linesRDD
                    .flatMap(_.split("\\W+"))
                    .map((_, 1))
                    .reduceByKey(_+_)
```

Listing 1: WordCount implementation in Spark

Table I: WordCount time (sec) by the end of each phase

	End of P1	End of P2	End of P3	End of P4
Spark	51.9	57.5	58.1	72.9
Flink	15.4	32.2	32.9	126.3
Naiad	23.4	60.9	64.9	79.5
Husky	10.0	–	11.2	24.2

A. *WordCount*

WordCount is one of the most representative MapReduce workloads, which counts the occurrences of each distinct word in a corpus. In the Map phase, input sentences are split into words. These words are then mapped to (*key*, *value*) pairs, where the *key* is simply the word and the *value* is 1 for each word. The system then shuffles these (*key*, *value*) pairs and sends them to different Reducers according to their *key*. Combiners may be applied before shuffling in order to combine all *value*'s (i.e., summing up all the '1's) with the same *key*. In the Reduce phase, the (combined) *value*'s of the same *key* are sent to the same Reducer, and this Reducer sums up all these *value*'s for the *key*, which is the count of the corresponding word.

1) Implementation of WordCount: The above MapReduce algorithm is implemented using transformation operators provided in the APIs of Spark, Flink and Naiad. We take the Spark implementation as an example, as shown in Listing 1. `textFile` first reads files from HDFS to obtain an RDD, `linesRDD`. Each line is then split into words by the `flatMap` transformation, and each *word* is then mapped to (*word*, 1) pairs. Finally, `reduceByKey` shuffles these pairs and sums up all values with the same key (i.e., each distinct *word*). Combiner is automatically enabled by the Spark system. The implementations in Flink and Naiad are similar to that in Spark. For the implementation in Husky, we define the words objects, each of which takes its content as the key. After splitting a sentence, we send out the (*word*, 1) pairs to these objects and each object will aggregate received messages(i.e. counts).

2) Performance on WordCount and Analysis: The WordCount algorithms described in Section IV-A1 can be divided into four phases: (P1) *loading*: input data is loaded from HDFS and partitioned among workers, (P2) *splitting*: sentences are split into words, (P3) *mapping*: each *word* is mapped to a (*word*, 1) pair, and (P4) *aggregating*: the (*word*, 1) pairs are first combined and then shuffled to different workers to obtain the final count for each *word*. We used random text generator on Hadoop to generate dataset for WordCount. In this experiment, we generated 90GB dataset.

Table II: WordCount time in each phase as % of total time

	P1	P2	P3	P3
Spark	71.2%	7.6%	0.9%	20.3%
Flink	12.2%	13.3%	0.6%	73.9%
Naiad	29.4%	47.2%	5.0%	18.4%
Husky	41.3%	–	5.0%	53.7%

Section VII will present more results on larger datasets. We analyzed the design and implementation of each of these four systems and have the following findings that can explain the performance result.

For Spark, the dominating cost is the loading phase, which is about 71.2% of Spark's total running time for WordCount, as shown in Table II. The loading time of Spark is also much longer than that of all the other three systems. We examined the cause and found that there is a totaling about 81GB network traffic(out of 90GB input data) in the phase of loading in Spark, which becomes the performance bottleneck. Flink, Naiad and Husky do not have this problem since they all have strategies that try to load local files directly from local disks of each machine. Apart from data loading, Spark is quite efficient on the other three phases of WordCount, and its running time for these three phases is also significantly shorter than that of Flink and Naiad.

For Flink, it finishes the first three phases in only 32.9 seconds, which is significantly faster than both Spark and Naiad. However, Flink spends much more time on the last phase, and its overall running time is longer than all the other systems. Examining the source codes of Flink, we found that when applying combiner in each local worker in the last phase, Flink uses a sort-based implementation to group by the (*word*, 1) pairs. This is expensive since the number of distinct words is comparatively much smaller than the total number of words in a corpus. On the contrary, the other three systems all employ a hash-based implementation to group by the (*word*, 1) pairs, which is significantly more efficient.

