Towards Hybrid Programming in Big Data

Peng Wang
Institute of Information Engineering, Chinese Academy of Sciences

Hong Jiang
University of Nebraska-Lincoln

Xu Liu
College of William and Mary

Jizhong Han
Institute of Information Engineering, Chinese Academy of Sciences

Abstract

Within the past decade, there have been a number of parallel programming models developed for data-intensive (i.e., big data) applications. Typically, each model has its own strengths in performance or programmability for some kinds of applications but limitations for others. As a result, multiple programming models are often combined in a complimentary manner to exploit their merits and hide their weaknesses. However, existing models can only be loosely coupled due to their isolated runtime systems.

In this paper, we present Transformer, the first system that supports hybrid programming models for data-intensive applications. Transformer has two unique contributions. First, Transformer offers a programming abstraction in a unified runtime system for different programming model implementations, such as Dryad, Spark, Pregel, and PowerGraph. Second, Transformer supports an efficient and transparent data sharing mechanism, which tightly integrates different programming models in a single program. Experimental results on Amazon's EC2 cloud show that Transformer can flexibly and efficiently support hybrid programming models for data-intensive computing.

1 Introduction

Nowadays, data-intensive applications, such as spam detection, web search, and friend recommendation, become critically important in handling an increasing amount of data produced everyday. Such applications usually have multiple computation phases and share intermediate results across different phases. For example, a typical machine-learning application [8] can consist of five phases: data preprocessing (cleaning and normalization), feature extraction, model training (e.g., classification, regression, or ranking), model evaluation, and optimization. Some phases require intensive relational algebraic operations, while others may need graph analytics. An internal phase needs to pass intermediate results to its subsequent phases for further processing.

To support efficient data processing in large-scale data centers, there exist a number of programming models, such as MapReduce [7], Dryad [14], Pregel [16], and Spark [24]. These programming models provide high-level APIs, transparent fault tolerance, and scalable computation. Their runtime systems handle intricate issues in distributed computing, e.g., task scheduling, data communication, resilience, and synchronization.

Programming systems based on existing programming models can be categorized into *Monolithic* ones and *Same-Cluster-Sharing* ones. The *Monolithic* systems provide a single programming model and use a centralized scheduling algorithm for all jobs, e.g., Hadoop [2]. Such systems have limited applicability due to the restriction of using a single programming model. The *Same-Cluster-Sharing* systems, on the other hand, decouple programming models from resource managements, allowing multiple programming models to collaborate on the same cluster. The example resource managements are Mesos [13] and YARN [21]. While this approach provides a method of combining multiple programming models to develop complex applications, it has several disadvantages. First, Same-Cluster-Sharing systems partition code according to different models, which puts extra burdens on software maintenance. Second, they have poor programmability. Programmers need to get familiar with all these models and design hybrid code by continuously switching their coding among different models. Third, multiple models are loosely coupled for collaboration, which can seriously degrade the performance when, for example, they share data via a slow distributed file system.

To address these issues, we propose and develop a hybrid programming system, Transformer, which seamlessly integrates multiple existing models. There are two unique contributions in Transformer.

- Transformer provides a programming abstraction: Unified Data Space (UDS) to easily express and hybridize common high-level programming models.
- Transformer provides an in-band data sharing mechanism to tightly couple multiple programming models for high performance.

To evaluate Transformer, we study two applications running on a 50-node EC2 cluster. Experiments show that Transformer eases the programming of end-to-end data processing workflows. It avoids the complexity of programming an application with multiple separate runtime systems. Instead, programmers utilize a single runtime system only, which improves programmability and code maintenance. Moreover, with the help of efficient in-band data sharing, Transformer achieves performance better than or comparable to the manual combinations of existing systems.

The rest of this paper is organized as follows. Section 2 describes the importance of hybrid programming, which motivates our work. Section 3 describes the design and implementation of Transformer. Section 4 studies two applications to evaluate Transformer. Section 5 reviews related work and distinguishes our approach. Section 6 presents some conclusions and previews the future work.

