Riffle: Optimized Shuffle Service for Large-Scale Data Analytics

Haoyu Zhang[★], Brian Cho[†], Ergin Seyfe[†], Avery Ching[†], Michael J. Freedman[★]

[★]*Princeton University* [†]*Facebook, Inc.*

ABSTRACT

The rapidly growing size of data and complexity of analytics present new challenges for large-scale data processing systems. Modern systems keep data partitions in memory for pipelined operators, and persist data across stages with wide dependencies on disks for fault tolerance. While processing can often scale well by splitting jobs into smaller tasks for better parallelism, all-to-all data transfer—called *shuffle* operations—become the scaling bottleneck when running many small tasks in multi-stage data analytics jobs. Our key observation is that this bottleneck is due to the superlinear increase in disk I/O operations as data volume increases.

We present Riffle, an optimized shuffle service for big-data analytics frameworks that significantly improves I/O efficiency and scales to process petabytes of data. To do so, Riffle efficiently merges fragmented intermediate shuffle files into larger block files, and thus converts small, random disk I/O requests into large, sequential ones. Riffle further improves performance and fault tolerance by mixing both merged and unmerged block files to minimize merge operation overhead. Using Riffle, Facebook production jobs on Spark clusters with over 1,000 executors experience up to a 10x reduction in the number of shuffle I/O requests and 40% improvement in the end-to-end job completion time.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Ultra-large-scale systems**; • **Applied computing → Enterprise data management**;

KEYWORDS

Shuffle Service, I/O Optimization, Big-Data Analytics Frameworks, Storage

ACM Reference Format:
Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, Michael J. Freedman. 2018. Riffle: Optimized Shuffle Service for Large-Scale Data Analytics. In *EuroSys '18: 13th European Conference on Computer Systems, April 23–26, 2018, Porto, Portugal.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3190508.3190534

EuroSys '18, April 23–26, 2018, Porto, Portugal

ACM ISBN 978-1-4503-5584-1/18/04.
https://doi.org/10.1145/3190508.3190534

1 INTRODUCTION

Large-scale data analytics systems are widely used in many companies holding and constantly generating big data. For example, the Spark deployment at Facebook processes 10s of PB newly-generated data every day, and a single job can process 100s of TB of data. Efficiently analyzing massive amounts of data requires underlying systems to be highly scalable and cost effective.

Data analytics frameworks such as Spark [57], Hadoop [1], and Dryad [31] commonly use a DAG of *stages* to represent data transformations and dependencies inside a *job*. A stage is further broken down to *tasks* which process different partitions of the data using one or more operations. Data transformations for grouping and joining data require all-to-all communication between *map* and *reduce* stages, called a *shuffle* operation. For example, a `reduceByKey` operation in Spark requires each task in the reduce stage to retrieve corresponding data blocks from all the map task outputs. Jobs that execute shuffle are prevalent—over 50% of Spark data analytics jobs executed daily at Facebook involve at least one shuffle operation.

The amount of data processed by analytics jobs is growing much faster than the memory available. At Facebook, data can be 10x larger than the total memory resource allocated to a job, and thus the shuffle intermediate data has to be kept on disks for scalability and fault tolerance purposes. The fast-growing data and complexity of analytics pose a fundamental performance tension in big-data systems.

Research work highly encourages running a large number of small tasks. Recent work [16, 32, 41–43] illustrates the benefit of slicing jobs into small tasks: small tasks improve the parallelism, reduce the straggler effect

with speculative execution, and speed up end-to-end job completion. Solutions have also been presented to minimize task launch time [37] as well as scheduling overhead [44] for a large number of small tasks.

However, engineering experience often argues against running too many tasks. In fact, large jobs processing real-world workloads observe significant performance degradation because of excessive shuffle overhead [4, 5, 14]. While the tiny tasks execution plan works well with single-stage jobs, it introduces significant I/O overhead during shuffle operations in multi-stage jobs. Engineers often execute jobs with fewer bulky, slow tasks to mitigate shuffle overhead, paying the price of stragglers and inefficient large tasks that do not fit in memory.

We observe that the root cause of the slowdown is due to the fact that the number of shuffle I/O requests between map and reduce stages *grows quadratically* as the number of tasks grows, and the average size per request actually *shrinks linearly*. At Facebook, data is preserved on spinning disks for fault tolerance, so a large amount of small, random I/O requests (e.g., 10s or 100s of KB) during shuffle leads to a significant slowdown of job completion and resource inefficiency. Executing jobs with large numbers of tasks over splits the I/O requests, further aggravating the problem. Thus, neither approach for tuning the number of tasks provides efficient performance at large scales.

We present Riffle, an optimized shuffle service for big-data analytics frameworks that significantly improves I/O efficiency and scales to processing PB-level data. Riffle boosts shuffle performance and improves resource efficiency by converting large amounts of small, random shuffle I/O requests into much fewer large, sequential I/O requests. At its core, Riffle consists of a *centralized scheduler* that keeps track of intermediate shuffle files and dynamically coordinates merge operations, and a *shuffle merge service* which runs on each physical cluster node and efficiently merges the small files into larger ones with little resource overhead.

Challenges and solutions. In designing Riffle, we had to overcome several technical challenges.

First, Riffle has to be efficient in handling shuffle files without using much computation or storage resources. Riffle overlaps the merge operations with map tasks, and always accesses large chunks of data sequentially with minimal disk I/O overhead when performing merge operations. To reduce the additional delay caused by stragglers, Riffle allows users to set a *best-effort merge* threshold, so that reducers consume some late-arriving intermediate outputs in unmerged form, together with the majority of outputs in merged form.

Second, Riffle should be easy to configure to best fit different storage systems and hardware. While merging files generally reduces the number of I/O requests, making the block sizes too large leads to only marginal improvement in I/O requests but slowdown in merge operations. Riffle explores the inherent tradeoff between maximizing the gain of large request sizes and minimizing the overhead of aggressive merges, and supports merge policies with different fan-ins and target block sizes, to get the best efficiency for disk I/Os and merge operations.

Third, Riffle must tolerate failures during merge and shuffle. Since failure is the norm at large scale, we must handle failures without affecting correctness or incurring additional slowdown in job execution. Riffle keeps track of intermediate files in both merged and unmerged forms, and on failure falls back to files in unmerged format within the scope of failure.

Finally, Riffle should not create prohibitive overhead. The merge operations of Riffle come at the cost of reading and writing more shuffle data for the merged intermediate files. Riffle makes this tradeoff a performance win, by issuing all merge requests as large, sequential requests, keeping the overhead significantly less than the savings. In terms of space, the intermediate files are soon garbage collected after job completion, so they occupy disk space only temporarily.

We implemented the Riffle shuffle service within the Apache Spark framework [3]. Riffle supports unmodified Spark applications and SparkSQL queries [19]. This paper presents the results of Riffle on a representative mix of Facebook's production jobs processing 100s of TB of data: Riffle reduces disk I/O requests by up to 10x and the end-to-end job completion time by up to 40%.

2 BACKGROUND AND MOTIVATION

The past several years has seen a rapid increase in the amount of data that is being generated and analyzed every day. Distributed data analytics engines, like Spark [57], MapReduce [22], Dryad [31], are widely used for executing SQL queries and user-defined functions (UDFs) on large datasets, or preprocessing and postprocessing in machine learning jobs. The key challenge in analyzing massive amounts of data arises from the fact that the volume and complexity of data processing is growing much faster than hardware speed and capacity improvements. Riffle aims to solve the problem at large scale by significantly improving the efficiency of hardware resource usage.

This section motivates and provides background for Riffle. §2.1 briefly reviews the DAG computation model commonly used in big-data analytics frameworks. §2.2 discusses the memory constraints of data processing, and the quadratic relationship between data volume and disk I/O during shuffle. §2.3 presents existing solutions to mitigate the problem, and explains why they fall short in fundamentally solving the problem at large scale.

(a) Logical operators. (b) DAG execution plan.

Figure 1: DAG representation of a Spark job, which `joins` data processed from two tables and uses `groupByKey` to aggregate the key-value items, then `filters` the data to get the final results.