For Naiad, the processing is relatively slow at the splitting phase, which takes 47.2% of its total running time. This is because Naiad is implemented in C# and the C# function used to split a string is inefficient. Apart from splitting, the mapping phase in Naiad also takes much longer time than that in the other systems. The mapping phase in Naiad is about 5.0% of its total running time, while it is less than 1% for Spark and Flink (note that the 5.0% for Husky includes both splitting and mapping). To analyze why mapping is more expensive in Naiad, we rewrite the program and simply let the *words* stream flow into a new processing element without any data transformation(i.e. map each word to (*word*, 1) pair). We measured that mapping now takes about 2.4 seconds. This implies that the transformation takes $(64.9 - 60.9 - 2.4) = 1.6$ seconds only and there is an overhead of 2.4 seconds in the system. In order to support both low-latency stream processing as well as high-throughput batch processing, Naiad introduces and implements distributed progress tracking protocol, which coordinates the above two executions but brings extra workload to the system.

For Husky, it is the fastest among all the systems for WordCount. It uses 41.3% of the total time on the data loading, which is much more than the 12.2% and 29.4% of Flink and Naiad. But this is not an indicator that Husky's loading is not efficient, as it only takes 10.0 seconds as shown Table I. On the contrary, this shows that Husky is very efficient since for the simple WordCount workload, the computation is only a linear scan of the data and counting the occurrences of each distinct word is a light-weight process, and thus the total computation time should not be much longer than the data loading time. Husky's is implemented in C++ and thus it can use the efficient tokenizer function provided in the C++ library for splitting, while the splitted words are immediately pipelined to the mapping phase and thus we cannot clearly separate the splitting and mapping phases, i.e., P2 and P3 in Tables I and II. Although C++ implementation leads to more efficient data loading and splitting, it requires more development cost (e.g., lines of code) compared with Spark, Flink and Naiad.

3) Insights from WordCount Workload: From the analysis on the WordCount performance, we obtain the following insights on how to design and implement an efficient system for non-iterative bulk workloads such as WordCount.

Data partitioning. HDFS or any DFS is often deployed on the same cluster which the system runs on. Thus, data loaded from a local disk should go to a local partition whenever possible, so that the data can be processed as much as possible by the local worker, especially for bulk data workloads. For example, if Spark does not distribute much of the data loaded from a local disk to partitions in remote machines, its overall performance can be comparable with that of other systems on WordCount.

Programming language. Although the development time is usually shorter if system developers use JAVA or C# to implement a system, a C++ implementation may still have many advantages when efficiency is considered, and it also provides libraries with many highly efficient functions. Thus, it is desirable to implement the back-end execution engine in C++, especially when performance is critical. However, as many data scientists (who may not be programmers who are familiar with C++) today prefer to use a high-level functional or declarative language to develop their applications, it becomes also important to provide an API that supports high-level languages such as Python.

Model-specific overheads. Message passing on Naiad is asynchronous but notification is synchronized. In order to guarantee the correct delivery of notification, Naiad proposes the distributed progress tracking protocol [17]. Such a protocol is only required when timely dataflow is the underlying computing model. The protocol counts the in- and out-messages for each timestamp and thus may limit the scalability of the system. However, Naiad is the only system that supports low-latency stream processing that can handle one record at a time. When designing a system, a developer should consider such tradeoff between more functionalities and fast performance.

Table III: TeraSort running time (in sec)

	Spark	Flink	Naiad	Husky
90GB data	300.6	132.1	125.6	122.7
18GB data	67.8	32.7	25.4	24.4

B. TeraSort

Sorting is one of the most common operations in data processing. Thus, it is important to see how the systems perform on sorting. We use TeraSort, one of the most scalable distributed sorting algorithm. TeraSort is also known as a typical non-iterative bulk MapReduce workload [8]. We generated 90GB records, where each record has 100 bytes. We take the first 10 bytes as the key and the remaining 90 bytes as the value. TeraSort sorts the records by their keys.

Implementation. The implementations of TeraSort in the four systems all follow the typical MapReduce TeraSort implementation [8]. First, the system reads records from HDFS. Then, the records are partitioned by their keys and distributed to different workers. Finally, each worker sorts their records independently. By designing a range partitioning function, records are distributed to workers based on the ranges and the final sorted result is just to concatenate the ranges of sorted records.

Performance on TeraSort and analysis. We report the running time of each system for TeraSort in Table III. We first conducted experiments on 18GB dataset and then increased the input size to 90 GB.

Husky is the fastest among all the systems, but Naiad and Flink also have comparable performance and the difference may be because Husky was in C++, as discussed in Section IV-A3. As the dataset size increases 5 times, the running time of the systems also increases roughly 5 times. Spark is much slower, mainly because Spark separates the execution of loading from shuffling, as shuffling starts only after loading all records. In the other systems, loading and shuffling are pipelined, where records are shuffled after they are read.

