2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)

EC-Shuffle: Dynamic Erasure Coding Optimization for Efficient and Reliable Shuffle in Spark

Xin Yao, Cho-Li Wang, Mingzhe Zhang
Department of Computer Science
The University of Hong Kong
Hong Kong, China
Email: {xyao, clwang, mzzhang}@cs.hku.hk

***Abstract*—Fault-tolerance capabilities attract increasing attention from existing data processing frameworks, such as Apache Spark. To avoid replaying costly distributed computation, like shuffle, local checkpoint and remote replication are two popular approaches. They incur significant runtime overhead, such as extra storage cost or network traffic. Erasure coding is another emerging technology, which also enables data resilience. It is perceived as capable of replacing the checkpoint and replication mechanisms for its high storage efficiency. However, it suffers heavy network traffic due to distributing data partitions to different locations. In this paper, we propose EC-Shuffle with two encoding schemes and optimize the shuffle-based operations in Spark or MapReduce-like frameworks. Specifically, our encoding schemes concentrate on optimizing the data traffic during the execution of shuffle operations. They only transfer the parity chunks generated via erasure coding, instead of a whole copy of all data chunks. EC-Shuffle also provides a strategy, which can dynamically select the per-shuffle biased encoding scheme according to the number of senders and receivers in each shuffle. Our analyses indicate that this dynamic encoding selection can minimize the total size of parity chunks. The extensive experimental results using BigDataBench with hundreds of mappers and reducers shows this optimization can reduce up to 50% network traffic and achieve up to 38% performance improvement.**

***Keywords*-fault tolerance; distributed computation; erasure coding**

I. Introduction

As the runtime data loss becomes a fairly serious issue in distributed computation, there are various methodologies to support fault tolerance. Logging is one of the widely deployed schemes [1, 2]. It appends all executed operations to a log list at runtime and the system could recompute the operations in this log list to recover the lost data. Some studies suggest persisting the runtime data via two other schemes: (1) local checkpoints: storing the data in the non-volatile storage, like SSD and non-volatile memory [3]; (2) remote replication: duplicating the data to the remote machines [4]. The system can directly recover the lost parts from the backup, instead of replaying the logs. However, both schemes need extra storage space to duplicate the data. Existing cloud storage systems [5, 6] exploit erasure coding [7] since it can effectively reduce storage redundancy while achieving good durability. It splits the data into several data chunks and encodes them to generate the parity chunks. If data loss happens, erasure coding reconstructs the lost data chunks by decoding the available data chunks and parity chunks. The system can recover r lost data chunks with r parity chunks. The encoding and decoding costs are negligible because of increasingly powerful cores. For instance, a single Intel Xeon E5-2650v4 CPU core can encode data at 5.30GB/s [8] for Reed-Solomon(10,4) codes, which is faster than a current high-end NIC with 40Gb/s bandwidth [9]. The heavy network traffic in these schemes raises more concerns.

Spark classifies the transformations based on different dependencies: narrow/wide dependencies [1, 10]. There are four common communication patterns between mappers and reducers: *Map*, *Aggregation*, *Partition*, and *Shuffle*. Map and Aggregation are narrow dependencies while the other two patterns are wide dependencies. It is obvious that some transformations (*Shuffle*) are more expensive than the others (*Map*) because of the network traffic. Spark creates a lineage graph [11] according to these dependency relationships among RDDs (RDD is a data structure in Spark). The system spends much time replaying a large lineage graph to recover a small data partition. The usage of local checkpoint and remote replication can speed up the recovery process by providing the backup of intermediate data, instead of replaying from the beginning. However, these schemes cause significant runtime overhead and extra storage cost. Under the age of "Big Data", these disadvantages become apparent and they shadow the benefits brought by fast recovery. Some researchers [12, 13] use erasure coding to reduce memory cost. Compared with remote replication, the state-of-the-art mechanisms (e.g., FTI [12]) based on erasure coding improve memory efficiency but fail to save data traffic in its encoding process and decoding process. Therefore, under potential network jam, effective mechanisms and techniques are still needed for these serious problems with the growing data size and communication complexity. In this paper, we answer two questions: (1) how to reduce the network traffic of fault-tolerance mechanisms when executing shuffle-like operations, (2) how to minimize the total size of generated parity chunks to optimize the runtime performance.

978-1-7281-0912-1/19/$31.00 ©2019 IEEE
DOI 10.1109/CCGRID.2019.00014

We propose EC-Shuffle, a new fault-tolerance mechanism in Spark framework, to face these challenges. Firstly, EC-Shuffle can recover the lost runtime data via decoding data chunks and parity chunks on the surviving nodes, instead of replaying logs from the beginning. In addition, our mechanism only transfers the generated parity chunks via two encoding schemes, namely, forward-coding (FC) and backward-coding (BC). Compared with previous studies, our schemes can remove the heavy network traffic of data chunks. Furthermore, we observe that these two encoding schemes exhibit different performance behaviors while executing different shuffle operations. It is related to generating the parity chunks, determined by the number of mappers and reducers in each shuffle. To reduce the total size of parity chunks, we design an adaptive selection scheme in EC-Shuffle, which can choose a proper encoding scheme for each shuffle operation at runtime. Our experiments of real-life applications illustrate EC-Shuffle respectively achieves up to 43% and 38% performance improvement (e.g., Sort) than the replication-based and FTI-based shuffle in Spark. Compared with FTI [12], EC-Shuffle achieves 32% and 18% performance improvement for a single shuffle operation and the whole job in PageRank. Our main contributions are:

- **EC-based Fault-tolerance Mechanisms**: Different from exploiting erasure coding in storage systems, shuffle-like operations have frequent data transformation among nodes. With this feature, we propose two optimized encoding schemes, forward-coding and backward-coding, to provide fault tolerance support for these operations. Our schemes hide most data traffic of erasure coding in executing these operations and the overhead is reduced to be generating and transferring the parity chunks. Compared with traditional studies (e.g., FTI [12]), it can save much more bandwidth with the same cost of memory.
- **Dynamic Encoding Selection**: We find there is a further trade-off between forward-coding and backward-coding. We design a dynamic encoding selection, which can determine the better encoding scheme on a per-shuffle operation basis, to minimize the generated parity chunks at runtime according to the numbers of senders and receivers. The prototype of these core functions is implemented in Spark, for supporting the reliability of distributed computation on a large cluster.

II. Motivations

A. *Shuffle Mechanism in Spark*

In Figure 1, we show the procedure of a shuffle operation in Spark. With the wide dependencies, each mapper splits its data into different blocks (4 blocks in the example). Then, it calls *ShuffleWriter* to persist these blocks by storing them to local hard disk. After entering the next stage, each reducer calls *ShuffleReader* to fetch the local blocks (read from disk) and remote blocks (network traffic). Finally, each reducer deserializes these blocks to get a key/value iterator.