2.1 Shuffle: All-to-All Communications

Data analytics frameworks typically use a DAG to represent the computation logic of a *job*, with stages as its vertices, and the dependencies between stages as its edges. A *stage* is further comprised of a set of tasks, each processing a partition of the datasets. A *task* typically includes a pipeline of one or more programmer specified operators that need to be applied to transform a data partition from input to output. Tasks in the first and last stages of a job are responsible to read in data from external sources (e.g., file systems, table storage, streams) and persist results, while tasks in the middle stages take the output generated by tasks in the previous stage as input, perform the transformation based on the specified operators, and then generate data for tasks in the next stage. Data dependencies thus can be classified in two types [57]: *narrow dependencies*, where the partition of data processed by a child task only depends on one parent task output, and *wide dependencies*, where each child task processes outputs from multiple or all parent tasks.

For example, Figure 1(a) is a logical view of a Spark job. It applies transformations (`map` and `filter`) on data from two separate tables, `joins` and aggregates the items over each key (a certain field of items) using `groupByKey`. After `filtering`, it stores the output data in the result table. Figure 1(b) shows the Spark execution plan of this job. For narrow dependencies (`map` and `filter`), Spark pipelines the transformations on each partition and performs the operators in a single stage. Internally, Spark tries to keep the intermediate data of a single task in memory (unless the size of data cannot fit), so the pipelined operators (a `filter` operator following a `map` operator in Stage 1) can be performed efficiently.

Spark triggers an all-to-all data communication, called *shuffle*, for the wide dependency between Stages 1 (map) and 2 (reduce). Each map task reads from a data partition (e.g., several rows of a large table), transforms the data into the intermediate format with the map task operators, sorts or aggregates the items by the partitioning function of the reduce stage (e.g., key ranges) to produce *blocks* of items, and saves the blocks to on-disk intermediate files. The map task also writes a separate *index file* which shows the offsets of blocks corresponding to each reduce task. To organize reduce stage data with `groupByKey`, each reduce task brings together the designated data blocks and performs reduce task operators. By looking up the offsets in index files, each reduce task issues fetch requests to the target blocks from all the map output files. Thus, data that was originally partitioned according to table rows are processed and shuffled to data partitioned according to reduce key ranges.

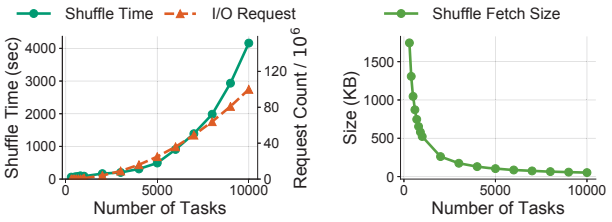
The large amount of intermediate files, written by the map tasks and read by the subsequent reduce tasks, are persisted on disks in both Spark and MapReduce for fault tolerance purposes. For large jobs, 10s to 100s of TB, or even PB of data are generated during each shuffle. Between stages with wide dependencies, each reduce task requires reading data blocks from all the map task outputs. If the intermediate shuffle files were not persisted, even a single reduce task failure could lead to recomputing the entire map stage. In fact, failure of tasks or even cluster nodes is the norm at large scale deployment of big-data frameworks [30, 34, 52], so it is crucial to persist shuffle data for strong fault tolerance.

As a result, shuffle is an extremely resource intensive operation. Each block of data transferred from a map task to a reduce task needs to go though data serialization, disk and network I/O, and data deserialization. Yet shuffle is heavily used in various types of jobs—those requiring data to be repartitioned, grouped or reduced by key, or joined all involve shuffle operations. At Facebook, we observe that over 50% of our daily batch analytics jobs have at least one shuffle operations. A key approach to better completion time and resource efficiency of these jobs is improving the performance of shuffle operations.

2.2 Efficient Storage of Intermediate Data

Even though there is a trend towards keeping data in memory wherever possible to improve resource efficiency [2, 21, 23, 35], in real-world settings the amount of data is growing much faster than the available memory, which makes it infeasible to keep the data entirely in memory. For example, a job at Facebook processes data that is over 10x larger than the allocated resources. Instead intermediate data must be pushed to permanent storage for scalability and fault tolerance.

At Facebook, the current generation of warehouse clusters use HDDs for permanent storage. For large amount of data, this is significantly more cost effective than SSDs given current hardware [27, 33]. With spinning HDDs, the number of IOPS (I/O Operations Per Second) available is a limiting factor for the system throughput. While HDDs continue to grow in capacity, the available IOPS will not increase accordingly due to the physical limits of mechanical spin time [55]. Thus, we must be careful to use IOPS wisely for intermediate data, both for disk spills and shuffles.



(a) Shuffle time and I/O requests. (b) Average I/O request size.

Figure 2: When the number of tasks in each stage grows, the shuffle time and the number of I/O requests increase quadratically, and the average shuffle fetch size in each request decreases.

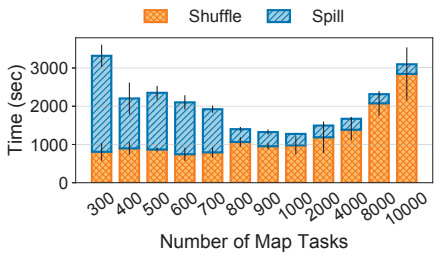


Figure 3: Shuffle-spill tradeoff when varying number of map tasks (with fixed number of reduce tasks). Bulky tasks (left) incur more spill overhead, while tiny tasks (right) incur significant shuffle overhead.

Disk spill I/O. When the size of the data partition assigned to a task exceeds the memory limit, the task has to *spill* intermediate data to permanent storage. Disk spills can incur a significant amount of additional overhead because of the increasing disk I/O and garbage collection.

For example, assume that a map task processes a partition of 4GB input data, and runs with 8GB memory.[1] Data have to be decompressed and deserialized from disks to get the in-memory objects. This process effectively enlarges the original data, in practice, by about 4x. Thus, reading and processing 2GB input data already consumes the entire memory. The map task has to perform the operations and sort the result items by keys according to the reduce partition function. To do so, it (1) reads in the first 2GB data, performs the computation, and spills a temporary output file on disk; (2) similarly, reads, processes, and spills the second 2GB data; (3) merges the two temporary files into one using external merge sort. The overhead of repeated disk I/O and serialization significantly slows down the task execution and harms resource efficiency.

Shuffle I/O. To avoid disk spills, the task input size (S) should be appropriate to fit in memory, and thus is determined by the underlying hardware. As the size of job data increases, the number of map (M) and reduce (R) tasks has to grow proportionally. Because each reduce task needs to fetch from all map tasks, the number of shuffle I/O requests $M \cdot R$ increases *quadratically*, and the average block size $\frac{S}{R}$ for each fetch *decreases* linearly.

Figure 2 shows the job completion time when we keep the task input size fixed at 512MB (incurring no disk spills), and increase the number of tasks in both stages from 300 to 10,000. We see that the shuffle time grows quadratically from 100 to over 4,000 seconds. This is because the number of shuffle fetch requests increases rapidly (30K to 100M), as the average size of each request shrinks (1.7MB to 50KB).

Since disks are especially inefficient in handling large amounts of small, random I/O requests, jobs suffer a severe slowdown at large scale. Our goal is to improve the efficiency by reducing the IOPS requirement of the underlying storage systems for large-scale data analytics.

2.3 Current Practices & Existing Solutions

Several solutions have been previously proposed to mitigate the problem of large amounts of small, random I/O requests during shuffle. We discuss the limitations of these solutions, and explain why they fall short in fundamentally solving the problem at large scale.

Reducing the number of tasks per stage. By tuning the number of tasks in job execution plan, engineers look for the optimal performance trading off between shuffle and spill I/O efficiency [5]. Since the number of I/O requests is determined by the corresponding map and reduce tasks, using fewer tasks reduces the total number of shuffle fetches, and thus improves the shuffle performance. However, this approach inevitably enlarges the average size of input data and creates very bulky, slow tasks with disk spilling overhead.

For example, Figure 3 shows how the shuffle and spill runtime changes when varying the number of map tasks in a job processing 3TB data. Towards the left, smaller number of tasks implies larger task partition sizes, making the shuffle operations more efficient. At the same time, larger tasks also mean each task needs to spill more data, slowing down the task completion time. In this case, at around 1,000 tasks the job reaches its optimal value in terms of the total runtime of shuffle and spill.

