Sailfish: A Framework For Large Scale Data Processing

Sriram Rao[*]
Microsoft Corp.
sriramra@microsoft.com

Raghu Ramakrishnan[*]
Microsoft Corp.
raghu@microsoft.com

Adam Silberstein[*]
LinkedIn
asilberstein@linkedin.com

Mike Ovsiannikov
Quantcast Corp
movsiannikov@quantcast.com

Damian Reeves
Quantcast Corp
dreeves@quantcast.com

Abstract

In this paper, we present `Sailfish`, a new Map-Reduce framework for large scale data processing. The `Sailfish` design is centered around aggregating intermediate data, specifically data produced by map tasks and consumed later by reduce tasks, to improve performance by batching disk I/O. We introduce an abstraction called $\mathcal{I}$-files for supporting data aggregation, and describe how we implemented it as an extension of the distributed filesystem, to efficiently batch data written by multiple writers and read by multiple readers. `Sailfish` adapts the Map-Reduce layer in Hadoop to use $\mathcal{I}$-files for transporting data from map tasks to reduce tasks. We present experimental results demonstrating that `Sailfish` improves performance of standard Hadoop; in particular, we show 20% to 5 times faster performance on a representative mix of real jobs and datasets at Yahoo!. We also demonstrate that the `Sailfish` design enables auto-tuning functionality that handles changes in data volume and skewed distributions effectively, thereby addressing an important practical drawback of Hadoop, which in contrast relies on programmers to configure system parameters appropriately for each job, for each input dataset. Our `Sailfish` implementation and the other software components developed as part of this paper has been released as open source.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems–*Distributed applications*

General Terms

Design, Experimentation, Performance

1 Introduction

In recent years data intensive computing has become ubiquitous at Internet companies of all sizes, and the trend is extending to all kinds of enterprises. Data intensive computing applications (viz., machine learning models for computational advertising, click log processing, etc.) that process several tens of terabytes of data are common. Such applications are executed on large clusters of commodity hardware by using parallel dataflow graph frameworks such as Map-Reduce [10], Dryad [15], Hadoop [2], Hive [28], and Pig [20]. Programmers use these frameworks on in-house clusters as well as on public cloud settings (such as Amazon's EC2/S3) to build applications in which they structure the computation as a sequence of steps, some of which are blocking. The framework in turn simplifies distributed programming on large clusters by providing software infrastructure to deal with issues such as task scheduling, fault-tolerance, coordination, and data transfer between computation steps. We refer to the data that is transferred between steps as *intermediate data*.

In the cluster workloads at Yahoo!, we find that, a small fraction of the daily workload (about 5% of the jobs) use well over 90% of the cluster's resources. That is, these jobs (1) involve thousands of tasks that are run on many machines in the cluster, (2) run for several hours processing multiple terabytes of input, and (3) generate intermediate data that is at least as big as the input. Therefore, we find that the overall cluster performance tends to be substantially impacted by these jobs and in particular, how the computational framework handles intermediate data at scale.

At large data volumes, transporting intermediate data between computation steps typically involves writing data to disk, reading it back later, and often a network transfer. This is because tasks from successive computation steps are usually not co-scheduled and may also be scheduled on different machines to exploit parallelism. For computations in which tens of terabytes of data have to be transported across cluster machines, the associated overheads can become a significant factor in the job's overall run-time. Surprisingly, the overheads involved in the handling of intermediate data and its impact on performance at scale have not been well studied in the literature. This is the focus of our paper.

The main contributions of this paper are as follows:

- We demonstrate the importance of optimizing the transport of intermediate data in distributed dataflow systems such as Hadoop (Section 2).
- We argue that batching data for disk I/O (or aggregating data for disk I/O) should be a core design principle in handling intermediate data at scale (Section 3).
- Based on this design principle, we develop *$\mathcal{I}$-files* as an abstraction, to support batching of data written by multiple writers (Section 4). We develop an $\mathcal{I}$-file implementation by extending distributed filesystems that were previously designed for data intensive computing and are already deployed

[*]Work done at Yahoo! Labs

in computing clusters. $\mathcal{I}$-files are effectively a general framework for aggregating intermediate data whose specific instantiation can be based on the cluster configuration.

- Using $\mathcal{I}$-files to transport intermediate data (i.e., to transfer output of map tasks to relevant reduce tasks), we develop `Sailfish`, a new Map-Reduce framework that can run standard Hadoop jobs with no changes to application-code (Section 5). `Sailfish` provides the same level of fault-tolerance for jobs as Stock Hadoop.
- We experimentally demonstrate that `Sailfish` improves upon existing frameworks along the dimensions of both performance and auto-tuning functionality (Section 6):
  - **Performance:** Using benchmarks as well as production jobs run on actual datasets, we show significant performance gains with `Sailfish`: Using a representative mix of production jobs at Yahoo!, we demonstrate that the same job, run unmodified, with `Sailfish` is between 20% to 5 times faster when compared to the corresponding run with Stock Hadoop (see Section 6.3).
  - **Auto-tuning Functionality:** The core design principle of aggregating the output of map tasks allows us to gather statistics because intermediate data management is decoupled from the computation. This enables `Sailfish` to automatically exploit parallelism in the reduce phase of a computation, with no need for users to tune system parameters, unlike current systems. The results in Section 6.3 demonstrate that `Sailfish` can, in a data-driven manner:(1) handle changes in data volume by dynamically determining the number of reduce tasks, (2) handle skew in intermediate data by dynamically determining the work assignment (i.e., range of keys to process) for reduce tasks, and (3) by decoupling sorting of intermediate data from map task execution to better handle skew in map output.

We discuss related work in Section 7, and conclude in Section 8 by presenting directions for future work.

Our `Sailfish` implementation and the other software components developed as part of this paper has been released as open source [6].

2 Importance of Intermediate Data Handling

2.1 Current Approaches to Handling Intermediate Data

To motivate the problem of handling intermediate data at scale, we briefly describe the mechanisms used in existing Map-Reduce implementations. For example, in Hadoop [2] (an open source Map-Reduce implementation), the underlying mechanism used for handling intermediate data in a Map-Reduce computation is essentially via a parallel merge-sort. A Hadoop map task generates intermediate data (i.e., key/value pairs) as it executes, stores them in RAM, and periodically sorts and spills the data to a file on disk. Whenever the map task completes execution it merges the sorted spills from disk (in effect, executing the merge phase of an external sort) to produce a single output file. This single output file contains sorted runs of key/value pairs, partitioned by key, with one sorted run destined for each reducer task (see Figure 1). Next, a reducer task pulls its portion of the input data from each of the mappers' output files. After the reducer has obtained all of its input from the map tasks,

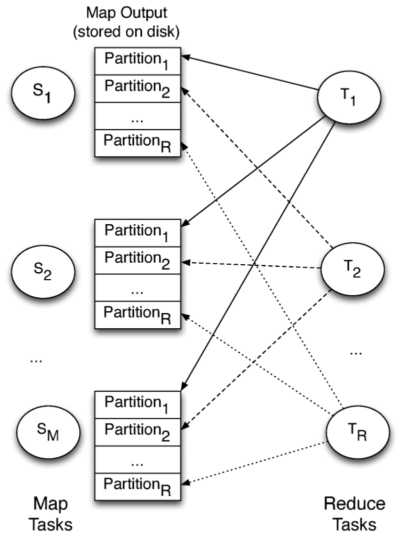

Figure 1: A reduce task retrieves its input from each of the map tasks. The number of distinct retrievals is proportional to $M * R$ and the data read per retrieval is proportional to $1/R$.

it merges the data (using a disk-based merge if necessary) and then invokes the user-supplied reduce function.

With this design, there is a potential for further inefficiency whenever the map task has to merge spills from disk to generate its final output file (i.e., whenever the map output exceeds the size of RAM). Interestingly, we find that this occurs frequently in practice, especially when there is skew in the output size of map tasks.

2.2 Impact of Intermediate Data Size and Tuning

At scale, the cost of handling intermediate data is dominated by the rate at which data can be read from (as well as written to) the disk subsystems on individual nodes. This is because disk performance is affected by seeks. The effective disk transfer rate is highly dependent on the number of seeks as well as the amount of data read per disk seek. Up to a point, the effects of disk seeks may be masked by memory-based filesystem buffer caches. Beyond that point, unless careful attention is paid to the seek overheads involved in handling intermediate data, cluster throughput will degrade.

To evaluate these overheads in practice, we use Hadoop [2] as the basis for our study. As a starting point, we measure Hadoop's runtime for executing jobs in our experimental cluster as the size of intermediate data in a given job varies from 1TB to 64TB. Figure 2 shows the result of this experiment (full details of our experimental setup are in Section 6).

Our findings are striking. As the volume of intermediate data increases, Hadoop performance degrades non-linearly (for example, beyond 16TB of intermediate data, there is a 2.5x increase in runtime as the volume of data processed increases by 2x). This degradation is due to the disk overheads involved in the data transfer. To estimate the disk overheads involved, in a computation involving M mappers and R reducers, as Figure 1 shows, there are $M * R$ distinct retrievals [11, 19]. Furthermore, as Figure 1 also shows, due to data partitioning, the amount of data retrieved by a reduce task from a map task is inversely proportional to the number of reducer tasks (i.e., $1/R$). Therefore, for computations in which the values of M and R are large (i.e., on the order of 1000's of tasks), the amount of data read per disk seek decreases substantially *and* the number of disk seeks grows super-linearly, causing a degra-

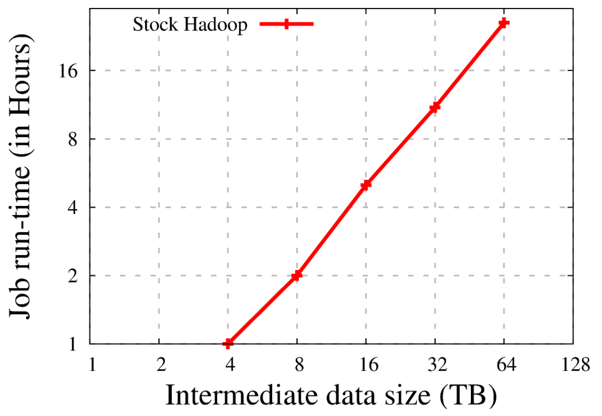

Figure 2: Hadoop performance vs. intermediate data size

Intermediate data size	Cache effects	Impact on data transfer
Int. data size < RAM size	RAM masks disk I/O	Transfer is network-bound
Int. data size > RAM size	High cache hit rate	Transfer is *likely* network-bound
Int. data size >> RAM size	Low cache hit rate	Seek-intensive (i.e, disk-bound)

Table 1: Intermediate data size vs. cache effects

dation in achievable disk throughput resulting in poor application performance.