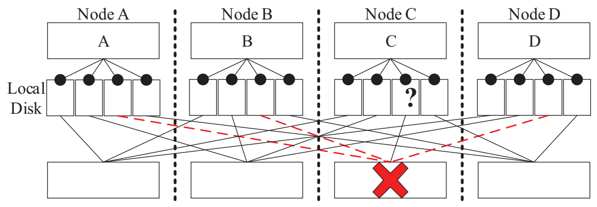


Figure 1: Shuffle process with four mappers and four reducers

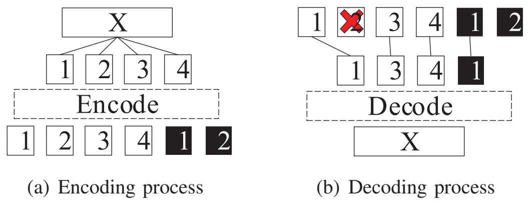


Figure 2: Two processes of erasure coding (k=4, r=2)

When some nodes are unavailable, Spark starts the recovery process and continues to execute the job. We assume Node C crashes (the block with a red cross mark in Figure 1). To resume the local block (the block with a question mark), it needs to replay the operations in the RDD lineage graph and regenerate the data at Node C. As for the remote blocks fetched from other nodes, the system can copy them from the checkpoints on the surviving nodes (the red lines). Therefore, the system only replays logs to recover parts of the lost data (i.e., the local block), instead of regenerating all intermediate data. Although checkpoints avoid re-execution of a whole job, it is still costly when the lineage graph is very large or contains many expensive operations. Spark also provides other schemes. After a shuffle, the user can call the *persist* function. With different storage levels, the system can duplicate the data to different locations and the system can instantly recover the lost data from its replication.

B. *Erasure Coding*

Erasure coding become widely used due to its high memory efficiency [6], such as f4 storage system [5] and HDFS [14, 15]. Facebook [16] has saved multiple petabytes of storage space by employing Reed-Solomon (RS) codes instead of replication in their data warehouse cluster. In the Encoding phase (Figure 2(a)), erasure coding divides the data into k data chunks and uses RS codes to generate r parity chunks. When some nodes are unavailable, erasure coding collects any k of (k+r) chunks and reconstructs the original data (Figure 2(b)). Erasure coding greatly improves memory efficiency by recovering the lost data with the help of parity chunks.

Fault Tolerance Interface (FTI)) [12] is a recent study to use the erasure encoding technique in distributed compu-

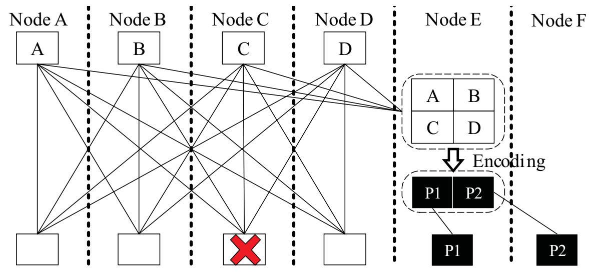


Figure 3: Runtime overhead of FTI (k=4, r=2)

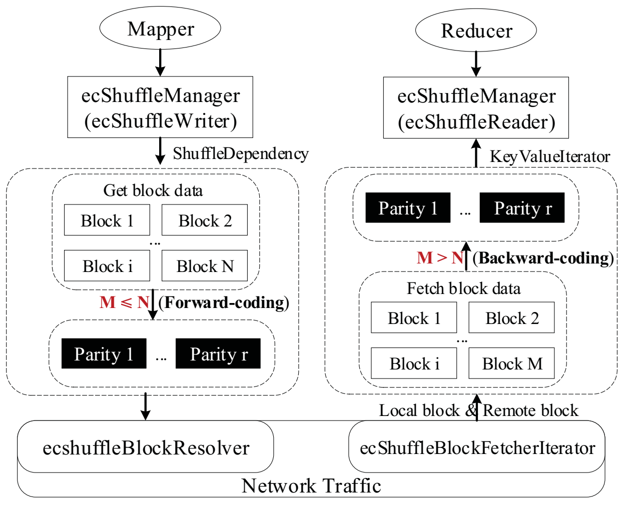


Figure 4: EC-Shuffle in Spark with M mappers and N reducers

tation. FTI [12] partitions the system in groups of k processes (k data chunks). Each group independently generates r parity chunks by encoding the data from k processes. FTI chooses the highest fault tolerance level (i.e., r=k), but the case that k data chunks are simultaneously unavailable is rare. Therefore, we assume $r \ll k$ in our analyses: FTI is memory-efficient (r parity chunks), while the network traffic (the total size of k+r-1 chunks) is heavy. In Figure 3, we show an example of FTI when executing shuffle-based operations. All data chunks (k chunks) are sent to one node (Node E), where the system produces all (r) parity chunks, then this node keeps one parity chunk and sends others (r-1 chunks) to different nodes.

This research proposes EC-Shuffle to provide a non-trivial fault tolerance mechanism for shuffle-like operations. We list the main difference between FTI and EC-Shuffle: (1) FTI generates the parity in each group (fixed k processes), while EC-Shuffle encodes the runtime data in a dynamic manner (decided by the number of mappers and reducers in each shuffle), (2) FTI produces the parity chunks after gathering k data chunks at one node. Forward-coding finishes the encoding process on each sender independently, while backward-coding generates the parity chunks on each receiver after the execution of shuffle operations; both of them only send these parity chunks to different nodes. Therefore, EC-Shuffle can hide the network traffic of k data chunks in FTI via using forward-coding and backward-coding. EC-Shuffle also provides a dynamic selection to choose the proper encoding scheme for each shuffle operation. It can minimize the total size of generated parity chunks.

Both FTI and EC-Shuffle provides similar fault-tolerant functions with remote replication. Their common aim is to avoid replaying too many shuffle operations in Spark's lineage graph. Given the worst case, i.e., an error occurs during encoding/duplicating phase before the shuffle operation is done, it can recover from the last finished shuffle and only redo all narrow transformations between these two shuffles.

III. Design

A. System Overview

The system architecture of EC-Shuffle is shown in Figure 4. We demonstrate it as a new data shuffle mechanism in Spark. It contains two components: (1) two encoding schemes (FC and BC), (2) dynamic encoding selection.

Firstly, EC-Shuffle contains FC and BC based on erasure coding. Erasure coding can split the data into k pieces and generate the parity chunks before sending all chunks to different nodes. According to the dependency of an M-to-N shuffle (i.e., M mappers and N reducers), each mapper partitions the data into N blocks for forward-coding to produce r parity chunks. These parity chunks will be sent to different nodes along with the data chunks in the execution of shuffle. We also claim that erasure coding can encode the data from different nodes to generate the parity chunks. The receiver fetches M blocks (including local blocks and remote blocks) from different mappers. With receiving these blocks after the shuffle, backward-coding can begin the encoding process and only distribute these parity chunks to different nodes. We separately describe these two encoding schemes in Section III-B and Section III-C.

Secondly, we propose a dynamic encoding selection, which chooses an encoding scheme between FC and BC for each shuffle. In our design, the proper encoding scheme is decided by the number of mappers and reducers, which is recorded in ShuffleHandle (including numMaps and ShuffleDependency). In Figure 4, forward-coding is executed if $M \leq N$, while backward-coding is better to left cases.