However, tuning the number of tasks is untenable to apply across the thousands of jobs at Facebook. Each job has different characteristics (e.g., distribution and skew of data), so it is not possible to find the optimal point without tedious experimentation. In addition, data characteristics change over

[1] In practice, only a portion of memory can be used to cache data and the remaining is reserved by the runtime and program. The example ignores this discussion for simplicity.

time, depending on outside factors such as Facebook user behavior. Jobs are typically configured in favor of having more tasks, which allows room for data growth.

More importantly, the effects of having a small number of bulky tasks can be very detrimental for job execution in production: such tasks run very slowly due to additional I/O and garbage collection overhead [42]. In practice we see that task number tuning could assign GBs of data to a single task, causing the tasks to run over 60 minutes. Bulky tasks amplify the straggler problem, in that jobs get significantly delayed if a few tasks become stragglers or retry after failure, and speculative execution can only provide limited help in these cases [16, 43].

Aggregation servers for reducers. Another solution is to use separate aggregation processes in front of each reducer to collect the fragmented shuffle blocks and batch the disk I/O for shuffle data. The in-memory buffering in the aggregators ensures sequential disk access when writing shuffle data, which can later be read by reduce tasks all at once. However, directly applying this approach to process 100s of TBs or PBs of data is still infeasible. One aggregator instance per reduce task could consume a large amount of computation (for task bookkeeping) and memory (for disk I/O buffering) resources for large jobs, so the solutions can only be applied at relatively small scale [47]. In addition, because each reduce task collects data from all the map tasks, even failure of a single aggregation process leads to data corruption and requires the entire map stage to be recomputed. As jobs further scale in number of processes and runtime, the frequency of aggregation process failures (due to machine or network failures, etc.) increases. The high cost of failure recovery makes the solution inadequate for deployment at large scale.

To improve Hadoop shuffle performance, Sailfish [45] leverages a new file system design to support multiple insertion points to store aggregated intermediate files. Besides the fact that it requires modification to file systems, the solution also impairs the fault tolerance: to recover a single corrupted aggregation file, a large number of map tasks need to be re-executed. Compromising fault tolerance leads to frequent re-computation and thus harms system performance at Facebook scale.

Instead of trading fault tolerance for I/O efficiency, our goals of designing an optimized shuffle service include highly efficient shuffle I/O performance, little resource overhead to the clusters, and no additional failures caused by the shuffle optimization. Riffle provides its service as a long-running process on each physical node, and requires much less memory space and almost no computation overhead compared to existing solutions. Riffle operates on persisted disk files and saves results as separate files, so the service failures will not lead to any recomputation of stages or tasks. In the rest of the paper, we will show how Riffle's design and implementation meet these design goals in detail.

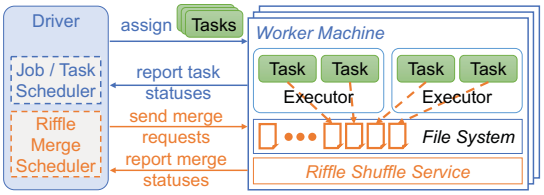


Figure 4: Riffle runs a shuffle merge scheduler as part of the analytics framework driver, and a merger instance per physical node. Since a physical node is typically sliced into a few executors, each running multiple tasks, it's common to have hundreds of tasks per job executed on each node.

3 SYSTEM OVERVIEW

Riffle is designed to work with multi-stage jobs running on distributed data analytics frameworks that involve all-to-all data shuffle between different stages. We describe how Riffle works with cluster managers and data analytics frameworks, as shown in Figure 4.

Shuffle merge scheduler. Tasks in data analytics frameworks are assigned by a global driver program. As explained in §2, the driver converts a data processing job to a DAG of data transformations, with several stages separated by shuffle operations. Tasks from the same stage can be executed in parallel on the executors, while tasks in the following stage typically need to be executed after the shuffle. The intermediate shuffle files are persisted on local or distributed file systems (e.g., HDFS [49], GFS [25], and Warm Storage [8]).

Riffle includes a shuffle merge scheduler on the driver side, which keeps track of task execution progress and schedules merge operations based on configurable strategies and policies. In practice, it is common to have hundreds of tasks assigned per physical node in processing large-scale jobs. The Riffle scheduler collects the state and block sizes of intermediate files generated by all tasks, and issues merge requests when the shuffle files meet the merge criteria (§4.1).

Shuffle service with merging. Data analytics frameworks provide an external shuffle service [10, 12] to manage the shuffle output files. A long-running shuffle service instance is deployed on each worker node in order to serve the shuffle files uninterruptedly, even if executors are killed or reallocated to other jobs running concurrently on the cluster with dynamic resource allocation policies [26, 29]. Riffle runs a merger instance as part of the shuffle service on each

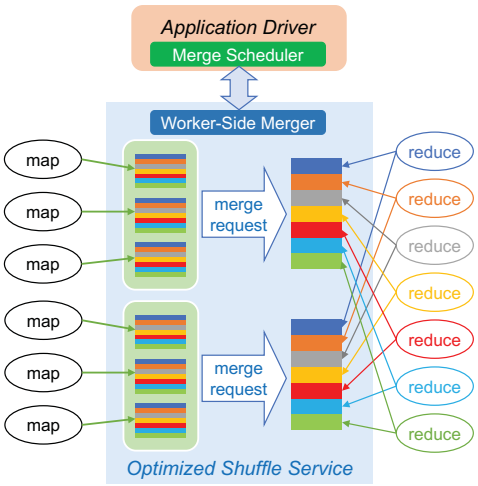


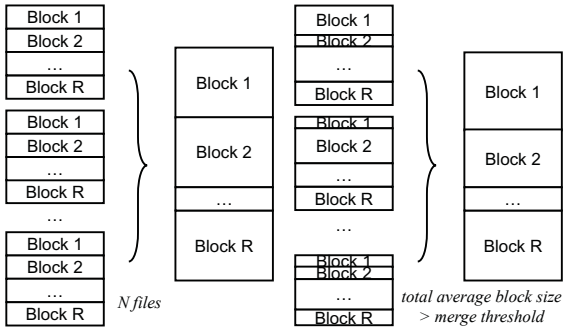
Figure 5: Merging intermediate files with Riffle.

physical node, which performs merge operations on shuffle output files.

The shuffle merge scheduler directly communicates with all the registered merger instances where some of the job tasks are executed, to send out merge requests and collect results from the mergers. Figure 5 illustrates the shuffle service side merger combining multiple intermediate shuffle files into larger files. Each mapper outputs data such that items are partitioned into the reducer it belongs to (indicated here by color). Without Riffle, each reducer would read partitions from all map outputs, which can be on the order of tens of thousands per reducer. Riffle merges the shuffle files block by block to preserve the reducer partitioning. After the merge operations, a reducer only needs to fetch a significantly smaller number of large blocks from the merged intermediate files instead. Note that these merge operations are performed on compressed, serialized data files. This process significantly improves the shuffle I/O efficiency without incurring much resource overhead.

4 DESIGN

This section describes the mechanisms by which Riffle addresses its key challenges. We explain the merge strategies and policies in the driver side scheduler, and the execution of merge operations in the worker side merger in §4.1. We discuss how Riffle minimizes the merge overhead with best-effort merging (§4.2), handles merge failures (§4.3), and balances merge requests using power of choices in disaggregated architecture (§4.4). We analyze Riffle's performance benefit in §4.5.



(a) Greedy N-way Merge. (b) Fixed Block Size Merge.

Figure 6: Riffle merge policies.

4.1 Merging Shuffle Intermediate Files

Riffle is designed to work with existing data analytics frameworks by introducing shuffle merge operations in the shuffle service instances coordinated by the driver. Specifically, Riffle builds additional communication channels between the scheduler and mergers, allowing the driver to issue requests and coordinate mergers.

The merge scheduler starts merge operations immediately as map outputs become available, according to merge policies (§4.1.1). This causes most merges to overlap with the ongoing map stage, *hiding* their merge time if they finish before the map stage. When the map stage finishes, outstanding merge requests can incur additional delay, which makes policy configuration and merger efficiency (§4.1.2) important.