Table 1 summarizes our observations from the experiments.

Some of the performance degradation caused by very large intermediate data sizes can be mitigated by tuning system parameters. Unfortunately, we also find that parallel dataflow frameworks are very sensitive to system parameters that users are expected to tune. For instance, with Hadoop, a user has to choose the number of reduce tasks for a Map-Reduce job and tune the parameters related to sorting the output of a map task (e.g., buffer sizes, merge width). Though various guidelines are provided [19], such tuning is known to be critical for performance and is non-trivial [14]. This problem is made worse by continuously changing data volumes. For example, in Yahoo! workloads, we find that computations run on a periodic basis (e.g., daily click-stream log processing) see as much as a 2x variation in the size of their input depending on the time of day and day of the week. In practice, we find that once programmers choose a set of parameters for their job, they rarely tune it further. For instance, anecdotal evidence in our Yahoo! cluster workloads suggest that some of the jobs have been sped up by nearly a factor of two with careful parameter tuning. Such speedups through parameter tuning have also been observed elsewhere [14]. In the absence of such careful tuning, the net result is that overall cluster performance degrades.

3 Our Approach: Batching Data I/O

The techniques we develop for handling intermediate data are in general applicable to any parallel dataflow graph frameworks that contain a *blocking* step (i.e., whenever the output from one step has to be materialized by writing to disk-based storage before it can be consumed by a later step). Such a blocking step is present in dataflow graphs such as Map-Reduce, joins in parallel databases, Group-By computations in SQL systems, Pig (Yahoo!'s version of SQL) programs which are compiled into Hadoop Map-Reduce computations, etc. In this paper, we use Map-Reduce as a sample application since *every* step in the flow is blocking.

Our approach is driven by practical constraints. Clusters for data intensive computing are built using commodity hardware in which hard disks are currently the only cost-effective high capacity storage mechanism. Hence, we focus on minimizing the disk overheads involved in handling large volumes of intermediate data. While storage alternatives that avoid some disk overheads altogether are available, they are not yet viable for our setting. RAM-based systems (e.g., RAM-Clouds [21]) are not yet cost-effective on the scale of tens to hundreds of terabytes; and while solid-state drives (SSDs) excel at random I/O, their price to storage density ratio is much higher than disk drives, and thus they also are not cost-effective for multi-terabyte scale settings.

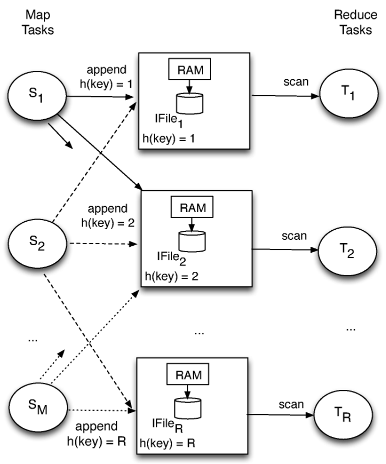

Figure 3: Batching data for disk I/O (or aggregation) lowers the overheads involved in transfer of intermediate data. Map output is batched and committed to disk. The number of distinct retrievals in the reduce phase is proportional to R.

As noted in Figure 1, to lower the disk overheads involved in transporting intermediate data it is crucial to minimize the number of disk seeks. When retrieving a given volume of data from disk, fewer disk seeks translates to increasing the amount of data read per seek. This suggests that *batching data for disk I/O* (or *aggregating data for disk I/O*) should be a core design principle in handling intermediate data.

3.1 Intermediate Data Aggregation: $\mathcal{I}$-files in `Sailfish`

Figure 3 illustrates the benefits of aggregating intermediate data on a per-partition basis. A reduce task reads aggregated data to obtain its input, thereby reducing the number of distinct retrievals in the reduce phase from $M * R$ to R. At scale, such a drastic reduction in the number of retrievals enables substantial performance improvements.

A straightforward way to aggregate records in a file is to employ an *aggregator task* that receives records from map tasks and commits them to disk. We considered implementing custom aggregator tasks that work on a per-partition basis. However, this approach of employing ad-hoc custom aggregators suffers from some practical limitations. In particular, whenever there is data skew, some aggregators may need to handle substantially more data than others (i.e., become hot-spots) and hence, limit performance. Furthermore, dealing with failures of aggregator tasks is another issue.

In our work, we enhance the distributed filesystem which are originally designed for data-intensive computing to also support data aggregation through the notion of an $\mathcal{I}$*-file* (intermediate data file). Consequently, $\mathcal{I}$-files make the intermediate data available outside the context of a computation, enabling additional introspection functionality (such as debugging, reuse, etc.).

3.2 Benefits of Aggregation for Addressing Skew

While data aggregation lowers the disk overheads in the reduce phase, as noted in Section 2.1 disk overheads can affect the map phase as well if the map output does not fit in memory and must be spilled to disk before it is merged. Since a reduce task cannot begin execution until *all* map tasks generate their output file, the *sort overhead* incurred by even a single map task can substantially increase job completion time. Such a slow-down occurs, for instance, if there is skew in map output: some map tasks may incur a multi-pass external merge to produce their final output file while the remaining tasks may produce it via a single pass in-memory sort. While DISC frameworks employ speculative execution to work around "straggler" tasks, note that such a mechanism has limited benefit in a setting where the slow-down in task completion is due to the external sort.

To mitigate these effects and also to handle skew, we decouple the sorting of intermediate data from map task execution. That is, the intermediate data is first aggregated, and then sorted outside the context of map tasks. Therefore, *sorting of intermediate data is now a separate phase of execution which can be optimized by the DISC framework*. This decoupling relieves the programmer from having to tune framework parameters related to map-side sorting of the intermediate data. For instance, specifically for Hadoop, our approach eliminates 4 parameters related to map-side sorting (see Table 2 (a)) that a programmer has to tune.

Furthermore, we can take advantage of the aggregation of map outputs to build an index, as noted above, which enables us to auto-tune the subsequent reduce phase effectively. This alleviates the need for programmers to select the number of reduce tasks (as required, for example, by Hadoop). Using the index, based on the distribution of keys across the intermediate data files, the number of reduce tasks as well as a task's key-range assignment can be determined *dynamically* in a data dependent manner. The index also allows each reduce task to efficiently retrieve (only) its input.

Figure 3 shows how `Sailfish` uses $\mathcal{I}$-files for transporting the intermediate data between map tasks and reduce tasks. Briefly, in our design, the output of map tasks (which consists of key/value pairs) is first partitioned by key and then aggregated[1] on a per-partition basis using $\mathcal{I}$-files. This intermediate data is sorted and augmented with an index to support key-based retrieval. Finally, the reduce phase of execution is automatically parallelized in a data-driven manner. We elaborate on this design in the next two sections.

Since $\mathcal{I}$-files are a key building block in the `Sailfish` design, in what follows (Section 4) we first describe the design and implementation of the $\mathcal{I}$-file abstraction. Next, in Section 5, we describe the `Sailfish` system, and how it uses $\mathcal{I}$-files to implement a Map-Reduce framework.

4 $\mathcal{I}$-files for Aggregating Intermediate Data

We extended the Kosmos distributed filesystem (*KFS*) [4], an alternative to HDFS, to implement the $\mathcal{I}$-file abstraction. We chose KFS purely for implementation convenieance (see Section 4.1) since it already implements some of the features for constructing $\mathcal{I}$-files. Our ideas are general and applicable to HDFS as well. In what follows, we first provide an overview of KFS and then describe how we extended KFS to implement $\mathcal{I}$-files.

[1]In the rest of the paper, in the context of `Sailfish` and $\mathcal{I}$-files we use *aggregation* to mean batching data for disk I/O.

4.1 Background: KFS Overview

KFS is designed to support high throughput access to large files in clusters built using commodity hardware. KFS's design is similar in spirit to Google's GFS [13] and Hadoop's HDFS [3]. Blocks of a file (referred to as *chunks*) are striped across nodes and replicated for fault-tolerance. Chunks can be variable in size but with a fixed maximal value (which by default is 128MB). KFS consists of three components: (1) a single metadata server (*metaserver*) that provides a global namespace, (2) a set of chunkservers that run on each of the cluster machines and store chunks as files on the machine's local disk(s), and (3) a client library which should be linked with applications to access files stored in KFS. Finally, KFS is integrated with Hadoop for storing the input/output in MapReduce computations and has been deployed in practical settings.

4.2 Adapting KFS to Support $\mathcal{I}$-files

An $\mathcal{I}$-file is like any other KFS file, with a few exceptions. First, the chunks of an $\mathcal{I}$-file are written to using an atomic record append primitive and hence, by definition, append-only. Second, in our implementation, once a chunk is closed for writing, it is immutable (i.e., *stable*). We leverage the immutable property of stable chunks to sort the records stored in a chunk (see Section 4.2.3).