There are also some other challenges in our design. For example, some applications have large numbers of mappers and reducers (e.g., MovieLensALS has 100 mappers and 200 reducers). This critical issue influences the recovery efficiency of erasure coding because the system needs to collect and decode many chunks to recover only one lost data chunk. We will introduce how our implementation addresses them in Section IV-A.

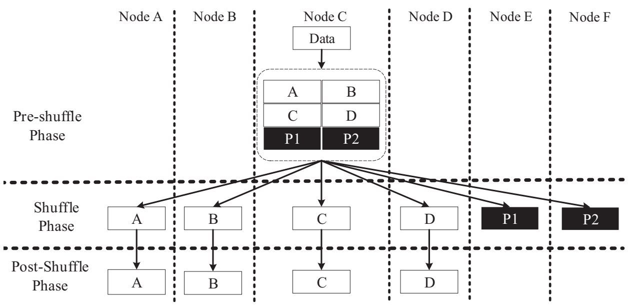


Figure 5: Forward-coding in sending partitions to nodes (*r*=2)

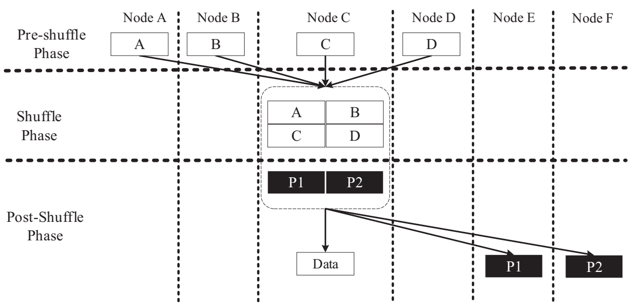


Figure 6: Backward-coding in collecting partitions from nodes (*r*=2)

B. Forward-coding Scheme

1) Runtime of forward-coding:

Phase I - **Pre-Shuffle**: Each sender node (e.g., mapper) splits the data into N pieces. Then it encodes these data chunks to generate r parity chunks. These chunks will be sent to different nodes and some meta information (e.g., the destination of receiving these chunks) is recorded. In Figure 5, the data in Node C is split into 4 pieces (N=4). Then, EC-Shuffle encodes four data chunks to generate two parity chunk (r=2) locally.

Phase II - **Shuffle**: Each node transfers its chunks including the parity part to their destinations. Compared with the original shuffle procedure, the overhead is the network traffic of r additional parity chunks. The number of parity chunks is r, which is smaller than N. The system only transfers this parity chunk, instead of the whole data copy in remote replication or FTI.

Phase III - **Post-Shuffle**: When M is greater than one, each receiver needs to combine the data chunks from senders. After deserializing the combined data chunks, the system continues to handle the next operation.

2) Recovery mechanism:

We consider two cases: (1) when all sender nodes are surviving, we directly copy the lost data from the sender node which has the original data. The overhead is the network traffic of M data chunks. (2) when one sender node is unavailable, we need the available data chunks from the receivers, along with the corresponding parity chunks to reconstruct the lost data. The system collects remote data chunks again from surviving senders (M-1 data chunks). Then, the system selects any N chunks from (N+r-1) chunks stored at receivers, who fetch data or parity from the unavailable sender. Finally, it decodes these chunks to reproduce the lost data chunk. The overhead of recovery process contains the network traffic of (N+M-1) chunks and the negligible decoding cost.

In Figure 5, if the data in Node A or Node B or Node D is lost, Node C re-sends the correspondent data chunk to them. If Node C crashes, which means the sender's data is lost, the system recovers the Data C via decoding any four chunks from three data chunks at Node A, Node B, Node D, and two parity chunks at Node E and Node F.

C. Backward-coding Scheme

1) Runtime of backward-coding:

Phase I - **Pre-Shuffle**: Each sender partitions its data into N chunks and decides the destination of these data chunks based on the shuffle dependency.

Phase II - **Shuffle**: one data chunk from each sender node and totally M data chunks are collected at each receiver node. In Figure 6, Node C gets the data from Node A, B, C, and D.

Phase III - **Post-Shuffle**: After collecting M data chunks from all senders, the receiver encodes them to create r parity chunks. Then, this node transfers these parity chunks to different locations for fault tolerance. When r is small, the overhead of transferring parity chunks is lighter than transferring the whole data in remote replication. In our example (see Figure 6), the parity chunk (r=2) is sent to Node E and F, respectively. At the same time, the receiver combines the data chunks for executing the next operations.

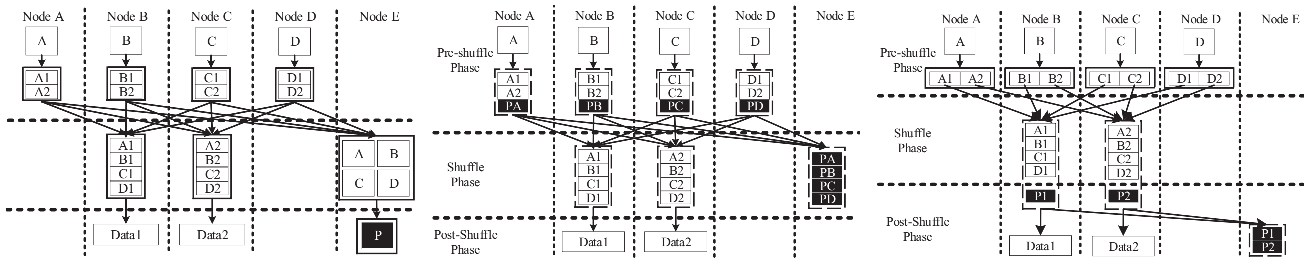
2) Recovery mechanism:

Similar to forward-coding, if all sender nodes are surviving, the system can fetch data partitions from M sender nodes. When one sender node is unavailable, it can still recover $M - 1$ data chunks from other senders. Since these data chunks take part in the encoding process of backward-coding at receiver, the system only chooses 1 of r parity chunks to reproduce the lost data. The recover overhead includes the network traffic of M chunks and the elapsed time of this decoding process.

In our example (see Figure 6), if Node C is unavailable, the system resumes some lost data chunks from Node A, Node B, and Node D. With these three data chunks, the system only reads the parity chunk P1 from Node E (or P2 from Node F) to reproduce the Data C. Compared with forward-coding, this scheme reduces the network traffic in the recovery process.

D. Dynamic Encoding Strategy in EC-Shuffle

An M-to-N shuffle operation has M mappers and N reducers. Each mapper sends its partitions to reducers (see



(a) Fault tolerance interface mechanism (b) Forward-coding mechanism (c) Backward-coding mechanism

Figure 7: Three fault-tolerance mechanisms in a shuffle with four mappers and two receivers case (r=1)

Table I: Overhead comparison among five mechanisms

	RR	**FTI**	**FC**	**BC**	**EC-Shuffle**
Network traffic	$r*M*X$	$(M+r-1)*X$	$r*X*\frac{M}{N}$	$r*X$	$r*X*\min\{\frac{M}{N}, 1\}$
Memory usage	$r*M*X$	$r*X$	$r*X*\frac{M}{N}$	$r*X$	$r*X*\min\{\frac{M}{N}, 1\}$

1. RR = remote replication. FTI = fault tolerance interface. FC = forward-coding. BC = backward-coding.
2. r: The replication factor (in RR) or the number of parity chunks (in other schemes). X: The size of data on each sender. M: The number of nodes on the sender side. N: The number of nodes on the receiver side.

