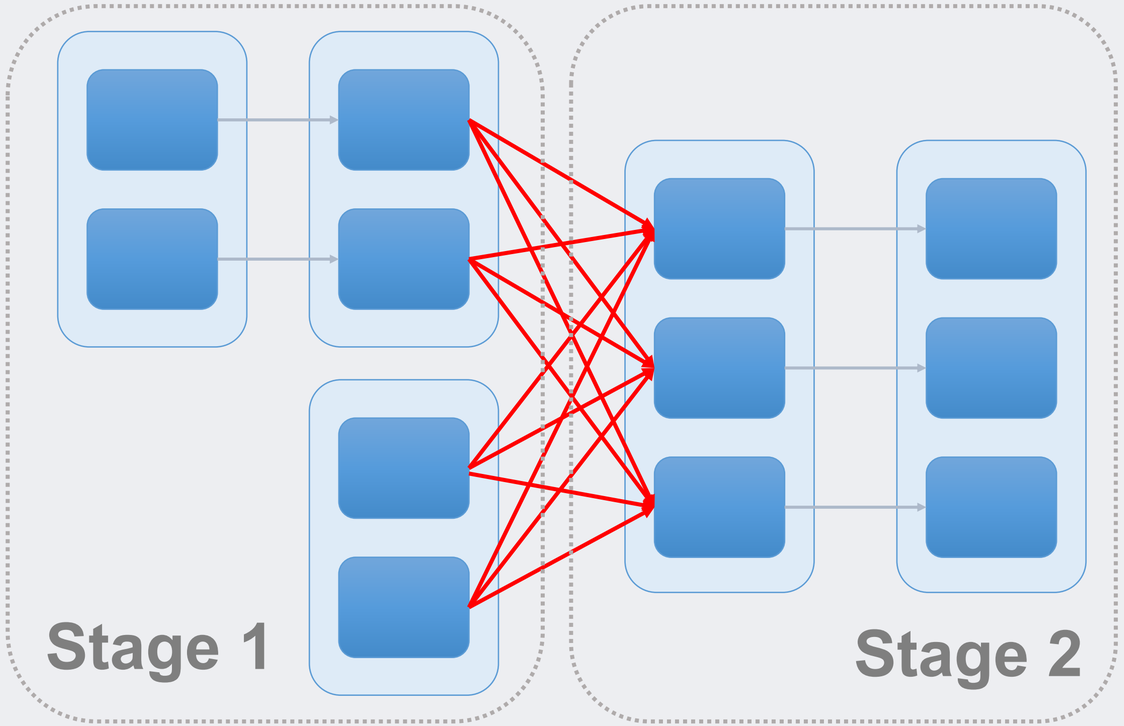
facebook

SOS: Optimizing Shuffle I/O

Brian Cho and Ergin Seyfe, Facebook
Haoyu Zhang, Princeton University

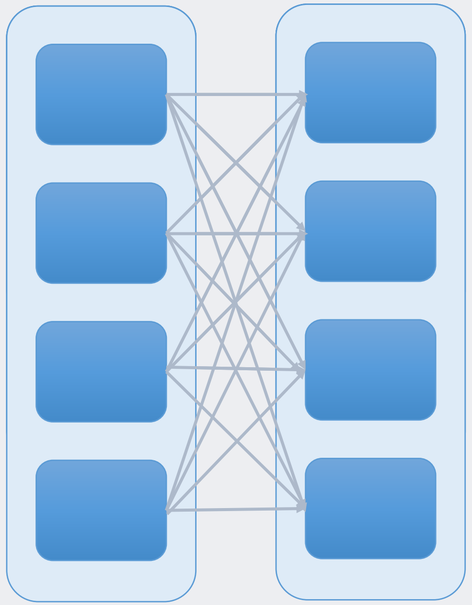
1. Shuffle I/O at large scale

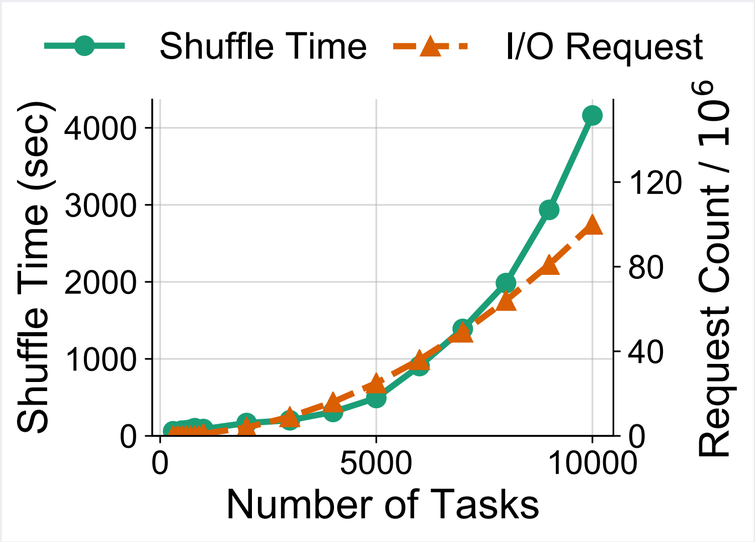
Large-scale shuffle

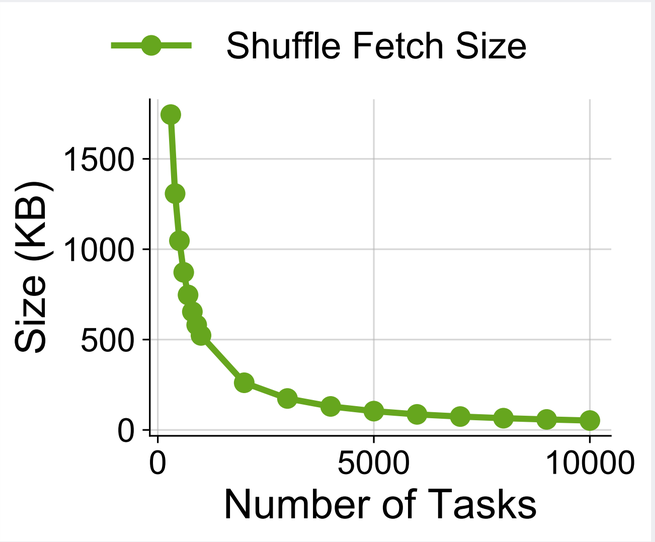


- Shuffle: all-to-all communication between stages
- >10x larger than available memory, strong fault tolerance requirements
  → on-disk shuffle files