When a client appends a record to an $\mathcal{I}$-file, this translates to an append operation on a chunk of that $\mathcal{I}$-file. The chunkserver storing that chunk plays the role of the aggregator task that we outlined above (see Section 3.1). Next, to allow data aggregation in $\mathcal{I}$-files to be done in a distributed manner, we (1) restrict the number of concurrent writers to a given chunk of an $\mathcal{I}$-file, and (2) allow multiple chunks of an $\mathcal{I}$-file to be concurrently appended to. This approach has multiple advantages. First, by allowing a chunk to be written to by multiple writers, data for an $\mathcal{I}$-file can be packed into fewer chunks. Second, since chunks are striped across nodes, data aggregation for the chunks of an $\mathcal{I}$-file is handled by multiple chunkservers which allows us to avoid hot-spots.

In the following subsections, we present the APIs for supporting $\mathcal{I}$-files, describe how we implement the key operations of appending records and retrieving records by key, and finally, discuss some efficiency considerations.

4.2.1 $\mathcal{I}$-file APIs

Conceptually, the APIs we have developed for $\mathcal{I}$-files to support record-based I/O are the following:

- `create_ifile(filename)`: An application uses this API to create an $\mathcal{I}$-file. The call returns a file descriptor for use with subsequent append operations.
- `record_append(fd, <key, value>)`: A writer uses this API to append records to an $\mathcal{I}$-file.[2]
- `scan(fd, buffer, lower_key, upper_key)`: A reader uses this API to retrieve records from an $\mathcal{I}$-file that are in the specified key range. The KFS client library implements the functionality of retrieving the matching records from all the chunks of the $\mathcal{I}$-file.

4.2.2 Appending Records To An $\mathcal{I}$-file

Appending a record to an $\mathcal{I}$-file translates to an append operation on a chunk of that file. The steps involved in appending a record to an $\mathcal{I}$-file chunk are as follows:

[2]Our implementation of this API was released as part of the KFS open-source software distribution in version 0.5.

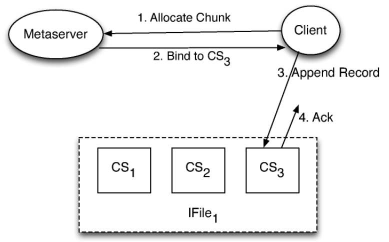

Figure 4: Appending records to an $\mathcal{I}$-file: The metaserver binds the client to the chunkserver (CS_3) which stores one of the 3 open chunks in the $\mathcal{I}$-file.

1. To append a record to an $\mathcal{I}$-file, a client must be bound to a chunk of that $\mathcal{I}$-file. The client obtains a binding by issuing an allocation request to the KFS metaserver. The metaserver then binds the client to one of the chunks of that $\mathcal{I}$-file that is currently open for writing. If there is no such chunk, then the KFS metaserver allocates a new chunk. In either case, in response to the client's request, the metaserver binds the client to a chunkserver. This is illustrated by steps 1 and 2 in Figure 4. The client can continue appending records to that chunk as long as the binding is valid. Note that a client has such a binding for *each* $\mathcal{I}$-file it is appending records to.

2. The client sends the record (i.e., *just data*, without the offset) to the chunkserver that it is currently bound to. This corresponds to step 3 in Figure 4.

3. The chunkserver assigns an offset to the record within the chunk, buffers the record in memory (while asynchronously writing data to disk), and responds with an ack message to the client. This is step 4 in Figure 4.

4. When the client receives an ack message from the server, the client considers the operation to be successful.

5. If the client fails to receive an ack message in a timely manner, the client will retry. The client, with suitable timeouts, queries the chunkserver for the operation's status. If the status retrieval fails, the client will first give up its binding to the chunkserver and then retry the operation starting from the chunk allocation step (step #1). As part of the retry for a given operation, the client may get bound to a different chunk of the $\mathcal{I}$-file.

6. The chunkserver may also fail the operation if appending the record will the cause the chunk's size to exceed the maximal value In this case, the chunkserver will return an error to the client. The client will give up its chunkserver binding and then retry the operation by starting from the allocation step (step #1).

Our record append primitive provides *at least once* semantics for each operation (since the retry logic above does not preclude a record being appended multiple times). `Sailfish` employs per-record framing to filter out such duplicate records (see Section 5.2.4).

With the above protocol (see step #1), concurrent writers to an $\mathcal{I}$-file can be bound to the same chunk. Extending the protocol to handle concurrent appends is fairly straightforward. For each record append operation, since the *chunkserver* assigns the offset within the chunk, it can serialize the appends it receives from multiple writers in a lock-free manner. The record append operation is therefore atomic. Lock-free atomic append operations are not new and have been implemented in filesystems. For instance, using the `write` system call supported by the Linux filesystem, concurrent writers can append to a file in a lock free manner by having the operating system serialize the writes. That is, the clients provide the data but the *server* chooses the offset. The Google filesystem paper [13] also mentions implementing a similar atomic append primitive. From the perspective of appending records to a single chunk, our primitive is similar to these implementations. A novel aspect of our $\mathcal{I}$-file implementation is the extension of allowing multiple chunks of an $\mathcal{I}$-file to be concurrently appended to.

As an aside, we note that we have also extended our record append implementation to support replication. This is described in [26]. Since we do not use replication for $\mathcal{I}$-files in this paper, we do not discuss this further.

4.2.3 Retrieving Records From An $\mathcal{I}$-file By Key

To support key-based retrieval of records from an $\mathcal{I}$-file, we augment each chunk with an index. Since the client provides a key when appending a record, the chunkserver saves the key in a per-chunk index that is written out past the end of the chunk[3]. Consequently, construction of the per-chunk index incurs minimal overhead. In addition, as a performance optimization to support data retrieval by key (via the `scan()` API), we use an offline process to sort the records within a chunk. The sort operation is done on *stable* chunks which are closed for writing. Whenever a chunk becomes stable (i.e., the chunk is full or the metaserver forced chunkserver to make a chunk stable), the chunkserver notifies a *chunksorter* daemon process that is running locally. The chunksorter, in turn, reads the data from disk (at most 128MB), sorts it in-memory, and writes the data back to disk (along with an updated in-chunk index). Finally, chunksorter failures are handled via re-execution.

The in-chunk index is necessarily *sparse* to minimize both index-processing overheads as well as storage space requirements. We record an index entry for approximately each 64KB of data; with a 128MB chunk size, this translates to about 2000 entries per chunk. In practice, we find that a per-chunk index is about a few tens of KB in size.

4.2.4 Efficiency Considerations In Building $\mathcal{I}$-files

To maximize the available write parallelism in constructing an $\mathcal{I}$-file, we restrict the number of concurrent appenders to a chunk. This is accomplished using a space reservation protocol, in which, prior to appending records to a chunk, a client first reserves *logical* space within a chunk with the chunkserver. The client then appends records as long as it has space. If the space reservation fails, the client gives up the current chunkserver binding and then obtains a new binding from the metaserver. For details, see [26].

In our design, for a given chunk, a chunkserver batches records appended by different clients and writes them to disk in a physically contiguous manner. Any unwritten portion of a chunk is only between the end of the last record and the end of the chunk. Unwritten portions in a chunk will exist only if the remaining amount of space in a chunk is too small for any client to use. A key property of our record append implementation is that records do not span chunk boundaries. This also means that records have a fixed maximal size, which is the same as the chunk size (currently, 128MB).

Finally, the allocation policy we have used to bind clients to chunks forces new $\mathcal{I}$-file chunks to be allocated judiciously—only when there are no chunks open for writing or if a client reports to

[3]The per-chunk index is written at a fixed offset in the chunk file—offset corresponding to 128M. This enables efficient storage and retrieval of just the index alone.

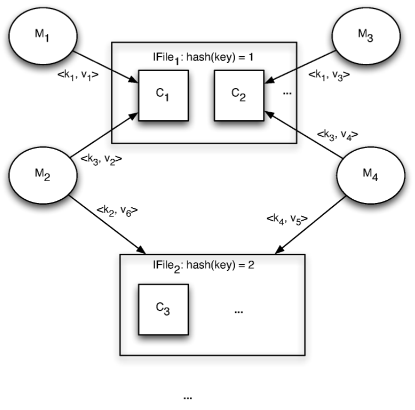

Figure 5: Mappers appending their output, partitioned by key, to $\mathcal{I}$-file chunks. Note that multiple chunks of multiple $\mathcal{I}$-files are appended to concurrently.

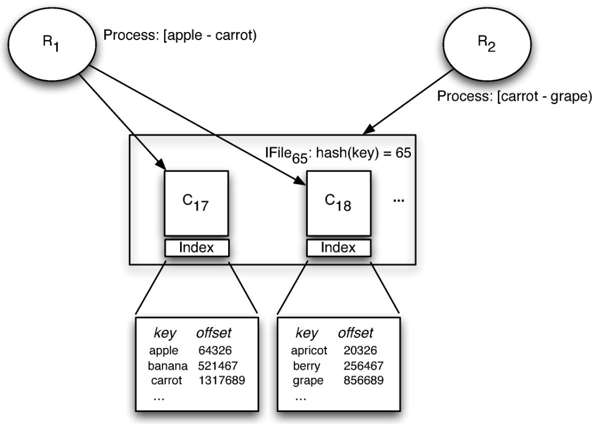

Figure 6: Reducers retrieving their assigned key ranges from the chunks of an $\mathcal{I}$-file. Multiple reducer tasks are assigned non-overlapping key ranges from a single $\mathcal{I}$-file.

the metaserver that it was unable to append to a chunk (e.g., due to contention). This enables us to batch data from multiple writers into as few chunks as possible.

5 `Sailfish`: MapReduce Using $\mathcal{I}$-files

We use Hadoop as a starting point for implementing `Sailfish`. Hadoop consists of a Map-Reduce framework and HDFS (Hadoop distributed filesystem). In building `Sailfish`, we made modifications to Hadoop's Map-Reduce components, and used KFS extended to support $\mathcal{I}$-files as a drop-in replacement for HDFS.

In this section, we first present a conceptual overview of how Map-Reduce computations are executed using `Sailfish`, and specifically, how $\mathcal{I}$-files are used to transport intermediate data from map tasks to reduce tasks. We then describe salient implementation details for `Sailfish`, and finally provide an estimate of the disk seeks involved for I/O related to $\mathcal{I}$-files in `Sailfish`.

