High Performance Computing Paper Review

Hiroki Kanezashi

13M38152

Reviewed Paper 1

Sailfish: a framework for large scale data processing

[SoCC '12 Proceedings of the Third ACM Symposium on Cloud Computing]

Sriram Rao[1], Raghu Ramakrishnan[1], Adam Silberstein[2], Mike Ovsiannikov[3], Damian Reeves[3]

[1]Microsoft Corp, [2]LinkedIn, [3]Quantcast Corp

Reviewed Paper 2

Breaking the speed and scalability Barriers for Graph exploration on distributed-memory machines

[2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)]

Checconi F, Petrini F, Willcock J, Lumsdaine A, Choudhury A.R, Sabharwal Y.

IBM TJ Watson, Yorktown Heights, NY, USA

Reviewed Paper 3

Parallel breadth-first search on distributed memory systems

[SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis]

Aydin Buluç and Kamesh Madduri

Lawrence Berkeley National Laboratory, Berkeley, CA

Summarized Abstract

- Developed *Sailfish*: MapReduce framework for large scale data processing.
- *Sailfish* improved performance of Hadoop by 20% ~ 5 times on real jobs and datasets.
- *Sailfish* design enabled auto-tuning functionality that changes data volume and distributions effectively.

Outline

1. Introduction
2. Intermediate Data Handling
3. Batching Data I/O
4. I-files for Aggregating Intermediate Data
5. *Salifish*: MapReduce Using I-files
6. Evaluations
7. Related Work
8. Summary

1. Introduction

- Data intensive computing applications commonly process several tens of terabytes.
  - These applications run on large clusters by using parallel dataflow graph frameworks.
  - These frameworks enable to simplify procedures like task scheduling, handling data transferred between computation steps(**intermediate data**).

Contributions of this paper

- Optimized the transport of intermediate data in distributed dataflow systems.
- Found that data managing for disk I/O should be a core design principle.
- Developed I-files to support batching of data.
- Developed and demonstrated Sailfish, a new MapReduce framework.

2. Intermediate Data Handling

- Current MapReduce implementations have a problem about performance while intermediate data handling.
  - For example, Hadoop stores intermediate data to RAM, but sometimes spills them to disk.

Current Approaches (Hadoop)

1. Handles intermediate data using merge-sort.
2. Spills data from RAM to a file on disk.
3. The map task merges the spills to a file.
4. Each reduce task pull data from mappers' output files.
5. Reducer merges data.

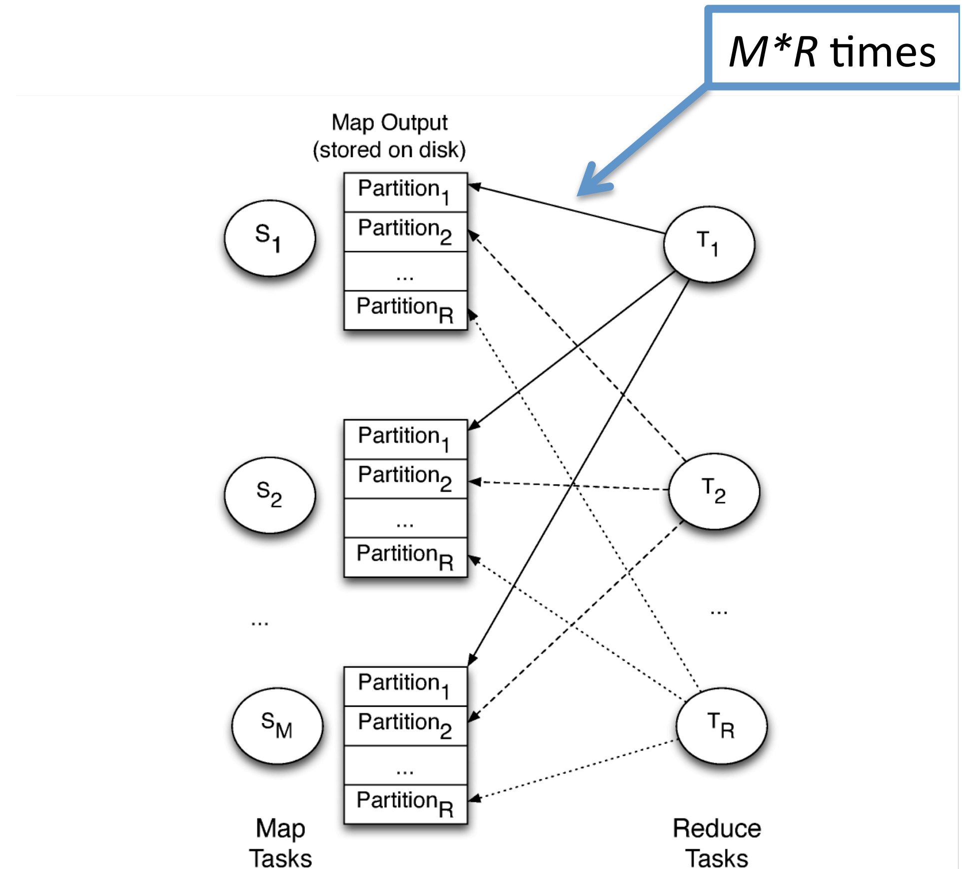


Figure 1: A reduce task retrieves its input from each of the map tasks. The number of distinct retrievals is proportional to $M * R$ and the data read per retrieval is proportional to $1/R$.

Cost of handling intermediate data

- It is dominated by the rate at which data can be read from the disk subsystems.
  - Disk performance is affected by the amount of data read per the number of disk seek.
- If memory-based filesystem buffer caches cannot mask disk seeks, overhead of them affects throughput.

The number of Mappers and Reducers

- There are many mappers and reducers, and the number of distinct retrievals is the product of them.
- The amount of data retrieved by a reduce task is proportional to the number of reducer tasks.

➢ The amount of data read per disk seek will decreases but the number of disk seeks grows super-linearly.

Inefficiency of Performance

- Hadoop performance degrades non-linearly.
- The reason is disk overheads involved in the data transfer.