Shuffle I/O grows *quadratically* with data

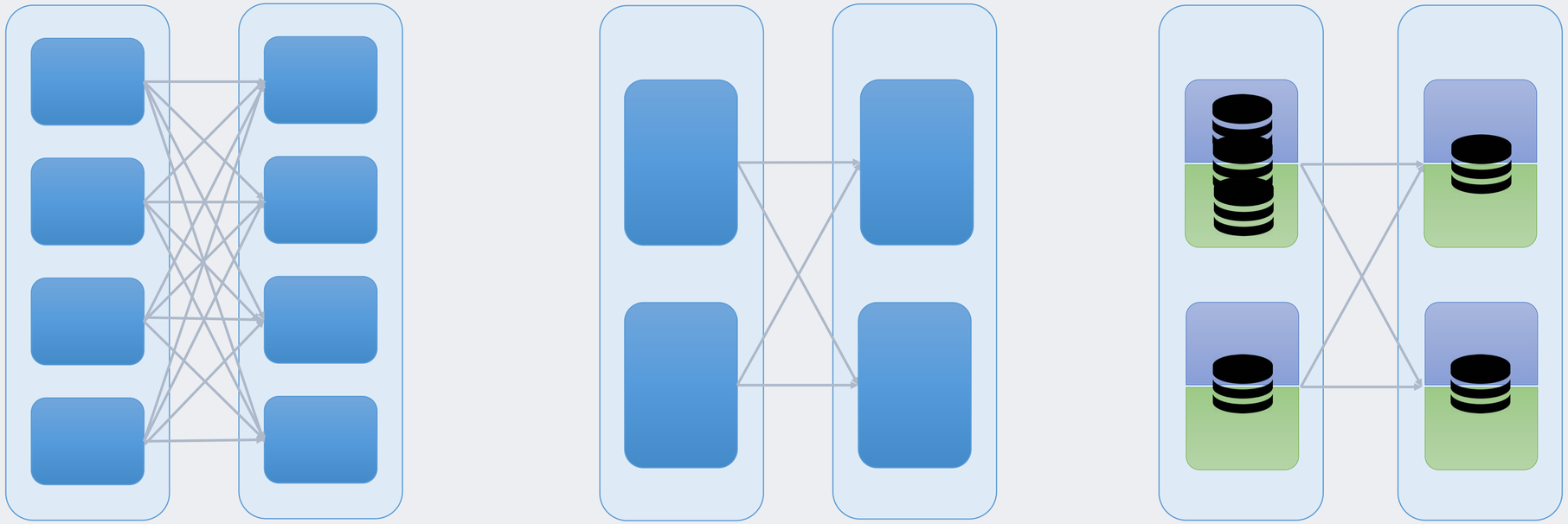






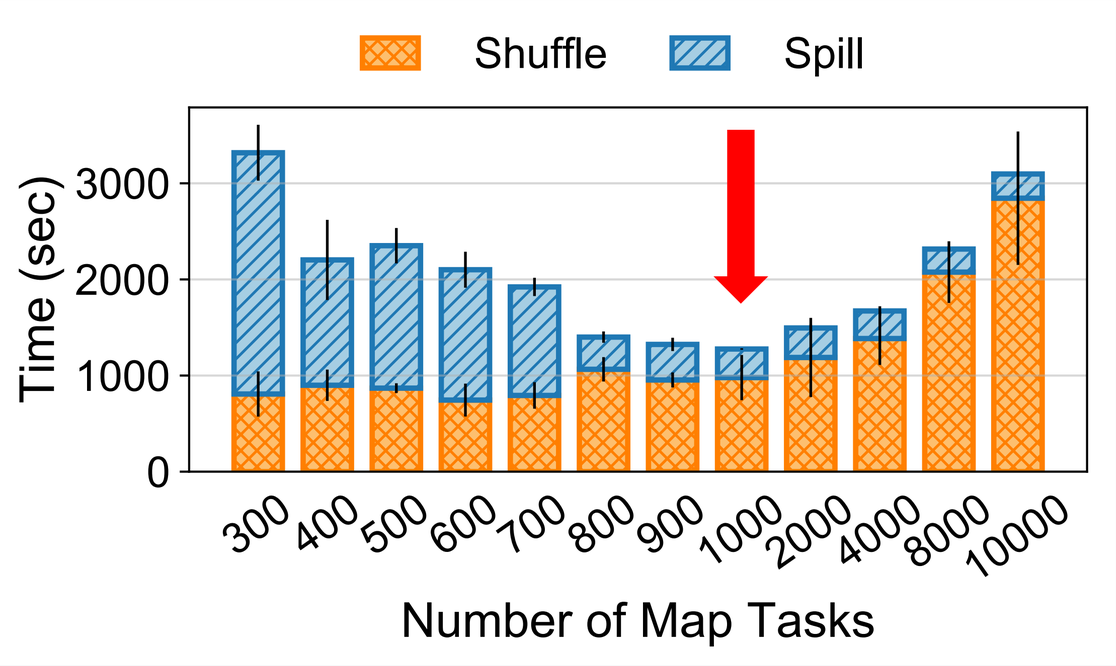
- M * R (number of mappers * number of reducers) shuffle fetches
- Large amount of fragmented I/O requests
  - Adversarial workload for hard drives!