5.1 `Sailfish` Overview

The key aspects that define the execution of a Map-Reduce computation with `Sailfish` are as follows:

1. **Writing map task output to $\mathcal{I}$-file**: Map tasks append their output (i.e., key/value pairs) to $\mathcal{I}$-files using the record append API described earlier (in Section 4.2.2). Map output is partitioned by key (the choice of hash partitioning or range partitioning is application-specific) and written to the $\mathcal{I}$-file associated to the partition. That is, there is exactly one $\mathcal{I}$-file per partition. This is illustrated in Figure 5. $\mathcal{I}$-file$_i$ is associated with keys whose partition is i. For instance, for partition 1, mappers M_1, M_2 append records to chunk C_1 while M_3, M_4 append records to C_2. The chunkservers storing C_1, C_2 serialize the appends to their respective chunks. Implementation details of how map output is appended to $\mathcal{I}$-files are covered in Section 5.2.1.

2. **Sorting and indexing $\mathcal{I}$-file chunks:** As motivated in Section 3.2, in `Sailfish` sorting of map output is decoupled from map task execution. As described in Section 4.2.3, whenever an $\mathcal{I}$-file chunk becomes stable, it is sorted and augmented with an in-chunk index. Implementation details are covered in Section 5.2.2.

3. **Determining the number of reducers:** `Sailfish` tries to automatically parallelize execution in the reduce phase by choosing the appropriate number of reduce tasks based on data properties (such as the number of keys per $\mathcal{I}$-file) and run-time properties (such as the number of machines, available RAM, etc.). Our goals in the reduce phase are to divide work evenly among the reduce tasks and meet a target amount of work per task. The details are described in Section 5.2.3.

4. **Retrieving reduce task input from an $\mathcal{I}$-file:** Reduce tasks obtain their input from the appropriate $\mathcal{I}$-file based on their assigned key range. This is illustrated in Figure 6. Reduce tasks R_1, R_2 are assigned to $\mathcal{I}$-file$_{65}$ and they use the per-chunk index to retrieve (only) their respective input from the $\mathcal{I}$-file$_{65}$'s chunks. Implementation details of how reduce input is generated by merging the records from $\mathcal{I}$-file chunks is described in Section 5.2.4.

In addition to handling task failures (map/reduce tasks), there is also the issue of handling data loss in $\mathcal{I}$-files (i.e., when a chunk of an $\mathcal{I}$-file is lost due to disk failure). In `Sailfish`, data loss will trigger the recomputation of all the map tasks that wrote to the lost $\mathcal{I}$-file chunk. Section 5.2.5 covers how recomputations are handled.

5.2 `Sailfish` Implementation

Figure 7 illustrates the dataflow path in `Sailfish`. We describe the steps in the following sub-sections.

5.2.1 Appending Map Output To $\mathcal{I}$-files

A map task on start up spawns a child process `iappender` for appending records to $\mathcal{I}$-files. Each record generated by the map task is streamed to the `iappender` as a tuple: <partition, key, value> (step 1 in Figure 7). The `iappender` buffers the records and periodically flushes the data to the $\mathcal{I}$-file associated with the partition, using the record append API. As described in Section 4.2.2, the `iappender` is bound to a chunkserver, and to minimize network overheads when appending records to a chunk, the `iappender` gathers multiple records (by default, upto 64KB of data) and writes them to the chunkserver in a single operation (step 2 in Figure 7). The chunkserver buffers records sent by multiple `iappender`'s in RAM and commits the records to disk (step 3 in Figure 7). Finally, when the parent map task has processed all of its input, it notifies the `iappender` to flush any remaining buffered records. After the outstanding record append operations finish successfully, the task execution is complete.

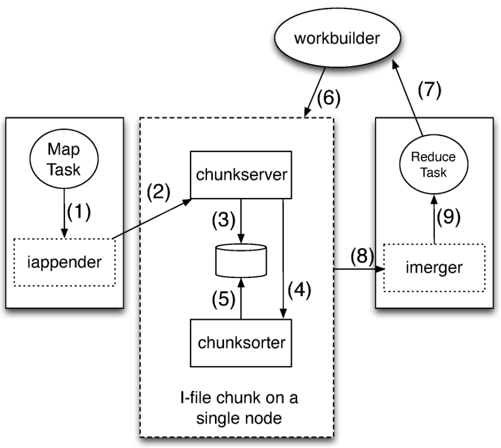

Figure 7: Dataflow in `Sailfish` as it corresponds to a single $\mathcal{I}$-file chunk. The `iappender` and `imerger` are one per task. There is one `workbuilder` daemon per job.

In our implementation, if the `iappender` task ever crashes, we cause the parent map task to also crash. The Hadoop infrastructure detects the task failure and automatically spawns a new task to attempt re-execution.

Observe that the records emitted by a map task are written to $\mathcal{I}$-files concurrently with task execution. If a map task ever fails then the records emitted by the failed map task need to be discarded. In our design discarding such records is done in the reduce phase. To do so, (1) the `iappender` prepends source information (namely, map task id/ attempt number, and record sequence number—12 bytes overhead) to each record, and (2) during the reduce phase, the `imerger` learns of the identities of the failed map tasks (such as, from the Hadoop JobTracker) and then uses the per-packet source information to filter out records emitted by failed map tasks. Note that the same mechanism can also be used handle speculative execution—the output of speculatively executed tasks which were killed will be discarded.

5.2.2 Sorting Stable $\mathcal{I}$-file Chunks

For the reasons discussed in Section 3, sorting of map output is decoupled from map task execution. Rather than wait for all the chunks of an $\mathcal{I}$-file to become stable, our implementation forces a chunkserver to schedule a chunk for sorting using a chunkserver daemon (running locally) whenever that chunk becomes stable (step 4 in Figure 7). The I/O path involved in sorting a chunk is primarily sequential: The chunksorter daemon loads a chunk's worth of data to RAM (i.e., 128MB), performs an in-memory sort, and then writes the records back to disk (step 5 in Figure 7) along with an in-chunk index (see Section 4.2.3). Note that an $\mathcal{I}$-file is piecewise sorted; while the records in each chunk are sorted, the file as a whole is not.

Sorting of stable chunks incurs a pair of additional disk I/O's. We considered an optimization path by sorting the chunks in RAM before committing to disk. Implementing this optimization without affecting performance requires substantial buffer space for holding chunk data. Specifically, data for (1) chunks that are stable but not yet sorted, (2) chunks that are being sorted, (3) chunks that have been sorted and enqueued for writing to disk, and (4) chunks currently being written to, should all be buffered in RAM. In the absence of such buffer space, either data will need to be flushed to disk whenever the system is under memory pressure or data generation must stall; otherwise, nodes will incur swapping. Since machines in our cluster are RAM-constrained, setting aside such buffer space meant that we had to significantly under program the CPU (i.e., run fewer map or reduce tasks per machine), which in turn can affect performance. Due to these reasons, we choose not to pursue this path.

5.2.3 Determining Number of Reducers

Reducers are assigned input by key ranges. In `Sailfish` the step of determining the split points (and therefore, the number of reducers) is implemented using a `workbuilder` daemon process. There is a single `workbuilder` process per job and it executes concurrently with the job. The `workbuilder` reads the per-chunk indexes from $\mathcal{I}$-files as they become available (step 6 in Figure 7). Once the map phase of a computation is complete, the `workbuilder` process first uses a target amount of work per task (for example, 2GB of data per reduce task) to determine the number of reduce tasks. Next, the `workbuilder` process uses the per-chunk indexes to construct suitable *split points* (i.e., key ranges) based on the key distributions and thereby decide the per-task work assignment. Note that, each *split* is handled by a separate reduce task. The `workbuilder` then notifies the Hadoop JobTracker to spawn the targeted number of reduce tasks. Each reduce task, on start up, obtains its work assignment (namely, an $\mathcal{I}$-file, a range of keys within that $\mathcal{I}$-file, and the set of chunks to read) from the `workbuilder` (step 7 in Figure 7). Restricting a reduce task input to a single $\mathcal{I}$-file was done to simplify implementation. Finally, to handle `workbuilder` failures, such as, if the `workbuilder` crashes, a new one is started in its place. It then rebuilds state from the per-chunk indexes of the $\mathcal{I}$-files.

In practice, we find that the overheads (CPU/memory) imposed by the `workbuilder` are fairly low. As noted in Section 4.2.3, per-chunk indexes are typically a few KB in size. Consequently, compared to task execution time, the `workbuilder` imposes a relatively low overhead for both retrieving the per-chunk indexes as well as processing them to gather statistics about the intermediate data.

5.2.4 Generating Reduce Task Input From $\mathcal{I}$-files

The mechanisms used to generate reducer input are similar to those in Stock Hadoop. As described in Section 2.1, in Stock Hadoop a reduce task generates its input records by performing a merge on the sorted runs of data it retrieves from the map tasks. Analogously, with `Sailfish`, the reducer input is generated by merging the appropriate sorted runs from each of the $\mathcal{I}$-file chunks. To perform the merge, a reduce task upon startup spawns a child process, `imerger`, which uses the `scan()` API (see Section 4.2.3) for retrieving its records from the chunks of the $\mathcal{I}$-file (step 8 in Figure 7): Since there is no index at the $\mathcal{I}$-file level, the per-chunk indexes are used by `imerger` to (only) retrieve those records that correspond to the key-range assigned to the reduce task. Subsequently, the `imerger` merges the records using a heap-based implementation [17] and then streams the records (ordered by key) to its parent reduce task (step 9 in Figure 7).

As part of the merge step, the `imerger` uses the per-record header information to filter out records generated by failed map tasks (see Section 5.2.1) as well as duplicate records (i.e., by using the per-record source assigned sequence number). For a more detailed explanation of duplicate filtering, we refer the reader to the Sailfish project wiki pages [6].