2 Hybrid Programming

There are many programming models developed by the data-intensive computing community. These models have different capabilities in handling different kinds of computation. For example, distributed dataflow frameworks, such as MapReduce [7], Dryad [14], and Spark [24] are well suited for analyzing tabular data. Graph-parallel models, such as Pregel [16] and PowerGraph [10] are designed for graph analysis. MadLINQ [20] and Presto [22] are efficient for performing matrix computation. However, a real-world application usually consists of multiple computation phases that are not amenable to obtaining high performance within a single programming model. For example, processing on social network graphs can have two phases: page ranking and value sorting. Graph-parallel models can obtain high performance for the page ranking phase because of their strengths in graph analysis. However, graph-parallel models have poor support for the sorting computation. Instead, dataflow models have better performance on the sorting computation, because they can perform data parallelism via map and reduce on large datasets. Thus, neither PowerGraph nor Hadoop efficiently performs this two-phase graph computation. A possible solution is to combine two jobs written by PowerGraph and Hadoop, simply piping the data from one to the other. This approach not only puts heavy burdens on programmers but also leads to performance degradation.

Unifying different programming models into a single system is a promising approach for data analytics. In this paper, we design and implement a general programming system — Transformer. Transformer integrates different programing models, including Dryad [14], Spark [24], Pregel [16], and PowerGraph [10]). With the support of Transformer, one can easily implement applications of multiple phases with high performance.

3 Programming Model in Transformer

To integrate different programming models into one single system, we introduce unified data space (UDS) as a common substrate. UDS offers two capabilities for hybrid programming. First, UDS facilitates *system programmers* to easily express the existing programming models. We will take MapReduce as an example to illustrate the implementation of a programming model on top of UDS in the remaining section. Second, UDS provides a memory-based data sharing mechanism for *application programmers*, which not only tightly integrates different programming models in a single program, but also does not require modification to any program logic.

A Transformer program executes in single program multiple data (SPMD) model. Listing 1 shows an example of a hybrid program, which performs the `pagerank` computation (lines 2-7) that is followed by the `sort` computation (lines 9-19). From the high-level view, the `main` function (lines 21-30) executes these two functions sequentially. The rest of this section elaborates the design and implementation of Transformer.

Listing 1: Hybridizing PowerGraph and Dryad computations in one program supported by Transformer.

```
1   // PowerGraph-based pagerank computation
2   func pagerank(input, output, ...) {
3       // Gather-Apply-Scatter interfaces
4       ...
5       prJob = NewGraphCompute("pagerank")
6       prJob.Execute(input, output,...)
7   }
8   // Dryad-based sort computation
9   func sort(input, output , ...) {
10      // dataflow interfaces
11      ...
12      sortJob = NewDataflow("sort")
13      mapStage = NewStage("map", ...)
14      mapStage.SetInput(input)
15      reduceStage = NewStage("reduce", ...)
16      reduceStage.SetOutput(output)
17      sortJob.Shuffle(mapStage, reduceStage)
18      sortJob.Execute(...)
19  }
20  // hybridization
21  func main() {
22      input = "hdfs://dataset/input"
23      share = "cache://dataset/share"
```

```
24          output = "hdfs://dataset/output"
25          engine = NewEngine()
26          engine.Start()
27          pagerank(input, share, ...)
28          sort(share, output, ...)
29          engine.Stop()
30      }
```

3.1 Unified Data Space

UDS stores all the data of a hybrid program in a four-sphere data space: HDFS, File, Cache, and Pipe. Each sphere structures data into the well-understood file system concepts like tree structure, directories, and files. Typically, a logically distributed dataset is partitioned into a collection of physical data objects. Similar to a file in file system, a data object is stored as a finite-length sequence of bytes, and can only be referenced by its unique name—*uniform resource identifier (URI)*, which exposes the network location (hostname-port pair) the object stored. It is worth noting that the mapping from a distributed dataset to UDS is a responsibility of *system programmers*.

The HDFS sphere usually stores the input and output datasets for a job. The File sphere mainly stores temporary data in local disks (e.g., the map outputs in MapReduce). Similar to the HDFS sphere, the Cache sphere provides a distributed file system abstraction, but it stores data across distributed memory among workers. It reduces I/O operations. That is, if a consumer task is placed on the producer task's physical node, it can directly access the data object stored in the Cache sphere without any data movement. The data object in the Pipe sphere is a special object, which is shared between a pair of tasks. Typically, a producer task writes data to the pipe object and a consumer task reads data from the pipe object. The Pipe sphere leads to pipeline parallelism. It requires the producer and consumer run simultaneously; otherwise the program may result in a deadlock.

UDS provides four spheres for system programmers to balance the tradeoff between performance and reliability. A novel feature of UDS is that data objects residing in any sphere can be retrieved and stored by the storage-agnostic data operations: read and write. By hiding the low-level details (e.g., location lookup, accessing disk, memory management, and network communication), storage agnosticism allows programmers to think about what kind of data to access instead of how to access the data.