Figure 5), while each reducer collects partitions from senders (see Figure 6). Therefore, both the forward-coding and backward-coding can recover this kind of complex operation. Since the encoding processes of forward-coding and backward-coding happen at different phases, they may bring different overhead even executing the same shuffle operation. In our former discussion, the overhead of forward-coding and backward-coding is directly related to the parity chunks: a smaller total size of the parity chunks brings less overhead. In EC-Shuffle, we summarize the costs in FC and BC (see Table I) and design a dynamic strategy which can minimize the size of parity chunks. In this section, we list some formulas of the generated parity chunks in these two schemes to present this strategy with the relationship of M and N. Without loss of generality, we assume the size of the original data at each sender is X (i.e., the total size of data enters the shuffle is $M*X$) and the partitioned data chunks are equal in size [17].

In the **Pre-Shuffle** phase of forward-coding, it encodes r parity chunks at M nodes (senders) separately, and the data on each sender is split into N parts. We use a formula to calculate the size of parity data generated in forward-coding:

$$Parity_{FC}(M,N,r) = \frac{X}{N}*r*M = \frac{M}{N}*r*X \tag{1}$$

Likewise, backward-coding encodes r parity chunks at each of N nodes (receivers) in **Post-Shuffle** phase. The data chunks on each receiver are collected from M nodes. In a similar approach, we derive the following expression:

$$Parity_{BC}(M,N,r) = \frac{X}{N}*r*N = r*X \tag{2}$$

Base on Equation 1 and Equation 2, we have: (1) When $M < N$: Forward-coding produces smaller size of the parity chunks than backward-coding. EC-Shuffle selects forward-coding as the proper encoding scheme. (2) When $M = N$: Forward-coding and backward-coding generate the parity chunks in the same size at runtime, thus have competitive overheads. (3) When $M > N$: Backward-coding is the better encoding scheme because it is more memory efficient.

In Figure 7, we show an example of executing a shuffle operation with four mappers and two reducers. Figure 7(b) presents the forward-coding process. This process produces four parity chunks (black blocks at Node E), which take up 50% of the whole data size ($4/8 = 50\%$). On the other hand, backward-coding (see Figure 7(c)) generates two parity chunks. It only produces the half size of parity chunks generated in forward-coding. Compared with FTI (see Figure 7(a)), our schemes (EC-Shuffle) gains obvious benefits. Since M>N, EC-Shuffle uses backward-coding to generate 2 parity chunks and transfers them, while FTI generates the same parity chunks but incur much more network traffic (total 8 data chunks, 4 times of EC-Shuffle).

Furthermore, EC-Shuffle brings a higher degree of parallelism than FTI in the encoding process. In FTI, each group executes the encoding process sequentially (P1 and P2 are generated at Node E in Figure 7(a)). EC-Shuffle can improve the parallelism degree of encoding process because each mapper/reducer produces the parity chunks independently. In our example (see Figure 7(c)), P1 is constructed at Node B while the reducer on Node C generates P2 simultaneously.

In general cases, we compare these fault tolerance mechanisms (e.g., remote replication, FTI) in Table I. Equation 3 shows that EC-Shuffle always produces a minimum size of parity chunks (no more than $r*X$ in FTI) and reduces data traffic for any M and N.

$$r*X*min\{\frac{M}{N},1\} \le r*X \le r*M*X \tag{3}$$

This dynamic selection in EC-Shuffle works based on the

Table II: Software specification

Software	Version	Software	Version
OS	Ubuntu 18.04	Intel isa-l	2.19
Kernel	4.15.0-22	Java	1.8.0_191
GCC	7.3.0	Hadoop	2.7.5
Maven	3.5.2	Spark	2.3.0
Yasm	1.3.0.0	BigDataBench	4.0

relationship of *M* and *N*. In many previous systems [1], the number of senders and the number of receivers can be determined before executing distributed operations. For instance, in Spark, *M* and *N* are passed to ShuffleManager component (recorded in ShuffleDependency). In the overview (see Figure 4), we show how Spark decides which scheme is used. When executing operations with wide dependencies, each mapper calls forward-coding scheme if *M* is no larger than *N*, or each reducer exploits backward-coding.

IV. EC-Shuffle in Spark

There are many open source erasure coding libraries, such as Jerasure [18]. We exploit Intelligent Storage Acceleration Library (ISA-L) [8] from Intel to implement this system. ISA-L is used in Hadoop 3.x [14], which contains a collection of optimized storage methods including CRC, RAID, compression and so on. The performance report [8] shows the single core throughput of erasure coding can achieve 12.7 GB/s. We provide a comprehensive performance evaluation of EC-Shuffle based on BigDataBench [19], a Big Data benchmark suite.

A. EC-Shuffle Manager Plugin for Spark

We deploy the pluggable shuffle system in Spark. The users only need to specify the shuffle manager when using EC-Shuffle (similar to SparkRDMA [20]). We also implement an RSCoder via JNI (Java Native Interface). It enables EC-Shuffle to call the ISA-L library to speed up the encoding process on Inter-chips. As discussed in Section III-A, we face some technical issues when we implement EC-Shuffle:

1) Large numbers of mappers and reducers in the shuffle: In some jobs (e.g., MovieLensALS), both *M* and *N* are very large. In EC-Shuffle, the system encodes *N* (or *M*) data chunks and generates the parity chunks. It brings overhead because the system needs to decode a large number of chunks to recover one single lost data chunk. Facebook introduces an improved erasure coding scheme, namely fLRC [21], to settle the repair problem. It encodes a subset of the data chunks (we assume the number of data chunks in this subset is *k*). Once a data chunk from this subset is lost, the system only needs to fetch data chunks and parity chunks of the subset to recover it. Likewise, we use a slicing method to divide the large numbers of data chunks at every mapper/reducer into groups. Each group contains no more than *k* data chunks and encodes its data chunks to generate the parity chunks independently.

2) Data Chunks in Different Sizes: When encoding k data chunks to generate the parity chunks, the system faces another problem: all data chunks should be in the same size. In real applications, those data chunks are usually different in sizes (i.e., load imbalance). FTI solves this issue by dividing these data chunks into 4KB blocks. However, in Spark, this solution brings significant overhead (e.g., metadata) to manage these blocks. We find the data chunk whose size is the largest among k data chunks. Then, we append an array of zero-valued bytes to each data chunk. This scheme formats all data chunks and they can be the same in size before the encoding process. We report how this issue makes a difference between our analyses in Table I and evaluation results.

