Exoshuffle: An Extensible Shuffle Architecture

Frank Sifei Luan
UC Berkeley
Berkeley, CA, USA

Stephanie Wang
UC Berkeley and Anyscale
Berkeley, CA, USA

Samyukta Yagati
UC Berkeley
Berkeley, CA, USA

Sean Kim
UC Berkeley
Berkeley, CA, USA

Kenneth Lien
UC Berkeley
Berkeley, CA, USA

Isaac Ong
UC Berkeley
Berkeley, CA, USA

Tony Hong
UC Berkeley
Berkeley, CA, USA

SangBin Cho
Anyscale
San Francisco, CA, USA

Eric Liang
Anyscale
San Francisco, CA, USA

Ion Stoica
UC Berkeley
Berkeley, CA, USA

Abstract

Shuffle is one of the most expensive communication primitives in distributed data processing and is difficult to scale. Prior work addresses the scalability challenges of shuffle by building monolithic shuffle systems. These systems are costly to develop, and they are tightly integrated with batch processing frameworks that offer only high-level APIs such as SQL. New applications, such as ML training, require more flexibility and finer-grained interoperability with shuffle. They are often unable to leverage existing shuffle optimizations.

We propose an extensible shuffle architecture. We present Exoshuffle, a library for distributed shuffle that offers competitive performance and scalability as well as greater flexibility than monolithic shuffle systems. We design an architecture that decouples the shuffle control plane from the data plane without sacrificing performance. We build Exoshuffle on Ray, a distributed futures system for data and ML applications, and demonstrate that we can: (1) rewrite previous shuffle optimizations as application-level libraries with an order of magnitude less code, (2) achieve shuffle performance and scalability competitive with monolithic shuffle systems, and break the CloudSort record as the world's most cost-efficient sorting system, and (3) enable new applications such as ML training to easily leverage scalable shuffle.

CCS Concepts

• **Computing methodologies** → *Distributed computing methodologies*; • **Information systems** → *Parallel and distributed DBMSs.*

SIGCOMM '23, 2023, New York City, NY

ACM ISBN 979-8-4007-0236-5/23/09…$15.00
https://doi.org/10.1145/3603269.3604848

Keywords

Shuffle, MapReduce, distributed computing, extensibility

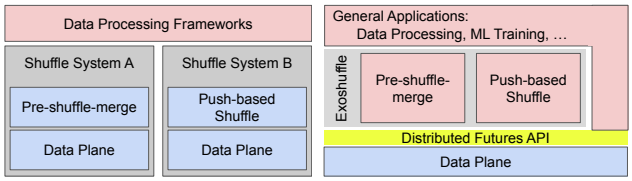
ACM Reference Format:
Frank Sifei Luan, Stephanie Wang, Samyukta Yagati, Sean Kim, Kenneth Lien, Isaac Ong, Tony Hong, SangBin Cho, Eric Liang, and Ion Stoica. 2023. Exoshuffle: An Extensible Shuffle Architecture. In *Proceedings of SIGCOMM (SIGCOMM '23).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3603269.3604848

1 Introduction

Shuffle is a fundamental operation in distributed data processing systems. It refers to the all-to-all data transfer from mappers to reducers in a MapReduce-like system [10]. Shuffle is one of the most expensive communication primitives in these systems and is difficult to scale. Scaling shuffle requires efficiently and reliably moving a large number of small blocks from each mapper to each reducer across memory, disk, and network. It requires both high I/O efficiency, and robustness to failures and data skew. Furthermore, as the data size increases, the number of shuffle blocks grows quadratically, making shuffle the most costly operation in some workloads.

The difficulty of scaling shuffle has inspired many solutions from both the industry and research community. These shuffle implementations improve the performance and reliability of large-scale shuffle by optimizing I/O in different storage environments, such as HDD, SSD and disaggregated storage [3, 35, 41, 56]. Since performance at scale is a priority, these prior solutions are built as monolithic shuffle systems from scratch using low-level system APIs. However, these systems are costly to develop and integrate. For example, cloud providers each have to build proprietary services to support shuffle on their own storage services [1, 3, 42]. Magnet, a push-based shuffle system for Spark [41], took 19 months between publication and open-source release in the Spark project [12] because it required significant changes to system internals [40].

Furthermore, existing shuffle systems only work with batch processing frameworks, which offer high-level abstractions such as SQL or dataframe APIs. Most are synchronous in nature: the results are available only after the entire shuffle operation completes. This



(a) Monolithic shuffle systems. **(b)** Exoshuffle.

Figure 1: Exoshuffle builds on an extensible architecture. Shuffle as a library is easier to develop and more flexible to integrate with applications. The data plane ensures performance and reliability.

poses challenges for applications that require *fine-grained* integration with the shuffle operation to improve their performance by processing data as it is being shuffled, i.e., pipeline data processing with the shuffle operation. For example, ML training often requires repeatedly shuffling the training dataset between epochs to improve learning quality [23, 24]. Doing this efficiently requires fine-grained pipelining between shuffle and training: ML trainers should consume partial shuffle outputs as soon as they become ready. Today's ML developers are faced with two undesirable choices: (1) they either rebuild shuffle from scratch, once again dealing with the performance challenges of large-scale shuffle, or (2) interface with existing shuffle systems through the synchronous APIs: the shuffle results can only be consumed after all partitions are materialized, leaving pipelining opportunities on the table.

To simplify the development of new shuffle optimizations targeting different environments, and to provide fine-grained pipelining for new applications, we propose an *extensible* architecture for distributed shuffle that enables flexible, efficient, and scalable implementations. Unlike previous solutions built as monolithic systems (Fig 1a), we propose building distributed shuffle as a library (Fig 1b). Such an architecture allows: (1) shuffle builders to easily develop and integrate new shuffle designs for new environments, and (2) a broader set of applications to leverage scalable shuffle in a more flexible manner.

How can we implement shuffle at the application level (as a library) while providing high performance? To answer this question, we first identify the optimizations in past shuffle systems that are key to performance and reliability. (1) Coordination: managing the timing and placement of mapper and reducer tasks, and implementing optimizations such as merging intermediate shuffle blocks. (2) Efficient data transfer: pipelining I/O with computation to maximize throughput, and spilling data to disk to accommodate larger-than-memory datasets. (3) Fault tolerance: guaranteeing data is reliably transferred to reducers via retries or replication.

Our key observation is that we can split these optimizations between a control and a data plane. Optimizations for coordinating shuffle are implemented by the *control plane* at the application layer, while the *data plane* provides efficient data transfer and fault tolerance at the system layer. This enables developers to easily implement a variety of shuffle solutions at the application layer, while having the underlying system handle efficient data transfer and fault tolerance.

The next question is what interface should the data plane provide to the application. Our answer is *distributed futures*, an extension of RPC that allows referencing data objects in distributed memory. It allows the caller of a remote task to pass objects *by reference*, regardless of their physical locations, thus decoupling remote task invocations from physical data transfers, the latter implemented by the data plane. Distributed futures can also be passed before the data object is created, allowing the system to parallelize remote calls and pipeline data transfer with task execution. We show that this abstraction can express a variety of shuffle algorithms, including dynamic strategies to handle data skew and stragglers (§3).

Although many distributed futures implementations exist, none of the these systems have been able to match the scale and performance of a monolithic shuffle system. CIEL [28] is the first to show MapReduce programs can be implemented using distributed futures, but it lacks an in-memory object store which is crucial for efficient pipelining and data transfers. Dask [38], another distributed futures-based dataframe system, supports in-memory objects but cannot scale beyond hundreds of GBs (§5.4.1). Previous versions of Ray [26] support shuffle within the capacity of its distributed shared memory object store, but lack disk spilling mechanisms and therefore do not support out-of-core processing.

In this work, we extend Ray with the necessary features to support large-scale shuffle (§4). These include: (1) locality scheduling primitives to enable colocating tasks to better exploit shuffle data locality; (2) a full distributed memory hierarchy with disk spilling and recovery; (3) asynchronous object fetching to pipeline task execution with disk and network I/O. We present Exoshuffle, a flexible and scalable library for distributed shuffle built on top of Ray. We demonstrate the advantages of this extensible shuffle architecture by showing that (§5):

- A variety of previous shuffle optimizations can be written as distributed futures programs in Exoshuffle, with an order of magnitude less code.
- The Exoshuffle implementations of these shuffle optimizations match or exceed the performance of their monolithic counterparts.
- Exoshuffle can scale to 100 TB, outperforming Spark and Magnet by 1.8×, and breaking the CloudSort record as the world's most cost-efficient sorting system.
- Exoshuffle can easily integrate with a diverse set of applications such as distributed ML training, improving end-to-end training throughput by 2.4×.