Transformer organizes a parallel computation as a collection of independent tasks, which communicate with each other via UDS. A Transformer task is an instance of function with input and output signatures, as well as user-defined code. A signature specifies one or more object URIs. Unlike typical tasks in MapReduce, a Transformer task works in a more restricted manner: it retrieves data objects from UDS, then performs computation, and finally stores data objects to UDS. The normalized behavior of a task brings statelessness and location independence, which offers much more freedom for the task scheduler to place tasks for data locality and fault tolerance. By considering the location of data objects, the task scheduler may place a task into an appropriate worker process with high data locality. When a task fails during its execution on a worker process, the scheduler can retry to launch it on another worker process for fault recovery.

3.2 Hybridization and data sharing

Conceptually, a hybrid job, including multiple computations with different programming models, proceeds as follows: a driver program orchestrates multiple computations (sequentially or concurrently). A model-specific computation is initiated by specifying an input dataset (as a seed) and an output dataset (not yet produced). The master process launches a batch of independent tasks across worker processes, and each task performs the user-defined logic to process the data independently. After completing the process, each task returns its metadata about its output objects (added into the data space) to the master process. The output objects may satisfy the data dependence of another batch of tasks and activate their executions.

In Transformer, data sharing is accomplished by passing the data references (i.e., object URIs) between tasks. Traditionally, a dataset is shared via HDFS by offloading data to external storage systems, which we call *out-of-band*. To reduce the data movement overhead between the worker processes and the service processes (e.g., the data nodes in HDFS), *application programmers* may directly store data among the worker processes using the cache sphere, a data-sharing mechanism that we call *in-band*. Application programmers only need to specify a designated parameter as the cache sphere (see *share* in Lising 1) and do not need to modify any program logic.

3.3 Model Implementation

From a high-level perspective, the parallel computation in a model is expressed by the customized composition of independent tasks and communication patterns. As an illustrative example, Figure 1 shows how the MapReduce model is expressed on top of UDS. The MapReduce model can be expressed by a two-stage dataflow: map and reduce. In the map stage, map tasks (`m0` and `m1`) read data from the HDFS sphere of UDS, and output results to the cache sphere of UDS . The reduce tasks (`r0` and `r1`) follow the same well-defined behavior as map

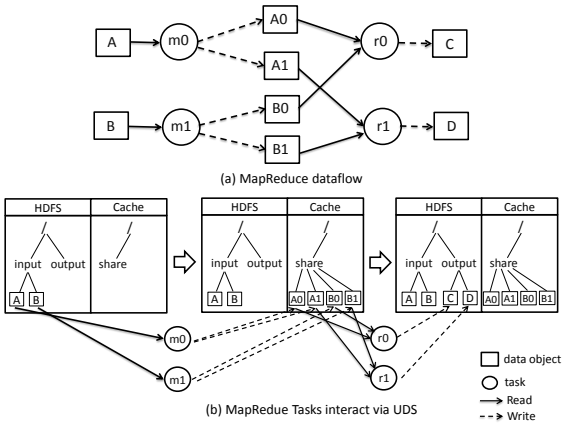
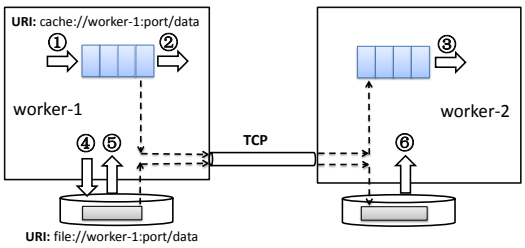


Figure 1: Using UDS to express MapReduce programming model



API examples	Task location	
	worker-1	worker-2
NewWriter("cache://worker-1:port/data")	①	N/A
NewReader("cache://worker-1:port/data")	②	③
NewWriter("file://worker-1:port/data")	④	N/A
NewReader("file://worker-1:port/data")	⑤	⑥

Figure 2: Read/Write operation on different locations (Workflow and buffering in Read/Write).

tasks: read-compute-write. Typically, Hadoop's shuffle operation materializes the intermediate data to the local disks. In Figure 1, the output of map tasks is specified by the cache sphere, requiring no program effort to implement a memory-based shuffle.

3.4 Implementation Sketch of Transformer

The Transformer system is implemented in the Go [1] language. We briefly sketch the implementation of UDS operations.