After the map tasks and merge operations finish, the driver launches reduce tasks in the subsequent stage, and broadcasts the metadata (location, executor id, task id, etc.) of all the map outputs to the executors hosting reduce tasks. With Riffle, the driver sends out metatdata of the merged files instead of the original map output files, so the reduce tasks can fetch corresponding blocks from the merged files with more efficient reads.

4.1.1 *Merge Scheduling Policies*

Merge with fixed number of files. Users can configure Riffle to merge a fixed number of files. For N-way merge, the scheduler sends a merge request to the merger whenever there are N map output files available on that node (Figure 6(a)). The merger, upon receiving this request, performs the merge by reading existing shuffle files, grouping blocks based on reduce partitions, and generating a new pair of shuffle output file and index file as the merge result.

Merge with fixed block size. In real-world settings, we observe a large variance in block sizes of the shuffle output files (Figure 6(b)). Some shuffle blocks themselves are large enough, leading to few fragmented reads; some are very tiny

Algorithm 1 Merging Intermediate Shuffle Files

- *files*: shuffle files to be merged in request
- *index_files*: accompanying index files, which has offsets of shuffle file blocks corresponding to each reduce task
- *out_file*: merged shuffle file
- *out_index*: index file for the merged shuffle file
- *offset*: integer tracking offset of merged file

```
1:  function MergeShuffleFiles(in_ids, out_id)
2:      for all id in in_ids do
3:          files[id] = OpenWithAsyncReadBuffer(id)
4:          index_files[id] = CachedIndexFile(id)
5:      out_file = OpenWithAsyncWriteBuffer(out_id)
6:      out_index = NewIndexFile(out_id)
7:      offset = 0
8:      for p = 1.. number of reduce partitions do
9:          for all id in in_ids do
10:             start = index_files[id].GetOffset(p)
11:             end = index_file[id].GetOffset(p + 1)
12:             length = end − start
13:             BufferedCopy(out_file, offset, files[id], start, length)
14:             offset = offset + length
15:         AppendIndex(out_index, offset)
16:     FlushBufferAndClose(out_file)
17:     PersistIndexFile(out_index)
18:     return out_file, out_index
```

and we need to merge tens or hundreds of them to make shuffle reads efficient. Riffle also supports fixed block size merge. In this case, the driver sends out a merge request when the accumulated average shuffle block size across all partitions exceeds a configurable threshold. The Riffle scheduler avoids merging files that already have large blocks, and merges more files with tiny blocks for better I/O efficiency.

Configuring the merge policy. While merging files generally leads to more efficient shuffle, merging too aggressively can exacerbate the merge operation delay. Merge request processing is limited by the disk writing speed. For example, Riffle mergers achieve nearly the sequential speed at about 100MB/s when writing the merged files in our current deployment. The larger the merged output file is, the longer the merge operation will take. Riffle's file and block size based policies provide flexibility to trade off between shuffle and merge efficiency on a per-job basis.

In addition, these policies allow Riffle to be applied to file systems with different I/O characteristics. For example, if a file system provides 2MB unit I/O size, larger requests will be split into multiple 2MB chunk reads. Merging aggressively to get gigantic block files only provides marginal benefits for shuffle reads. In this case, Riffle's merge policy can be configured to a lesser number of files or smaller block size.

4.1.2 *Efficient Worker-Side Merger*

Upon receiving a merge request, the worker-side merger performs the merge operation and generates new shuffle files, as shown in Algorithm 1. A merge request includes a list of completed map task IDs. The merger locates the shuffle files previously generated by those tasks, and their accompanying index files which contain offsets of file blocks corresponding to individual reduce tasks. For each shuffle file, the merger allocates a buffer for asynchronously reads and caches its index file (normally no larger than a few tens of KB) in memory. The merger also allocates a separate buffer to asynchronously write the merged output file. During merge, it goes through each reduce partition, asynchronously copies over the corresponding blocks from all specified files into the merged file, and records the offsets in the merged index.

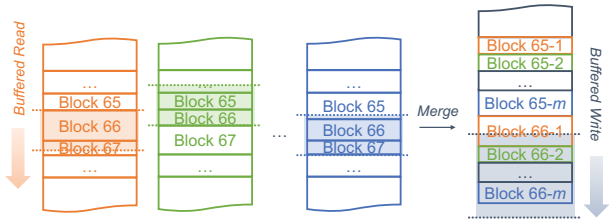


Figure 7: Riffle mergers trigger only sequential disk I/O for efficiency. The shadow sections of the input and output files are asynchronously buffered in memory to ensure sequential I/O behavior.

Riffle ensures the merge operation is efficient and lightweight on the worker side. First, Riffle merges compressed, serialized data files in their raw format on disks, incurring minimal computation overhead during merge. Second, the mergers prefetch data from original shuffle files, aggregate the blocks belonging to same reducers, and asynchronously write blocks into the result file. Thus, they always read and write large chunks of data sequentially, leading to minimal disk I/O overhead when performing merge operations.

Memory Management. The major resource overhead on the workers comes from the in-memory buffers for reading the original shuffle output files and writing the merged file, as shown in Figure 7. Buffering files ensures large, sequential disk I/O requests, at the cost of more memory consumption when the number of files and the number of concurrent merge operations grow.

For example, assume that we keep a 4MB read buffer and a 20MB write buffer. To merge 20 shuffle files, the merger has to buffer 80MB data for all input files, and 20MB for the output file, ending up consuming 100MB memory. Using a dedicated buffer for each file parallelizes the reads and writes and accelerates the merge speed. However, since a merger is responsible to handle hundreds of map output files per job generated by tens of executors on the worker node, the memory consumption can be significant when handling a large number of concurrent merge requests.

Riffle deploys mergers with a fixed memory allocation on each physical node. Upon receiving a new merge request, the merger estimates the memory consumption of processing the request based on the fan-in (i.e., number of files) and average block sizes, and only starts the operation if there is enough memory. When exceeding the memory limit, new incoming merge requests will be queued up and waiting for the memory to become available. We find that allocating 6–8GB of memory to a merger is sufficient to process 10–20 concurrent merge requests in most use cases.[2] With this configuration, Riffle mergers can achieve nearly sequential disk I/O speed when writing merged files. Given that each physical node typically has 256GB or even larger memory in modern datacenters, and tens of GB of memory per machine is reserved for OS and framework daemons, we consider the memory overhead of Riffle acceptable.

4.2 Best-Effort Merge

When processing large-scale jobs with Riffle, there are usually some merger processes still working on performing merge operations while most of the other mergers have already completed the assigned requests. These *merge stragglers* exist mainly for two reasons. First, there are always shuffle files that are generated by the final few map tasks, and the late merge operations need to wait for these tasks to complete before starting to merge. Second, the mergers on the worker nodes could also crash and get restarted, which slows down the pending merge requests on that node. We find that when deployed at large scale, Riffle merge stragglers can sometimes significantly increase the end-to-end job completion time.

To alleviate the delay penalty caused by stragglers, we introduce *best-effort merge*, which allows the driver to mark the map stage as finished and start to launch reduce tasks when *most* merge operations are done on workers. Riffle allows users to configure a percentage threshold, and when the completed merge operations exceed this threshold, the driver does not wait for additional merge requests to return. The job execution directly proceeds to the reduce stage, and all pending merge operations are cancelled by the driver to save resources.

When using best-effort merge, the Riffle driver sends to reducers the metadata for merged shuffle files for successful merge operations, and the metadata of original unmerged files for cancelled merge operations. By eliminating merge stragglers, best-effort merge improves the end-to-end job completion time as well as the overall resource efficiency despite a small portion of shuffle fetches being done on less efficient unmerged files. We demonstrate this improvement in §6.2.

[2] The memory allocation of the merger determines the number of concurrent requests it can handle. In general, increasing the memory space leads to higher merge throughput, until a certain point where the effective disk output rate becomes a limiting factor.

4.3 Handling Failures

Since failure is the norm at scale, Riffle must guarantee the correctness of computation results, and should not slow down the recovery process when failures happen. This requires Riffle to efficiently handle both merge operation failures and loss of shuffle files. To handle these cases well, Riffle keeps both the original, unmerged files as well as the merged files on disks.