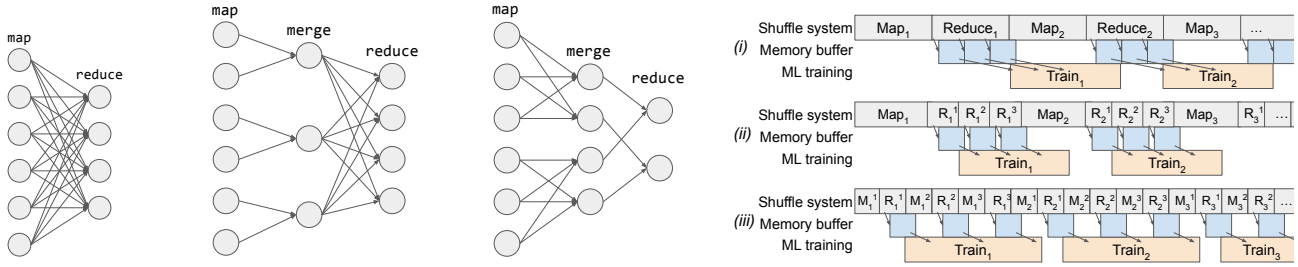
2 Motivations

In this section, we overview two lines of previous work in building shuffle systems to illustrate the challenges in simultaneously achieving shuffle scalability and flexibility.

2.1 Shuffle Systems

Storage target	Shuffle systems
Hard disk	Sailfish [35], Riffle [56], Magnet [41]
SSD	Zeus [4]
Cloud storage	Alibaba E-MapReduce Shuffle [1], AWS Glue Shuffle [3], Google Cloud Dataflow Shuffle [42]

Table 1: Different shuffle systems are built to optimize shuffle for deployment in different storage environments.



(a) "Simple" shuffle [10]. **(b)** Pre-shuffle merge [56]. **(c)** Push-based shuffle [5, 41, 45]. **(d)** ML training pipeline.

Figure 2: Shuffle algorithms for various applications. Exoshuffle uses distributed futures to execute these DAGs.

In a MapReduce operation with M map tasks and R reduce tasks, shuffle creates $M \times R$ intermediate blocks. Each of these blocks must be moved across memory, disk, and network. As the number of tasks grow, the number of blocks increases and the block size decreases both quadratically. At terabyte scale, this can result in hundreds of millions of very small blocks. This creates great challenges for I/O efficiency, especially for hard drives with low IOPS limits. Many shuffle systems have been built to optimize I/O efficiency in different storage environments. Table 1 shows an incomplete list of these systems, grouped by their target storage environments.

Previous I/O optimizations fall under two general categories: (1) reducing the number of small and random I/O accesses by *merging* intermediate blocks into larger ones at various stages [5, 41, 56] (Figures 2b and 2c), and (2) using *pipelining* to overlap I/O with execution [18, 41]. For example, *push-based shuffle* [17, 45] involves pushing intermediate outputs directly from the mappers to the reducers, allowing network and disk I/O to be overlapped with map execution, and optionally merging results on the reducer (Figure 2c) to improve disk write efficiency [5, 41].

While these solutions can improve throughput, they also come with high development cost. Each new operation, such as reduce-side merge, requires building additional protocols for managing block transfers. However, although the ideas may be system-agnostic, the physical artifacts are often tightly integrated with proprietary storage systems, making them difficult to port to open-source frameworks. For example, many cloud providers build proprietary shuffle services to work with their own disaggregate storage offerings [1, 3, 42]; meanwhile, Magnet [41] is open-sourced as part of Spark but has yet to support disaggregated storage.

Furthermore, large-scale shuffle systems often come with more complicated deployment models. They are often deployed as auxiliary services to existing data processing systems. Shuffle services decouple block lifetimes from task executors to minimize interruptions upon executor failures [51], which are more frequent in large clusters. Shuffle services are also used to coordinate more sophisticated shuffle protocols, such as push-based shuffle and reduce-side merge [41]. However, because these shuffle services are only necessary at very large scale, they are not enabled by default in systems like Spark and require a separate deployment process.

Thus, while there has been significant innovation in new shuffle designs, few of these are widely deployed. Furthermore, it is difficult for an application to choose on the fly whether to use a particular shuffle algorithm; it requires both a priori knowledge of the application scenario and potentially an entirely different system deployment.

2.2 Random Shuffle in ML Training Pipelines

While much of the existing shuffle literature has focused on large-scale batch processing, there is also a need for performant shuffle in other application scenarios, such as online aggregation [8] and pipelining with more complex applications. An example of the latter is the *random shuffle* operation commonly used in machine learning training jobs. Note that by random shuffle, we mean the application-level transform that randomly permutes the rows of a dataset, rather than the generic system-level shuffle that is used to execute MapReduce applications.

To improve model convergence in deep learning, it is common practice to randomly shuffle the training dataset before feeding into GPU trainers to avoid bias on the order of the data [23, 24]. To minimize GPU pauses, the shuffle should be pipelined with the training execution (Figure 2d). Furthermore, it is desirable for developers to be able to trade off between performance and accuracy: they might wish to run shuffle in a smaller window to reduce training latency, at the cost of overall end model accuracy.

These differences make it difficult for ML pipelines to directly leverage existing monolithic shuffle systems. Systems like Hadoop and Spark are highly optimized for global shuffle operations, but are not designed to pipeline the shuffle with downstream executions: shuffle results cannot be read until the full shuffle is complete [8]. The results must be written out to an external store before they can be read by the training workers (Figure 2d*i*). However, this leads to either high memory footprint, as it requires holding an additional copy of the dataset, or higher I/O overhead, if the shuffled data is written to disk before transfer to the GPU.

Fine-grained pipelining can improve efficiency. Figure 2d*ii* shows an example in which the reduce tasks for a particular epoch are pipelined with the training computation. This allows results to be used as they become available while limiting memory footprint to a single partition. Alternatively, the application can also choose to shuffle the dataset in windows (Figure 2d*iii*), improving pipelining at the cost of accuracy. Unfortunately, existing shuffle systems are not built for such fine-grained pipelining, and most big data systems that offer high-performance shuffle use an execution model that is incompatible with deep learning systems [9].

Instead, ML training frameworks often end up re-implementing shuffle within specialized data loaders and thus run into known

problems that have been solved by traditional shuffle systems. Typically, data loaders are implemented with a pool of CPU-based workers colocated with the GPU trainers [16, 29, 43]. Each worker loads a partition of the dataset from storage (e.g., Amazon S3), preprocesseses it, and feeds the resulting data into the colocated trainers. To support random shuffle, the workers may read a random partition of the dataset on each epoch. However, to improve I/O efficiency, data must still be read in batches. Thus, to de-correlate data within the same batch, workers further shuffle the data by mixing records within a fixed-size local memory buffer. This effectively ties the shuffle window size to the size of the memory buffer. Setting the buffer size too large results in out-of-memory errors and poor pipelining, but if the buffer is too small, data de-correlation may be insufficient. In Section 5.3.3, we demonstrate how Exoshuffle can bring distributed shuffle optimizations to ML training applications, achieving both high performance and flexibility.

3 Shuffle with Distributed Futures

For distributed futures to serve as an intermediate abstraction layer for shuffle, they should: (1) abstract out the common implementation details of different shuffle implementations, (2) be general enough to allow heterogeneous end applications to interface with the shuffle library, and (3) provide the same performance and reliability as monolithic shuffle systems. This narrow waist for distributed shuffle would enable both faster development for new shuffle implementations and extensibility to new application use cases.

Monolithic shuffle systems use messaging primitives, like RPC, as an intermediate abstraction layer. RPC is both general-purpose and high-performance, but it is too low-level to be a useful intermediate layer for shuffle. Integrating push-based shuffle into Spark, for example, required 1k+ LoC for the RPC layer changes alone [40]. Much of this development effort lies in implementing new inter-task protocols for data transfer and integrating them alongside existing ones.

In contrast, distributed futures decouple the shuffle control plane from the data plane. This abstraction enables different shuffle libraries to share a common data plane. Optimizations like push-based shuffle can be implemented in an order of magnitude less code as a result (§5.2).

In this section, we show how to express previous shuffle optimizations and application-specific shuffle variants as application-level programs with the distributed futures API. These simplified examples capture the logical execution DAG of the shuffle. Section 4 describes the physical execution of these programs and the details in achieving performance parity with monolithic shuffle systems.

3.1 The Distributed Futures API

A distributed futures program invokes remote functions, known as *tasks*, that execute and return data on a remote node. When calling a remote function, the caller immediately gets a distributed future that represents the *eventual* return value. The future is "distributed" because the return value may be stored anywhere in the cluster, e.g., at the node where the task executes. This avoids copying return values back to the caller, which can become expensive for large data.

The caller can make use of a distributed future in two ways. First, it can create a DAG by passing a distributed future as an argument to another task. The system ensures that the dependent task runs only after all of its arguments are computed. Note that the caller can specify such dependencies before the value is computed and that the caller need not see the physical values. This gives the system control over parallelism and data movement, e.g., pipelining task execution with dependency fetching for other tasks, and allows the caller to manipulate data larger than local memory. Second, the caller can get the value of a distributed future using a `get` call, which fetches the value to the caller's local memory. This is useful when consuming the output of a shuffle, as it allows the caller to pipeline its own execution with the shuffle. The caller can additionally use a `wait` call, which blocks until a set of tasks complete (without fetching the return values), for synchronization and for avoiding scheduling too many concurrent tasks.