The Transformer runtime handles the low-level details of the data naming space, such as communication protocol, data transport, and memory management. All the data transferring across nodes is through the TCP socket. In UDS, the HDFS sphere is implemented by wrapping the libhdfs C library using the `cgo` tool in Go, while the other spheres are implemented in the native Go language. Figure 2 shows how the data movement happens between distributed nodes for the cache and file spheres. Two data objects (cache and file) are first generated by some task being executed on worker1 process (① and ④ operations). When a subsequent task is scheduled to execute on worker1 process, it may directly read the two data objects from memory buffer (②) or local disk (⑤). For a task being executed on other locations (e.g., worker2 in Figure 2), reading a data object may incur overhead. For the cache object, the operation ③ will trigger a network communication to copy data from the remote buffer in worker1 to the local buffer in worker2, and then read data from the local memory buffer. For the file object, the operation ⑥ replicates from remote data to local disk, leading to disk and network overhead. Since the data transport happens among workers, the master is not the performance bottleneck. The current Transformer prototype implementation stores a cache data object entirely in the memory buffer, and the system does not ensure that a cache data object fit in the available memory of a single node.

4 Case Studies

We evaluate the performance of Transformer with two case studies. We compare it with Hadoop 1.2.1 and Spark 1.2. The experiments are performed on an Amazon EC2 cluster using 50 m3.xlarge nodes. We measure the end-to-end execution times for all these experiments.

4.1 Machine Learning

This application consists of two phases of data transformation. The first phase leverages the principle component analysis (PCA) algorithm [3] to reduce the feature dimension. The second phase utilizes logistic regression (LR) [9] for classification. The dataset was synthetically generated and contains a series of text records representing <label, feature vector> pairs. Each line begins with a label that is followed by a feature vector with 100 double-precision floats. For comparison, we implement PCA and LR using Hadoop and Spark, respectively. The PCA is implemented by two Hadoop jobs for reusing a third-party matrix library, while LR is implemented by Spark as it is well suited for iterative computation. In Transformer, PCA is developed with a Dryad-like model, and LR is implemented by Dryad-like and Spark-like models, labeled T-Dryad and T-Spark respectively.

Figure 3 compares the overall execution times of PCA and a ten-iteration execution of the logistical regression algorithm using different systems. In the case of HDFS sharing, the workflow consumes 5258 seconds with Hadoop, 618 seconds with Hadoop+Spark, 921 seconds with T-Spark, and 543 seconds with T-Dryad. It

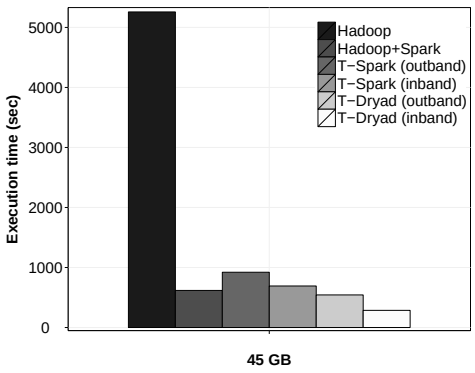


Figure 3: Execution time comparison of PCA-LR

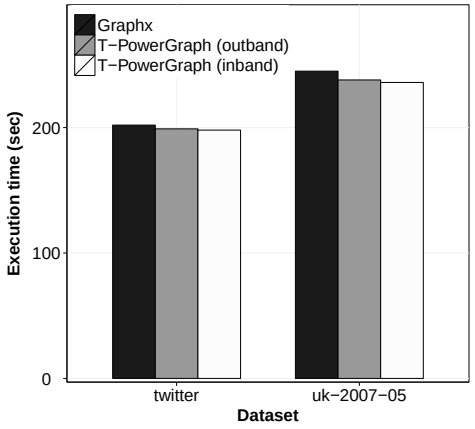


Figure 4: Execution time comparison of Pagerank-Topk

shows that our Spark implementation is slower than the original Spark. T-Spark currently stores the records as data objects in the cache sphere of UDS, requiring type conversion from bytes to the Go objects on each iteration, while Spark avoids this parsing overhead by storing the records in memory directly as Java objects. In both T-Spark and T-Dryad evaluations, the in-band mode shows clearly better performance than that of the out-of-band mode by 25% to 47%, respectively.

4.2 Graph Analysis

We evaluate a graph analysis application to compare Transformer with GraphX [11]. Figure 4 compares the end-to-end execution times of a ten-iteration execution of PageRank followed by TopK on two social-network graphs (twitter [5] and uk-2007 [4]). We see that Transformer shows competing performance with GraphX, a graph computing system using many optimizations (e.g., join elimination, vertex mirroring and delta updates). Note that for the two graph datasets, the in-band sharing scheme provides limited improvement over the out-of-band approach. This happens simply because the intermediate results generated by PageRank (i.e., collection of vertex and its value) are relatively small in volume.