A merge operation can fail if the merge service process crashes, or merging takes too long and the request times out. When that happens, Riffle is designed to fall back to original unmerged files in similar manner to best-effort merge. This leads to a slight performance degradation during shuffle, while avoiding delaying the map stage. Correctness is guaranteed in the same way as best-effort merge, by the Riffle driver sending a mixture of metadata for merged and unmerged files to reduce tasks.

Spark and Hadoop deal with shuffle data loss or corruption by recomputing only the map tasks that generated the lost files. Riffle follows this strategy if unmerged files are lost, but can recover faster if only merged files are lost. For lost merged files, the original shuffle file is used as a fallback, avoiding any recomputations in the map stage while slightly degrading shuffle by fetching more files. Note that this is different from previous solutions using aggregators to collect data on the reducer side. Sailfish [45] modifies the underlying file system with a new file format that supports multiple insertion points for reduce block aggregation. However, a data loss which involves a single chunk of the aggregated file requires re-execution of all map tasks which appended to that chunk. Thus, data losses can lead to heavy recomputation for the tasks in the map stage, and it falls short to meet our key requirement of efficient failure handling.

4.4 Load Balancing on Disaggregated Architecture

Recent development in datacenter resource disaggregation [7, 24, 36] replaces individual servers with a rack of hardware as the basic building block of computing. The new-generation disaggregated architecture provides efficiency through gains in flexibility, latency, and availability. At Facebook, disaggregated clusters are widely used: the compute nodes (with powerful CPUs and memory) and storage nodes (with weaker CPUs and large disk space) on separate racks. The distributed file system abstracts away the physical file locations, and leverages fast network connections to achieve high I/O performance across all storage nodes. While deploying a data analytics framework such as Spark on the

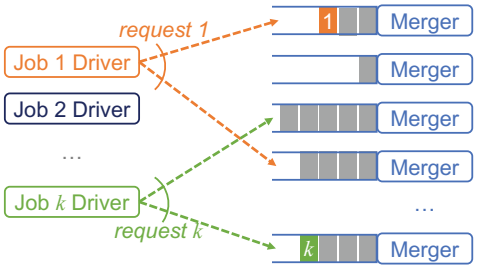


Figure 8: Multiple Riffle jobs on a disaggregated architecture balances the merge requests leveraging the power of two choices.

disaggregated clusters, all workers experience nearly homogeneous rates reading and writing files regardless of their physical locations in the storage nodes.

Riffle on disaggregated clusters runs one merger process on each compute node. In the context of resource disaggregation, merge operations are no longer limited to work with "local" shuffle files generated from the same physical nodes. In fact, the driver can send a request to an arbitrary merger to merge a number of available shuffle output files generated by multiple executors on different physical nodes. For example, when the fixed block size policy is used, the driver will pick a merger and send out a merge request whenever the accumulated average block sizes of shuffle files generated by all workers exceed the minimum merging block size.

Because of the merger memory limits, merge requests can queue up when the cluster experiences high workload (as described in §4.1.2). Note that the mergers, located on the physical nodes, are shared across all concurrent jobs running on the cluster. The Riffle enabled drivers need to consider the workload of mergers when sending out their requests, so that the merge operations are balanced among the mergers.

In order to efficiently balance the dynamic merge workload in a distributed manner, Riffle leverages "power of two choices" [40]. As shown in Figure 8, each driver only needs to query the pending merge workload of two (or a few) randomly picked mergers and choose the one with the shortest queue length. Theoretical analysis and experiments [38, 44] show that the approach can efficiently balance the distributed dynamic requests while incurring little probing overhead.

4.5 Discussion

Analysis of I/O operation savings. Assume a two-stage job has M map tasks and R reduce tasks. The total amount of data it processes is T. To simplify the discussion, we assume the partitions of processed by all tasks can fit in memory (i.e., no disk spills). With unmodified shuffle, the number of total shuffle I/O requests is $M \cdot R$.

Using N-way complete merge, $\frac{M}{N}$ merged files are generated by the mergers. During shuffle, each reducer only sends $\frac{M}{N}$ read requests. Assuming data is evenly partitioned, the total shuffle I/O requests during is now $\frac{M}{N} \times R$.

Merge operations also trigger additional I/O. Specifically, a complete merge of all intermediate files requires an additional read and write of T data. Since Riffle mergers only incur sequential disk I/O, the total number of I/O requests is $2 \cdot \frac{T}{s}$, where s is the buffer size in the Riffle mergers. Putting them together, the total number of I/O requests is

$$\frac{M}{N} \times R + 2 \cdot \frac{T}{s}$$

For example, assume a job processing 100GB data uses 1,000 map tasks and 1,000 reduce tasks. It triggers 1,000,000 I/O requests during shuffle. If the Riffle merger uses 10MB I/O buffers, then with 40-way merge, the total number of I/O requests becomes $\frac{1000}{40} \times 1000 + 2 \times \frac{100GB}{10MB}$ =45,000, reduced by 22x.

This calculation does not consider the effect of disk spills. In fact, Riffle's efficient merge alleviates the quadratic increase of shuffle I/O. Thus users can run much smaller tasks instead of bulky tasks, which further reduces disk IOPS requirement due to less spills.

Note that the amount of additional I/O incurred by Riffle is similar to that required in Sailfish [45]. More specifically, the *chunkservers* and *chunksorters* in Sailfish also need to make a complete pass reading and writing shuffle data to reorganize the key-values and generate new index files. Both systems move this process off the critical path to unblock the execution of map and reduce tasks. Riffle's configurable merge policy and best-effort merge mechanism further minimize the merge overhead. In contrast, ThemisMR [46] provides exactly twice I/O property. Compared with Riffle and Sailfish, it completely avoids materializing intermediate files to disks, at the cost of impaired fault tolerance. Thus, the solution only applies to relatively small scale deployment.

Deployment on different clusters. Riffle works best when there are multiple executors processing tasks on each physical machine. As computing nodes getting larger and more powerful, it is desirable to slice them into smaller executors for efficient resource multiplexing (i.e., shared by multiple concurrent jobs) and failure isolation. In addition, Riffle fits well with recent research and industry trends in resource disaggregation, where merge operations are no longer limited to "local" files (§4.4). Large jobs running on small machines can still benefit from Riffle: in this case, tasks in map stage come in *waves*, ending up with many shuffle files on each physical node to merge.

5 IMPLEMENTATION

We implemented Riffle with about 4,000 lines of Scala code added to Apache Spark 2.0. Riffle's modification is completely transparent to the high-level programming APIs, so it supports running unmodified Spark applications. We implemented Riffle to work on both traditional clusters with collocated computation and storage, and the new-generation disaggregated clusters. Riffle as well as its policies and configurations can be easily changed on a per-job basis. It is currently deployed and running various Spark batch analytics jobs at Facebook.

Garbage collection. Storage space, compared to other resources, is much cheaper in the system. As described in §4.3, Riffle keeps both unmerged and merged shuffle output files on disks for better fault tolerance. Both types of shuffle output files share the lifetime of the running Spark job, and are cleaned up by the resource manager when the job ends.

Correctness with compressed and sorted data. Compression is commonly used to reduce I/O overhead when storing files on disks. The data typically needs to go through compression codecs when transforming between its on-disk format and in-memory representation. Riffle concatenates file blocks directly in their compressed, on-disk format to avoid compression encoding and decoding overhead. This is possible because the data analytics frameworks typically use concatenation friendly compression algorithms. For example, LZ4 [9] and Snappy [11] are commonly used in Spark and Hadoop for intermediate and result files.

Merging the raw block files breaks the relative ordering of the key-value items in the blocks of merged shuffle files. If a reduce task does require the data to be sorted, it cannot assume the data on the mapper side is pre-sorted. Sorting in Spark (default) and Hadoop (configurable) on reduce side uses the TimSort algorithm [13], which takes advantage of the ordering of local sub-blocks (i.e., segments of the concatenated blocks in merged shuffle files) and efficiently sorts them. The algorithm has the same computational complexity as Merge Sort and in practice leads to very good performance [6]. The sorting mechanism ensures that reducer tasks will get the correctly ordered data even with the Riffle merge operations. In addition, since merge will not affect the internal ordering of data in sub-blocks (i.e., sorted regions in map outputs), the sorting time using TimSort with Riffle will be the same as the no merge case.