In terms of failure handling, whenever a reduce task fails, the Hadoop infrastructure detects the failure and spawns a new task. The newly spawned task has the same task id, but different attempt number, when compared to its crashed counterpart. This task contacts the workbuilder to obtain the work assignment and thereby recovers the lost execution.

5.2.5 Recovering Lost Map Task Output

Whenever a chunk of an $\mathcal{I}$-file is lost (e.g., a chunk of an $\mathcal{I}$-file is lost due to disk failure), the records in that chunk are irretrievably lost. Since chunks of an $\mathcal{I}$-file were generated by multiple map tasks appending data via the record append API, the lost data will need to be regenerated by re-executing the appropriate map tasks. To regenerate the lost data, additional bookkeeping information to track the identity of map tasks that wrote to a given $\mathcal{I}$-file chunk has to be maintained. In our implementation, the workbuilder maintains this bookkeeping information and uses it to appropriately trigger re-execution: First, when a map task completes execution, the iappender notifies the workbuilder about the set of chunks it wrote its output to. Second, whenever a chunk is lost, the workbuilder notifies the Hadoop JobTracker to re-run the map tasks that wrote to that chunk. In our implementation, a lost chunk is detected when an imerger is unable to retrieve data from the chunk. The imerger notifies the workbuilder, which then triggers re-execution.

5.3 Disk Seek Analysis

To derive the number of disk seeks involved in the map phase with Sailfish, note that the map output is committed to disk by the chunkservers and then subsequently, read back, sorted, and written back to disk by the chunksorter. The number of seeks is effectively data dependent: Let the number of $\mathcal{I}$-files be i, and the number of chunks in an $\mathcal{I}$-file be c. Now, (1) a lower bound on the number of disk seeks incurred by the chunkservers for writing out the data is $i * c$, and (2) since the chunksorters perform sequential I/O, the minimum number of seeks incurred by the sorters is $2*i*c$. Hence, a lower bound on the number of seeks is $3 * i * c$.

To derive the number of disk seeks involved in the reduce phase with Sailfish, observe that each reduce task retrieves its input from a single $\mathcal{I}$-file and in the worst case must access every chunk of that $\mathcal{I}$-file. With R reducers and c chunks per $\mathcal{I}$-file, the number of disk seeks is proportional to $c*R$. Note that, in contrast to Stock Hadoop in which disk overheads are also dependent on the number of map tasks, the disk overheads with Sailfish are independent of the number of map tasks, but are dependent on the data volume. Therefore, batching intermediate data in to as few chunks as possible is critical for Sailfish.

5.4 Miscellaneous Issues: Topology Aware $\mathcal{I}$-files

The $\mathcal{I}$-file abstraction provides the flexibility of choosing *where* (i.e., local versus remote) to aggregate map output. Fault-tolerance considerations influence this choice. Recall that a data loss which involves a single chunk of an $\mathcal{I}$-file requires all the map tasks that appended to that chunk to be re-executed. With local aggregation, map tasks (from a given job that run on a machine) append their output to $\mathcal{I}$-files whose chunks are stored locally; thus if a disk fails, only map tasks on this machine may be affected. With per-rack aggregation, map tasks write to chunks on the same rack; thus, if a disk fails, only map tasks on that rack may be affected. With global $\mathcal{I}$-files, map tasks can write to any chunk; if a disk fails, *all* map tasks are potentially affected.

While the local approach is best with respect to fault-tolerance, unfortunately, it did not scale on our cluster (which has 4 drives per node) for two reasons. First, the number of files that can be concurrently written to while still obtaining reasonable disk subsystem performance is relatively low on a single machine (viz., about 32 files in our cluster). This causes the number of $\mathcal{I}$-files per job to be small (i.e., 32). Second, for a given volume of data, fewer $\mathcal{I}$-files means that a larger number of reduce tasks will need to retrieve their input from each $\mathcal{I}$-file; effectively this increases the number of scans on each chunk of the $\mathcal{I}$-file which lowers disk throughput. We settled on per-rack aggregation since it provides reasonable fault-containment while allowing for a large number of $\mathcal{I}$-files to be concurrently written (viz., 512 files in our cluster). An evaluation of the various approaches for aggregation based on different cluster/node configurations is outside the scope of this paper.

6 Experimental Evaluation

We deployed our Sailfish prototype in a 150-node cluster in our lab and used it to drive a two-part experimental evaluation.

- The first part of the study involves using a synthetic benchmark to (1) evaluate the effectiveness of $\mathcal{I}$-files in aggregating intermediate data and (2) study the system effects of the Sailfish dataflow path (see Section 6.2).
- The second part of the study involves using Sailfish to run a representative mix of real Map-Reduce jobs with their actual input datasets (see Section 6.3).

In summarizing our results, we find that job completion times with Sailfish are in general faster when compared to the same job run with Stock Hadoop (see Figure 8 and Table 3/Figure 11). There are four aspects to the performance gains:

- $\mathcal{I}$-files enable better batching of intermediate data (see Section 6.2.3). As a result, this leads to higher disk throughput during the reduce phase (see Figure 10) and in turn, translates to a faster reduce phase.
- Due to batching of intermediate data, Sailfish provides better scale when compared to Stock Hadoop (see Figure 8).
- Dynamically planning the execution of the reduce phase enables Sailfish to exploit the parallelism in a data dependent manner (see Table 3 and Section 6.3.2). This approach possibly simplifies program tuning.
- Map-phase execution with Sailfish is in general slower when compared to Stock Hadoop. This is because records are written to the RAM of a remote machine as opposed to local RAM with Stock Hadoop. However, whenever there is a skew in map output, the sorting of the map output can cause the map phase of Stock Hadoop to be slower when compared to Sailfish: with Stock Hadoop, due to the skew, some tasks are able to sort the data entirely in RAM while others incur the overheads of a multi-pass external sort. In contrast, with Sailfish, since map output is aggregated and then sorted, the decoupling allows Sailfish to better parallelize the sorting of map output and thereby better handle the skew (see Figure 12).

In what follows, we describe the details of our setup in Section 6.1 and then present the results of our evaluation.

6.1 Cluster Setup

Our experimental cluster has 150 machines organized in 5 racks with 30 machines/rack. Each machine has 2 quad-core Intel Xeon E5420 processors, 16GB RAM, 1Gbps network interface card, and four 750GB drives configured as a JBOD, and runs RHEL 5.6. The connectivity between *any* pair of nodes in the cluster is 1Gbps.

We run Hadoop version 0.20.2, KFS (version 0.5) with modifications for the key-based variants defined in Section 4.2.1, and the

Parameter	Values
Map tasks per node	6
Reduce tasks per node	6
Memory per map/reduce task	1.5GB
Map-side sort parameters	*io.sort.mb* = 512 *io.sort.factor* = 100 *io.sort.record.percent* = 0.2 *io.sort.spill.percent* = 0.95

(a) Stock Hadoop

Parameter	Values
Map tasks per node	6
Reduce tasks per node	6
Memory per map/reduce task	512MB
Memory per `iappender`	1GB
Memory per `imerger`	1GB

(b) `Sailfish`

Table 2: Parameter settings

other `Sailfish` components. On each machine we run an instance of a Hadoop TaskTracker, a KFS chunkserver, and 4 KFS chunksorter daemon processes (one sorter process per drive). The disks on each machine are used by all the software components.

Parameter Settings: We configure Stock Hadoop using published best practices [19] along with settings from Yahoo! clusters for the Hadoop map-side sort parameters. Table 2(a) shows the parameters we used. Due to the differences in intermediate data handling, the parameter settings for `Sailfish` (shown in Table 2(b)) are different from Stock Hadoop. The total memory budget imposed by either system is similar. Finally, during the experiments none of the nodes in the cluster incurred swapping.

`Sailfish` Notes: For `Sailfish`, we use the rack-aware variant of $\mathcal{I}$-files described in Section 5.4. In the experiments, we limit the number of concurrent appenders per chunk of an $\mathcal{I}$-file to 128, enforced by having each `iappender` reserve 1MB of logical space before it appends records to a chunk. We set the number of $\mathcal{I}$-files to be 512 (the largest possible value given our system configuration). Choosing a large value makes `Sailfish` performance less sensitive to the specific choice. Furthermore, this setting relieves our users from choosing the number of $\mathcal{I}$-files for their specific job. We configure each of the chunksorter deamons to use 256MB RAM. Finally, for the merge involved in generating reducer input, if `imerger` determines that the reducer input exceeds the amount of RAM, it does an external merge. (Our implementation for merging records is similar to that of Stock Hadoop's.)

6.2 Evaluation With Synthetic Benchmark

In this part of the study, we evaluate `Sailfish` for handling intermediate at scale (viz., for data volumes ranging from 1TB to 64TB). We then discuss aspects of the `Sailfish` dataflow path as it relates to (1) packing intermediate data in chunks, (2) overheads imposed by chunksorter daemon, and (3) system effects of aggregating map output on a rack-wide basis. We begin by describing our synthetic benchmark program and then present the results.

6.2.1 Benchmark Description

To highlight the overheads of transporting intermediate data in isolation, we implemented a synthetic MapReduce job in which, intentionally, there is *no* job input/output. Our program, `Benchmark`, performs a partitioned sort: (1) each map task generates a configurable number of records (namely, strings with 10-byte key, 90-byte value over the ASCII character set), (2) the records are hash-partitioned, sorted, and merged and then provided as input to the reduce task, and (3) each reduce task validates its input records and discards them. Our `Benchmark` is very similar to the Daytona Sort benchmark program that is used in data sorting competitions [7]. Finally, with `Benchmark`, there is *no skew*: (1) all map tasks generate an equal amount of data such that the keys are uniformly random and (2) all reduce tasks process roughly the same number of keys.