V. Iterative Graph Workloads

We focus on two categories of graph workloads: *all-active* and *partially-active*. For all-active workloads, all vertices in the input graph are involved in the computation in each iteration. PageRank computation belongs to this category. For partially-active workloads, only partial vertices of the input graph are involved in the computation and vertices may be active in different iterations. The computation of single-source shortest path (SSSP) belongs to this category. PageRank and SSSP are the most typical workloads used in the evaluation of existing graph processing systems, and are thus chosen in our benchmark.

A. *PageRank*

Implementing distributed graph algorithms in existing general-purpose systems are more complicated than implementing tasks in Section IV. GraphX (i.e., Graph processing library on Spark), Gelly (i.e., Graph processing library on Flink) and Husky adopt Pregel's vertex-centric API [15] for implementing distributed graph algorithms. As shown in Listing 2, each vertex v simply updates its PageRank value according to the PageRank equation and then sends a value, i.e., the new PageRank value divided by its out-degree, to each of its out-neighbors for use in the next iteration.

```
def vertex_exec(v):
  pr = (v.get_msgs())*0.85+0.15
  v.set_value(pr)
  for nb in v.get_nbs():
    v.send_msg(pr/len(v.get_nbs()), nb.id)
```

Listing 2: PageRank implementation in Spark/GraphX, Flink/Gelly, and Husky

Naiad provides the GraphLINQ library for graph processing. As shown in Listing 3, Naiad first creates the `edges`, `degrees`, and `vertices` sets, which keep the sets of edges, out-degrees of vertices, and vertices in the input graph, respectively. In the i-th iteration, Naiad first joins `vertices` and `degrees` to obtain $pr'(u) = 0.85 * (pr_{i-1}(u)/|\Gamma_{out}(u)|)$ for each vertex u. Then, `GraphReduce` joins `vertices` and `edges` to obtain $\sum_{u \in \Gamma_{in}(v)} pr'(u)$ for each vertex v, which is then used to compute the final $pr_i(v)$.

```
Function PageRank(
  this vertices, degrees, edges){
    vertices.NodeJoin(degrees,
      (rank, degree) => rank*0.85/degree)
    .GraphReduce(edges, (x, y) => x+y, false)
    .Select(v =>
      v.node.WithValue(v.value + 0.15) )
}
```

Listing 3: PageRank implementation in GraphLINQ

Performance on PageRank and analysis. We use two graphs, WebUK[1] and Twitter[2], and some of their statistics are given in Table IV. Both graphs are popularly used in the evaluation of existing specialized graph processing systems.

Table V reports the preprocessing time taken by each system and their running time per iteration. Husky has better performance than the other systems in all cases. Besides it is developed in C++, there are two more reasons. First, husky presorts vertices by ids to accelerate the join operation of messages and vertices. Second, husky requires much smaller communication cost, as reported in Table VI, due to effective machine-level combiner.

Table IV: Graph Datasets

Graph	—V—	—E—	max deg	avg deg
Twitter	52,579,682	1,963,263,821	779,958	37.33
WebUK	133,633,040	5,507,679,822	22,429	41.21

Table V: PageRank preprocessing & running time (in sec)

	Twitter		WebUK	
	Pre-processing	Per-iteration	Pre-processing	Per-iteration
Spark	83.2	29.5	180.6	104.8
Flink	19.7	33.2	47.5	89.2
Naiad	56.4	9.1	148.0	22.7
Husky	11.9	2.9	26.0	3.3

Table VI: PageRank network traffic (NT) per iteration and memory consumption (MC) (in GB)

	Twitter (NT)	WebUK (NT)	Twitter (MC)	WebUK (MC)
Spark	19.1	53.6	23.0	34.6
Flink	10.1	21.1	<30.0	<40.0
Naiad	16.0	44.1	5.7	10.6
Husky	2.3	1.9	5.5	11.9

Spark has much longer preprocessing time as reported in Table V, because GraphX needs to build a property graph and other data structures such as bitmask, routing table, and local indexes. These data structures are used to accelerate the execution of iterative graph algorithms. However, its running time is still worse than the other systems (only slightly better than Flink on Twitter). There are two main reasons. First, GraphX requires shuffling in every iteration for joining vertices and edges. The shuffling incurs high network traffic as shown in Table VI, since GraphX applies vertex-cut partitioning and the edges of a vertex may be stored in different machines. Second, as RDDs are immutable, GraphX creates new RDDs, while in Naiad and Husky most data structures are just updated and re-used (Flink only re-creates a new vertex set). These RDDs are large and hence costly to construct, which also uses three times more memory than Naiad and Husky, as reported in Table VI. Note that Flink's actual memory consumption is less than that reported in Table VI, because Flink requires users to pre-allocate memory to the computation, which is hard to guess and we assigned memory a bit more than that used by Spark to ensure that Flink does not run out of memory.