Strawman: tune number of tasks in a job

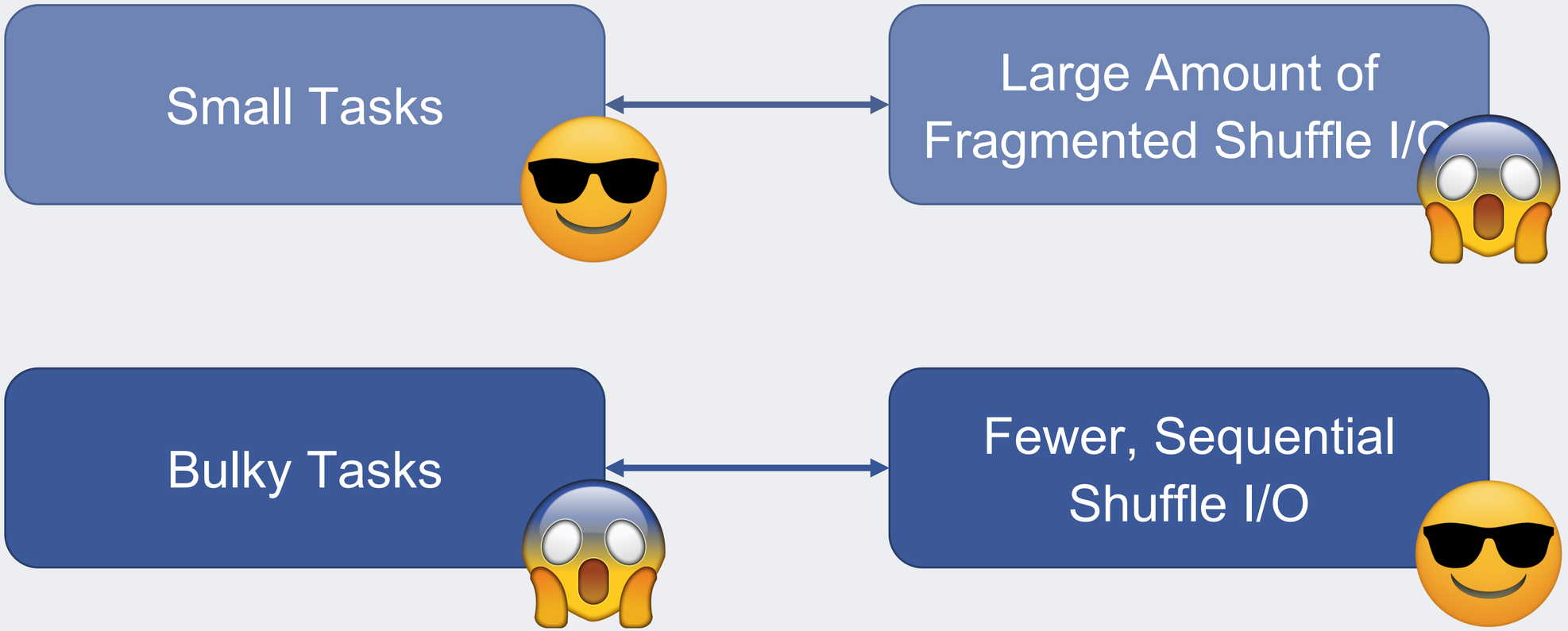


- Tasks spill intermediate data to disk if data splits exceed memory capacity
- Larger task execution reduces shuffle I/O, but increases spill I/O

Strawman: tune number of tasks in a job



- Need to retune when input data volume changes for each individual job
- Small tasks run into the quadratic I/O problem
- Bulky tasks can be detrimental [Dolly NSDI 13] [SparkPerf NSDI 15] [Monotask SOSP 17]
  - straggler problems, imbalanced workload, garbage collection overhead


Small Tasks
Large Amount of Fragmented Shuffle I/O
Bulky Tasks
Fewer, Sequential Shuffle I/O