6 EVALUATION

In this section, we present evaluation results on Riffle. We demonstrate that Riffle significantly improves the I/O efficiency by increasing the request sizes and reduces the IOPS requirement on the disks, and scales to process 100s of TB of data and reduces the end-to-end job completion time and total resource usage.

	Data	Map	Reduce	Block
1	167.6 GB	915	200	983 K
2	1.15 TB	7,040	1,438	120 K
3	2.7 TB	8,064	2,500	147 K
4	267 TB	36,145	20,011	360 K

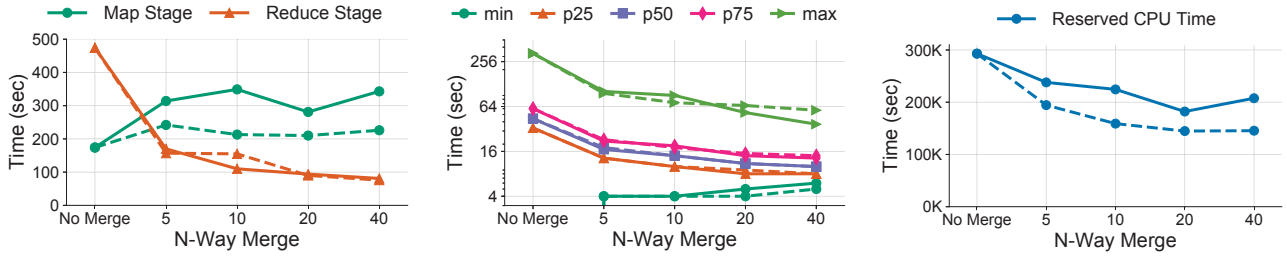
Table 1: Datasets for 4 production jobs used for Riffle evaluation. Each row shows the total size of shuffle data in a job, the number of tasks in its map and reduce stages, and the average size of shuffle blocks.

6.1 Methodology

Testbed. We test Riffle with Spark on a disaggregated cluster (see §4.4). The computation *blade* of the cluster consists of 100 physical nodes, each with 56 CPU cores, 256GB RAM (with 200GB allocated to Spark executors), and connected with 10Gbps Ethernet links. Each physical node is further divided into 14 executors, each with 4 CPU cores and 14 GB memory. In total, the jobs run on 1,414 executors. 8GB memory on each physical node is reserved for in-memory buffering of the Riffle merger instance. The storage *blade* provides a distributed file system interface, with 100MB/s I/O speed for sequential access of a single file. Our current deployment of file system supports 512KB unit I/O operation. We also use emulated IOPS counters in the file system to show the performance benefit when the storage is tuned with larger optimal I/O sizes.

Workloads and datasets. We used four production jobs at Facebook with different sizes of shuffle data, representing small, medium and large scale data processing jobs, as shown in Table 1. To isolate the I/O behavior of Riffle, in §6.2 we first show the experiment results on synthetic workload closely simulating Job 3: the synthetic job generates 3TB random shuffle data and uses 8,000 map tasks and 2,500 reduce tasks. With vanilla Spark, each shuffle output file, on average, has a $3TB/8000/2500 = 150KB$ block for each reduce task (approximating the 147KB block size in Job 3). Without complex processing logic, experiments with the synthetic job can demonstrate the I/O performance improvement with Riffle. We further show the end-to-end performance with the four production jobs in §6.3.

Metrics. Shuffle performance is directly reflected in the reduce task time, since each reduce task needs to first collect all the blocks of a certain partition from shuffle files, before it can start performing any operations. To show the performance improvement of Riffle, we focus on measuring (i) task, stage, and job completion time, (ii) reduction in the



(a) Stage runtime. (b) Reduce task runtime. (c) Reserved CPU time.

Figure 9: Riffle performance improvement in runtime with synthetic workload. 9(a) and 9(b) show the wall clock time to complete stages and tasks, and 9(c) plots the total reserved CPU time representing the job resource efficiency. Map time includes time to execute both map tasks and Riffle merge operations. Reduce time includes time to perform both shuffle fetch and reduce tasks. No complex data processing is in the synthetic applications, so shuffle fetch dominates the reduce time. Dashed lines show the performance with best-effort merge.

number of shuffle I/O requests, and (*iii*) the total resource usage in terms of reserved CPU time and estimated disk IOPS requirements.

Baseline. In the experiments with the synthetic workload, we compare the time and resource efficiency of Riffle with different merge policies. In the experiments with real-world workloads, we compare the performance improvement of Riffle against the engineering tuned execution plans (numbers of map and reduce tasks in Table 1) that have best shuffle-spill tradeoffs with vanilla Spark.

6.2 Synthetic Workload

6.2.1 *Stage and Task Completion Time*

We compare the performance improvement of Riffle when doing 5-way, 10-way, 20-way and 40-way merge, respectively. The merged shuffle files will on average get 750KB, 1.5MB, 3MB and 6MB block sizes.

Map and reduce stage execution time. Figure 9(a) shows the map and reduce stage completion time when running the job with vanilla Spark ("no merge") vs. Riffle with different merge policies (note the log scale on x-axis). As N grows, the merge operation generates larger block files, yet also takes longer time to finish. Since Riffle merge operations block the execution of reduce stage, map is only considered as completed when the merge is done. We see the map stage time increases gradually from 174 to 343 seconds. Despite the delay in map, we have a much larger reduction in the reduce stage time, which drops from 474 to 81 seconds. Overall, the job completion time (i.e., sum of the two stages) drops from 648 down to 424 seconds, 35% faster.

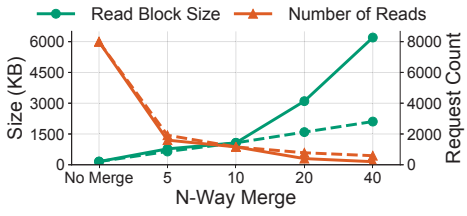
Improvement with best-effort merge. Riffle uses best-effort merge mechanism (§4.2) to further reduce the delay penalty of merge operations. In Figure 9(a), the dashed lines show the results of best-effort merge (threshold = 95%). We can see the map stage overhead, compared to full merge, is much smaller (343 down to 226 seconds with 40-way merge), while the reduce stage time stays almost the same. Overall, the job completes 53% faster compared with vanilla Spark.

To better understand the reduce stage time improvement, we break down the stage time by plotting the distribution of all task completion time. We show the minimum, 25/50/75 percentile, and maximum for different merge policies in Figure 9(b) (note the log scale of both axes). Similarly, results with best-effort merge are in dashed lines. The medium task time is reduced from 44 seconds (no merge) down to 10 seconds (40-way merge). The improvement comes from the fact that a reduce task only has to issue hundreds of large reads, as opposed to thousands of small reads, after the merge.

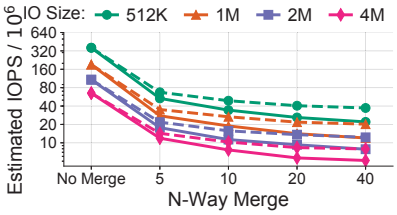
6.2.2 *Improvement in Resource Efficiency*

We measure the resource efficiency via metrics reported by the cluster resource manager. Figure 9(c) shows the total *reserved CPU time*. When merge is disabled, the entire job takes 293K reserved CPU seconds to finish; with over 20-way merge, the reserved CPU time is reduced to 207K seconds, or by 29%. In addition, when we enable best-effort merge, the saving in job completion time is also reflected in the resource efficiency—the total reserved CPU time is further decreased down to 145K seconds. That means we can finish the job with only 50% of the computation resource.

Note that the synthetic workload rules out the heavy data computation from the jobs, in order to isolate the I/O performance during shuffle. With production jobs, the overall resource efficiency also highly depends on the nature of the specific data processing logic. However, we expect to see the same resource efficiency gains when considering the shuffle operations alone.



(a) Shuffle I/O requests.



(b) IOPS requirement with different unit I/O sizes.

Figure 10: Riffle I/O performance during shuffle. The dashed lines show best-effort merge performance.