3.2 Expressing Shuffle with Distributed Futures

We demonstrate how these APIs can be used to express various shuffle optimizations (Fig 2) as application-level programs. We use Ray's distributed futures API for Python [26] for illustration. The `@ray.remote` annotation designates remote functions, and the `.remote()` operator invokes tasks.

```
def simple_shuffle(M, R, map, reduce):
  map_out = [map.remote(m) for m in range(M)]
  return ray.get([
    reduce.remote(map_out[:,r]) for r in range(R)]

def shuffle_riffle(M, R, F, map, reduce, merge):
  map_out = [map.remote(m) for m in range(M)]
  merge_out = [
    merge.remote(map_out[i*F:(i+1)*F, :])
    for i in range(M/F)]
  return ray.get([
    reduce.remote(merge_out[:,r]) for r in range(R)]

def shuffle_magnet(M, R, F, map, reduce, merge):
  map_out = [map.remote(m) for m in range(M)]
  merge_out = [
      [merge.remote(map_out[i*F:(i+1)*F, r])
      for i in range(M/F)] for r in range(R)]
  return ray.get(
    [reduce.remote(merge_out[:,r]) for r in range(R)]
```

Listing 1: Shuffle algorithms as distributed futures programs.

3.2.1 Simple Shuffle In Listing 1, `simple_shuffle` shows a straightforward implementation of the MapReduce paradigm illustrated in Figure 2a. The shuffle routine takes a `map` function that returns a list of map outputs, and a `reduce` function that takes a list of map outputs and returns a reduced value. `M` and `R` are the numbers of map and reduce tasks respectively. The two statements produce the task graph shown in Figure 2a. Note that the `.remote()` calls are non-blocking, so the entire task graph can be submitted to the system without waiting for any one task to complete.

This is effectively *pull-based shuffle*, in which shuffle blocks are *pulled* from the map workers as reduce tasks progress. Assuming a fixed partition size, the total number of shuffle blocks grows quadratically with the total data size. Section 5.1 shows empirical evidence of this problem: as the number of shuffle blocks increases, the performance of the naive shuffle implementation drops due to decreased I/O efficiency. Prior work [5, 41, 56] have proposed solutions to this problem, which we study and compare next.

3.2.2 Pre-Shuffle Merge Riffle [56] is a specialized shuffle system built for Spark. Its key optimization is merging small map output blocks into larger blocks, thereby converting small, random disk I/O into large, sequential I/O before shuffling over the network to the reducers. The merging factor F is either pre-configured, or dynamically decided based on a block size threshold. As soon as F map tasks finish on an executor node, their output blocks ($F \times R$) are merged into R blocks, each consisting of F blocks of data from the map tasks. This strategy, illustrated in Figure 2b, is implemented in Listing 1 (`shuffle_riffle`). The code additionally takes `F` as the merging factor, and a merge function which combines multiple map outputs into one.

Riffle's key design choice is to merge map blocks *locally* before they are pulled by the reducers, as shown in the highlighted lines. For simplicity, the code assumes that the first F map tasks are scheduled on the first worker, the next F map tasks on the second worker, etc. In reality, the locality can be determined using scheduling placement hints or runtime introspection (§4.2) Section 5.1 shows that this implementation of Riffle-style shuffle improves the job completion time over simple shuffle.

3.2.3 Push-based Shuffle Push-based shuffle (Fig 2c) is an optimization that pushes shuffle blocks to reducer nodes as soon as they are computed, rather than pulling blocks to the reducer when they are required. Magnet [41] is a specialized shuffle service for Spark that performs this optimization by merging intermediate blocks on the reducer node before the final reduce stage. This improves I/O efficiency and data locality for the final reduce tasks. `shuffle_magnet` in Listing 1 implements this design.

3.2.4 Straggler Mitigation Distributed futures enable dynamic task graphs by nature, making it ideal for detecting and reacting to stragglers during runtime.

Speculative Execution One way to handle stragglers is through speculative execution. Tasks that are suspected to be stragglers can be duplicated, and the system chooses whichever result is available first. This can be accomplished with distributed futures using the `ray.wait` primitive, as shown in Listing 2.

```
map_out = ...
_, timeout_tasks = ray.wait(map_out, timeout=TIMEOUT)
duplicates = []
for task in timeout_tasks:
  duplicates.append(map.remote(task.args))
for t1, t2 in zip(timeout_tasks, duplicates):
  t, _ = ray.wait([t1, t2], num_returns=1)
  map_out[t1.id] = t
```

Listing 2: Mitigating stragglers with speculative execution.

Best-effort Merge Shuffle systems including Riffle and Magnet also implement "best-effort merge", where a timeout can be set on the shuffle and merge phase [41, 56]. If some merge tasks are cancelled due to timeout, the original map output blocks will be fetched instead. This ensures straggler merge tasks will not block the progress of the entire system. Best-effort merge can be implemented in Exoshuffle as shown in Listing 3 using an additional `ray.cancel()` API which cancels the execution of a task. The cancelled task's input, which are the original map output blocks, will then be directly passed to the reducers. This way, the task graph is dynamically constructed as the program runs, adapting to runtime conditions while still enjoying the benefits of transparent fault tolerance provided by the system.

```
map_out = ...
merge_out = ...
_, timeout_tasks = ray.wait(merge_out, timeout=TIMEOUT)
for task in timeout_tasks:
  ray.cancel(task)
  merge_out[task.id] = task.args
out = [reduce.remote(merge_out[:, r]) for r in range(R)]
ray.wait(out)
```

Listing 3: Mitigating stragglers via task cancellation.

3.2.5 Data Skew Data skew can be prevented at the data management level using techniques such as key salting, or periodic repartitioning. However, it is still possible for skews to occur during ad-hoc query processing, especially for those queries involving joins and group-bys. Data skew during runtime can cause the working set of a reduce task to be too large to fit into executor memory.

Dynamic repartitioning solves this problem by further partitioning a large reducer partition into smaller ones. This is straightforward to implement since the distributed futures programming model enables dynamic tasks by nature. Listing 4 shows that we can recursively split down a reducer's working set until it fits into a predefined memory threshold.

```
@ray.remote
def reducer(*parts):
  total_size = [part.size() for part in parts]
  if total_size > THRESHOLD:
    L = len(parts) // 2
    return flatten([
      reducer.remote(*parts[:L]),
      reducer.remote(*parts[L:])])
```

Listing 4: Dynamic repartitioning for skewed partitions.

3.3 Applications

Because Exoshuffle implements shuffle at the application level, it can easily interoperate with other applications. Here, we demonstrate two example applications that use fine-grained pipelining with shuffle to improve end-to-end performance. These applications are evaluated in Section 5.3.

3.3.1 Online Aggregation with Streaming Shuffle Online aggregation [19] is an interactive query processing mode where partial results are returned to the user as soon as some data is processed, and are refined as progress continues. This is especially useful when the query takes a long time to complete. Online aggregation is difficult to implement in MapReduce systems because they require all outputs to be materialized before being consumed. Past work made in-depth modifications to Hadoop and Spark to support online aggregation [8, 55].

Online aggregation is straightforward to implement in Exoshuffle without the need to modify the underlying distributed futures system. Listing 5 shows the `streaming_shuffle` routine. It requires a modified `reduce` function that takes a reducer state and a list of map outputs and returns an updated state, and an `aggregate` function which combines the reducer states to produce aggregate statistics. Shuffle is executed in rounds. At the end of each round, the aggregation function is invoked with the reducer outputs, and will asynchronously print an aggregate statistic (e.g. sum) to the user. Note

```
1  def streaming_shuffle(map, reduce, print_aggregate):
2    reduce_states = [None] * R
3    for rnd in range(N):
4      map_results = [map.remote(M*rnd+i) for i in range(M)]
5      ray.wait(reduce_states)
6      reduce_states = [
7        reduce.remote(reduce_state, *map_results[:, r])
8        for r, reduce_state in enumerate(reduce_states)]
9      print_aggregate.remote(reduce_states)
10   return ray.get(reduce_states)
11
12 def model_training(trainer, data):
13   shuffle_out = shuffle(data, ...)
14   for epoch in range(EPOCHS):
15     next_shuffle_out = shuffle(data, ...)
16     for block in shuffle_out:
17       trainer.train(ray.get(block))
18     shuffle_out = next_shuffle_out
```

Listing 5: Streaming shuffle and pipelined data loading for ML.

that the Exoshuffle user can simply swap between `simple_shuffle` and `streaming_shuffle` to get the semantics they desire.

3.3.2 Distributed ML Training with Pipelined Shuffle Exoshuffle also enables fine-grained pipelining for ML training, as illustrated in Figure 2d. In Listing 5, `model_training` shows the code skeleton. On line 13, the `shuffle` function (could be any in Listing 1) returns a set of distributed futures pointing to reducer outputs. They are passed immediately to the model trainer while shuffle executes asynchronously. As soon as a reducer block becomes available, the model trainer acquires it (line 17) and send it to the GPU for training. This achieves the fine-grained pipelining described in Figure 2d.

4 System Architecture

Section 3 shows how shuffle DAGs can be expressed as distributed futures programs. However, achieving high performance shuffle also requires a set of critical system facilities. In this section, we describe the architecture of Exoshuffle via a realistic implementation of the push-based shuffle described in Section 3.2.3. We describe the additional system APIs used by Exoshuffle (§4.2), and the transparent features provided by the underlying distributed futures implementation (§4.3) that are key to performance.