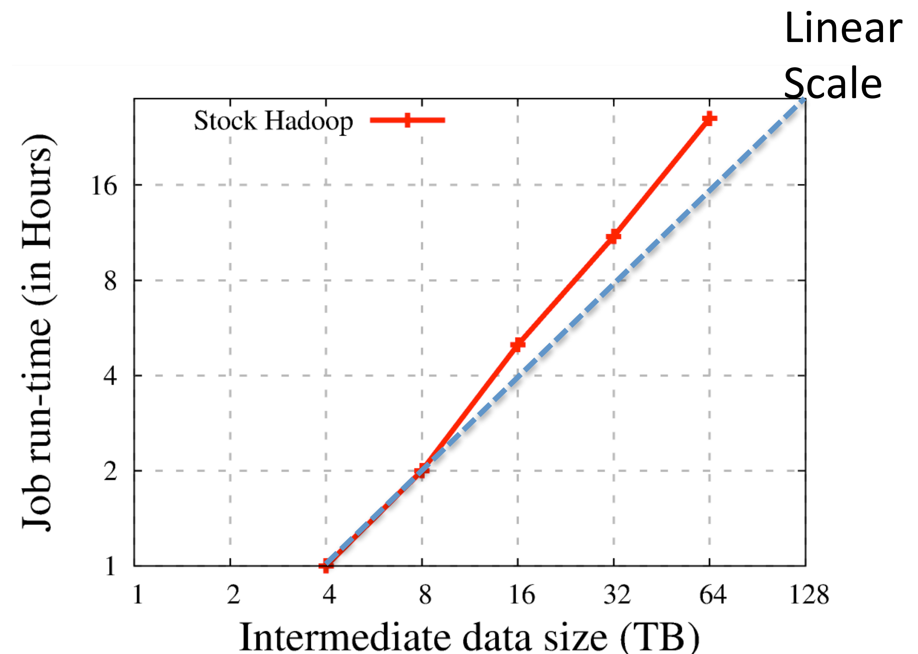


Figure 2: Hadoop performance vs. intermediate data size

System parameter tuning

- Users have to tune system parameters of such parallel dataflow frameworks.
- However, many programmers set parameters only once and rarely do it further.
- Data volumes will change continuously, so performance will degrade without tuning.

3. Batching Data I/O

- Used “blocking step” techniques.
  - Already exists in MapReduce, SQL Systems, Pig
- Used MapReduce as a sample application.
  - Every step in the flow is blocking.

Clusters for data intensive computing

- Using commodity hardware
  - Hard disks are currently the only cost-effective and high capacity storage.
  - Only focus on minimizing the disk overheads.
- Other storage systems to avoid some disk overheads are not yet viable.
  - RAM-based system will be expensive.
  - Using SSD is not applicable for multi-terabyte scales.

Intermediate Data Aggregation

- The number of reduce tasks is reduced from M*R to R.
- Enhanced the distributed file system.

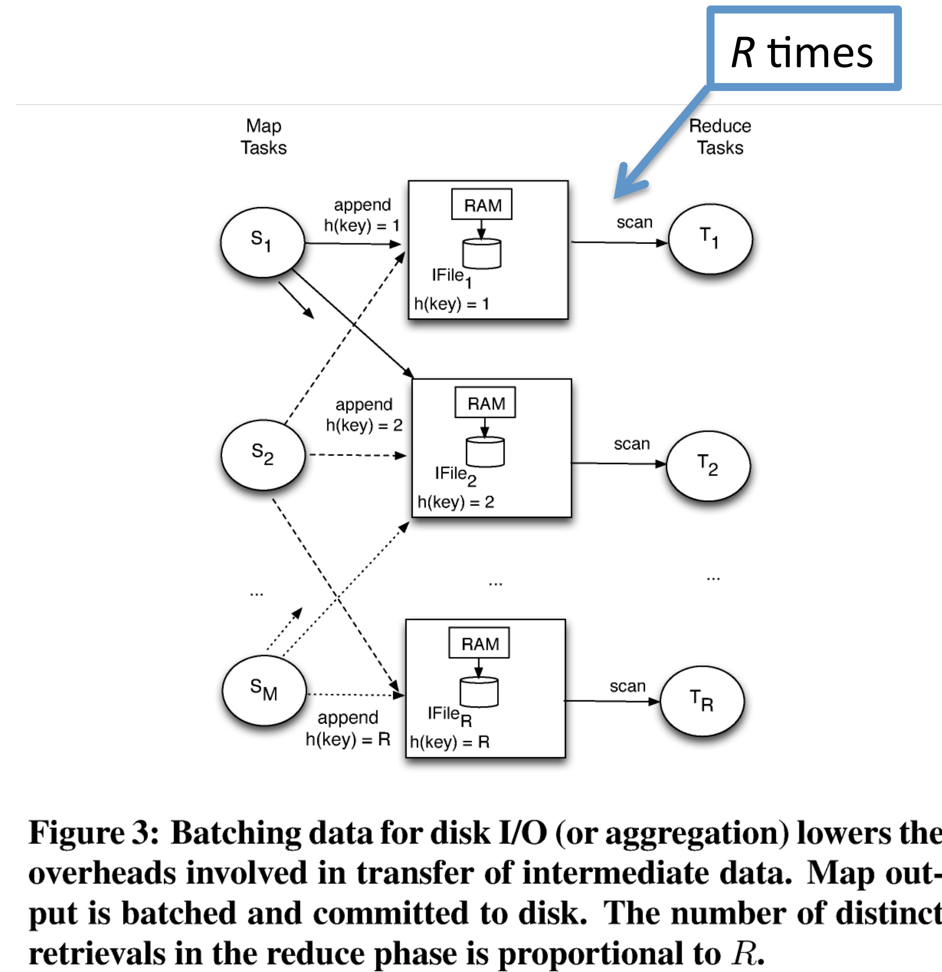


Figure 3: Batching data for disk I/O (or aggregation) lowers the overheads involved in transfer of intermediate data. Map output is batched and committed to disk. The number of distinct retrievals in the reduce phase is proportional to R.

4. I-files for Aggregating Intermediate Data

- Extended the KFS to implement the I-file abstraction besides HDFS.
  - KFS already contains some I-file features.
  - KFS is designed for handle large files in clusters.