B. Environment Setup

The testbed, which is used to evaluate BigDataBench [19], contains 64 nodes and each node is equipped with 2 quad-core (Intel Xeon E5540 @ 2.53GHz), 8GB or 16GB DDR3 memory, and 250GB RPM SATA hard disk. These machines connect to 1000Mb/s Ethernet network and the whole Spark cluster runs 200 containers. Table II shows the environment and software we used to implement our system.

C. Big Data Workloads

In Table III, we show a summary of the wide transformations used in popular distributed applications discussed below. Our experiments exploit the datasets from some famous companies (see the BigDataBench handbook [19]).

Search Engine. We run two workloads in this domain: Sort and PageRank. Sort uses the same Wikipedia entries generated by BigDataBench tools with "lda_wiki1w" model. In this dataset, we have 80 files and each file has around 8,000 lines. Sort is based on **sortByKey** to sort the lines in all text files. The dataset of PageRank is the information of Google web graph. In this graph, there are 2,097,152 nodes and 9,928,630 edges. Different from Sort, it executes one **reduceByKey** in each iteration. We set the number of iterations equals 10.

Social Network. KMeans is a widely used clustering algorithm. It is done after some iterations and divides vector points into k groups. Each vector point belongs to its cluster with the minimum mean value. In our experiment, it uses the social network from Facebook, which has 69,000,000 vector points. Each vector point has nine features (nine elements in one vector). Similar to PageRank, the number of **reduceByKey** used in KMeans equals the number of iterations (4 iterations in KMeans). We set the number of centers equals 32.

Machine Learning. Many machine learning studies rely on vector/matrix multiplication [22], such as classifications.

Table III: Target distributed applications in our experiments

Workload	Dataset	Key Operations used in Spark	(# of mappers, # of reducers)[1]
Sort	Wikipedia Entries	sortByKey * 1	(96, 96)
MatrixMultiply	2D Matrix: $Z^{2000*2000}$	reduceByKey * 1	(200, 200)
PageRank	Google Web Graph	reduceByKey * (# of iterations)	(4, 8)
KMeans	Facebook Social Network	reduceByKey * (# of iterations)	(100, 100)
MovieLensALS	Movielens 20m Dataset	{groupByKey, join} * (# of iterations)	(100, 200)
SVMWithSGD	Pascal Large-Scale Challenge	{foldByKey[2]} * (# of iterations)	(54, 6)

[1] We only record the number of mappers and reducers of these key operations used in these workloads.
[2] In each iteration, the number of **foldByKey** is decided by the multi-level treeAggregate.

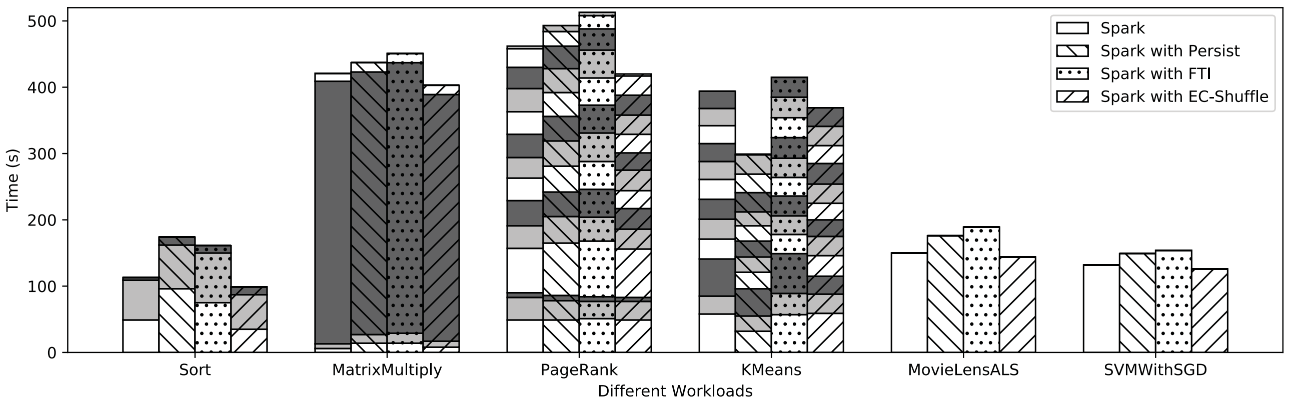


Figure 8: Completion time of multiple stages (different colors) in different workloads without task failures (MovieLensALS and SVMWithSGD have over 100 stages and we do not show every stage of them in this figure).

We evaluate the multiplication of two 2D matrices (the size of each matrix is 2000*2000) and two ML workloads.

MovieLensALS is a recommendation application based on the Alternating Least Squares (ALS) algorithm. It works with 20 million ratings from 138,000 users on 27,000 movies, collected by MovieLens [23]. In the runtime of MovieLensALS, we observe that M is 100 and N is 200 in some shuffle operations. Stochastic gradient descent SGD is another useful algorithm in data mining. It can minimize an objective function within several iterations. SVMWithSGD is to solve a Support Vector Machines (SVM) optimization problem with SGD. We deploy the dataset provided by Pascal Large-Scale Challenge [24] (Alpha version).