6.2.2 Handling intermediate data at scale

For scale, we ran `Benchmark` while varying the volume of intermediate data generated by the map tasks from 1TB to 64TB. For both Stock Hadoop and `Sailfish`, we configure the number of mapper tasks such that each mapper generates 1GB of output. For the reduce phase, (1) with Stock Hadoop we provide a value for the number of reduce tasks and (2) with `Sailfish` we configure the `workbuilder` process to assign each reduce task approximately 1GB of data. In the experiments, the number of map/reduce tasks varied from 1024 (for handling 1TB of data) to 65536 (for handling 64TB of data).

Figure 8 shows the results of our experiments. A key takeaway from this graph is that the performance of `Sailfish` for handling intermediate data scales linearly even upto large volumes of data (viz., 64TB). On the other hand, the performance of Stock Hadoop grows non-linearly as the volume of intermediate data to be transported begins to exceed 16TB.

The following discussion focusses on the system characteristics during the reduce phase of execution. We defer the discussion of the map phase of execution to Section 6.2.5.

Recall that, in this set of experiments, the amount of input data to a reduce task is approximately 1GB. Based on the parameter settings, the reducer input fits entirely in RAM. Furthermore, in both systems, a reducer retrieves its input from the multiple sources concurrently: with Stock Hadoop, a reduce task obtains its input multiple mapper machines (viz., 30 by default) in parallel; with `Sailfish`, an `imerger` issues concurrent reads to all the chunks of the $\mathcal{I}$-file. However, the difference between the two systems is in the efficiency with which the reduce task obtains its input, namely, the amount of data read per seek which effectively determines the disk throughput that can be achieved.

For Stock Hadoop, Section 2.2 details why data retrieved per I/O shrinks and why this hurts its performance: the amount of data a reducer pulls from a mapper, on average, is $(1GB/R)$. For `Sailfish`, since the number of $\mathcal{I}$-files is fixed (i.e., 512), there is an increase in both the number of chunks in an $\mathcal{I}$-file as well as the number of reduce tasks assigned to a given $\mathcal{I}$-file. While the amount of data consumed by a reduce task is fixed (namely, 1GB), this data is spread over almost all the chunks of the $\mathcal{I}$-file. Consequently, the amount of data retrieved per I/O by a reduce task from a single $\mathcal{I}$-file chunk begins to decrease. However, due to better batching (see Section 6.2.3), the amount of data read per I/O with `Sailfish` is an order of magnitude higher when compared to Stock Hadoop (see Figure 9). The difference in the amount of data read per seek translates to higher disk read throughput for `Sailfish` in the reduce phase leading to better job performance. We highlight this effect next.

Figure 10 shows the disk throughput obtained with Stock Hadoop as well as `Sailfish` for runs of `Benchmark` in which the volume of intermediate data is 16TB. Given our 1GB limit of data for each map or reducer task, this job involved executing 16384 mappers and 16384 reducers. For Stock Hadoop, the average amount data retrieved by a reducer from a map task is about 70KB. For `Sailfish`, the average amount data retrieved by a reducer from an $\mathcal{I}$-file chunk is about 1.5MB. With fewer seeks and higher amount of data read per seek, the disk read throughput obtained by `Sailfish` on a single machine averages to about 35MB/s. On the other hand, with Stock Hadoop, due to higher seeks and less amount of data read per seek, the observed disk throughput averages to about 20MB/s. As a result, this effect causes the reduce phase in Stock Hadoop to be substantially longer when compared to `Sailfish`'s reduce phase for the same job (viz., 3.5 hours when compared to 1.75 hours).

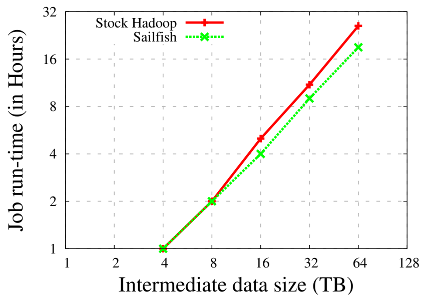

Figure 8: Variation in job run-time with the volume of intermediate data. Note that the axes are log-scale.

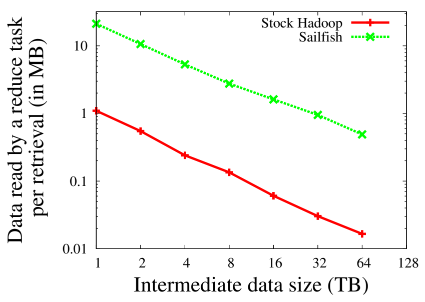

Figure 9: Data read per retrieval by a reduce task with Stock Hadoop and `Sailfish`.

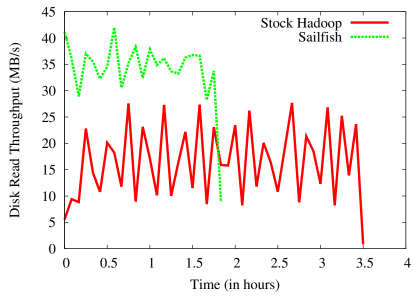

Figure 10: Disk read throughput in the reduce phase with `Sailfish` is higher and hence reduce phase is faster (int. data size = 16TB).

Note that our implementation of `Sailfish` can be tuned further. For instance, rather than sorting individual chunks when they become stable, multiple chunks can be locally aggregated and sorted. This optimization increases the length of the "sorted runs" which can lead to better scale/performance. That is, when compared to Figure 9, this optimization can increase the data read per retrieval at larger values of intermediate data size.

Finally, there is also the issue of implementation differences between the two systems when transporting intermediate data. Based on the above results (coupled with the observation that there is no data skew), much of the gains in `Sailfish` are due to better disk I/O in the reduce phase. Therefore, unless the I/O sizes in both systems are comparable, we do not believe the implementation differences have a significant impact on performance.

6.2.3 Effectiveness of $\mathcal{I}$-files

For a given $\mathcal{I}$-file our atomic record append implementation tries to minimize the number of chunks required for storing the intermediate data. The experimental results validated the implementation. The chunk allocation policy ensured that the number of chunks per $\mathcal{I}$-file was close to optimal (i.e., size of $\mathcal{I}$-file / KFS chunksize). Furthermore, except for the last chunk of an $\mathcal{I}$-file, the remaining chunks were over 99% full. This is key to our strategy of maximizing batching: a metric discussed in Section 4.2.4 was the number of files used to store intermediate data.

6.2.4 Chunk sorting overheads

For $\mathcal{I}$-files, whenever a chunk becomes stable, the chunkserver utilizes the chunksorter daemon to sort the records in the chunk. The chunksorter daemon is I/O intensive and performs largely sequential I/O: First, it spends approximately 2-4 seconds loading a 128MB chunk of data into RAM. Second, it spends approximately 1-2 seconds sorting the keys using a radix trie algorithm. Finally, it spends approximately 2-4 seconds writing the sorted data back to disk[4].

6.2.5 Impact of network-based aggregation

The improvements in the reduce phase of execution with `Sailfish` come at the expense of an additional network transfer. During the map phase, in Stock Hadoop, a map task writes its output to the local filesystem's buffer cache (which writes the data to disk asynchronously). With `Sailfish`, the map output is committed to RAM on remote machines (and the chunkserver aggregates and writes to disk asynchronously). This additional network transfer causes the map phase of execution to be higher by about 10%.

From a practical standpoint, aggregating map output on per-rack basis (see Section 5.4) minimizes the impact of the additional traffic on the network for two reasons. First, clusters for data intensive computing are configured with higher intra-rack connectivity when compared to inter-rack capacity. For instance, in Yahoo!'s clusters, the connectivity is 1Gbps between any pair of intra-rack nodes when compared to 200Mbps between inter-rack nodes. Second, due to locality optimizations in the Hadoop job scheduler (i.e., schedule tasks where the data is stored) the intra-rack capacity is relatively unused and `Sailfish` leverages the unused capacity.

Finally, network capacity within the datacenter is slated to substantially increase over the next few years. Clusters with 10Gbps inter-node connectivity (on a small scale) have been deployed [25]; larger clusters with such connectivity will be commonplace. We expect `Sailfish` to be deployed in such settings, where maximizing the achievable disk subsystem bandwidth and in turn effectively utilizing the available network bandwidth becomes paramount.

6.3 Evaluation With Actual Jobs

For this part of the study we first construct a job mix by developing a simple taxonomy for classifying Map-Reduce jobs in general. Our taxonomy is based on an analysis of jobs we see in Yahoo!'s clusters (see Section 6.3.1). Using this taxonomy, we handpicked a representative mix of jobs to drive an evaluation using actual datasets and jobs (from data mining, data analytics pipelines) run in production clusters at Yahoo!. We then present the results of our evaluation.

6.3.1 Constructing a Job Mix

Based on conversations with our users as well as an analysis of our cluster workloads, we use the following taxonomy for classifying MapReduce jobs in general:

1. **Skew in map output**: Data compression is commonly used in practice, and users organize their data so as to obtain high compression ratios. As a result, the number of input records processed by various map tasks can be substantially different, and this impacts the size of the output of the task.

2. **Skew in reduce input**: These are jobs for which some partitions get more data than others. The causes for skew include poor choice of partitioning function, low entropy in keys, etc.

3. **Incremental computation**: Incremental computation is a commonly used paradigm in data intensive computing environments. A typical use case is creating a sliding window

[4]Since writing out the sorter output file is a large sequential I/O, as a performance optimization, our implementation uses the `posix_fallocate()` API for contiguous disk space allocation.

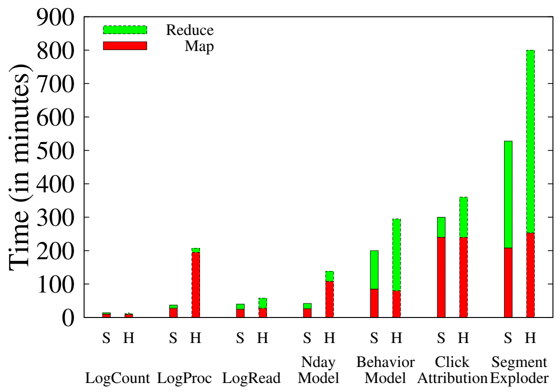

Figure 11: Time spent in the Map and Reduce phases of execution for the various MapReduce jobs. At scale, `Sailfish` (S) outperforms Stock Hadoop (H) between 20% to a factor of 5.

over a dataset. For instance, for behavioral targeting, N-day models of user behavior are created by a key-based join of a 1-day model with the previous N-day model.