[1]http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05/
[2]http://konect.uni-koblenz.de/networks/twitter_mpi

Table VII: SSSP performance (time in sec) on Twitter

	Total time	Pre-processing	Running time	5th Iter	Last Iter
Spark	496.6	50.9	445.7	26.2	37.1
Flink	169.1	19.5	149.6	23.3	8.3
Naiad	83.6	55.5	28.1	4.3	0.1
Husky	27.2	12.1	15.1	2.7	0.2

B. SSSP

SSSP computes the shortest-path distance from a source vertex to every vertex in the input graph. Unlike PageRank, SSSP computation only accesses partial vertices in each iteration. Thus, GraphX, Gelly and GraphLINQ all attempt to separate active vertices from inactive vertices in each iteration. GraphX clusters edges so that edges belonging to inactive vertices will not be scanned when joining the vertex set and the edge set. Gelly uses incremental `Iteration` operator and GraphLINQ uses `IterateAndAccumulate` operator, which processes only active vertices and corresponding edges in each iteration.

By identifying the active vertices, an implementation of SSSP using a Pregel-like API in GraphX, Gelly and Husky is more efficient than that an implementation using coarse-grained transformations. The implementation of vertex program, as shown in Listing 4, is similar to PageRank shown in Listing 2, except that the aggregation used here is `min` and the message to be sent to v's neighbors is the sum of v's distance (from the source) and the weight of the edge to its neighbor.

```
def vertex_exec(v):
  dist = min(v.get_msgs())
  if(v.value > dist){
    v.set_value(dist)
    for nb in v.get_nbs():
      v.send_msg(dist + nb.weight, nb.id)
  }
```

Listing 4: SSSP implementation in Spark/GraphX, Flink/Gelly, and Husky

The implementation in GraphLINQ is similar to its PageRank implementation in Listing 3. In each iteration, only active vertices will join with the set of edges. The distance values gathered from neighbors of a vertex will be aggregated by a user-specified stateful processing element, `BlockingAggregate`, which maintains a hash map with *key* being the vertex id and *value* being the current distance of the vertex from the source.

Performance on SSSP and analysis. Table VII reports the total time, which includes the pre-processing time (2nd column) and running time on SSSP (3rd column), of the four systems. The results are for the Twitter graph only, for which the number of active vertices is very large in the 5th iteration and very small in the last (i.e., 15-th) iteration. Thus, we also report the time taken to process these two iterations.

Husky gives the best performance because it allows fine-grained access to vertices and edges, and thus in each iteration only the active vertices and their edges are processed. Naiad also only involves only the edges of active vertices in the join in each iteration. However, Naiad does not support combiner, which makes it slower than Husky. The use of combiner in Husky has an obvious advantage as shown in the 5th iteration, which has many active vertices and hence many messages can be combined. On the contrary, in the last iteration when there are few active vertices and messages, Husky is not better than Naiad.

Spark and Flink involve the whole set of edges in the join in each iteration, and are thus slower than both Naiad and Husky. This is also the reason why in the last iteration when there are few active vertices, Spark and Flink still spend much time in the join. Spark takes 37.1 seconds in the last iteration, which is much longer than its 5th iteration. We found that the running time of Spark increases after each iteration, which is probably due to the long lineage of RDDs.

SSSP on the WebUK graph, however, takes 664 iterations. Due to the large number of iterations, Spark failed to compute SSSP on WebUK. Flink took 5,534.2 seconds to finish the job mainly because involving the whole set of edges in the join in each iteration is time consuming. Naiad took 144.6 seconds to read the whole graph (vs. Husky's 25 seconds) but it finished computing SSSP in 139.8 seconds, which is even considerably shorter than Husky's 216.5 seconds. This is mainly because for WebUK, only 3 out of 664 iterations involve many active vertices, while the other 661 iterations have few active vertices and hence few messages, thus the benefit gained by applying combiner in Husky does not pay off its overhead.