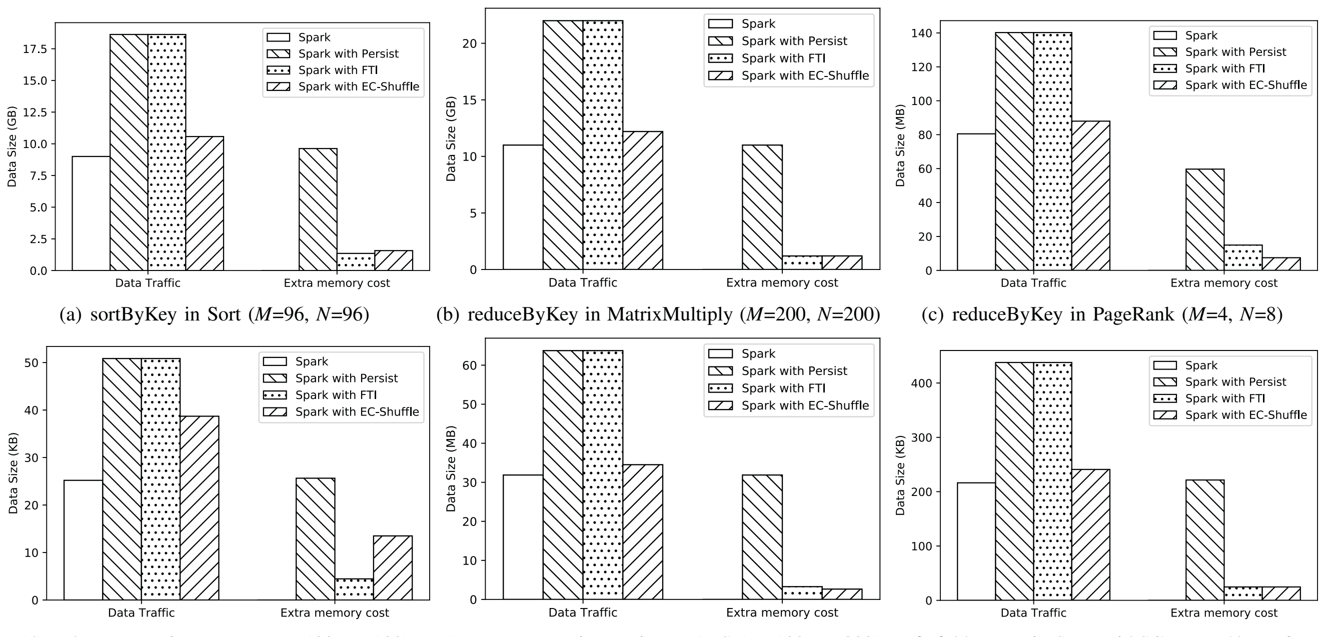
D. Performance Evaluation

In our experiment, we compare four cases: (1) Spark with SortShuffle: Spark persists intermediate blocks of mappers to on-disk files during each shuffle. (2) Spark with Persist (*persist(StorageLevel.MEMORY_ONLY_2)*): Spark replicates each RDD partition on two locations after each shuffle. (3) Spark with FTI: we implement the idea of FTI in Spark via ISA-L, which is discussed at Section II-B. One group has k mappers and k=min{M,10}. (4) Spark with EC-Shuffle: Spark with EC-Shuffle mechanism, while k=min{max{M, N}, 10} (see our discussion in Section IV-A1). We also compare the performance of (3) and (4) under different reliability levels (i.e., with larger r, it can gain higher reliability).

1) Runtime Optimization (r=1):

In Figure 8, we evaluate the completion time of detailed stages. Overall, EC-Shuffle can achieve up to 1.38x speedup than FTI. Compared with saving local checkpoints for each block in native implementation, it also gains competitive performance and sometimes is even faster (e.g., Sort and MatrixMultiply). An application benefits from EC-Shuffle should have the following feature: (1) there is heavy data traffic between mappers and reducers in a shuffle-based operation (e.g., Sort), (2) it contains many shuffle-based operations (e.g., PageRank). In some iterations, the computing process is also very costly, which takes up a large part of the completion time. For example, EC-Shuffle could save 45% network traffic (around 9.8GB shuffled data) than FTI in MatrixMultiply, but only achieves 8.8% performance improvement for this whole job.

In Figure 9, we show both network traffic and extra memory costs (e.g., replications, parity chunks) in our distributed applications. It explains how network traffic reduction actually improves the performance of these real-life applications. In all applications, EC-Shuffle can reduce network traffic, varied from 24% to 50%, compared with traditional schemes (Spark with Persist, Spark with FTI). In most cases (except for KMeans), FTI produces



(a) sortByKey in Sort (M=96, N=96) (b) reduceByKey in MatrixMultiply (M=200, N=200) (c) reduceByKey in PageRank (M=4, N=8)

(d) reduceByKey in KMeans (M=100, N=100) (e) groupByKey in MovieLensALS (M=100, N=200) (f) foldByKey in SVMWithSGD (M=54, N=6)

Figure 9: Network traffic and extra memory cost of the shuffle operations in different workloads

a larger size of parity chunks than EC-Shuffle, e.g., 50% and 38% parity reduction in PageRank and MatrixMultiply, respectively. However, there are also some limitations in our implementation: (1) Erasure coding performs not well on encoding data chunks in small size: although both KMeans and SVMWithSGD also have large numbers of iteration, they only have a few shuffled data in each iteration, thus they do not have obvious speedup, (2) Load balance: whether the workloads are balanced or not will cause difference memory costs. FTI has similar problems (see Section IV-A2).

We make several conclusions from Figure 8 and Figure 9: (1) with EC-Shuffle, the network-intensive workloads (i.e., more shuffle-based operations) has more performance improvement than the computing-intensive workloads, (2) if the workload is load balanced, the total size of parity chunks in EC-Shuffle is no more than FTI; if the workload is load imbalanced, sometimes EC-Shuffle still saves extra memory space than FTI via using dynamic encoding strategy (e.g., M<N), (3) EC-Shuffle outperforms FTI (or replications) by saving 24-50 percent data traffic and it also avoids writing data to hard disks when checkpointing in native Spark.

2) Different reliability levels:

Unlike remote replication, increasing r in FTI only transfers additional parity data and the computational overhead of these parity chunks is light (i.e., it causes milliseconds-level overhead to compute more parities in FTI). We compare EC-Shuffle with FTI for different reliability levels (i.e., different values of r). With growing r, EC-Shuffle and FTI produce competitive parity data because M equals N. However, EC-Shuffle can be 4.8%-6.4% faster than FTI because the data chunks transfer still dominates the extra network traffic.

3) Recovery Overhead:

We randomly kill a container process, which is running a task at the data node, to cause failures. In Figure 11, it shows the normal stages and retries some stages in PageRank. Since Spark needs to replay all iterations (11 retry stages in our experiments) to recover the lost partition, it needs 1.32x the completion time of the same job without failures. Different from replaying all recorded logs, the other schemes can instantly recover the lost data via copying from replication or decoding the available data chunks and parity chunks. Although the recovery process of EC-Shuffle is 20% slower than replication, the total completion time of our scheme still outperforms replication by 17%. We also compare network traffic during the recovery process. Since computing a shuffle operation leads to extra data traffic, Spark replays all stages and incurs much significant network overhead (see Figure 12). Spark with Persist only needs to resume the lost data from replications, which has minimal data traffic.

E. Dynamic Encoding Strategy in EC-Shuffle (r=1)

In our experiment, we change the numbers of mappers and reducers (24 groups) in Sort to show the parity chunks reduction via deploying our dynamic encoding strategy. The dataset of this experiment is a large file with more than 80000 lines, generated by BigDataBench tools with "lda_wiki1w" model. We present the ratio ($1 : \frac{FC}{FTI} : \frac{BC}{FTI}$) of the generated parity data size among these schemes. This

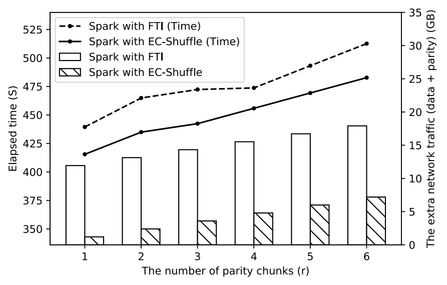


Figure 10: Time span and extra network traffic with different values of *r* in MatrixMultiply.

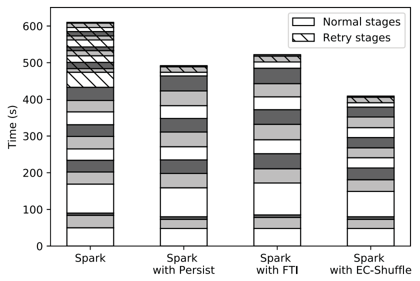


Figure 11: Completion time of multiple stages in PageRank with task failure.