Adapting KFS to Support I-files

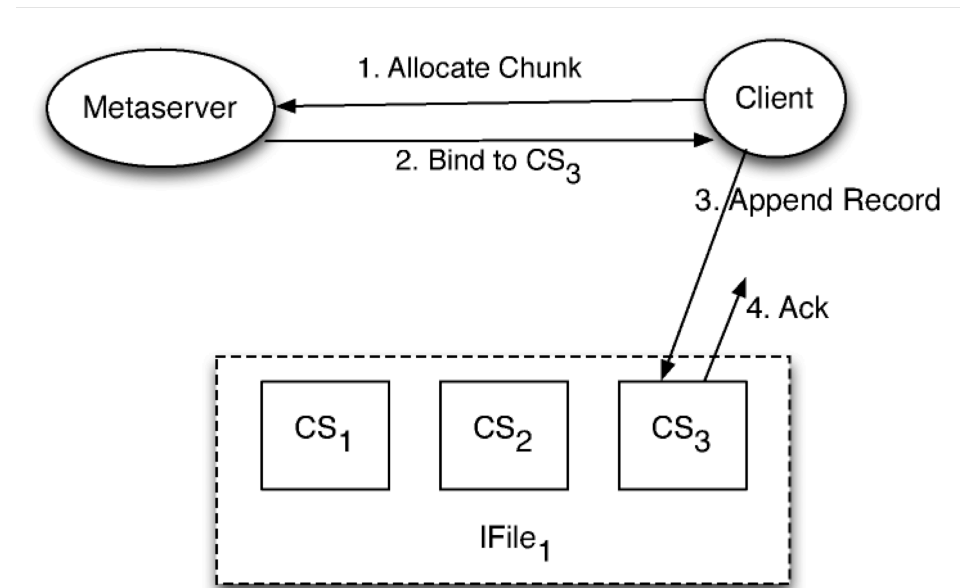
- I-files is different from KFS in:
  - File chunks is append-only (primitive).
  - Once a chunk is closed for writing, it is immutable.
- Set rules to I-files for data aggregation.
  - Restricts the number of writers for an I-file.
  - Allows multiple chunks of I-file to be appended to.

I-file APIs to support record-based I/O

- create_ifile(filename)
  - Creates an I-file
- record_append(fd, <key, value>)
  - Writes(appends) records to an I-file.
- scan(fd, buffer, lower_key, upper_key)
  - Retrieves records from an I-file.
  - Data is specified by key range.

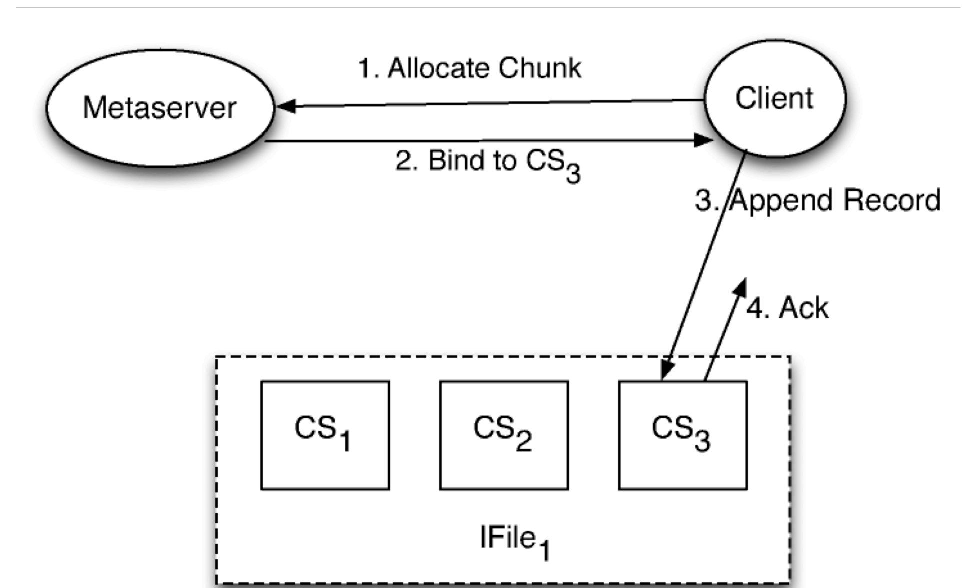
Appending Records to an I-file

1. Client send an allocate request to KFS metaserver to write a record to an I-file.
2. If there is an available chunk, this server binds the client to a chunkserver(CS).
   – If not, it allocates new chunk.



Appending Records to an I-file

3. The client sends the record to the bound chunkserver.
4. When client receives an ACK message, client considers it succeeds.
   – If fails to receive, It will retry. After failing for some time, gives up binging to chunkserver.



5. *Salifish*: MapReduce Using I-files

- It is a MapReduce framework replaced I-files for HDFS.
- Computation Overviews
  1. Writing map task output to I-file
  2. Sorting and indexing I-file chunks
  3. Determining the number of reducers
  4. Retrieving reduce task input from an I-file

Writing map task output to I-file

- Map output (I-file) is partitioned by key.
- Each mappers append records to designated chunks.
- Chunkservers storing chunks serialize the appends.

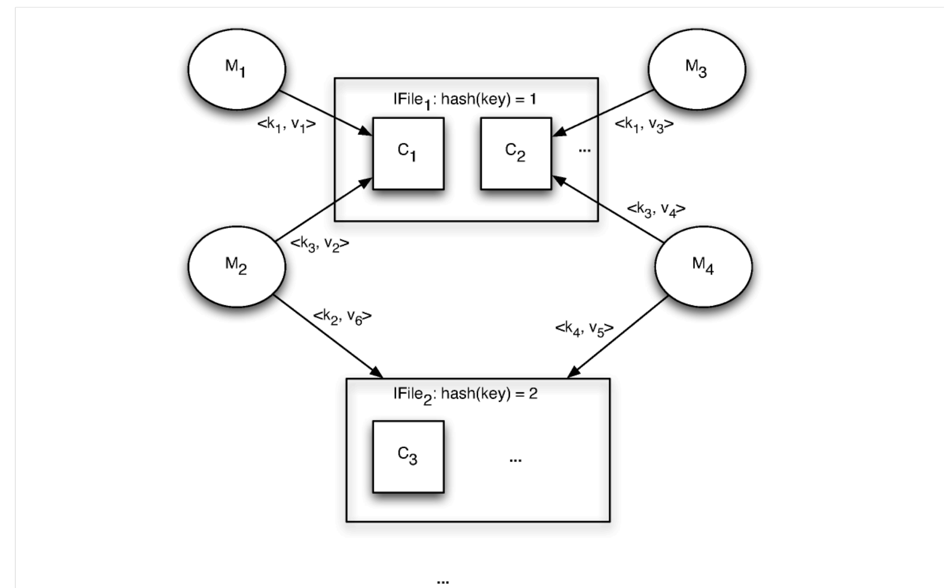


Figure 5: Mappers appending their output, partitioned by key, to $\mathcal{I}$-file chunks. Note that multiple chunks of multiple $\mathcal{I}$-files are appended to concurrently.