C. Insights from Graph Workloads

We obtain the following insights from the PageRank and SSSP workloads.

Data access pattern. The much superior performance of Husky for graph processing shows that coarse-grained data abstractions such as RDDs (in Spark) and DataSets (in Flink), as well as coarse-grained transformations, may not be suitable for fine-grained graph operations. On the other hand, Naiad can support efficient fine-grained operations with its fine-grained primitives, but it will be desirable to implement a more user-friendly API such as the vertex-centric API to support higher-level graph operators (instead of using join as in GraphLINQ).

Combiner. An effective combiner can significantly reduce network traffic. As our results show that Husky's machine-lever combiner, i.e., combining messages for all workers in the same machine, leads to lower network traffic than Flink's worker-level combiner, which in turn has lower network traffic than Naiad which does not use a combiner (due to a bug in GraphLINQ). However, for SSSP on WebUK where there are few messages to be combined in most iterations, our comparison on Husky and Naiad shows that the overhead of applying combiner is higher than directly sending the messages without first combining them. Thus, Husky (and also other systems) may adopt a strategy that uses a combiner only when there are sufficient number of messages.

Graph Partitioning. The default partitioning algorithm used in all the systems are simply by random hashing. A number of effective graph partitioning algorithms have been proposed recently [16], [22], and it has also been shown [9] that vertex-cut partitioning can lead to more efficient graph computation than edge-cut partitioning. The general-purpose systems may apply the more effective partitioning algorithms or vertex-cut partitioning to improve performance, although one also needs to study how to efficiently apply these partitioning strategies in existing systems.

VI. Iterative Machine Learning Workloads

We further compare these systems on two iterative machine learning workloads, *logistic regression* (*LR*) with gradient descent and *alternating least squares* (*ALS*). In each iteration, LR needs to broadcast a huge amount of parameters and update them. Thus, we can evaluate the ability of various systems in handling *large broadcast variables* (i.e., variables with many items to be broadcast, e.g., a long vector may have millions of such items). ALS is popular in collaborating filtering to learn latent factors. These latent factors can be furthered used to predict the rating of a user to an item. Other than per iteration time, data scientists are more interested in how fast ALS can converge. Therefore, we will evaluate the rate of convergence for different systems. We implemented LR and ALS on Naiad but they did not scale well. Since there is no programming guide for machine learning algorithms on Naiad, it is hard to provide better implementations and thus we skip Naiad in this section.

A. Logistic Regression

Given two set of data points, LR can train these points and find a boundary to separate these points. A boundary is usually represented as a D dimensional vector, where D is the number of features of these points (e.g., the number of distinct words in a corpus). We use ω to denote a boundary, which is usually initialized with random numbers. Gradient descent will be used to improve ω iteratively. In each iteration, ω is made visible to all the points, so that each point can be trained and generate partial gradient values to ω. These partial gradient values are then summed up and added to ω.

1) Implementation of Logistic Regression: The main difference in the implementations of LR in the three systems lies in the way ω is maintained, either *centralized* or *distributed*. In the centralized way, ω is stored in the master, which is responsible for broadcasting ω to all other machines, and collecting the partial gradient values and updating ω. In the distributed way, each of the n machines in a cluster manages a fraction (e.g., $1/n$) of ω, then the system will require each machine to broadcast its share of ω to all the other machines. We may use a vector to represent the features of a data point, but this requires more space since many positions of this vector could be zero (e.g., a text to be classified may contain only a small number of distinct words in a corpus). We adopt the format in LIBSVM [7] to represent features by a list of (idx, fv) pairs, where idx is the feature position in the vector and fv is the feature value.

Spark and Flink implementation. Spark maintains ω by the master node in a centralized way, stored in an array. In each iteration, broadcasting ω to workers will be done by the system automatically. When processing each data point, partial gradient values will be generated by mapping every feature. Next, `reduceByKey` aggregates these partial values to generate the final gradient. Then, `collect` gathers the gradients from all the workers to the master to update ω. Flink handles ω similarly as Spark.