2. SOS: optimizing shuffle I/O

SOS: optimizing shuffle I/O

a.k.a. Riffle, presented at Eurosys 2018

Riffle: Optimized Shuffle Service for Large-Scale Data Analytics

Haoyu Zhang*, Brian Cho†, Ergin Seyfe†, Avery Ching†, Michael J. Freedman*

**Princeton University* †*Facebook, Inc.*

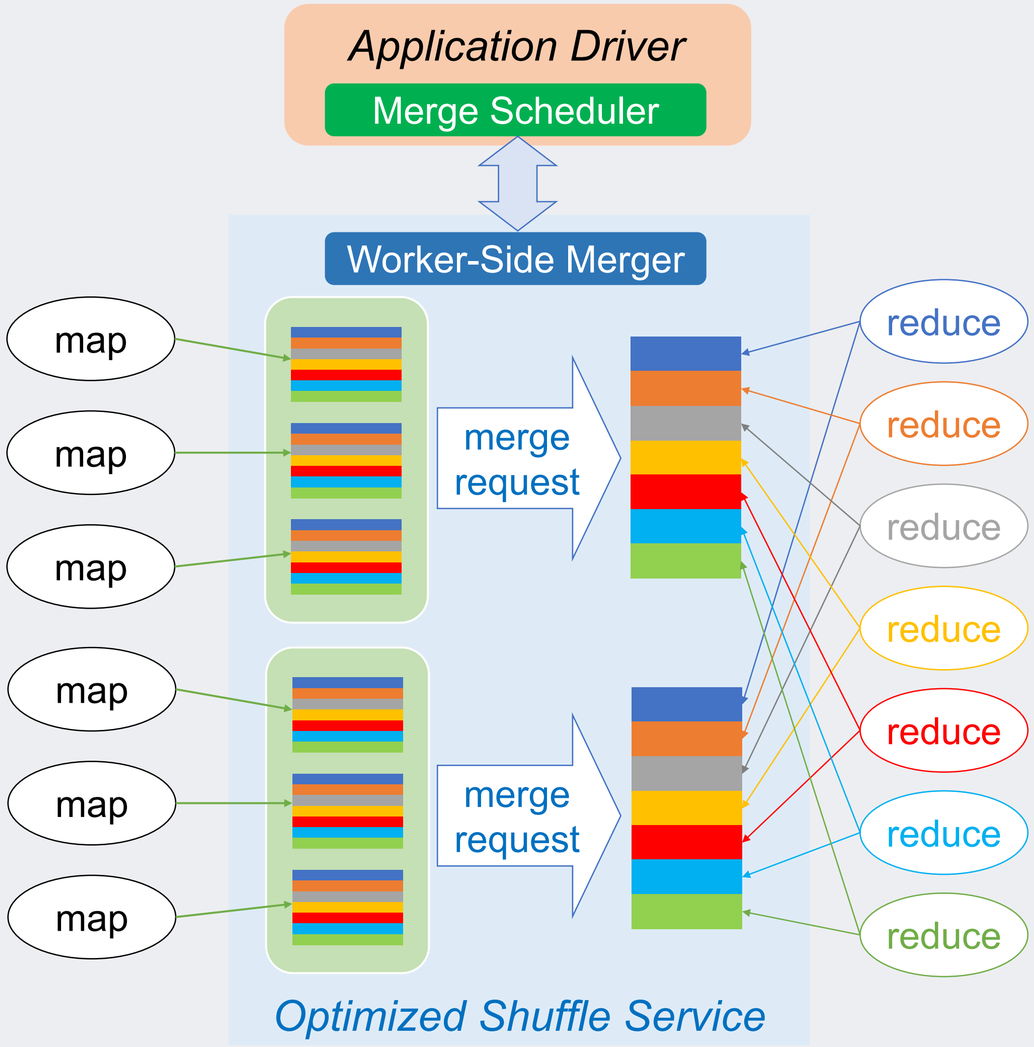