Sorting and indexing I-file chunks

- Sorting of map output is decoupled from map task execution.
  - If an I-file chunk becomes stable, it is sorted and augmented with an in-chunk index.

Determining the number of reducers

- It tries to automatically parallelize execution.
  - Calculates the number of reduce tasks from data properties and run-time properties.
- The aim of this function is to divide reduce phase from works and to gain amount of work per task.

Retrieving reduce task input from an I-file

- Two reduce tasks R_1 & R_2 are assigned I-file$_{65}$.
- These tasks use the per-chunk index to retrieve their input from chunks C_{17} & C_{18} in I-file$_{65}$.

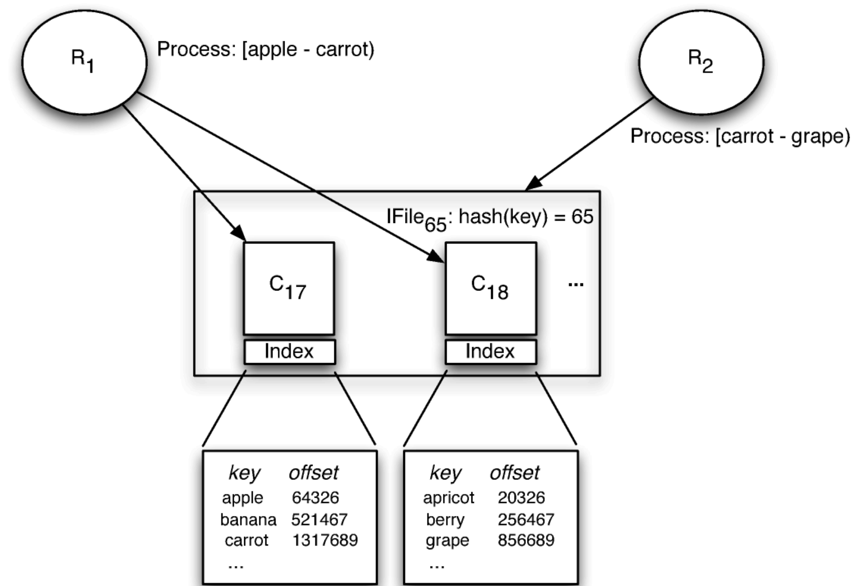


Figure 6: Reducers retrieving their assigned key ranges from the chunks of an $\mathcal{I}$-file. Multiple reducer tasks are assigned non-overlapping key ranges from a single $\mathcal{I}$-file.

Sailfish Implementation

- Appending Map Output to I-files
- Sorting Stable I-file Chunks
- Determining Number of Reducers
- Generating Reduce Task Input From I-files
- Recovering Lost Map Task Output

Dataflow

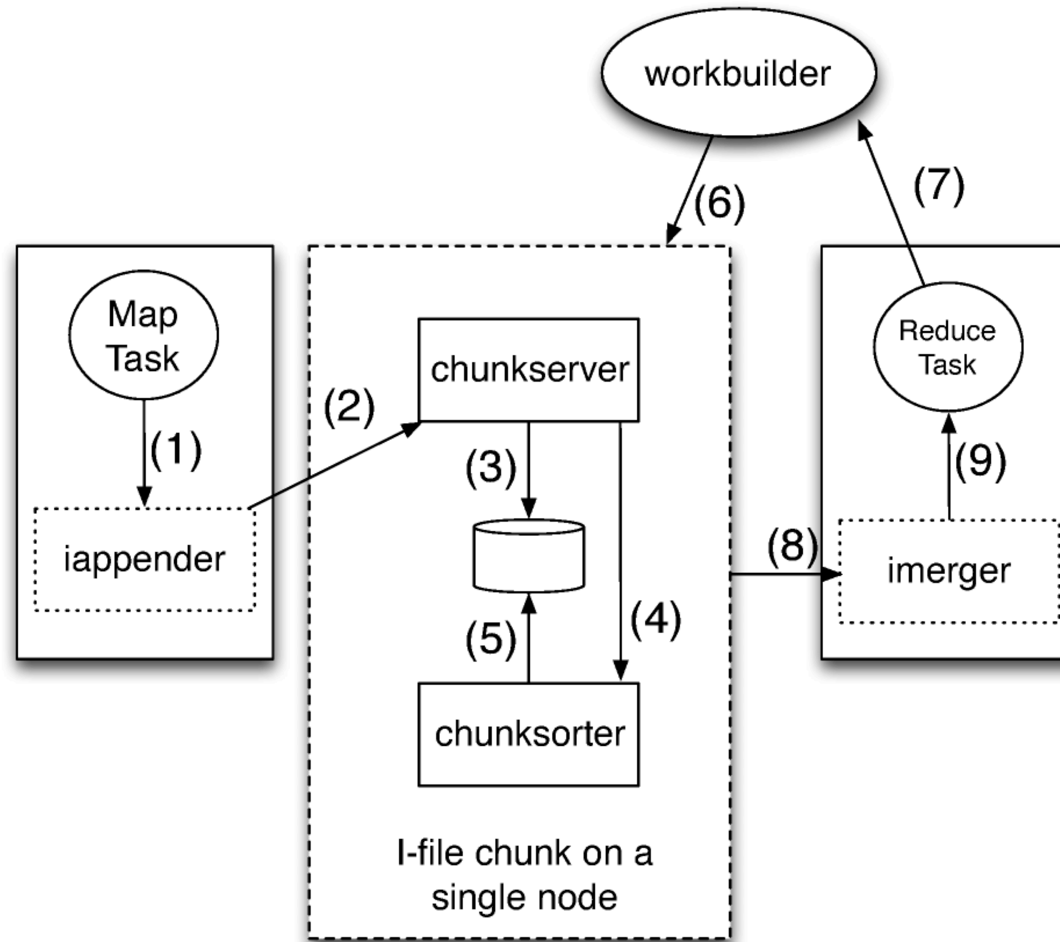
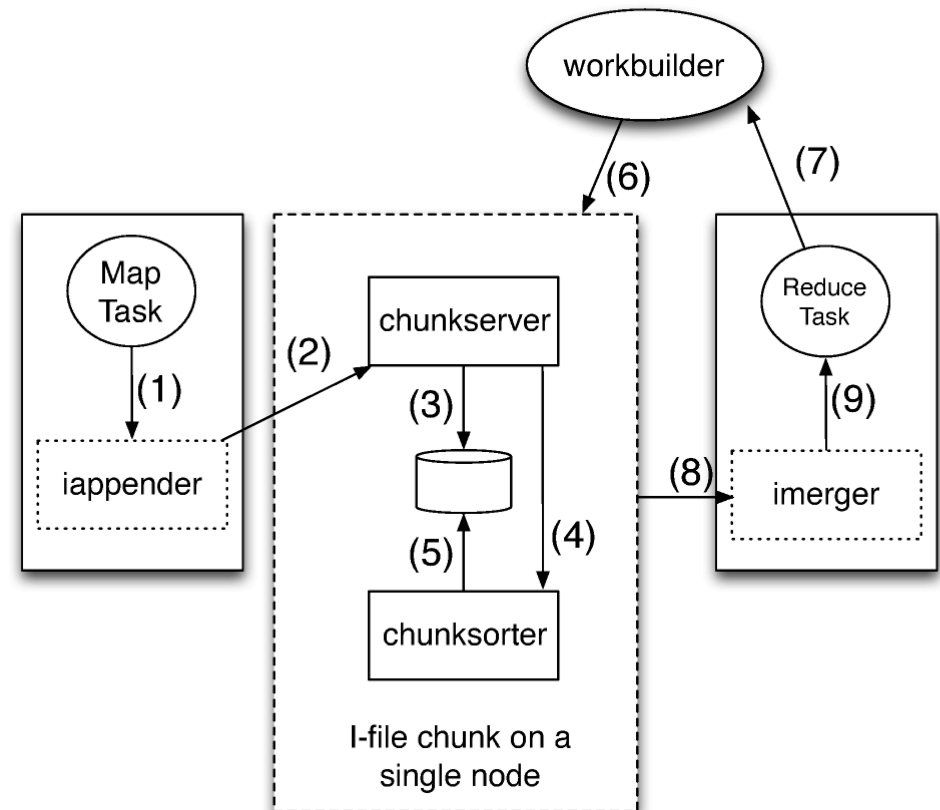