Husky implementation. Husky supports managing large variables (e.g., vector, matrix) in a distributed manner. We first register a broadcast variable, `W`. `W` maintains two fields, each with D elements, one is to store ω and another is to sum partial gradient values to form the gradient. Then, the `add` function defines how to operate on the two fields of `W` at the end of each iteration, in which the elements in the gradient will be added to the corresponding elements in ω. The system manages the distributed maintenance automatically.

2) Performance on LR and Analysis: We used the RCV1 and KDD10b datasets from LIBSVM [7]. RCV1 has 677,399 points and 47,236 features. KDD10b has 19,264,097 points and 29,890,095 features. We set the learning rate to 0.1. Table VIII reports the per-iteration time and GUB time of running LR with each system, where GUB time is the time taken to maintain ω, which involves gathering the gradients(G), updating(U) and broadcasting(B) ω.

Table VIII: LR performance (running time in sec)

	RCV1		KDD10b	
	Per-Iter	GUB	Per-Iter	GUB
Spark	1.9	< 1	48.3	41.1
Flink	4.9	< 1	-	-
Husky	2.2	< 1	11.4	6.8
Husky-C	1.7	< 1	50.8	49.8

Since RCV1 has only 47K features, ω is small. Thus, the GUB time is short and hard to measure. The per-iteration time is also short and hence there is no significant difference in performance between the systems.

The KDD10b dataset has a much larger ω (nearly 30 million features). Flink failed on KDD10b since they cannot handle more than one million features. Spark can run on

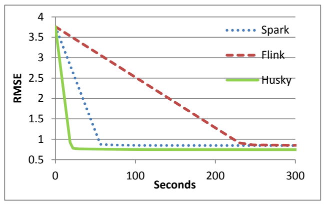
KDD10b but its centralized maintenance of ω leads to a significantly longer GUB time than Husky, since the master bears all the gathering and broadcasting workloads, while all the workers are idle and waiting to receive ω. Thus, the GUB workload in Spark is extremely imbalanced, while the GUB workload in Husky are evenly distributed among all the workers, and hence Husky is 6 times faster than Spark for GUB. On the other hand, the training time, i.e., per-iteration time minus GUB time, of Spark and Husky are comparable.

To further verify that the performance difference between Spark and Husky is due to the way how ω is maintained, we simulated Spark's centralized broadcast and implemented the corresponding LR algorithm in Husky, denoted by Husky-C. As also reported in Table VIII, the GUB time of Husky-C for KDD10b is now comparable with that of Spark due to the bottleneck of GUB with the master. On the other hand, for RCV1 when ω is small, GUB is not a bottleneck in Husky-C and it is even faster than Husky.

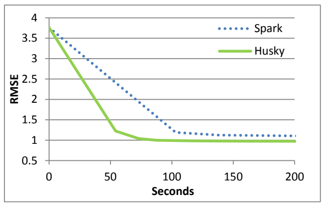
B. Alternating Least Squares

Many recommender systems store ratings in a matrix R, in which the element r_{ij} in the i-th row and the j-th column of R denotes the rating of user i for item j. To predict missing entries in R, a common way is to factorize R into the product of two smaller matrices U and V such that $R \approx U^{\mathrm{T}}V$. U and V store the latent factors for each user and item, respectively. The latent factors for a user (or an item) form a feature vector. ALS learns feature vectors iteratively to make the product of U and V become closer and closer to R. Specifically, the process starts by initializing U with random numbers and uses U to estimate V as follows. For each item, it needs to get the feature vectors of all the users who have rated the item and the corresponding ratings. Then, a least squares problem can be modeled and a new feature vector of the item can be obtained by solving this problem. After obtaining V, we alternate to use V to estimate U in a similar way. This process continues for a user-specified number of iterations or until U and V do not change much.

1) Implementation of ALS: We first describe Husky's implementation(about 200 lines of code), as it follows closely the ALS logic. We define two types of objects, *user* and *item*. Each *user* object stores its feature vector as its state, and is also associated with a set of neighbors, which are the *item* objects this user rated (and the corresponding ratings are also kept). Each *item* object constructs its state and neighbors symmetrically. In each iteration, each *user* sends its id together with its state (i.e., its feature vector) to its *item* neighbors. Since multiple neighbors of the same *user* may be in the same machine, the feature vector of the *user* can be sent to that machine multiple times. The Husky system does the de-duplication automatically. For each *item*, it gets the feature vectors of its *user* neighbors by the *user* ids and updates its feature vector. Then, the updated feature vectors



(a) ALS on Netflix



(b) ALS on YahooMusic

Figure 1: ALS performance

will be sent to its *user* neighbors to update the feature vectors of *user* objects in a symmetric way.