6.2.3 *I/O Performance*

Figure 10(a) demonstrates how the number and size of shuffle fetch requests change with different merge policies. The average read size (left y-axis) increases from 150KB (no merge) to up to 6.2MB (40-way merge), and the number of read requests (right y-axis) decreases from 8,000 down to 200. With best-effort merge, since shuffle files are partially merged, each reduce task still has to read 5% of data from the unmerged block files. With 40-way merge, we observe an average of 589 read requests per task, and the average read request size of 2.1MB. Riffle effectively reduces the number of fetch requests by 40x (10x) with complete (best-effort) merge.

To show the performance implication of the underlying file system, we look at the IOPS requirement for running the job with different policies. We measure the IOPS requirement with 512KB unit I/O size provided in our current deployment, and the estimated IOPS counters when the file system supports larger I/O sizes. Figure 10(b) shows how the shuffle IOPS changes (note the log scale of both axes) with different merge policies. We can see that Riffle reduces the job IOPS from 360M with no merge down to 22M (37M), or by 16x (9.7x), with complete (best-effort) 40-way merge. We see the 10x reduction carries over as we increase the file system I/O sizes to 1M, 2M or even larger.

6.3 Production Workload

In this section, we demonstrate Riffle's improvement in processing 4 production jobs, representing small (Job 1), medium (Job 2 and Job 3), and large (Job 4) jobs at Facebook Compared with synthetic workload, the production jobs are different in several ways:

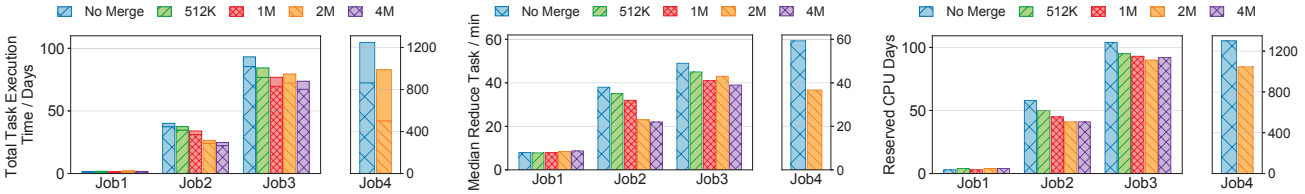
- They involve heavy computation in each task, instead of only I/O in the synthetic case;
- Jobs are deployed in real settings with limited memory resources that best fit the hardware configurations, and data will be spilled to disks if the memory space is insufficient;
- The block sizes of the intermediate shuffle files vary based on the user data distribution and the partitioning functions, and Riffle should merge based on block sizes instead of a fixed fan-in.

Improvement in I/O performance and end-to-end job completion time is crucial to production workload. For instance, Job 4 is processing a *key data set*, which is in the most upstream data pipeline for many other jobs under the same namespace. It processes hundreds of TB of data and consumes over 1,000 CPU days to finish. Accelerating this job will not only improve resource efficiency significantly, but also help improve the landing time of many subsequent jobs. We show the performance of Riffle with fixed block size merge, varying the block size threshold (512KB, 1MB, 2MB, and 4MB for first three jobs, and 2MB for the last job). All the experiments enable best-effort merge with a threshold of 95%.

Stage and task completion time. Figure 11(a) shows that Riffle significantly helps decrease the reduce stage time by 20–40% for medium to large scale jobs, without affecting the map time much. Compared to the gain in synthetic workload, Riffle gets less relative time reduction because of the fixed computing cost in the tasks. Note that in the case of running small-scale jobs (like Job 1), Riffle does not help because of the delay penalty incurred by the additional merge requests. Figure 11(b) further explains that the saving of reduce stage time comes from shorter reduce task time. The reduce task can be shorten by up to 42% (39%) when running medium (large) scale jobs.

Resource efficiency. The big saving in job completion time leads to more efficient resource usage. Figure 11(c) measures the resource usage of running the jobs. We can see that Riffle in general saves 20–30% reserved CPU time for medium to large scale jobs.

Figure 12 compares the total I/O requests during shuffle. Riffle reduces the total shuffle I/O requests by 10x for Jobs 2 and 3, and by 5x for Job 4. For Jobs 2 and 3, Riffle effectively converts the average request size from the original 100–150KB (see Table 1) to 512KB or larger, and thus significantly reduces the number of read requests needed during shuffle operations. Similarly, for Job 4, Riffle increases the average 360KB reads to 2MB and thus reduces the number of I/O requests.



(a) Map (top) & reduce (bottom) stage runtime. (b) Median reduce task runtime. (c) Reserved CPU time.

Figure 11: Riffle performance improvement with production workload.

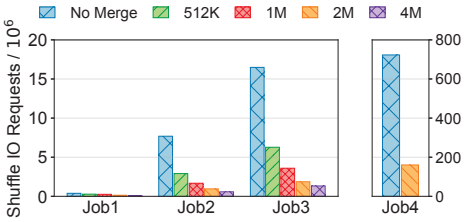


Figure 12: Number of shuffle I/O requests (million), including all additional I/O requests in Riffle mergers.

Riffle incurs additional I/O requests for merging shuffle files. The mergers use up to 64MB in-memory buffers to ensure that the merge operations only issue large, sequential I/O requests to disks. The overhead of merge I/O requests is almost negligible compared with the order of magnitude savings in shuffle I/O requests.

7 RELATED WORK

Shuffle optimization in big-data analytics. ThemisMR [46] improves the performance of MapReduce jobs by ensuring the intermediate data (including shuffle and spill) are not repetitively accessed through disks. However, the solution does not avoid large amounts of small random I/O during shuffle. In addition, as the paper stated, ThemisMR eliminates the task-level fault tolerance, and thus only applies to relatively small scale deployment. TritonSort [48] minimizes disk seeks by carefully designing the layout of output files without using huge in-memory buffers. However, since it targets the specific sorting problem, the solution can hardly generalize to other data analytics jobs. Sailfish [45] leverages a new file system design to support multiple insertion points to aggregate intermediate files. However, it requires modifications to file systems, and a single corrupted aggregation file requires recomputation of a large number of map tasks.

Parameter tuning for data analytics frameworks. Previous work [20, 39, 56, 59] provide guidelines on how to best configure system parameters (such as number of tasks in each stage) with given cluster resources. Starfish [28] is a self-tuning system which provides high performance without requiring users to understand the Hadoop parameters. However, the tuning process for a large number of jobs is expensive, and jobs have to be retuned when their characteristics such as the distribution and skew of input data change over time.

IOPS optimization. Sailfish [45] leverages a new file system design to support multiple insertion points to aggregate intermediate files. However, it requires modifications to file systems, and a single corrupted aggregation file requires recomputation of a large number of map tasks. Hadoop-A [54] accelerates Hadoop by overlapping map and reduce stages, and uses RDMA to speed up the data collection process. However, this solution relies on the reducer task to collect and buffer intermediate data in memory, which limits its scalability and fault tolerance. Recent development on hardware accelerates the handling of I/O requests and starts to get deployed in big-data analytics and storage systems [50, 53], but they do not targets the problem of small, random shuffle fetch for large-scale jobs.

The case for tiny tasks. Recent work [32, 42, 44] proposes tiny tasks which run faster and lead to better job completion time when investigating the performance of data analytics jobs. While solutions have been studied to minimize the task launch time [37] and overcome the scheduler overhead [44], tiny tasks hit the performance bottleneck of shuffle when used for large-scale jobs with multiple stages. Riffle merges intermediate files and significantly improves shuffle efficiency, so that the jobs can benefit from both fast task execution and efficient shuffle with small tasks.

Straggler mitigation. The original MapReduce paper [22] introduces the straggler problem. Previous work on data analytics leverages speculative execution [16, 18, 58] or approximate processing [15, 17, 51] to mitigate stragglers. Riffle avoids merge stragglers using best-effort merge, which allows shuffle files to be partially merged to avoid waiting for merge stragglers and accelerate job completion.

8 CONCLUSION

We present Riffle, an optimized shuffle service for big-data analytics frameworks that significantly improves the I/O efficiency and scales to process large production jobs at Facebook. Riffle alleviates the problem of quadratically increasing I/O requests during shuffle by efficiently merging intermediate files with configurable policies. We describe our experience deploying Riffle at Facebook, and show that Riffle leads to an order of magnitude I/O request reduction and much better job completion time.