Figure 7: Dataflow in *Sailfish* as it corresponds to a single I-file chunk.
The **iappender** and **imerger** are one per task. There is one **workbuilder** daemon per job.

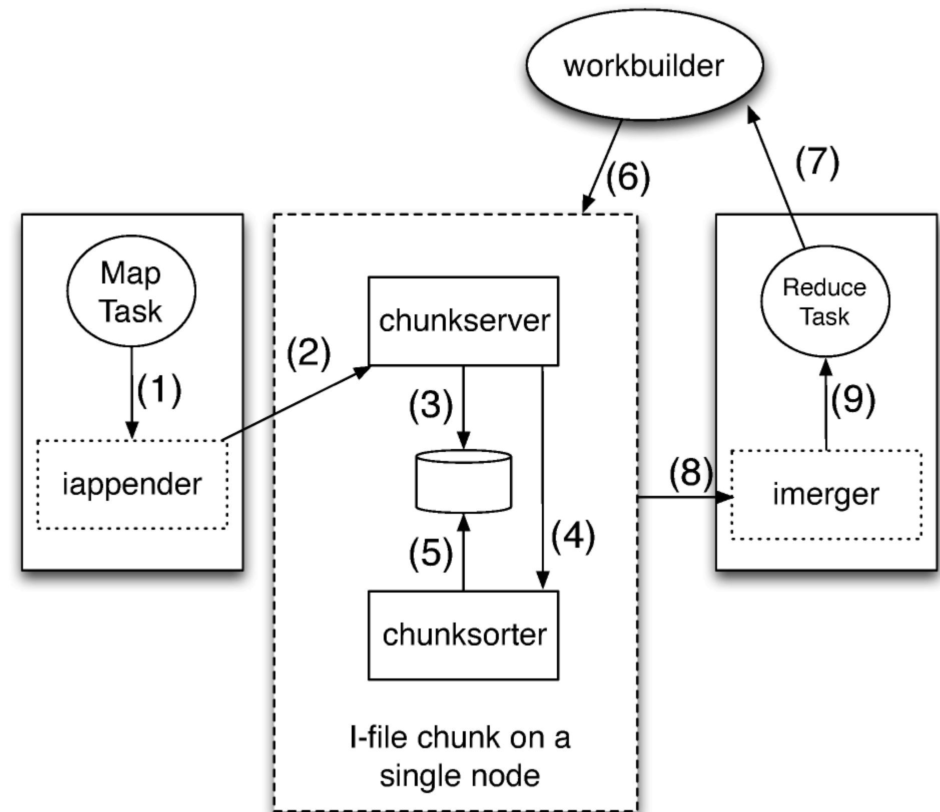
Appending Map Output To I-files

1) Map task generate and send each record to **iappender** (child process).
2) The iappender buffer flushes the record to chunkserver.
3) The chunkserver buffer sends the record to disk.