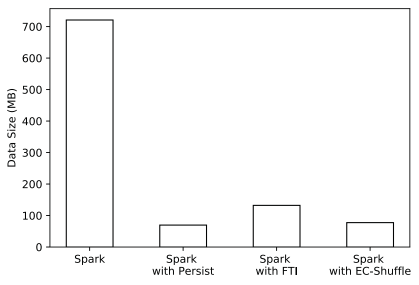


Figure 12: Network traffic in the recovery process of PageRank.

Table IV: The ratio of the generated parity data size (*r*=1) in Sort with different numbers of mappers (*M*) and reducers (*N*)

FTI : FC : BC \ **N** / **M**	**2**	**4**	**8**	**16**	**32**	**64**
2	1:1.008:1.000	1:0.542:1.000	1:0.290:1.000	1:0.294:1.000	1:0.298:1.002	1:**0.199**:1.007
4	1:2.237:1.006	1:1.077:1.012	1:0.593:1.011	1:0.608:1.024	1:0.676:1.036	1:0.551:1.047
8	1:4.445:1.014	1:2.466:1.030	1:1.074:1.055	1:1.139:1.078	1:1.263:1.022	1:1.134:1.078
16	1:4.377:1.022	1:2.721:1.039	1:1.263:1.049	1:1.250:1.076	1:1.330:1.114	1:1.259:1.161
32	1:4.453:1.025	1:2.150:1.046	1:1.187:1.067	1:1.233:1.098	1:1.393:1.145	1:1.289:1.210

experiment indicates EC-Shuffle, using dynamic encoding strategy, can reduce up to 80% parity data than FTI.

Table. IV shows the comparison between EC-Shuffle and FTI: (1) FTI and backward-coding produce the parity data in the same size ($r*X$), (2) when *M*<*N*, forward-coding ($\frac{M}{N}*r*X$) produces fewer parity data than FTI and backward-coding. The slicing method sometimes limits the group size (e.g., 10 in our experiment), which provides efficient recovery at the cost of more memory space.

1) When M is less than N: Forward-coding can reduce up to 80% parity data than FTI and backward-coding (*M*=2 and *N*=64). In this case, the group size k in forward-coding is $min\{N,10\}$ while k in FTI and backward-coding are $min\{M,10\}$. Therefore, k in forward-coding is no less than those in FTI and backward-coding, which explains why forward-coding generates fewer parity data. Some other cases, like *M*=8 and *N*=16, forward-coding performs worse than FTI and backward-coding because each mapper divide 16 data chunks into two groups (10 + 6) and each group generates *r* parity chunks based on its data chunks.

2) When M equals N: forward-coding, backward-coding, and FTI should produce the parity chunks in the same size. To encode these blocks, we format these data chunks (in Section IV-A2) into the same length. FTI and EC-Shuffle may generate parity chunks in larger sizes than our analyses because of the load imbalance problem (i.e., data chunks/-partitions are usually not in the same length).

3) When M is greater than N: In this case, forward-coding usually generates more parity data ($\frac{M}{N}*r*X>r*X$) and backward-coding performs more effectively. The result is similar to our analysis in Section III-D. *M* (2, 4, 8) is smaller than 10, each receiver in backward-coding only produces one parity; while *M* is larger than 10, each receiver has multiple groups and each group generates *r* parity.

V. Related Work

A. General Fault Tolerance Techniques

Lineage. Tachyon [25], which is currently named Alluxio, utilized the lineage mechanism [1, 2] to provide fault tolerance. The system recorded the operations applied on an object in the runtime as the history trace. Once the object is lost, the system could backtrack these historical records to find its parent object. Then, the system replayed all operations on this parent object to recover the lost object. The lineage-based recovery mechanism in Spark [1] was similar to the recovery mechanism used within a computation (job) in MapReduce [26] and Dryad [27], which tracked dependencies among a Directed Acyclic Graph (DAG) of tasks. To replay an operation with wide dependencies, the system needs to collect the data partitions from several nodes. The lineage mechanism could not guarantee the required data still exists. In the worst case, the system replayed all operations on each node. Therefore, after each shuffle, Spark applied local checkpoint to store the immediate shuffle outputs.

Checkpoint. The checkpoint technique [28, 29] can store runtime data to disk and resume them when the system needs them for recovery. The system could easily acquire the data persisted at surviving nodes and only replayed logs to recover the data stored on the failed node. MPICH-V [29] proposed the uncoordinated checkpointing technique, which meant each node or process can take its own checkpoint when it is most convenient. On the other hand, LAM/MPI [28] pointed out several disadvantages in uncoordinated checkpointing, including the domino effect [30].

In the coordinated checkpointing technique, each process always restarted from its most recent checkpoint and get rid of the domino effect. Both LAM/MPI and Spark strongly supported coordinated checkpointing, which simplified recovery from failure and minimized storage overhead.

Replication. Remote replication [31] was a widely used scheme in the distributed systems. Instead of storing the data in local non-volatile storage, it copies the data to other nodes. With the mirror data, the system can instantly recover the lost data. However, when the system synchronizes the replication frequently, it causes much runtime overhead, e.g., network traffic. CORFU [32] combined this technique with the logging mechanism to optimize the performance.

B. Erasure Coding Techniques

FTI. FTI [12] was the most related study, which exploited erasure coding technology in distributed computation. It gathered data blocks from *k* processes and generated *r* parity chunks. The static coding method sometimes only achieves sub-optimization of saving the storage space. Furthermore, it was still costly to transfer all data blocks to one node for encoding parity chunks. EC-Shuffle focuses on addressing these serious concerns of FTI (discussed in Section II-B).

Many optimized erasure coding schemes, such as mLRC [33] from Window Azure and fLRC [21] from Facebook, aimed at reducing the network traffic in the recovery via encoding a subset of the data chunks. Once losing a data chunk, the system primarily finds the subset to which it belongs. Then, it decodes the generated parity chunk and the other data chunks in this subset, instead of all data chunks, to reconstruct the lost data chunk. Thus, the network traffic in their recovery processes is reduced. They focus on in-memory storage systems with timely recovery but suffer the network traffic issues at runtime. Carousel codes [34] is another improved maximum distance separable (MDS) coding scheme, which can improve the data parallelism of MapReduce job by embedding the original data into more blocks and running more map tasks.

EC-Shuffle avoids data chunks transfer, does not concern itself with which erasure coding techniques are used. With a better coding scheme, EC-Shuffle may speed up the encoding/decoding process (e.g., XOR codes [35]) or the recovery process (e.g., slicing methods in our implementation).

VI. Conclusion

With the rapid growth of shuffled data, it is necessary to explore new methodology that can provide fault tolerance guarantee with light-weight network traffic. This paper proposes EC-Shuffle, exploiting erasure coding as a fault tolerance mechanism for high-performance and reliable distributed computation. In runtime, EC-Shuffle presents an efficient strategy to dynamically choose the proper encoding scheme between forward-coding and backward-coding. During the recovery process, EC-Shuffle can instantly recover the lost data via the decoding process, instead of replaying the large lineage graph. Some key findings are listed below:

- EC-Shuffle only transfers the generated parity chunks to provide fault-tolerance. Compared with collecting all data chunks in FTI, it reduces the network traffic.
- EC-Shuffle evaluates BigDataBench with hundreds of mappers and reducers and it can reduce up to 50% network traffic and achieve up to 38% performance improvement than FTI.
- Our analyses, including formulas and tables, indicate EC-Shuffle with dynamic encoding strategies can generate the minimum size of parity chunks than existing state-of-the-art mechanism. It outperforms FTI by reducing up to 80% of the total size of the parity chunks.