4.1 Example: Push-based Shuffle

Listing 6 implements push-based shuffle (§3.2.3) for a cluster of `NUM_WORKERS` nodes. The library takes a `map` and a `reduce` function as input. The remaining constants are chosen by the library according to the user-specified number of input and output partitions.

Lines 11–25 comprise the map and merge stage, in which map results are shuffled, pushed to the reducer nodes, and merged. This stage pipelines between CPU (`map` and `merge` tasks), network (to move data between `map` and `merge`), and disk (to write out `merge` results). The map and merge tasks are scheduled in rounds for pipelining: Lines 18–19 ensures that there is at most one round of merge tasks executing, and that they can overlap with the following round's map tasks. Each round submits one merge task per worker node. Each merge task takes in one intermediate result from each map task from the same round and returns as many merged results as there are reduce partitions on that worker.

Once all map and merge tasks are complete, we schedule all reduce tasks (lines 28–31) and return the distributed future results. Each reduce task performs a final reduce on all merge results for its given partition. To minimize unnecessary data transfer, the reduce tasks are co-located with the merge tasks whose results they read.

```
1  def push_based_shuffle(map, reduce):
2    @ray.remote
3    def merge(*map_results):
4      for results in zip(*map_results):
5        yield reduce(*results)
6
7    merge_results = numpy.empty((NUM_WORKERS,
8      NUM_ROUNDS, NUM_REDUCERS_PER_WORKER))
9
10   # Map and shuffle.
11   for rnd in range(NUM_ROUNDS):
12     for i in range(NUM_TASKS_PER_ROUND):
13       map_results = [
14         map.options(num_returns=NUM_WORKERS).remote(
15           parts[rnd * NUM_TASKS_PER_ROUND + i])
16         for i in range(NUM_TASKS_PER_ROUND)]
17
18     if rnd > 0:
19       ray.wait(merge_results[:, rnd - 1, :])
20
21     for w in range(NUM_WORKERS):
22       merge_results[w, rnd, :] = merge.options(
23         worker=w, num_returns=NUM_REDUCERS_PER_WORKER
24       ).remote(*map_results[:, w])
25     del map_results
26
27   # Reduce.
28   return flatten(
29     [[reduce.remote(*merge_results[w, :, rnd])
30       for rnd in range(NUM_REDUCERS_PER_WORKER)]
31       for w in range(NUM_WORKERS)])
```

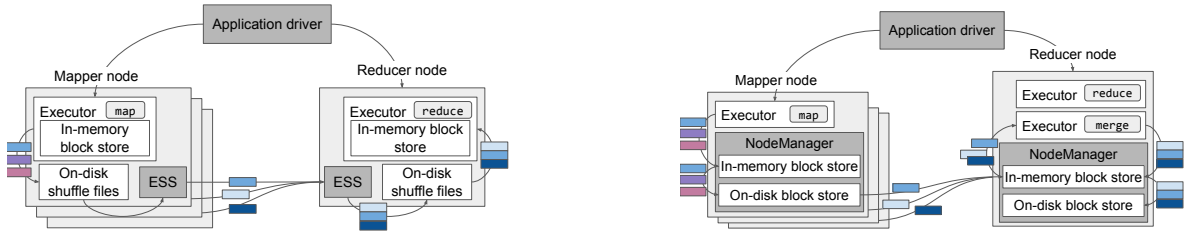
Listing 6: Implementation of two-stage shuffle.

4.2 Scheduling Primitives

For complex applications like distributed shuffle, it is difficult for a general-purpose system to make optimal decisions in every context. For instance, optimally scheduling a computation DAG on a set of nodes is NP-hard [7]. It is therefore more robust to allow the application or library developer to apply domain-specific knowledge to achieve better performance.

By default, Ray provides a two-level distributed scheduler that balances between bin-packing vs. load-balancing [26]. This is sufficient for map and reduce tasks in simple shuffle, as these can be executed anywhere in the cluster. However, more advanced shuffle strategies (§§3.2.2 and 3.2.3) require more careful placement and scheduling of tasks to improve performance. In this section, we describe the additional APIs designed to give the shuffle library more control over the physical execution of the shuffle DAG.

4.2.1 Scheduling for Data Locality Ray provides automatic locality-based scheduling when possible. For example in Listing 6, lines 28–31, Ray automatically schedules the reduce tasks on the workers on which the upstream merge results reside. In some other cases, hints must be provided to the system to achieve better data locality. For example, a group of merge tasks must be colocated with the downstream reduce task, but this is impossible for the system to determine because the reduce task's dependency is not known to the system yet. To handle this problem, we introduce *node-affinity scheduling* in Ray, which allows the application to pin tasks to a particular node. For example, Listing 6 uses this in line 23 to colocate merge tasks for the same reducer. Node affinity is soft, meaning that Ray will choose another suitable node if the specified node fails.



(a) Example of a monolithic shuffle architecture.

(b) Exoshuffle.

Figure 3: Comparing a monolithic vs. application-level shuffle architecture. (a) implements all coordination and block management through an external shuffle service on each node, in this case implementing the Magnet shuffle strategy (§3.2.3). (b) shows the same shuffle strategy but implemented as an application on a generic distributed futures system.

4.2.2 Scheduling for Task Pipelining The map and merge tasks should be pipelined to allow map results to be shuffled concurrently with map execution. This task-level pipelining is challenging for a distributed futures system to determine automatically: Too many concurrent map tasks will reduce resources available to downstream merge tasks, and scheduling the wrong set of map and merge tasks concurrently prevents map outputs from being consumed directly by merge tasks, resulting in unnecessary disk writes. The shuffle library is better placed to determine that it should apply backpressure by limiting the number of concurrent map and merge tasks. The library can also determine that a round of merge tasks should be executed concurrently with the following round of map tasks. Exoshuffle achieves this with the `wait` API (Listing 6, line 19), which blocks until a task completes.

4.2.3 Controlling Redundancy with Reference Counting Distributed futures are reference-counted in Ray. While an object reference is in scope, Ray attempts to ensure its value exists in the cluster. By selecting which references to keep or drop, the shuffle library can make tradeoffs between reducing write amplification and improving data redundancy. For example, line 25 of Listing 6 deletes the intermediate map results from the current round. This reduces write amplification, as the map results can be immediately dropped from memory without spilling to disk, but requires additional re-execution upon failure. Alternatively, the shuffle library can instead keep the intermediate references, resulting in additional disk writes but improved data redundancy.

4.3 Transparent System Facilities

The actual data transfer, or *shuffle*, is managed by the distributed futures system according to the application specifications. For example in Listing 6, lines 23–25 specifies that one column of the distributed futures in `map_results` should be sent to one merge task. This prompts the data plane to transfer the corresponding physical data to the `merge` task's location. In this section, we describe the transparent storage and I/O mechanisms provided by the distributed futures system to facilitate this data movement.

4.3.1 Shared Memory Object Store Previous monolithic shuffle systems implement distributed coordination via an *external shuffle service*, a specialized process deployed to each node that orchestrates block transfers (Fig 3a). This process is external to the executors, decoupling block transfers from map and reduce task execution. In Exoshuffle, we replace this service with a generic node manager that is responsible for both in-memory and spilled objects (Fig 3b).

We build on Ray's shared memory object store [26] for immutable objects. Each node manager hosts a shared memory object store shared by all executors on that node (Fig 3b). This decouples executors from blocks: once a task's outputs are stored in its local object store, the node manager manages the block. This keeps executors stateless and allows them to execute other tasks or exit safely while the node manager coordinates block movement. Shared memory enables *zero-copy* reads of object data on the same node, which avoids CPU and memory overhead. By making objects immutable, we also avoid consistency concerns between object copies.

Next, we describe extensions to the original Ray architecture [26] made in this work that improves pipelining disk and network I/O with task execution. These improvements are made at the system level without knowledge of the application-level shuffle semantics, and thus can benefit a wide range of data-intensive applications.

4.3.2 Pipelined Object I/O

Object Allocation and Fetching. There are two categories of object memory allocations: new objects created for task returns (e.g., `map` task outputs), and copies of objects fetched remotely as task arguments (e.g., `merge` task inputs). The memory subsystem queues and prioritizes object allocations to ensure forward progress while keeping memory usage bounded to a limit. This is critical for reducing thrashing within the object store, caused by requesting objects for too many concurrent requests, while leaving sufficient heap memory for task executors.

All memory allocations on a Ray worker node go into an allocation queue for fulfillment. If there is spare memory, the allocation is fulfilled immediately. Otherwise, requests are queued until the spilling process or garbage collection frees up enough memory. If memory is still insufficient, Ray falls back to allocating task output objects on the filesystem to ensure liveness. Spare memory besides the memory allocated to executing task arguments and returns is used to fetch the arguments of queued tasks. This enables pipelining between execution and I/O, i.e. restoring objects from disk or fetching objects over the network. For example, at line 28 in Listing 6, all merge results are already spilled to disk and all reduce tasks are submitted at once. While earlier reduce tasks execute, the system uses any spare memory to restore merge results for the next round of reduce tasks from disk.

Object Spilling. Object spilling is transparent, so the application need not specify if or when it should occur. When the memory allocation subsystem has backlogged requests, the spilling subsystem migrates referenced objects to disk to free up memory. When a spilled object's data is required locally for a task, e.g., because it is the argument of a queued task, the node manager copies it back to memory as described above. When requested by a remote node, the spilled object is streamed directly from disk across the network to the remote node manager. To improve I/O efficiency, Ray coalesces small objects into larger files before writing to the filesystem.