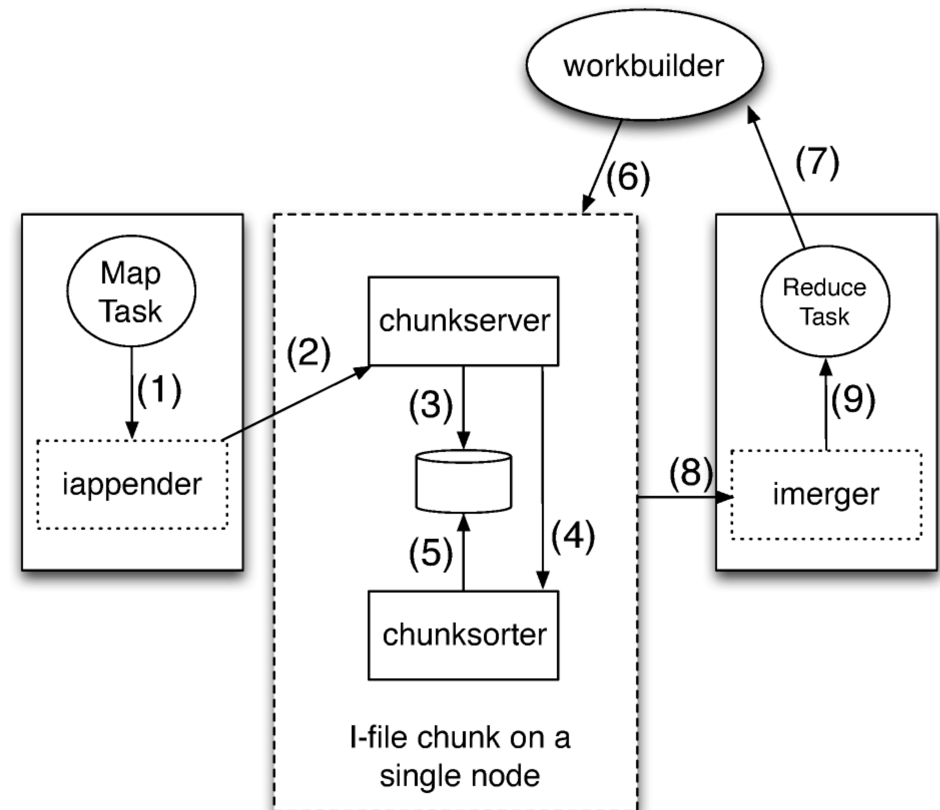
Sorting Stable I-file Chunks

4) When the chunk becomes stable, **chunkserver** will become **chunksorter**.
   – Performs in-memory sorting.
5) When sorting is finished, the chunksorter write sorted records to disk.



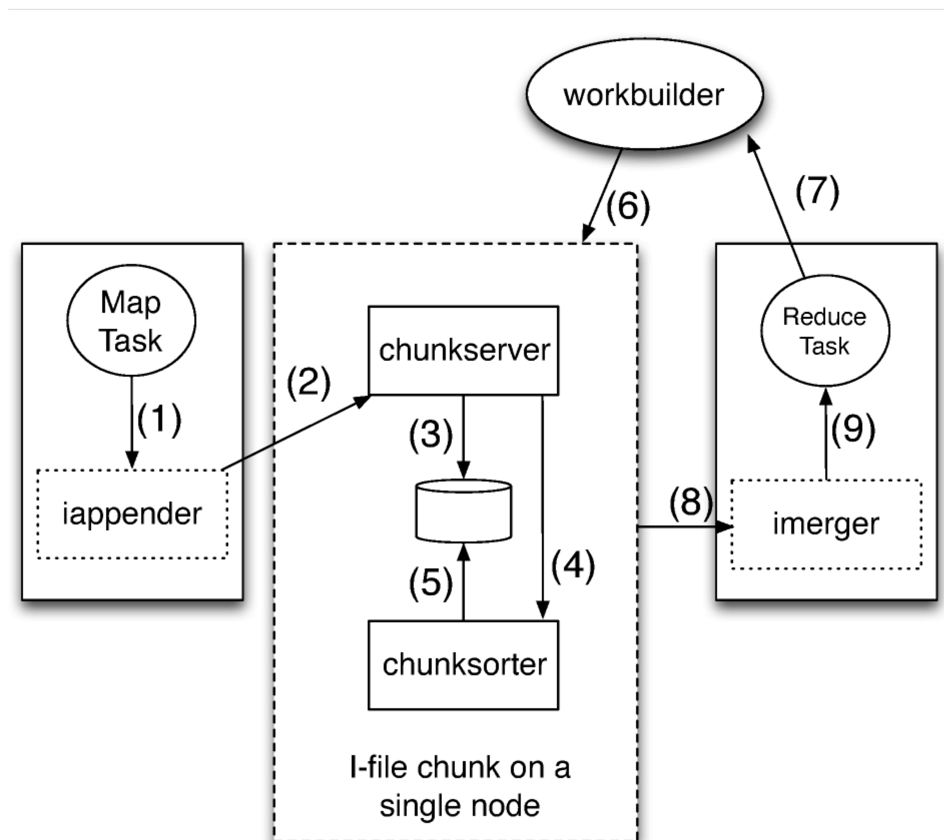
Determining Number of Reducers

6) A **workbuilder** daemon process reads the per-chunk indexes from I-files in order to determine split points.
7) Each reduce task obtains its work assignment from workbuilder.



Generating Reduce Task Input From I-files

8) The reducer startup **imerger** process and it retrieves records from the chunks of the I-file.
9) When the imerger used all of indexes in the I-file, it merges the records and send them to the reduce task.



Recovering Lost Map Task Output

- A chunk of an I-file may be lost and the containing records will be lost.
- To regenerate the lost data, the workbuilder maintains additional bookkeeping information.
  - When a map task executing finished, the iappender notifies the workbuilder about written chunks.
  - If a chunk is lost, workbuilder tells JobTracker to re-run the map tasks to generate the chunk.

Disk Seek Analysis

- Disk seeks occur when map output is committed to disk by the chunkservers.
  - read back, sort, write back
- The number of disk seeks is data dependent.
  - Writing: (I-files) * (chunk filesper I-file)
  - Sorting: 2 (I-files)*(chunk filesper I-file)

  →Lower bound seeks: 3 (I-files) * (chunk filesper I-file

6. Evaluations

- With Synthetic Benchmark
  - For evaluating the effectiveness of I-files in aggregating intermediate data
  - For studying the system effects of the *Sailfish* dataflow path
- With Actual Jobs
  - To evaluate representative mix of real MapReduce jobs with real datasets

Parameter settings

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

Evaluation With Synthetic Benchmark

- Evaluated *Sailfish* for handling 1TB~16TB data
  - Packing intermediate data in chunks
  - Overheads imposed by chunksorter daemon
  - System effects of aggre- gating map output on a rack-wide basis

Changing intermediate volume

- Intermediate data scales linearly even handling maximum of volume(64TB).