4. **Big data**: These are data mining jobs that process vast amounts of data, e.g., jobs that process a day of server logs (where the daily log volume is about 5TB in size). With these jobs, the output is proportional to the input (i.e., for each input record, the job generates an output record of proportional size).

5. **Data explosion**: These are jobs for which the output of the map step is a multiple of the input size. For instance, to analyze the effectiveness of an online ad-campaign by geo-location (e.g., impressions by (1) country, (2) state, (3) city), the map task emits multiple records for each input record.

6. **Data reduction**: These are jobs in which the computation involves a data reduction step which causes the intermediate data size (and job output) to be a fraction of the job input. For example, there are jobs that compute statistics over the data by processing a few TB of input but producing only a few GB of output.

Table 3 shows the jobs that we handpicked for our evaluation. We note that several of these are Pig scripts containing joins and co-grouping, and produce large amounts of intermediate data. Of these jobs, BehaviorModel, ClickAttribution are CPU and data intensive, while the rest are data intensive. Finally, note that in all of these jobs, with the exception of LogCount, there is *no* reduction in the intermediate data size when compared to the job input's size.

6.3.2 Evaluation With Representative Jobs

Hadoop best practices [19] recommend using compression to minimize the amount of I/O when handling intermediate data. Hence, for this set of experiments, for handling intermediate data we enabled LZO-based compression with Stock Hadoop and extended our `Sailfish` implementation to support an LZO codec.

Table 3 shows the data volumes for the various jobs as well as the number of map/reduce tasks. Note that multiple waves of map/reduce tasks per job is common.

For this set of experiments, the `workbuilder` was configured to assign upto 2GB of data per reduce task (independent of the job). This value represents a trade-off between fault-tolerance (i.e., amount of computation that has to be re-done when a reducer fails) versus performance (i.e., a large value implies fewer reducers, possibly improving disk performance due to larger sequential I/Os). As part of follow-on work [8], we are exploring ways of eliminating this parameter. This would then allow the reduce phase of execution to be adapted completely dynamically based on the available cluster resources (viz., CPUs).

Figure 11 shows the results of running the various jobs using Stock Hadoop as well as `Sailfish`. Our results show that as the volume of intermediate data scales, job completion times with `Sailfish` are between 20% to 5x faster when compared to the same job run with Stock Hadoop. There are three aspects to the gains:

- **Using $\mathcal{I}$-files for aggregation**: In terms of the reduce phase of computation, except for the LogProc and LogRead jobs in which the volume of intermediate data is relatively low (see Table 3), for the remaining jobs there is a substantial speedup with `Sailfish`. The speedup in the reduce phase is due to the better batching of intermediate data in $\mathcal{I}$-files, similar to what we observed with `Benchmark`.

- **Decoupling sorting from map task execution:** From our job mix, we found that skew in map output impacted LogProc and NdayModel jobs: (1) in the LogProc job, a few of the map tasks generated as much as 30GB of data, and (2) in the NdayModel job, which involves a `JOIN` of an N-day dataset with a 1-day dataset, about half the map tasks that processed files from the N-day dataset generated about 10GB of data while the remaining tasks generated 450MB of data. Figure 12 shows the distribution of map task completion times for NdayModel job. While the skew affects map task completion times in both Stock Hadoop and `Sailfish`, the impact on Stock Hadoop due to the sorting overheads incurred by map tasks is much higher. This result validates one of our design choices: decoupling the sort of map output from map task execution. In these experiments, particularly for the LogProc job, such a separation yielded upto a 5x improvement in application run-time.

- **Making reduce phase dynamic:** Dynamically determining the number of reduce tasks and their work assignment in a data dependent manner helps in skew handling as well as in automatically exploiting the parallelism in the reduce phase. We illustrate these effects using the LogRead job in which there is a skew in the intermediate data (particularly, as Figure 13 shows, partitions 0-200 had more data than the rest—4.5GB vs 0.5GB). As shown in Table 3 `Sailfish` used more reduce tasks than Stock Hadoop (800 compared to 512), and proportionately more reducers were assigned to those partitions (i.e., as shown in Figure 14, with 2GB of data per reduce task, $\mathcal{I}$-file$_0$ to $\mathcal{I}$-file$_{200}$ were assigned 3 reducers per $\mathcal{I}$-file while the remaining $\mathcal{I}$-files were assigned 1 reducer apiece). As a result, by better exploiting the available parallelism, the reduce phase in `Sailfish` is much faster compared to Stock Hadoop. Our approach realizes these benefits in a seamless manner *without* re-partitioning the intermediate data and simplifies program tuning.

Finally, to study the effect of change in data volume, we ran the ClickAttribution job using `Sailfish` where we increased the input data size (from 25% to 100%). We found that the `workbuilder` deamon automatically caused the number of reduce tasks to increase proportionately (i.e., from 4096 to 8192) in a data dependent manner.

Job Name	Job Characteristics	Operators	Input size	Int. data size	# of mappers	# of reducers		Run time	
						Stock Hadoop	`Sailfish`	Stock Hadoop	`Sailfish`
LogCount	Data reduction	`COUNT`	1.1TB	0.04TB	400	512	512	0:11	0:14
LogProc	Skew in map output	`GROUP BY`	1.1TB	1.1TB	400	1024	1024	3:27	0:37
LogRead	Skew in reduce input	`GROUP BY`	1.1TB	1.1TB	400	512	800	0:58	0:40
NdayModel	Incr. computation	`JOIN`	3.54TB	3.54TB	2000	4096	4096	2:18	0:42
BehaviorModel	Big data job	`COGROUP`	3.6TB	9.47TB	4000	4096	5120	4:55	3:15
ClickAttribution	Big data job	`COGROUP, FILTER`	6.8TB	8.2TB	21259	4096	4096	6:00	5:00
SegmentExploder	Data explosion	`COGROUP, FLATTEN, FILTER`	14.1TB	25.2TB	42092	16384	13824	13:20	8:48

Table 3: Characteristics of the jobs used in the experiments. The data sizes are post-compression. The job run times reported in this table are end-to-end (i.e., from start to finish). As data volumes scale, `Sailfish` outperforms Stock Hadoop between 20% to a factor of 5. See Figure 11 for a break-down in the time spent in Map/Reduce phases of the job.

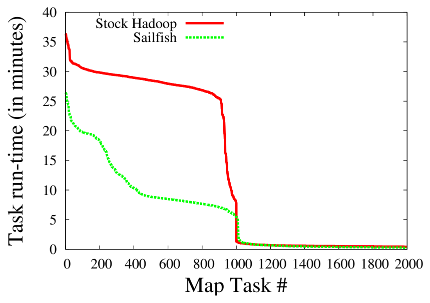

Figure 12: Distribution of map task run time for NdayModel job. Skew in map output sizes affects task completion times for both Stock Hadoop and `Sailfish`, but the impact for Stock Hadoop is much higher.

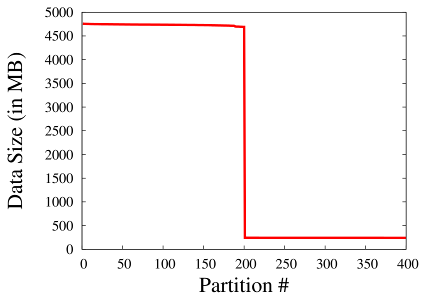

Figure 13: Distribution of the size of the intermediate data files in LogRead job. For this job there is a skew in distribution of data across partitions (i.e., skew in reducer input).

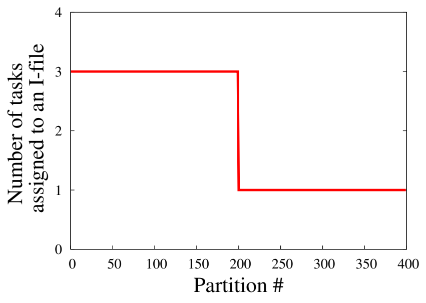

Figure 14: For the LogRead job `Sailfish` handles skew in intermediate data by assigning reduce tasks in proportion to the data in the $\mathcal{I}$-file (see Figure 13).

6.3.3 Impact of data loss in $\mathcal{I}$-files

In our setting the intermediate data is not replicated: Stock Hadoop does not implement it and for a fair comparison, we did not enable replication for $\mathcal{I}$-files. Hence, in the event of data loss (viz., caused by a disk failure), the lost data has to be regenerated via recomputation. When a disk fails, the required recomputations on the two systems are:

- Stock Hadoop: Since map tasks store their output on the local disks by arbitrarily choosing a drive, the expected number of recomputes is: $\frac{\text{\# of tasks run on a node}}{\text{\# of drives on a node}}$.
- `Sailfish`: With 512 $\mathcal{I}$-files and 30 machines per rack, with per-rack $\mathcal{I}$-files, a map task running on a node will write to all the chunkservers in the rack. Since a chunkserver on a node arbitrarily chooses a drive to store a chunk file, the expected number of recomputes is: $\frac{\text{\# of tasks run on a rack}}{\text{\# of drives on a node}}$.

Though these recompute tasks can be run in parallel, their effect on job runtimes is data dependent. For jobs where there is a skew in map output, the cost of recompute with Stock Hadoop is much higher than `Sailfish`. For instance, for the LogProc job in which over 90% of the job run-time is in the map phase of computation, recomputes can cause the overall job run-time to nearly double: The run-time with Stock Hadoop increases from 2:18 to 4:06, while with `Sailfish` it increases from 0:42 to 1:08. For the other jobs where there is no skew in map output, regenerating the lost data requires about 1 to 2 waves of map task execution which causes a 10-20% increase in job runtime in either system.