4.3.3 Fault Tolerance Exoshuffle relies on lineage reconstruction for distributed futures to recover objects lost to node failures [48], a similar mechanism to previous shuffle systems [10, 53]. In Ray, the application driver stores the object lineage and resubmits tasks as needed upon failure. This process is transparent to Exoshuffle, which runs at the application level. Still, Exoshuffle can use object references (§4.2.3) to specify reconstruction or eviction for specific objects.

Executor process failures are much more common than node failures. If reconstruction is required each time an executor fails, it can impede progress [41, 51]. Many previous shuffle systems use an external shuffle service to ensure map output availability in the case of executor failures or garbage collection pauses. Similarly, in Exoshuffle, executor process failures do not result in the loss of objects, because the object store is run inside the node manager as a separate process.

More sophisticated shuffle systems require additional protocols such as deduplication to ensure fault tolerance [5]. Distributed futures prevent such inconsistencies because they require objects to be immutable, task dependencies to be fixed, and tasks to be idempotent.

To reduce the chance of data loss, some shuffle system uses on-disk [41] or in-memory [5] replication of intermediate blocks to guard against single node failures. In Ray, objects are spilled to disk and transferred to remote nodes where they are needed, which also results in multiple copies as long as the object is in scope. The application can also disable this optimization by deleting its references to the object (Listing 6, L25). In the future, we could allow the application to more finely tune the number of replicas kept, e.g., by passing this as a parameter during task invocation.

5 Evaluation

We study the following questions in the evaluation:

- Can Exoshuffle libraries achieve performance and scalability competitive with monolithic shuffle systems? (§5.1)
- Is it easier to implement shuffle optimizations in Exoshuffle? (§5.2)
- What benefits does Exoshuffle provide for applications, including CloudSort, online aggregation and ML training? (§5.3)
- How do the features in the distributed futures backend contribute to Exoshuffle performance? (§5.4)

5.1 Shuffle Performance

5.1.1 Setup We create test environments on Amazon EC2 using VMs targeted at data warehouse use cases. We test on a HDD cluster of d3.2xlarge instances (8 CPU, 64 GiB RAM, 6× HDD, 1.1 GB/s aggregate sequential throughput, 18K aggregate IOPS, 15 Gbps network), and a SSD cluster of i3.2xlarge instances (8 CPU, 61 GiB RAM, NVMe SSD, 720 MB/s throughput, 180K write IOPS, 10 Gbps network).

Workload. We run the Sort Benchmark (a.k.a. TeraSort or CloudSort) [39], as it is a common benchmark for testing raw shuffle system performance. This benchmark requires sorting a synthetic dataset of configurable size, consisting of 100-byte records with 10-byte keys.

Baselines. We compare to the push-based shuffle service in Spark, a.k.a. Magnet, and a theoretical baseline.

Magnet is integrated into Spark in its 3.2.0 release as an external push-based shuffle service. We run Spark 3.2.0 on Hadoop 3.3.1 with Magnet shuffle service enabled. We disable compression of shuffle files according to the rules of TeraSort. This allows for a fair comparison in terms of total bytes of disk I/O.

For the theoretical baseline, we assume disk I/O is the bottleneck since empirically we find that disk I/O takes longer than networking and CPU processing in this benchmark. The baseline is calculated by $T = 4D/B$, where D is the total data size and B is the aggregate disk bandwidth. D is multiplied by 4 because each datum needs to be read twice and written twice, a theoretical minimum for external sort [36].

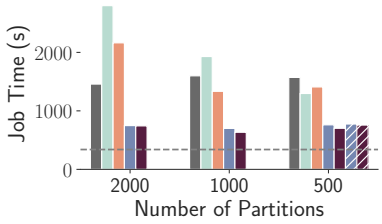
Exoshuffle variants. We run Exoshuffle on Ray 1.11.0. We compare implementations of the following shuffle libraries:

- ES-simple, the simple shuffle variant (§3.2.1).
- ES-merge, pull-based shuffle with pre-shuffle merge, similar to that in Riffle (§3.2.2).
- ES-push, push-based shuffle similar to Magnet (§3.2.3).
- ES-push*, push-based shuffle further optimized to reduce write amplification (§4.1).

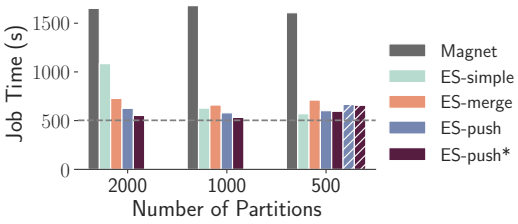
5.1.2 Performance Comparison of Shuffle Algorithms

Performance on HDD. Figure 4a shows the job completion times of Exoshuffle variants running 1 TB sort on 10 HDD nodes. ES-simple shows the well-known scaling problem: performance degrades as the number of partitions increases, because the intermediate shuffle blocks become more in number and smaller in size both quadratically, quickly reaching disk IOPS limit. The push-based shuffle variants (ES-push, -push*) achieve better performance regardless of the number of partitions, thanks to the merging of shuffle blocks to increase disk I/O efficiency and the pipelining of disk and network I/O. ES-merge runs slower than -simple because merging the map output blocks incurs additional disk writes, which outweighs the I/O efficiency savings when the number of partitions is small, and only shows benefits when the number of partitions increases. The Magnet baseline shows comparable performance. In summary, Exoshuffle libraries demonstrate performance benefits that match the characteristics of their monolithic counterparts.

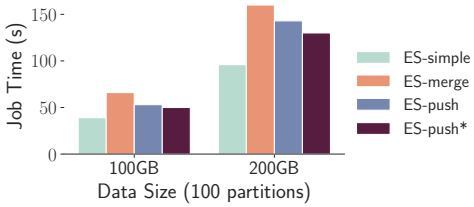
Performance on SSD. Figure 4b shows the same benchmark and variants running on the SSD cluster. All variants of Exoshuffle outperform the PBS baseline, and display similar trends as on the HDD cluster. The run times of the optimized versions of Exoshuffle are also close to the theoretical baseline. Since the NVMe SSD supports much higher random IOPS, the I/O efficiency gains are less pronounced.



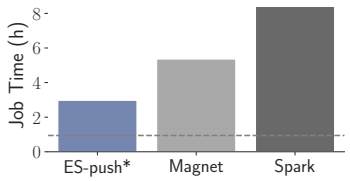
(a) 1 TB sort on 10 HDD nodes.



(b) 1 TB sort on 10 SSD nodes. Semi-shaded bars are runs with failures (§5.4.3).



(c) In-memory sort on 10 SSD nodes.



(d) 100 TB on 100 HDD nodes.

Figure 4: Comparing job completion times on the Sort Benchmark. The dashed lines indicate the theoretical baseline (§5.1.1). Exoshuffle is abbreviated as ES.

In-memory Performance. Figure 4c shows that when data fits in memory, ES-simple is actually the fastest algorithm compared to all other variants. This is because the other algorithms create copies of data by merging them, triggering unnecessary disk spilling. Magnet observes similar behavior for small datasets[1].

Conclusion. These experiments show that the shuffle algorithms provided by Exoshuffle offer the same performance benefits as their monolithic counterparts. Furthermore, the most performant shuffle algorithm depends on the data size and hardware configuration, and Exoshuffle offers the flexibility to choose the most suitable algorithm at the application level, without having to deploy multiple systems.

5.1.3 Shuffle Scalability To test performance at large scale, we run the Sort Benchmark on 100 TB data with 50 000× 2 GB input partitions on a cluster of 100× d3.2xlarge VMs. For Exoshuffle, we run the ES-push* variant since it is the most optimized for scale. For baselines, we run both Spark's native shuffle (Spark) and its push-based shuffle service (Magnet). We run both baselines with compression on because Spark without compression becomes unstable at this scale.

Figure 4d shows the results. Exoshuffle outperforms both native Spark shuffle and the push-based shuffle service Magnet, despite Spark's compression reducing total bytes spilled by 40%. Magnet improves shuffle performance by 1.6× because it reduces random disk I/O. Exoshuffle further improves performance over push-based Spark by 1.8×. This difference comes from reduced write amplification in ES-push*, which spills only the merged map outputs, while Magnet also spills the un-merged map outputs. These additional writes provide faster failure recovery through improved durability, albeit at the cost of performance. Exoshuffle allows the application to choose between these tradeoffs by using ES-push vs. ES-push*.

[1]The Spark 3.3.1 documentation states: "Currently [Magnet] is not well suited for jobs/queries which runs quickly dealing with lesser amount of shuffle data."
[2]Total lines of code in `org.apache.spark.shuffle`.
[3]As reported by Zhang et al. [56]
[4]Total added lines in https://github.com/apache/spark/pull/29808/files.

Shuffle Algorithm	System LoC	Exoshuffle LoC
Simple (§3.2.1)	2600 (Spark[2])	**215**
Pre-shuffle merge (§3.2.2)	4000 (Riffle[3])	**265**
Push-based shuffle (§3.2.3)	6700 (Magnet[4])	**256**
with pipelining (§4.1)	–	**256**

Table 2: Approximate lines of code for implementing shuffle algorithms in Exoshuffle versus in specialized shuffle systems.