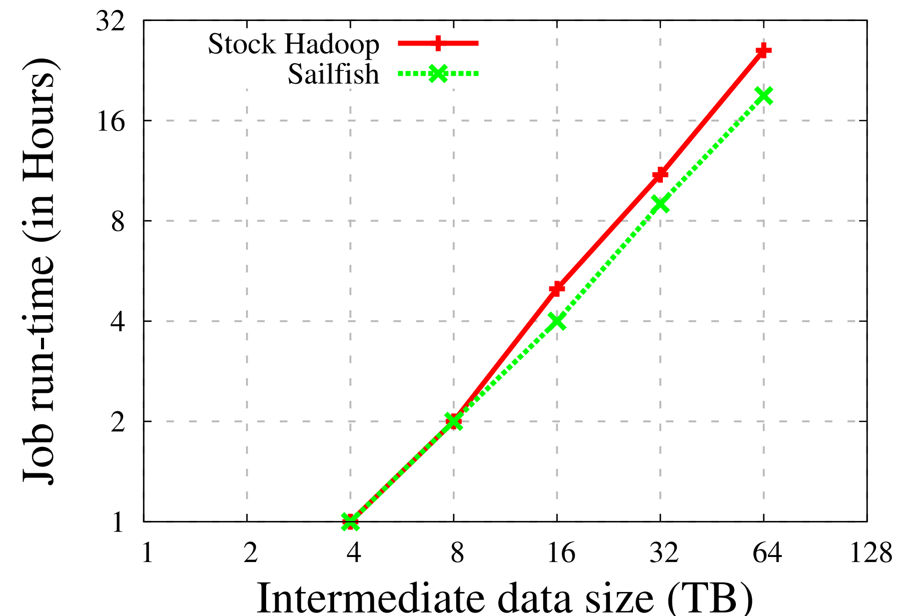


Figure 8: Variation in job run-time with the volume of intermediate data. Note that the axes are log-scale.

Frequency of data retrivals

- Fixing the number of I-files gained high performance.
  - The numbers of chunks and reduce tasks per I-file are increased.

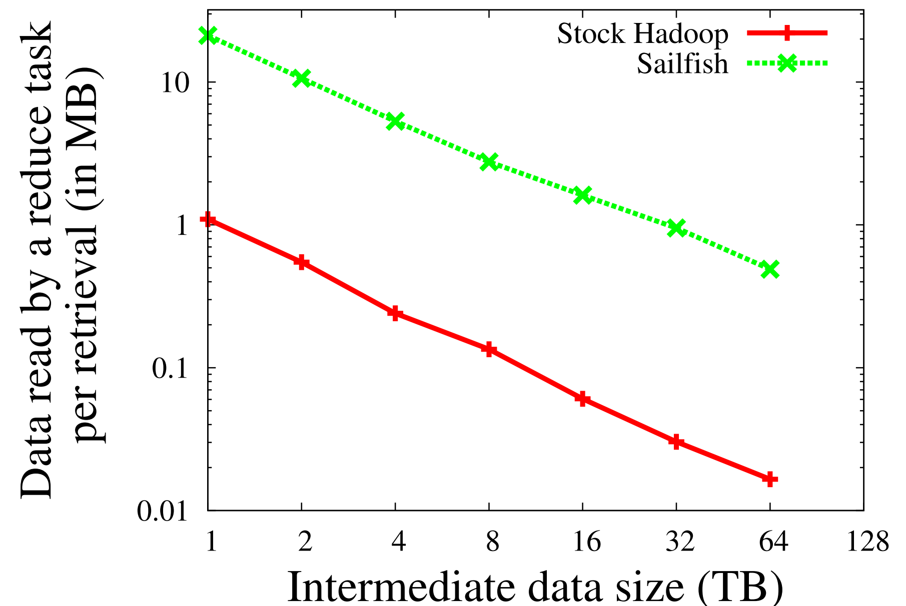


Figure 9: Data read per retrieval by a reduce task with Stock Hadoop and `Sailfish`.

Disk throughput

- *Sailfish* has twice as fast as Stock Hadoop.

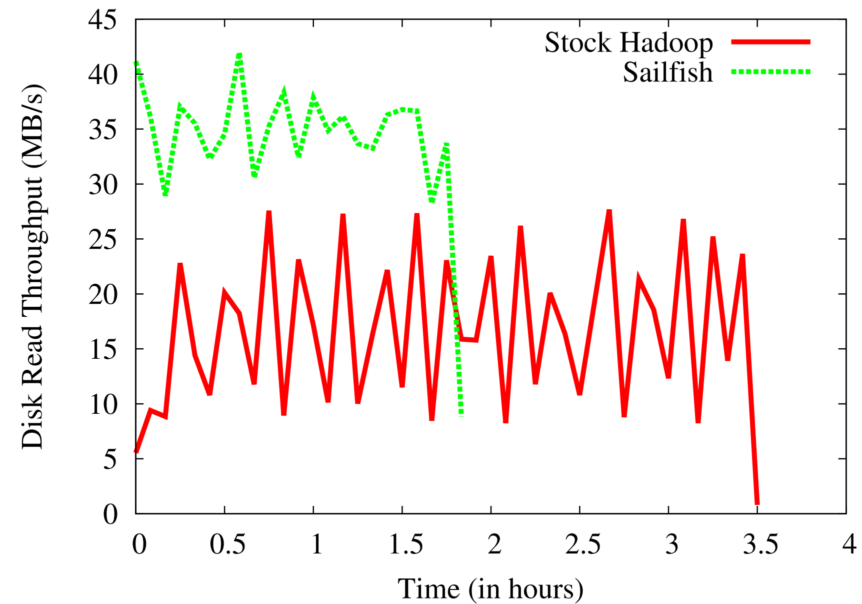


Figure 10: Disk read throughput in the reduce phase with `Sailfish` is higher and hence reduce phase is faster (int. data size = 16TB).

Evaluation With Actual Jobs

- Classify MapReduce jobs with these taxonomy:
  - Skew in map output (e.g. data compression)
  - Skew in reduce input (e.g. data partition)
  - Incremental computation (e.g. join)
  - Big data (e.g. handling huge daily logs)
  - Data explosion (e.g. ad-campaign by geo-location)
  - Data reduction (e.g. statistical narrowing down)

List of experiment jobs

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.

Elapsed time of Map and Reduce

- There are between 20% to 5x speed-ups.