There are still some limitations in this work: (1) the extra costs of aligning the data chunks in different sizes, (2) erasure coding performs not efficient for encoding small data chunks. In the future, it can design a better erasure coding scheme, which can encode the data chunks in different sizes without appending an array of zero-valued bytes. Some other works, such as Riffle [36], introduce how to merge the fragmented intermediate shuffle outputs into large block files. It is also a challenge to improve EC-Shuffle with the adaptive coding in HACFS [15], which can optimize the uncoordinated checkpointing technique in streaming computing frameworks (e.g., Flink).

Acknowledgment

This research was supported by Hong Kong RGC Grant 106160098. We also thank three anonymous reviewers for their unique insights and comments, which remind us of several limitations about this work.

References

[1] 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 12*. USENIX, 2012, pp. 15–28.

[2] P. K. Gunda, L. Ravindranath, C. Thekkath, Y. Yu, and L. Zhuang, "Nectar: Automatic management of data and computation in datacenters," in *(OSDI)*, October 2010.

[3] M. Zhang, K. T. Lam, X. Yao, and C.-L. Wang, "Simpo: A scalable in-memory persistent object framework using nvram for reliable big data computing," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 1, pp. 7:1–7:28, Mar. 2018. [Online]. Available: http://doi.acm.org/10.1145/3167972

[4] P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan, "Replication-based fault-tolerance for large-scale graph processing," in *Proc. DSN*, 2014, pp. 562–573.

[5] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar, "f4: Facebook's warm BLOB storage system," in *Proc. OSDI*, 2014, pp. 383–398.

[6] O. Khan, J. P. Randal Burns, and C. H. William Pierce, "Rethinking erasure codes for cloud file systems: Minimizing i/o for recovery and degraded reads," in *fast 12)*, 2012.

[7] H. Weatherspoon and J. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Revised*

Papers from the First International Workshop on Peer-to-Peer Systems, ser. IPTPS '01. London, UK, UK: Springer-Verlag, 2002, pp. 328–338. [Online]. Available: http://dl.acm.org/citation.cfm?id=646334.687814

[8] Intel, "Intelligent storage acceleration library with its relevant documents," https://github.com/01org/isa-l.

[9] H. Zhang, M. Dong, and H. Chen, "Efficient and available in-memory kv-store with hybrid erasure coding and replication," in *Proc. (FAST 16)*, 2016, pp. 167–180.

[10] Rohit, "Wide vs narrow dependencies," https://github.com/rohgar/scala-spark-4/wiki/Wide-vs-Narrow-Dependencies.

[11] jaceklaskowski., "Rdd lineage — logical execution plan," https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-rdd-lineage.html.

[12] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "Fti: High performance fault tolerance interface for hybrid systems," in *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2011, pp. 1–12.

[13] J. Cao, K. Arya, R. Garg, S. Matott, D. K. Panda, H. Subramoni, J. Vienne, and G. Cooperman, "System-level scalable checkpoint-restart for petascale computing," in *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, Dec 2016, pp. 932–941.

[14] A. Hadoop, "Hdfs erasure coding," https://hadoop.apache.org/docs/r3.2.0/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html.

[15] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A tale of two erasure codes in HDFS," in *(FAST 15)*, Santa Clara, CA, 2015, pp. 213–226.

[16] D. Borthakur, R. Schmidt, R. Vadali, S. Chen, and P. Kling, "Hdfs raid," https://www.slideshare.net/ydn/hdfs-raid-facebook, Fackbook, Tech. Rep., 2010.

[17] S. R. Ramakrishnan, G. Swart, and A. Urmanov, "Balancing reducer skew in mapreduce workloads using progressive sampling," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12. New York, NY, USA: ACM, 2012, pp. 16:1–16:14. [Online]. Available: http://doi.acm.org/10.1145/2391229.2391245

[18] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in c/c++ facilitating erasure coding for storage applications," Tech. Rep., 2007.

[19] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "Bigdatabench: A big data benchmark suite from internet services," in *2014 IEEE 20th HPCA*, Feb 2014, pp. 488–499.

[20] Mellanox, "Rdma accelerated, high-performance, scalable and efficient shufflemanager plugin for apache spark," https://github.com/Mellanox/SparkRDMA.

[21] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "Xoring elephants: Novel erasure codes for big data," *Proceedings of the VLDB Endowment (to appear)*, 2013.

[22] Fastai, "Linear algebra for deep learning," http://wiki.fast.ai/index.php/Linear_Algebra_for_Deep_Learning.

[23] MovieLens, "Movielens 20m dataset," http://files.grouplens.org/datasets/movielens/ml-20m-README.html.

[24] "Pascal large-scale challenge, alpha," ftp://largescale.ml.tu-berlin.de/largescale/.

[25] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proc. SOCC*, ser. SOCC '14, 2014, pp. 6:1–6:15.

[26] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proc. 6th OSDI - Volume 6*, ser. OSDI'04, 2004, pp. 10–10.

[27] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc.2007 Eurosys*, March 2007.

[28] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, and A. Lumsdaine, "The lam/mpi checkpoint/restart framework: System-initiated checkpointing," *SAGE Journals, Vol 19, Issue 4, page(s): 479-493*, 2005.

[29] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, "Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes," in *Supercomputing, ACM/IEEE 2002 Conference*, Nov 2002, pp. 29–29.

[30] B. Randell, "System structure for software fault tolerance," in *Proceedings of the International Conference on Reliable Software*. New York, NY, USA: ACM, 1975, pp. 437–449. [Online]. Available: http://doi.acm.org/10.1145/800027.808467

[31] A. K. Pandey, A. Kumar, N. Malviya, and B. Rajendran, "A survey of storage remote replication software," in *2014 3rd International Conference on Eco-friendly Computing and Communication Systems*, Dec 2014, pp. 45–50.

[32] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber, "Corfu: A distributed shared log," *ACM Trans. Comput. Syst.*, vol. 31, no. 4, pp. 10:1–10:24, Dec. 2013.

[33] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *Proc. (USENIX ATC 12)*. USENIX, 2012, pp. 15–26.

[34] J. Li and B. Li, "On data parallelism of erasure coding in distributed storage systems," *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 45–56, 2017.

[35] J. Luo, L. Xu, and J. S. Plank, "An efficient xor-scheduling algorithm for erasure codes encoding," in *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, June 2009, pp. 504–513.

[36] H. Zhang, B. Cho, E. Seyfe, A. Ching, and M. J. Freedman, "Riffle: Optimized shuffle service for large-scale data analytics," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: ACM, 2018, pp. 43:1–43:15. [Online]. Available: http://doi.acm.org/10.1145/3190508.3190534