5 Related Works

Hybrid programming models. Many programming models, such as MapReduce [7], Dryad [14], FlumeJava [6], Spark [24], CIEL [18], Naiad [17], DryadLINQ [23], and Pig [19], employ dataflow or its variants for massive data processing. In contrast, Transformer recasts existing programming models into a range of computations on a global data space, and makes it possible for hybridization. GraphX [11] is closest to Transformer in terms of incorporating diverse computations. GraphX supports pre-defined graph parallel and data parallel operations on a common data structure (i.e., collections). In contrast, Transformer supports user-defined computations on the same physical dataset. Thus, Transformer offers much more flexibility in computation composition.

Efficient data sharing. Nectar [12] proposes to cache the intermediate results of common computations to reduce redundant computations. Tachyon [15] is a distributed file system designed for fast and reliable data sharing across jobs. In tachyon, data is stored in distributed memory, and the lost data can be reproduced by lineage information. In contrast, in-band mechanism in Transformer provides an efficient data sharing at the intra-job level instead of the inter-job level.

6 Conclusions and Future Work

This paper presents a hybrid programming system for big data applications, which supports good programmability, competing or better performance, high portability, and high productivity. Our future work consists of three parts. First, we plan to integrate more existing programming models in Transformer. Second, we plan to extend Transformer with transparent fault tolerance for hybrid programming. Third, we will further improve the performance and scalability of hybrid programming in Transformer.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (No. 61303060), 863 Project of China (No. 2012AA01A401), and the Strategic Leading Science and Technology Projects of CAS (No. XDA06030200).

References

[1] Go language. http://golang.org/.

[2] Hadoop. http://hadoop.apache.org/.

[3] Principal component analysis. http://en.wikipedia.org/wiki/Principal_component_analysis.

[4] BOLDI, P., ROSA, M., SANTINI, M., AND VIGNA, S. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International Conference on World Wide Web* (2011), WWW '11, pp. 587–596.

[5] BOLDI, P., AND VIGNA, S. The webgraph framework i: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web* (2004), WWW '04, pp. 595–602.

[6] CHAMBERS, C., RANIWALA, A., PERRY, F., ADAMS, S., HENRY, R. R., BRADSHAW, R., AND WEIZENBAUM, N. Flumejava: Easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2010), PLDI '10, pp. 363–375.

[7] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation* (2004), OSDI '04.

[8] FLACH, P. *Machine Learning: The Art and Science of Algorithms that Make Sense of Data*. Cambridge University Press, 2012.

[9] FREEDMAN, D. A. *Statistical Models: Theory and Practice*. Cambridge University Press, 2009.

[10] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (2012), OSDI'12, pp. 17–30.

[11] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14, pp. 599–613.

[12] GUNDA, P. K., RAVINDRANATH, L., THEKKATH, C. A., YU, Y., AND ZHUANG, L. Nectar: Automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), OSDI'10, pp. 1–8.

[13] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. 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*, NSDI'11, pp. 295–308.

[14] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (2007), EuroSys '07, pp. 59–72.

[15] LI, H., GHODSI, A., ZAHARIA, M., SHENKER, S., AND STOICA, I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), SOCC '14, pp. 1–15.

[16] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (2010), SIGMOD '10, pp. 135–146.

[17] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP '13, pp. 439–455.

[18] MURRAY, D. G., SCHWARZKOPF, M., SMOWTON, C., SMITH, S., MADHAVAPEDDY, A., AND HAND, S. Ciel: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (2011), NSDI'11, pp. 113–126.

[19] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (2008), SIGMOD '08, pp. 1099–1110.

[20] QIAN, Z., CHEN, X., KANG, N., CHEN, M., YU, Y., MOSCIBRODA, T., AND ZHANG, Z. Madlinq: Large-scale distributed matrix computation for the cloud. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), EuroSys '12, pp. 197–210.

[21] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., LOWE, J., SHAH, H., SETH, S., SAHA, B., CURINO, C., O'MALLEY, O., RADIA, S., REED, B., AND BALDESCHWIELER, E. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (2013), SOCC '13.

[22] VENKATARAMAN, S., BODZSAR, E., ROY, I., AUYOUNG, A., AND SCHREIBER, R. S. Presto: Distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys '13, pp. 197–210.

[23] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, U., GUNDA, P. K., AND CURREY, J. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pp. 1–14.

[24] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (2012), NSDI'12.