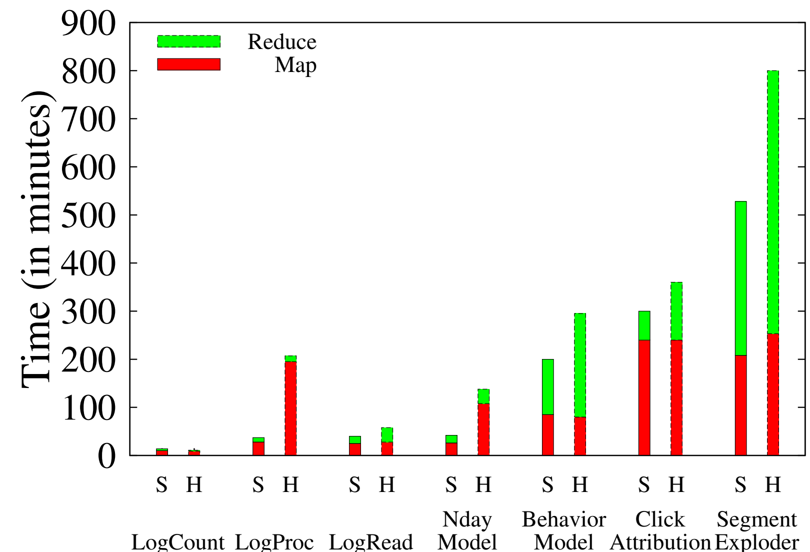


Figure 11: Time spent in the Map and Reduce phases of execution for the various MapReduce jobs. At scale, `Sailfish` (S) outperforms Stock Hadoop (H) between 20% to a factor of 5.

Speed-up aspects

- Using I-files for aggregation
- Decoupling sorting from map task execution
- Making reduce phase dynamic

Using I-files for aggregation

- There is a substantial speedup with *Sailfish* for the remaining jobs.
  - better batching of intermediate data in I-files, similar to what we observed with this benchmark.
- In **LogProc** and **LogRead** jobs, not speedup.
  - The volume of intermediate data is relatively low.

Decoupling sorting from map task execution

- In **LogProc** and **NdayModel** jobs, skew in map output impacted.

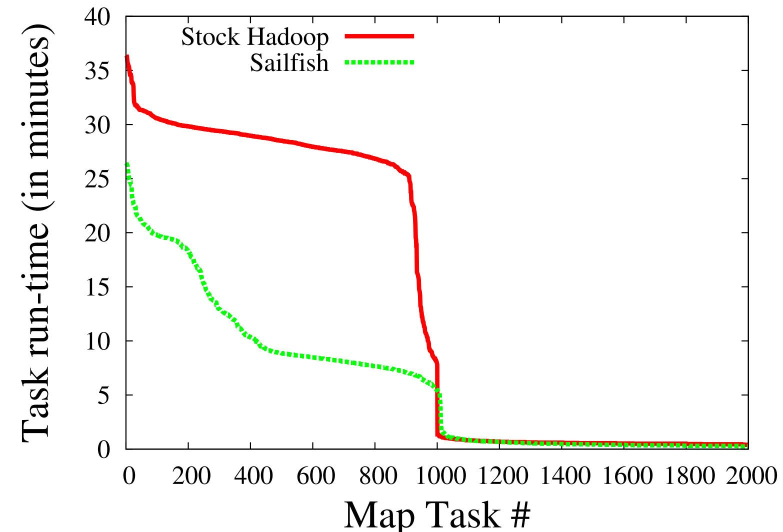
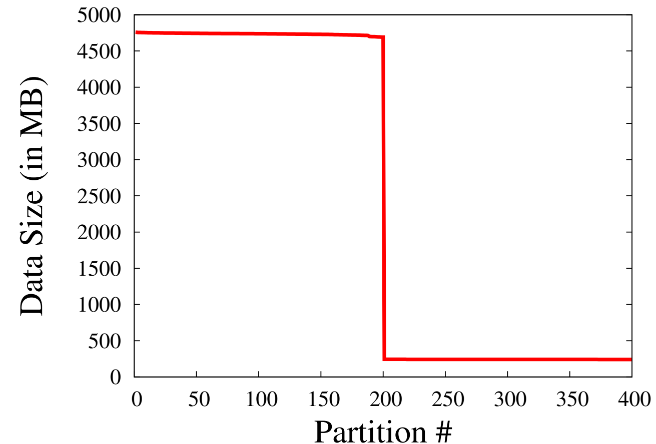
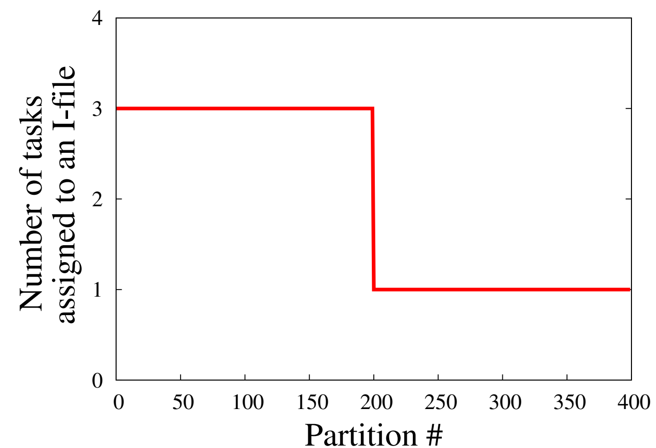


Figure 12: Distribution of map task run time for NdayModel job. Skew in map output sizes affects task completion times for both Stock Hadoop and `Sailfish`, but the impact for Stock Hadoop is much higher.

Making reduce phase dynamic

- Dinamically determining the number of reduce tasks helps in skew handling.
- More reducers were assigned to partitions with large data.





Conclusions from Results

- I-files enable better batching of intermediate data.
  - *Sailfish* provides better scale than Stock Hadoop.

7. Related Work

- TritonSort
  - Using MapReduce implementation “ThemisMR”.
  - It considers a point in the small design space.
- Starfish
  - Profiling has to be run to obtain suitable parameter values.

8. Summary

- Sailfish is an alternate MapReduce framework to aggregate intermediate data.
  - Developed I-files as distributed filesystem.

Future Work

- Add a feedback loop to the reduce phase of Sailfish to re-partition the key-boundary work.
- Evaluate mechanisms for replicating intermediate data.
- Have I-files to provide new opportunities for debugging (e.g. saving valuable programmer time with reducing phase of a job).

My Impressions

Strong Points

- Considers large-scale data and clusters.
  - It will be applicable to system handling larger data.
- *Sailfish* framework has good scalability compared to current Hadoop.

Weak Points

- Used only limited data and situations (Yahoo! data set).
- Not compared with other frameworks than Stock Hadoop.

Thank you for listening