For Spark and Flink, we implemented ALS using coarse-grained transformations, which however are very slow. But for Spark and Flink, they also provide an efficient blocked ALS algorithm in their libraries. The idea of blocked ALS is similar to Husky's implementation, except that users and items are further grouped into blocks. In this way, the feature vector of a user is only sent to the blocks that contain items rated by the user, instead of to all items. However, implement block ALS is not an easy task for most users, as it uses more than 1000 lines of codes in both Spark and Flink.

2) Performance on ALS and Analysis: We compared the converge rate for different systems. We used the Netflix and YahooMusic datasets, which are widely used to evaluate matrix factorization algorithms. Netflix contains 480,189 users, 17,770 items, and 100,480,507 ratings. YahooMusic has 1,823,179 users, 136,736 items, and 699,640,226 ratings. We set regularization parameter to 0.1.

Figure 1 reports the training root-mean-square error (RMSE) versus training time, where the faster the RMSE decreases over time, the better is the performance of a system. Although Spark and Flink use the same ALS algorithm, their performance varies quite differently. Spark converges much faster than Flink on Netflix, while Flink even failed to scale on the larger dataset YahooMusic. Flink uses a low-latency streaming engine to perform batch jobs. To offer low-latency execution, Flink pipelines all the operators of a job and runs as many of them as possible. However, these operators share the same resources such as main memory and may lead to severe contention. Blocked ALS uses many operators and hence the pipeline is long. Flink attempts to cut the pipeline by dividing the execution into three phases, i.e., building user indices, item indices, and running ALS. Between different phases, HDFS is used to transfer the intermediary results and connect the execution. On the contrary, Spark runs the operators one by one and thus has much lower overhead than Flink.

Husky converges faster than both Spark and Flink for both datasets. In fact, the per-iteration running time of both Spark and Husky is very fast and there is no obvious difference. However, to achieve this short per-iteration time, Spark has to maintain the block id indexes, which incurs extra overhead that Husky does not have.

C. Insights from LR and ALS Workloads

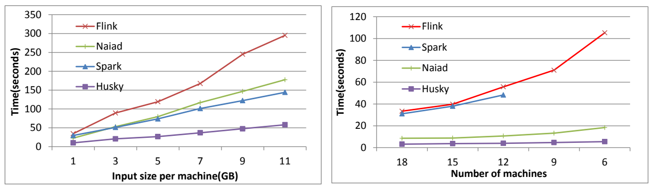
We obtain the following insights from the LR and ALS workloads.

Library support. From the implementations of LR and ALS in different systems, we can see that there is a lack of primitives that are common to many machine learning algorithms in the APIs of all the four systems. Spark and Flink provide an efficient ALS implementation in their library. Although the implementation requires non-trivial effort from most users, it is easy to call and use such library functions. Husky and Naiad may support even more efficient tailor-made implementations, and thus they should provide such library support to include common machine learning algorithms, which is particularly desirable since Husky and Naiad in general have a more efficient back-end system.

Maintenance of large variables. Large variables can be maintained in a distributed manner so that the workload of maintenance can be evenly distributed and does not cause a performance bottleneck in any single machine. This is particularly important for many machine learning algorithms which require to maintain large variables, e.g. vectors and matrices.

VII. Scalability Tests

In the section, we test the scalability of the four systems. We choose WordCount to represent non-iterative workloads and PageRank to represent iterative workloads. The results are reported in Figure 2.



(a) WordCount scalability (b) PageRank scalability

Figure 2: Performance on scalability

WordCount scalability. We fixed the number of machines to 18, each running 24 workers on 24 cores, and increased the input sizes linearly. Flink does not scale well when the input size becomes larger, mainly because its sort-based aggregation becomes the bottleneck when the number of intermediary results is large. Spark, Naiad and Husky use hash-based aggregation and scale linearly. Naiad's performance is mainly affected by sentence splitting and the distributed progress tracking, making it slower than Spark when the size of the input increases. Spark's performance is mainly limited by the overhead of sending local data to remote partitions. Husky does not have these overheads in the other systems and can thus scale better.