5.2 Implementation Complexity

In Exoshuffle, shuffle is expressed as application-level programs. Table 2 compares the amount of code of several monolithic shuffle systems with the lines of code needed to implement the corresponding shuffle algorithms in Exoshuffle. Exoshuffle libraries may not provide all the production features of the monolithic counterparts, but many shuffle optimizations can be implemented in Exoshuffle with an order of magnitude less code, while keeping the same performance benefits. By offering shuffle as a library, Exoshuffle also allows applications to choose the best shuffle implementation at run time without deploying multiple systems.

5.3 Shuffle Applications

Next, we show how Exoshuffle can extend distributed shuffle support for a broader set of applications.

5.3.1 CloudSort The CloudSort competition [39] calls for the most cost-efficient way to sort 100 TB of data on the public cloud. We ran Exoshuffle-CloudSort on a cluster of 40× i4i.4xlarge nodes with input and output data stored on Amazon S3, and set a new world record of $0.97/TB [21]. This is 33% more cost-efficient than the previous world record set in 2016. The previous entry used a heavily modified version of Spark for the CloudSort workload [45]. In contrast, Exoshuffle-CloudSort is only hundreds of lines of application code running on a release version of Ray.

To account for the fact that the cloud hardware costs have lowered since 2016, we take the setup from the previous record-winning entry and look up its cost on today's Alibaba Cloud. Table 3 shows that the same amount of cloud resources would cost $115 today.

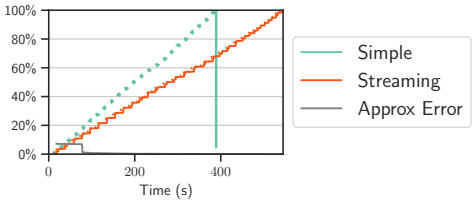


Figure 5: Online aggregation. Dotted lines show map progress; solid, reduce progress.

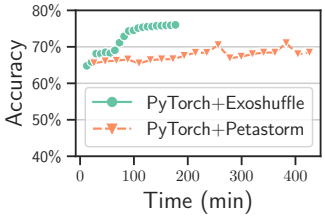


Figure 6: Single-node ML training for 20 epochs.

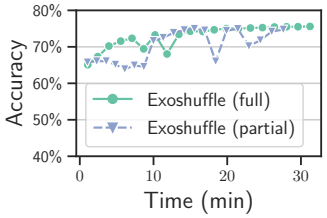


Figure 7: 4-node, distributed ML training for 20 epochs.

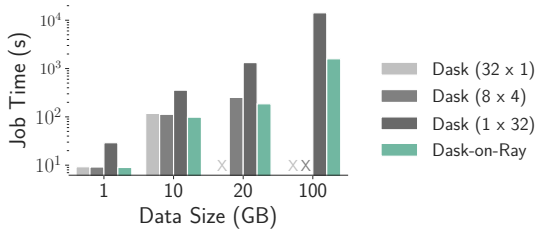


Figure 8: Comparing shuffle time in Dask and Ray. Legends show number of processes × threads.

Figure 9: Effect of I/O optimizations in Ray.

Still, Exoshuffle-CloudSort achieves another 15% cost reduction beyond this result. We calculate another theoretical baseline of simply shuffling 100 TB data through the AWS network (without sorting), which would cost $74. This puts our record within 31% of the theoretical limit. This result demonstrates that Exoshuffle can achieve state-of-the-art performance and cost-efficiency for large-scale shuffle.

System	Cost
NADSort (2016)	$1.44/TB
NADSort (2022, extrapolated)	$1.15/TB
Exoshuffle-CloudSort (2022)	**$0.97/TB**

Table 3: CloudSort costs over years.

5.3.2 Online Aggregation with Streaming Shuffle We use a 1 TB dataset containing 6 months of hourly page view statistics on Wikipedia. We run an aggregation to get the ranking of the top pages by language on 10× r6i.2xlarge nodes with data loaded from S3. Figure 5 shows the difference between regular and streaming shuffle. The streaming shuffle takes 1.4× longer to run in total due to the extra computation needed to produce partial results. However, with streaming shuffle, the user can get partial aggregation results within 8% error[5] of the final result in 18 seconds, 22× faster than regular shuffle. Exoshuffle makes it easy to switch between `simple_shuffle` and `streaming_shuffle` to choose between partial result latency and total query run time.

5.3.3 Distributed ML Training Many distributed training frameworks already run on top of Ray. By offering Exoshuffle as a library, we enable these workloads to leverage scalable shuffle. We demonstrate Exoshuffle's ability to support fine-grained pipelining for ML training using the Ludwig framework [25] to train a deep classification model TabNet on the HIGGS dataset (7.5 GB). Ludwig integrates ML data loaders with the PyTorch training framework [30]. Efficient training requires randomly shuffling the data per epoch before sending it into the GPU for training.

We first run the ML training on a single g4dn.4xlarge instance. We compare two versions of Ludwig: Ludwig 0.4.0 uses Petastorm [16], which prefetches data in batches into a per-process memory buffer and performs random shuffle in the buffer. This approach makes the shuffle window size limited by the memory buffer size (§2.2). In this experiment, we set the shuffle window size to 9% of the total data size to avoid OOM errors. In comparison, Ludwig 0.4.1 uses Exoshuffle offered through Ray Data [44]. It pipelines data loading and shuffling with GPU training (Fig 2d), and supports full shuffle across loading batches by storing data in the shared-memory object store. Figure 6 shows that model training with Exoshuffle is 2.4× faster end-to-end thanks to the fine-grained pipelining. The model also converges faster per-epoch and to a higher accuracy, because Exoshuffle performs complete random shuffling between epochs, whereas Petastorm's random shuffle is limited to subsets of the data.

Next, we run the training on 4 g4dn.xlarge nodes to show the distributed shuffle performance. Ludwig 0.4.*x* has known bugs with distributed training, so we could not compare Petastorm with Exoshuffle. Instead, we use the latest Ludwig 0.6.0 and compares two shuffle strategies with the Exoshuffle-based data loader: full shuffle (the default) and partial shuffle. For partial shuffle, we emulate the Petastorm behavior and perform random shuffling only in each in-memory batch. Figure 7 shows that per-epoch time is slightly faster with partial shuffle, since it is fully local, but the convergence accuracy is slightly lower because of the less random shuffling of training data. This example demonstrates that Exoshuffle gives the developer the flexibility to choose the best shuffle strategy based on

[5]Error is computed using the KL-divergence $D_{KL} = \sum p \log(p/\hat{p})$ where p is the true statistic and $\hat{p}$ is the sample statistic.

their training needs, while providing high-throughput data loading and shuffling.

5.4 System Microbenchmarks

The Exoshuffle architecture requires high-performance components from the distributed futures system to deliver good performance. In this section, we study the the impact of these system components on shuffle performance.

5.4.1 Shared-Memory Object Store We study the effect of a shared-memory object store that decouples objects from executors by comparing Dask and Ray. Dask and Ray are both distributed futures systems, but they differ in architecture. Ray uses a shared-memory object store that is shared by multiple executor processes on the same node (§4.3.1). Dask stores objects in executor memory and requires the user to choose between multiprocessing and multithreading. With multithreading, multiple Dask executor threads share data in a heap-memory object store, but the Python Global Interpreter Lock can severely limit parallelism. Dask in multiprocessing mode avoids this issue but uses one object store per worker process, so objects must be copied between workers on the same node. Thus, the lack of a shared-memory object store results in either reduced parallelism (multithreading) or high overhead for sharing objects (multiprocessing). It is also less robust as objects are vulnerable to executor failures.

We study these differences by running the same Dask task graph on Dask and Ray backends [46]. Figure 8 shows dataframe sorting performance on a single node (32 CPU, 244 GB RAM, 100 partitions). For Dask, we vary the number of executor processes and threads to show the tradeoff between memory usage and parallelism. Ray requires no configuration and uses 32 executor processes, 1 per CPU.

On small data sizes, Dask with multiprocessing achieves about the same performance as Ray, but it is 3× slower with multithreading due to reduced parallelism. On larger data sizes, Dask with multiprocessing fails due to high memory pressure from extra object copies. Meanwhile, Ray's shared-memory object store enables better stability and lower run time on all data sizes.

5.4.2 Small I/O Mitigations Ray implements two system-level optimizations for mitigating the small I/O problem: fusing writes of spilled objects to avoid small disk I/O, and prefetching task arguments to hide network and disk latency (§4.3.2). To show the impact of these optimizations, we run a single-node microbenchmark that creates 16 GB total objects in a 1 GB object store, forces them to spill to disk, then restores the objects from disk. We use object sizes ranging from 100 KB to 1 MB, as these are comparable to the shuffle block sizes. We use a sc1 HDD disk since the disk I/O bottleneck is more pronounced on slower storage.

Fusing Writes. Ray fuses objects into at least 100 MB files then writes them to disk. Figure 9 shows the total run time stays constant across object sizes with default fusing. When fusing is off, the run time is 25% slower for 1 MB objects, and up to 12× slower when spilling 100 KB objects.