ACKNOWLEDGMENTS

We are grateful to our shepherd Gustavo Alonso and the anonymous EuroSys reviewers for their valuable and constructive feedback. We also thank Byung-Gon Chun, Wyatt Lloyd, Marcela Melara, Logan Stafman, Andrew Or, Zhen Jia, Daniel Suo, and members of the BigCompute team at Facebook and the Software Platform Lab at Seoul National University for their extensive comments on the draft and insightful discussions on this topic. This work was partially supported by NSF Award IIS-1250990.

REFERENCES

[1] Retrieved 10/20/2017. Apache Hadoop. (Retrieved 10/20/2017). http://hadoop.apache.org/.
[2] Retrieved 10/20/2017. Apache Ignite. (Retrieved 10/20/2017). https://ignite.apache.org/.
[3] Retrieved 10/20/2017. Apache Spark. (Retrieved 10/20/2017). http://spark.apache.org/.
[4] Retrieved 10/20/2017. Apache Spark Performance Tuning âĂŞ Degree of Parallelism. (Retrieved 10/20/2017). https://goo.gl/Mpt13F.
[5] Retrieved 10/20/2017. Apache Spark @Scale: A 60 TB+ Production Use Case. (Retrieved 10/20/2017). https://code.facebook.com/posts/1671373793181703/.
[6] Retrieved 10/20/2017. Apache Spark the fastest open source engine for sorting a petabyte. (Retrieved 10/20/2017). https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html.
[7] Retrieved 10/20/2017. Facebook Disaggregate: Networking recap. (Retrieved 10/20/2017). https://code.facebook.com/posts/1887543398133443/.
[8] Retrieved 10/20/2017. Facebook's Disaggregate Storage and Compute for Map/Reduce. (Retrieved 10/20/2017). https://goo.gl/8vQdfU.
[9] Retrieved 10/20/2017. LZ4: Extremely Fast Compression Algorithm. (Retrieved 10/20/2017). http://www.lz4.org.
[10] Retrieved 10/20/2017. MapReduce-4049: Plugin for Generic Shuffle Service. (Retrieved 10/20/2017). https://issues.apache.org/jira/browse/MAPREDUCE-4049.
[11] Retrieved 10/20/2017. Snappy: A Fast Compressor/Decompressor. (Retrieved 10/20/2017). https://google.github.io/snappy/.
[12] Retrieved 10/20/2017. Spark Configuration: External Shuffle Service. (Retrieved 10/20/2017). https://spark.apache.org/docs/latest/job-scheduling.html.
[13] Retrieved 10/20/2017. Tim Sort. (Retrieved 10/20/2017). http://wiki.c2.com/?TimSort.
[14] Retrieved 10/20/2017. Working with Apache Spark. (Retrieved 10/20/2017). https://goo.gl/XbUA42.
[15] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *ACM EuroSys*.
[16] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective Straggler Mitigation: Attack of the Clones. In *USENIX NSDI*.
[17] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. 2014. GRASS: Trimming Stragglers in Approximation Analytics. In *USENIX NSDI*.
[18] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *USENIX OSDI*.
[19] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *ACM SIGMOD*.
[20] Josep Lluís Berral, Nicolas Poggi, David Carrera, Aaron Call, Rob Reinauer, and Daron Green. 2015. ALOJA-ML: A Framework for Automating Characterization and Knowledge Discovery in Hadoop Deployments. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.
[21] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce Online. In *USENIX NSDI*.
[22] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX OSDI*.
[23] Cliff Engle, Antonio Lupher, Reynold Xin, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Shark: Fast Data Analysis Using Coarse-grained Distributed Memory. In *ACM SIGMOD*.
[24] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *USENIX OSDI*.
[25] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *ACM SOSP*.
[26] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *USENIX NSDI*.
[27] Laura M. Grupp, John D. Davis, and Steven Swanson. 2012. The Bleak Future of NAND Flash Memory. In *USENIX FAST*.
[28] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A Self-tuning System for Big Data Analytics. In *CIDR*. 261–272.
[29] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *USENIX NSDI*.
[30] Qi Huang, Petchean Ang, Peter Knowles, Tomasz Nykiel, Iaroslav Tverdokhlib, Amit Yajurvedi, Paul Dapolito VI, Xifan Yan, Maxim Bykov, Chuen Liang, Mohit Talwar, Abhishek Mathur, Sachin Kulkarni, Matthew Burke, and Wyatt Lloyd. 2017. SVE: Distributed Video Processing at Facebook Scale. In *ACM SOSP*.
[31] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *ACM EuroSys*.
[32] S. Kambhampati, J. Kelley, C. Stewart, W. C. L. Stewart, and R. Ramnath. 2014. Managing Tiny Tasks for Data-Parallel, Subsampling Workloads. In *2014 IEEE International Conference on Cloud Engineering*.
[33] Vamsee Kasavajhala. 2011. Solid State Drive vs. Hard Disk Drive Price and Performance Study: A Dell Technical White Paper. *Dell PowerVault Storage Systems* (May 2011).

[34] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. 2010. An Analysis of Traces from a Production MapReduce Cluster. In *IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*.

[35] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *ACM SoCC*.

[36] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *ACM ISCA*.

[37] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. 2016. Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems. In *USENIX OSDI*. Savannah, GA.

[38] S. T. Maguluri, R. Srikant, and L. Ying. 2012. Stochastic Models of Load Balancing and Scheduling in Cloud Computing Clusters. In *IEEE INFOCOM*.

[39] M. D. McKay, R. J. Beckman, and W. J. Conover. 2000. A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. *Technometrics* 42, 1 (Feb. 2000), 55–61.

[40] Michael Mitzenmacher. 2001. The Power of Two Choices in Randomized Load Balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (Oct. 2001), 1094–1104.

[41] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. 2017. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *ACM SOSP*.

[42] Kay Ousterhout, Aurojit Panda, Joshua Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. 2013. The Case for Tiny Tasks in Compute Clusters. In *USENIX HotOS Workshop*. Santa Ana Pueblo, NM.

[43] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *USENIX NSDI*.

[44] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *ACM SOSP*.

[45] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. 2012. Sailfish: A Framework for Large Scale Data Processing. In *ACM SoCC*.

[46] A. Rasmussen, M. Conley, R. Kapoor, V.T. Lam, G. Porter, and A. Vahdat. 2012. ThemisMR: An I/O-efficient MapReduce. *Technical Report (University of California, San Diego. Department of Computer Science and Engineering)* (2012).

[47] Alexander Rasmussen, Vinh The Lam, Michael Conley, George Porter, Rishi Kapoor, and Amin Vahdat. 2012. Themis: An I/O-efficient MapReduce. In *ACM SoCC*.

[48] 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 *USENIX NSDI*.

[49] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*.

[50] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. 2017. Crail: A High-Performance I/O Architecture for Distributed Data Processing. *IEEE Data Eng. Bull.* 40, 1 (2017), 38–49.

[51] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J. Franklin, and Ion Stoica. 2014. The Power of Choice in Data-aware Cluster Scheduling. In *USENIX OSDI*.

[52] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. 2010. Characterizing Cloud Computing Hardware Reliability. In *ACM SoCC*.

[53] Y. Wang, R. Goldstone, W. Yu, and T. Wang. 2014. Characterization and Optimization of Memory-Resident MapReduce on HPC Systems. In *IEEE 28th International Parallel and Distributed Processing Symposium*.

[54] Yandong Wang, Xinyu Que, Weikuan Yu, Dror Goldenberg, and Dhiraj Sehgal. 2011. Hadoop Acceleration Through Network Levitated Merge. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*.

[55] Caesar Wu and Rajkumar Buyya. 2015. *Cloud Data Centers and Cost Modeling: A Complete Guide To Planning, Designing and Building a Cloud Data Center* (1st ed.). Morgan Kaufmann Publishers Inc.

[56] Tao Ye and Shivkumar Kalyanaraman. 2003. A Recursive Random Search Algorithm for Large-scale Network Parameter Configuration.

[57] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *USENIX NSDI*.

[58] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. 2008. Improving MapReduce Performance in Heterogeneous Environments. In *USENIX OSDI*.

[59] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *ACM SoCC*. Santa Clara, CA.