PageRank scalability. We ran PageRank on Twitter and report the per-iteration time by varying the number of machines from 18 to 6, each running 24 workers on 24 cores. Spark and Flink started to scale non-linearly when less than 12 machines were used. The main reason is that they used too much memory. This is because their API uses Scala, which makes them easier to program with high-level operators, but at the same time also easy to generate large amounts of intermediate results and lead to huge memory usage. Flink has a more robust virtual memory management implementation and so it is 3 and 5 times faster than Spark when only 9 and 6 machines were used. We do not show Spark's time for 9 and 6 machines in Figure 2 because showing them will make the line of Flink look linear.

Naiad and Husky scale much better and have low memory consumption. When less than 12 machines are used, Naiad's performance starts to be affected by large amounts of messages sent over the network, while Husky can effectively trade CPU for network by the use of an effective combiner.

VIII. Building a Good General-Purpose System

Based on the findings and insights obtained from our study, we list a number of important issues that may help researchers and system developers in building a good general-purpose distributed computing system for big data analytics.

Programming interface. Coarse-grained transformations supported in high-level functional programming interfaces make the implementation of algorithms such as WordCount and TeraSort straightforward. However, it is quite difficult to use high-level functional programming interfaces to implement efficient algorithms for many graph analytics and machine learning tasks that require fine-grained data access. Thus, high-level functional programming interfaces do not always make programming easier. Instead, supporting a high-level functional programming interface with a good set of transformation-based operators, while at the same time allowing an object-oriented programming interface to have fine control on data access and message passing, can provide users much more flexibility and lead to the development of more efficient algorithms using general-purpose systems.

Computing model and data abstraction. A suitable computing model and data abstraction are the foundation to a high-performance general-purpose system. Our study shows that Spark and Flink adopt a coarse-grained data abstraction and their computing model is also more suitable for coarse-grained transformations. Thus, for fine-grained data analytics tasks, these systems use an unnatural way to process them (e.g., instead of directly updating an item in a set, they use a join that accesses all items). Thus, a good computing model should consider to support both coarse-grained transformations and fine-grained data access.

Message combining. Message combining can effectively reduce communication cost and hence improve system scalability. Machine-level combiner can save more messages than worker-level combiner, and should be adopted when CPU is considerably faster than the network. However, if the network is fast and/or the number of messages is small, combiner may be better disabled in order to save its overhead.

Data partitioning. All the systems use very simple data partitioning algorithms. More advanced partitioning algorithms (e.g., those for graphs) can be applied to study the tradeoff between the partitioning overhead and the benefits gained by more balanced workload. This can be an important research problem as its solution can improve all existing systems. Dynamic partitioning to overcome imbalanced workload during runtime can also be an interesting problem, but its overhead may be too high.

References

[1] Apache Flink. https://flink.apache.org/.

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

[3] Apache Hive. https://hive.apache.org/.

[4] 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.

[5] Z. Cai, Z. J. Gao, S. Luo, L. L. Perez, Z. Vagena, and C. M. Jermaine. A comparison of platforms for implementing and running very large scale machine learning algorithms. In *SIGMOD*, pages 1371–1382, 2014.

[6] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI*, pages 363–375, 2010.

[7] C. Chih-Chung and L. Chih-Jen. LIBSVM: A library for support vector machines. *TIST*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.

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

[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[10] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[11] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *NIPS*, pages 1223–1231, 2013.

[12] Husky. http://www.husky-project.com/.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[14] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing. On model parallelization and scheduling strategies for distributed machine learning. In *NIPS*, pages 2834–2842, 2014.

[15] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[16] D. W. Margo and M. I. Seltzer. A scalable distributed graph partitioner. *PVLDB*, 8(12):1478–1489, 2015.

[17] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455, 2013.

[18] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[19] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178, 2009.

[20] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan. Clash of the Titans: Mapreduce vs. spark for large scale data analytics. *PVLDB*, 8(13):2110–2121, 2015.

[21] A. J. Smola and S. M. Narayanamurthy. An architecture for parallel topic models. *PVLDB*, 3(1):703–710, 2010.

[22] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *KDD*, pages 1222–1230, 2012.

[23] J. Tan, A. Chin, Z. Z. Hu, Y. Hu, S. Meng, X. Meng, and L. Zhang. DynMR: dynamic mapreduce with reducetask interleaving and maptask backfilling. In *EuroSys*, pages 2:1–2:14, 2014.

[24] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.

[25] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[26] 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.