Prefetching Task Arguments. Ray prefetches task arguments in a pipelined manner so that arguments are ready on a worker by the time the task is scheduled. Figure 9 shows that pipelined fetching of task arguments reduces the run time by 60–80%, comparing with a baseline implementation that only starts fetching objects after the task is scheduled.

5.4.3 Fault Tolerance To test fault recovery, we fail and restart a random worker node 30 seconds after the start of the run. This results in both executor failure and data loss, as the worker's local object store is also lost. In all cases, we rely on the distributed futures system to re-execute any lost tasks and to reconstruct any lost objects. Lineage reconstruction (§4.3.3) minimizes interruption time during worker failures. Figures 4a and 4b show run times with failures indicated with semi-shaded bars. For ES-simple and -merge, a known bug in Ray currently prevents fault recovery from completing. For ES-push and -push*, recovering from a worker failure adds 20–50 seconds to the job completion time. The system uses this time to detect node failures and re-execute tasks to reconstruct lost objects.

6 Related Work

Shuffle in Data Processing Systems. Many solutions to shuffle have been proposed [17, 20, 35–37, 49] since MapReduce [10] and Hadoop [2], with a focus on optimizing disk I/O and pipelining. Sailfish [35] is a notable example deployed at Yahoo which depends on a modified filesystem to batch disk I/O. Many recent shuffle systems have been built in industry for large-scale use cases [4, 5, 41, 56], but few have been open-sourced. Today's cloud providers often offer managed shuffle services [1, 3, 42]. However these are tightly integrated with proprietary cloud data services and are not accessible by other shuffle applications.

Hardware Environments. Hardware typically poses a range of constraints on shuffle design. For example, compute and memory may be either disaggregated [5, 33, 56] or colocated [35, 41, 51]. Disk constraints also affect system design, e.g., SSDs provide better random IOPS than HDDs but wear out more quickly. Many existing shuffle systems have been motivated by such hardware differences. In Exoshuffle, because the distributed futures API abstracts block management, a shuffle developer can plug in different storage backends and optimize shuffle at the application level.

Other Shuffle Applications. Machine learning research [23, 24] shows that SGD-based model training benefits from random shuffling of the training dataset. Both TensorFlow [29] and PyTorch [43] have built specialized systems designed specifically to pipeline data loading with ML training. These data loaders, in addition to Petastorm [16], support distributed data loading and random shuffling but shuffling is limited to a local buffer capped by worker memory (§5.3.3).

Dataframes [32, 38, 50] are another class of applications in data science that depend on shuffle for operations such as group-by. While systems like Dask [38] and Spark [54] provide distributed dataframes, developers continue to build new engines that optimize for specific application scenarios, such as multi-core [11], out-of-core performance [6], or supporting SQL [15]. These new dataframe libraries, along with new embedded query engines such as DuckDB [34] and Velox [31], can directly use Exoshuffle to support distributed query processing.

Distributed Programming Abstractions for Shuffle. CIEL [28] is the first to propose using distributed futures to express iterative distributed dataflow programs, including MapReduce. Its implementation does not include features critical to large-scale shuffle performance, including intra-node parallelism, in-memory object storage, and automatic garbage collection [27]. Dask [38] is another distributed futures-based system that has trouble scaling shuffle due to the lack of shared-memory objects (§5.4.1). While we build on Ray's design, such as a shared-memory object store [26] and lineage reconstruction [48], previous versions are not sufficient to support large-scale shuffle as they do not include spilling to disk or pipelining between execution and I/O. Thus, while others have implemented shuffle on distributed futures before, ours is the first that we know of to reach the scale, performance, and reliability of monolithic shuffle systems.

Serverless functions, as used in Locus [33], are one alternative to distributed futures. While Locus leverages an existing serverless cache and persistent storage, it still must manage block movement manually. In contrast, distributed futures abstract block management in full and manage execution, memory, and disk collectively on each node.

Hoplite [57] shows that it is possible to provide a high-performance and fault-tolerant collective communication layer on top of distributed futures, supporting operations such as scatter, gather, and reduce. Shuffle in MapReduce-like systems is a more challenging problem because it involves scheduling arbitrary compute tasks along with all-to-all communication. In this work we show that a distributed futures system can support shuffle at TB+ scale and provide competitive performance and reliability.

7 Discussion

Extensible Architectures. The decoupling of control and data planes in software-defined networking [22] has led to great innovations in the past two decades [14]. Operating systems research also advocates for extensible architectures to build OS kernels, such as microkernels [52] and exokernels [13]. We hope our work can drive more innovations in shuffle designs and applications through an extensible architecture for distributed shuffle.

Distributed Futures. Distributed futures are rising in popularity due to their ease of use and flexibility [26, 28, 47]. However, the question of flexibility versus performance remains. Large-scale shuffle is one of the most challenging problems in big data processing, inspiring years of work. By showing that large-scale shuffle is possible on a generic and flexible distributed futures system, we hope to show that other complex applications can be built on this framework, too.

Limitations. The ability to specify arbitrary tasks and objects with distributed futures is the key to its flexibility, but it is also the primary obstacle to performance. The system assumes that each task is independent for generality and stores metadata separately for each task and object. In contrast, monolithic shuffle systems have semantic information and can share metadata for tasks and objects in the same stage. Currently metadata overhead is the main limitation to executing Exoshuffle at larger scales. We plan to address this in the future by "collapsing" shared metadata, i.e., keeping one metadata entry for multiple outputs of a task.

Architecturally, the primary limitation in Exoshuffle is the fact that an object must be loaded in its entirety into the local object store before it can be read (§4.3.1). Generators allow tasks to "stream" large outputs by breaking them into many smaller physical objects; future improvements include the described metadata optimizations and/or introducing APIs to stream objects larger than the object store, similar to Ciel [28]. Another limitation is in scheduling. Currently the distributed futures system may require hints from the shuffle library to determine which tasks should be executed concurrently and where to place tasks (§4.2). A more sophisticated scheduler may be able to determine these automatically.

Finally, Exoshuffle does not yet address the problem of providing a single shuffle solution that can meet the requirements of all applications. Doing so would require automatically picking the best shuffle algorithm and parameters based on application, environment, and run-time information. Instead, we focus on the problem of shuffle *evolvability*, a necessary step towards this overarching goal.

8 Conclusion

There is a longstanding tension between performance and flexibility in designing abstractions for distributed computing. Monolithic shuffle systems sacrifice flexibility in the name of performance: they must essentially rebuild shuffle from scratch to handle varying application scenarios. In this work, we show that this need not be the case, by demonstrating an extensible architecture with a distributed futures system that makes it possible build efficient, flexible, and portable shuffle.

This paper does not raise ethical issues.

Acknowledgement

We thank the SIGCOMM reviewers and our shepherd Yiting Xia for their valuable feedback. We also thank members of the Sky Computing Lab at UC Berkeley for all the insightful discussion around this work. We thank Anyscale for providing the cloud resources necessary for completing the experiments in this work. This work is in part supported by NSF CISE Expeditions Award CCF1730628, and gifts from Astronomer, Google, IBM, Intel, Lacework, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Nexla, Samsung SDS, Uber, and VMware.

References

[1] Alibaba. 2021. EMR Remote Shuffle Service: A Powerful Elastic Tool of Serverless Spark - Alibaba Cloud Community. https://www.alibabacloud.com/blog/emr-remote-shuffle-service-a-powerful-elastic-tool-of-serverless-spark_597728. (Accessed on 02/01/2022).

[2] Apache Software Foundation. 2021. Hadoop. https://hadoop.apache.org.

[3] Anubhav Awasthi, Rajendra Gujja, and Mohit Saxena. 2021. Introducing Amazon S3 shuffle in AWS Glue. https://aws.amazon.com/blogs/big-data/introducing-amazon-s3-shuffle-in-aws-glue/. (Accessed on 10/16/2022).

[4] Mayank Bansal and Bo Yang. 2020. Zeus: Uber's Highly Scalable and Distributed Shuffle as a Service - Databricks. https://databricks.com/session_na20/zeus-ubers-highly-scalable-and-distributed-shuffle-as-a-service. (Accessed on 02/01/2022).

[5] Dmitry Borovsky and Brian Cho. 2019. Cosco: An Efficient Facebook-Scale Shuffle Service - Databricks. https://databricks.com/session/cosco-an-efficient-facebook-scale-shuffle-service. (Accessed on 01/19/2022).

[6] Maarten A. Breddels and Jovan Veljanoski. 2018. Vaex: big data exploration in the era of Gaia. *Astronomy & Astrophysics* 618 (oct 2018), A13. https://doi.org/10.1051/0004-6361/201732493

[7] Lingfeng Cai, Xianglin Wei, Changyou Xing, Xia Zou, Guomin Zhang, and Xiulei Wang. 2021. Failure-resilient DAG task scheduling in edge computing. *Computer Networks* 198 (08 2021), 108361. https://doi.org/10.1016/j.comnet.2021.108361

[8] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce Online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (San Jose, California) *(NSDI '10)*. USENIX Association, USA, 21.