7 Related Work

The performance of Hadoop Map-Reduce framework has been studied recently [16, 22, 27]. In [22], they find that Hadoop job performance is affected by factors such as, lack of a schema which impacts selection queries, lack of an index which affects join performance, and data parsing overheads. Reference [12] shows how to improve Hadoop performance by augmenting data with a schema as well as an index and then using the augmented information to substantially speed up select/join operators. These mechanisms are applicable with `Sailfish` as well. In [16], they identify additional factors that affect Hadoop performance such as, I/O mode for accessing data from disk, and task scheduling. The paper also describes ways to tune these factors to improve performance by 2.5 to 3.5x. The same paper also notes that another potential source of performance improvement is in modifying how the intermediate data is handled. Our work addresses this aspect and our results demonstrate substantial gains.

TritonSort [24, 25] was developed for doing large-scale data sorting. Recently, in an effort that parallels our `Sailfish` work, the TritonSort architecture has been extended with a Map-Reduce implementation called *ThemisMR* [23]. While an experimental comparison of the two systems is outside the scope of this paper, we briefly describe the ThemisMR design and then contrast it with `Sailfish`. The design goals of ThemisMR are similar to some our objectives in building `Sailfish`: (1) ThemisMR focuses on optimizing disk subsystem performance when handling intermediate data at scale and (2) ThemisMR tries to mitigate skew in reducer input. With ThemisMR, a Map-Reduce computation consists of two distinct phases, namely *map and shuffle* followed by *sort and* *reduce*. For handling intermediate data, the first phase involves large sequential writes, while the second phase involves large sequential reads. Through careful buffer management their design ensures that intermediate data touches disk exactly twice using I/O's that are long/sequential thereby maximizing disk subsystem performance. Next, for mitigating skew in reducer input, ThemisMR contains an optional sampling phase which is used to determine partition boundaries. Finally, ThemisMR considers a point in the design space where cluster sizes are small (on the order of 30-100 nodes) in which component failures are rare and hence, forgoes fault-tolerance (i.e., entire job must be re-run whenever there is a failure). However, for large clusters consisting of 100's to 1000's of nodes, it is well-known that failures are not uncommon [9]. For large clusters, the ThemisMR paper [23] notes that requiring entire jobs to be re-run whenever there is a failure can adversely impact performance. Contrasting the two systems, (1) `Sailfish` provides fault-tolerance while still improving application performance and (2) `Sailfish` tries to mitigate skew in reducer input without an explicit sampling step. In addition, within each of the two phases of ThemisMR, to avoid (unnecessary) spilling of data to disk (e.g., under memory pressure) their design relies on a memory manager that forces data generation to appropriately stall. However, as noted in their paper, carefully choosing memory management policies and tuning them to maximize performance (such as, by avoiding deadlocks and by minimizing stalls) is non-trivial.

Dealing with skew in the context of Map-Reduce has been studied by [18, 20, 23, 28]. In these systems, a job is executed twice: The first execution samples the input dataset to determine the split points, which are then used to drive the actual execution over the complete dataset. The objective here is to minimize the skew in intermediate data (i.e., skew in the reducer input). With `Sailfish`, by gathering statistics over the data at run-time, we try to achieve the same objective without requiring an explicit sampling step.

An alternate approach for handling skew in reducer input is to adaptively change the map-output partitioning function [29] for Hadoop jobs. In their work, the number of partitions is an input parameter and is fixed apriori; then, by sampling the output of a small number of map tasks, they mitigate skew by dynamically constructing the partitioning function (i.e., split points) such that the partitions will be balanced. Their methodology is similar to `Sailfish` in that they mitigate skew by sampling the intermediate data. However, since they assign one reduce task per partition and the number of partitions is fixed apriori, this parameter has to be carefully chosen. In particular, as the volume of intermediate scales, and jobs are run with larger number of partitions, while their techniques may mitigate skew, the performance gains are limited by Hadoop's intermediate data handling mechanisms.

Augmenting datasets with an index, particularly *after* the dataset has been generated is known to be expensive. For instance, in [12], they find that the one-time cost involved in building an index over a 2TB input dataset using 100 nodes takes over 10 hours. In contrast, with `Sailfish` the augmentation of an index to an $\mathcal{I}$-file chunk is done as part of intermediate data generation and hence, incurs little overhead.

Starfish [14] uses job profiling techniques to help tune Hadoop parameters including those related to handling of intermediate data (i.e., the map-side sort parameters and the number of reduce tasks). With Starfish, the computation has to be run once to obtain the job profile and it then suggests input parameter values for subsequent runs of the same job. The dynamic data-driven approach to parameter tuning in `Sailfish` achieves the same gains without having to run the job once to determine the job profile. Further, as our analysis and results show, the gains achievable by tuning Hadoop are inherently limited by Hadoop's mechanisms for handling intermediate data; `Sailfish` improves performance further by better batching of disk I/O.

8 Summary and Future Work

We presented `Sailfish`, an alternate Map-Reduce framework built around the principle of aggregating intermediate data for improved disk I/O. To enable aggregation, we developed $\mathcal{I}$-files as an abstraction, implemented as an extension of the distributed filesystem. Our `Sailfish` prototype runs standard Hadoop jobs, with no changes to application code, but uses $\mathcal{I}$-files to transport intermediate data (i.e., the output of the map step). We demonstrated both improved performance and less dependence on user-tuning.

As part of on-going research, there are several avenues of work that we are currently exploring. First, by adding a adding a feedback loop to the reduce phase of `Sailfish` it becomes possible to re-partition the work assigned to a reduce task at a key-boundary [8]. Such dynamic re-partitioning enables elasticity for the reduce phase, thereby improve utilization in multi-tenanted clusters. Second, for mitigating the impact of failures, we are evaluating mechanisms for replicating intermediate data thereby minimizing the number of recomputes. Third, $\mathcal{I}$-files provide new opportunities for debugging, particularly, the reduce phase of a MapReduce job, saving valuable programmer time.

The `Sailfish` design is geared towards computations in which the volume of intermediate data is large. As we noted in Section 1, for a vast majority of the jobs in the cluster, the volume of intermediate data is small. For such jobs alternate implementations for handling intermediate data may afford better performance. Though the current versions of the Hadoop framework forces *all* jobs to use the same intermediate data handling mechanism, the next generation of the Hadoop framework (namely, YARN [1]) relaxes this restriction. The YARN architecture includes hooks for customizing intermediate data handling, including a per-job *application master* that coordinates job execution. Incorporating many of the core ideas from this paper into an application master and task execution layer is the focus of ongoing work [5].

`Sailfish` is currently deployed in our lab and is being evaluated by our colleagues at Yahoo!. We have released `Sailfish` and the other software components developed as part of this paper as open source [6].

9 Acknowledgements

The authors would like thank Tyson Condie for providing detailed feedback and insightful suggestions on early drafts. We thank Chris Douglas, Carlo Curino, and the anonymous reviewers for helpful comments and feedback. We also thank Igor Gashinsky for providing us the cluster used in our experiments.

10 References

[1] Apache Hadoop NextGen MapReduce (YARN). `http://hadoop.apache.org/docs/r0.23.0/hadoop-yarn/hadoop-yarn-site/YARN.html`.

[2] Apache Hadoop Project. `http://hadoop.apache.org/`.

[3] HDFS. `http://hadoop.apache.org/hdfs`.

[4] KFS. `http://code.google.com/p/kosmosfs/`.

[5] Preemption and restart of mapreduce tasks. `http://issues.apache.org/jira/browse/MAPREDUCE-4585`.

[6] Sailfish. `http://code.google.com/p/sailfish/`.

[7] Sort benchmark home page. http://sortbenchmark.org/.

[8] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True Elasticity in Multi-Tenant Clusters through Amoeba. In *ACM Symposium on Cloud Computing*, SoCC'12, October 2012.

[9] J. Dean. Software engineering advice from building large-scale distributed systems. http://research.google.com/people/jeff/stanford-295-talk.pdf.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, 2004.

[11] J. Dean and S. Ghemawat. Mapreduce: A flexible data processing tool. *Communications of the ACM*, 53(1):72–77, January 2010.

[12] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.*, 3(1), 2010.

[13] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, volume 37 of *SOSP '03*, pages 29–43, New York, NY, USA, Oct. 2003.

[14] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. *Systems Research*, pages 261–272, 2011.

[15] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, 2007.

[16] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: an in-depth study. *Proc. VLDB Endow.*, 3(1), Sept. 2010.

[17] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley Professional, second edition, May 1998.

[18] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 75–86, New York, NY, USA, 2010. ACM.

[19] A. Murthy. Apache hadoop: Best practices and anti-patterns. http://developer.yahoo.com/blogs/hadoop/posts/2010/08/apache_hadoop_best_practices_a/.

[20] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, 2008.

[21] J. Ousterhout et al. The case for ramclouds: Scalable high-performance storage entirely in dram. *SIGOPS Operating Systems Review*, 43(4):92–105, December 2009.

[22] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM.

[23] A. Rasmussen, M. Conley, R. Kapoor, V. The Lam, G. Porter, and A. Vahdat. ThemisMR: An I/O Efficient MapReduce. Technical Report CS2012-0983, Department of Computer Science and Engineering, University of California at San Diego, July 2012.

[24] A. Rasmussen, M. Conley, G. Porter, and A. Vahdat. Tritonsort 2011. http://sortbenchmark.org/2011_06_tritonsort.pdf.

[25] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. Tritonsort: a balanced large-scale sorting system. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, Berkeley, CA, USA, 2011.

[26] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, D. Reeves. Sailfish: A framework for large scale data processing. Technical Report YL-2012-002, Yahoo! Labs.

[27] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53(1):64–71, Jan. 2010.

[28] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2), 2009.

[29] R. Vernica, A. Balmin, K. S. Beyer, and V. Ercegovac. Adaptive MapReduce using Situation-Aware Mappers. In *International Conference on Extending Database Technology (EDBT)*, 2012.