[9] Jason Jinquan Dai, Yiheng Wang, Xin Qiu, Ding Ding, Yao Zhang, Yanzhang Wang, Xianyan Jia, Cherry Li Zhang, Yan Wan, Zhichao Li, Jiao Wang, Shengsheng Huang, Zhongyuan Wu, Yang Wang, Yuhao Yang, Bowen She, Dongjie Shi, Qi Lu, Kai Huang, and Guoqiong Song. 2019. BigDL: A Distributed Deep Learning Framework for Big Data. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 50–60. https://doi.org/10.1145/3357223.3362707

[10] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (jan 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[11] Polars Developers. 2022. Polars – User Guide. https://pola-rs.github.io/polars-book/user-guide/index.html. (Accessed on 10/16/2022).

[12] Spark developers. 2021. Spark Release 3.2.0. https://spark.apache.org/releases/spark-release-3-2-0.html. (Accessed on 01/26/2022).

[13] D. R. Engler, M. F. Kaashoek, and J. O'Toole. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. Association for Computing Machinery, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076

[14] Nick Feamster, Jennifer Rexford, and Ellen Zegura. 2014. The Road to SDN: An Intellectual History of Programmable Networks. *SIGCOMM Comput. Commun. Rev.* 44, 2 (apr 2014), 87–98. https://doi.org/10.1145/2602204.2602219

[15] Apache Software Foundation. 2022. Apache Arrow DataFusion Documentation. https://arrow.apache.org/datafusion/. (Accessed on 10/16/2022).

[16] Robbie Gruener, Owen Cheng, and Yevgeni Litvin. 2018. Introducing Petastorm: Uber ATG's Data Access Library for Deep Learning. https://eng.uber.com/petastorm/. (Accessed on 01/19/2022).

[17] Yanfei Guo, Jia Rao, and Xiaobo Zhou. 2013. iShuffle: Improving Hadoop Performance with Shuffle-on-Write. In *10th International Conference on Autonomic Computing (ICAC 13)*. USENIX Association, San Jose, CA, 107–117. https://www.usenix.org/conference/icac13/technical-sessions/presentation/guo

[18] Ajay Gupta. 2020. Revealing Apache Spark Shuffling Magic. https://medium.com/swlh/revealing-apache-spark-shuffling-magic-b2c304306142. (Accessed on 02/01/2022).

[19] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. 1997. Online Aggregation. *SIGMOD Rec.* 26, 2 (jun 1997), 171–182. https://doi.org/10.1145/253262.253291

[20] Jingui Li, Xuelian Lin, Xiaolong Cui, and Yue Ye. 2013. Improving the Shuffle of Hadoop MapReduce. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, Vol. 1. IEEE, Bristol, UK, 266–273. https://doi.org/10.1109/CloudCom.2013.42

[21] Frank Sifei Luan, Stephanie Wang, Samyukta Yagati, Sean Kim, Kenneth Lien, Isaac Ong, Tony Hong, SangBin Cho, Eric Liang, and Ion Stoica. 2023. Exoshuffle-CloudSort. https://doi.org/10.48550/ARXIV.2301.03734

[22] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* 38, 2 (mar 2008), 69–74. https://doi.org/10.1145/1355734.1355746

[23] Qi Meng, Wei Chen, Yue Wang, Zhi-Ming Ma, and Tie-Yan Liu. 2019. Convergence Analysis of Distributed Stochastic Gradient Descent with Shuffling. *Neurocomput.* 337, C (apr 2019), 46–57. https://doi.org/10.1016/j.neucom.2019.01.037

[24] Konstantin Mishchenko, Ahmed Khaled, and Peter Richtarik. 2020. Random Reshuffling: Simple Analysis with Vast Improvements. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., Virtual, 17309–17320. https://proceedings.neurips.cc/paper/2020/file/c8cc6e90ccbff44c9cee23611711cdc4-Paper.pdf

[25] Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala. 2019. Ludwig: a type-based declarative deep learning toolbox. https://doi.org/10.48550/ARXIV.1909.07930

[26] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 561–577. https://www.usenix.org/conference/osdi18/presentation/moritz

[27] Derek Gordon Murray. 2012. *A distributed execution engine supporting data-dependent control flow*. Ph. D. Dissertation. University of Cambridge.

[28] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 113–126.

[29] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. 2021. Tf.Data: A Machine Learning Data Processing Framework. *Proc. VLDB Endow.* 14, 12 (jul 2021), 2945–2958. https://doi.org/10.14778/3476311.3476374

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc., Red Hook, NY, USA. https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[31] Pedro Pedreira, Orri Erling, Maria Basmanova, Kevin Wilfong, Laith S. Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. *Proc. VLDB Endow.* 15, 12 (2022), 3372–3384. https://www.vldb.org/pvldb/vol15/p3372-pedreira.pdf

[32] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. 2020. Towards Scalable Dataframe Systems. https://doi.org/10.48550/ARXIV.2001.00888

[33] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[34] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[35] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. 2012. Sailfish: A Framework for Large Scale Data Processing. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California) *(SoCC '12)*. Association for Computing Machinery, New York, NY, USA, Article 4, 14 pages. https://doi.org/10.1145/2391229.2391233

[36] Alexander Rasmussen, Vinh The Lam, Michael Conley, George Porter, Rishi Kapoor, and Amin Vahdat. 2012. Themis: An I/O-Efficient MapReduce. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California) *(SoCC '12)*. Association for Computing Machinery, New York, NY, USA, Article 13, 14 pages. https://doi.org/10.1145/2391229.2391242

[37] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. 2011. TritonSort: A Balanced Large-Scale Sorting System. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 29–42.

[38] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference*, Kathryn Huff and James Bergstra (Eds.). Python in Science Conference, Austin, Texas, 130 – 136.

[39] Mehul A. Shah, Amiato, and Chris Nyberg. 2014. CloudSort: A TCO Sort Benchmark. http://sortbenchmark.org/2014_06_CloudSort_v_0_4.pdf. (Accessed on 01/24/2022).

[40] Min Shen. 2020. RPC implementation to support pushing and merging shuffle blocks. https://issues.apache.org/jira/browse/SPARK-32915. (Accessed on 10/16/2022).

[41] Min Shen, Ye Zhou, and Chandni Singh. 2020. Magnet: Push-Based Shuffle Service for Large-Scale Data Processing. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3382–3395. https://doi.org/10.14778/3415478.3415558

[42] Sergei Sokolenko. 2018. How Distributed Shuffle improves scalability and performance in Cloud Dataflow pipelines. https://cloud.google.com/blog/products/data-analytics/how-distributed-shuffle-improves-scalability-and-performance-cloud-dataflow-pipelines.
[43] PyTorch Team. 2022. torch.utils.data – PyTorch documentation. https://pytorch.org/docs/stable/data.html. (Accessed on 10/16/2022).
[44] Ray Team. 2022. Ray Datasets: Distributed Data Preprocessing. https://docs.ray.io/en/latest/data/dataset.html. (Accessed on 10/16/2022).
[45] Qian Wang, Rong Gu, Yihua Huang, Reynold Xin, Wei Wu, Jun Song, and Junluan Xia. 2016. NADSort. http://sortbenchmark.org/NADSort2016.pdf. (Accessed on 01/26/2022).
[46] Stephanie Wang. 2021. Analyzing memory management and performance in Dask-on-Ray. https://medium.com/distributed-computing-with-ray/analyzing-memory-management-and-performance-in-dask-on-ray-930a2236b70d. (Accessed on 01/26/2022).
[47] Stephanie Wang, Benjamin Hindman, and Ion Stoica. 2021. In Reference to RPC: It's Time to Add Distributed Memory. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 191–198. https://doi.org/10.1145/3458336.3465302
[48] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. 2021. Ownership: A Distributed Futures System for Fine-Grained Tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Virtual, 671–686. https://www.usenix.org/conference/nsdi21/presentation/cheng
[49] Yandong Wang, Cong Xu, Xiaobing Li, and Weikuan Yu. 2013. JVM-Bypass for Efficient Hadoop Shuffling. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, Cambridge, MA, USA, 569–578. https://doi.org/10.1109/IPDPS.2013.13
[50] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman (Eds.). Python in Science Conference, Austin, Texas, 56 – 61. https://doi.org/10.25080/Majora-92bf1922-00a
[51] Reynold Xin. 2014. Apache Spark the Fastest Open Source Engine for Sorting a Petabyte. https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html. (Accessed on 01/19/2022).
[52] M. Young, A. Tevanian, R. Rashid, D. Golub, and J. Eppinger. 1987. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles* (Austin, Texas, USA) *(SOSP '87)*. Association for Computing Machinery, New York, NY, USA, 63–76. https://doi.org/10.1145/41457.37507
[53] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia
[54] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (oct 2016), 56–65. https://doi.org/10.1145/2934664
[55] Kai Zeng, Sameer Agarwal, Ankur Dave, Michael Armbrust, and Ion Stoica. 2015. G-OLA: Generalized On-Line Aggregation for Interactive Analysis on Big Data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 913–918. https://doi.org/10.1145/2723372.2735381
[56] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J. Freedman. 2018. Riffle: Optimized Shuffle Service for Large-Scale Data Analytics. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 43, 15 pages. https://doi.org/10.1145/3190508.3190534
[57] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. 2021. Hoplite: Efficient and Fault-Tolerant Collective Communication for Task-Based Distributed Systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 641–656. https://doi.org/10.1145/3452296.3472897