Deployed at Facebook scale

SOS: optimizing shuffle I/O

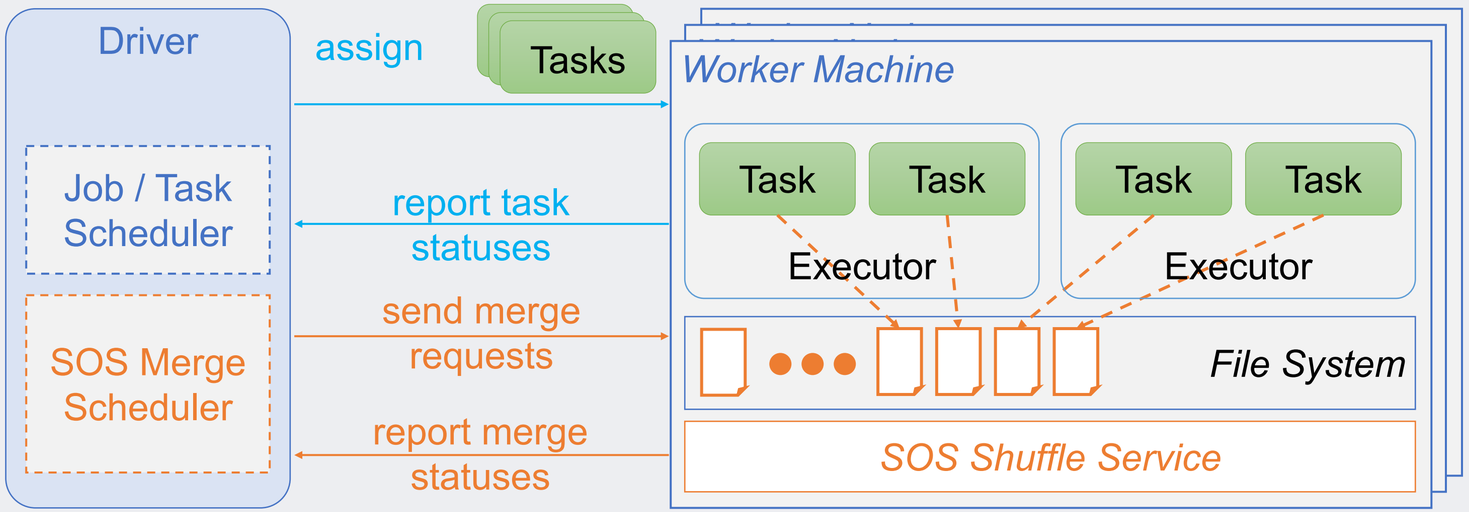
- Merge map task outputs into larger shuffle files

1. Combines small shuffle files into larger ones
2. Keeps partitioned file layout

- Reducers fetch fewer, large blocks instead of many, small blocks
- Number of requests:
  (M * R) **/ (merge factor)**

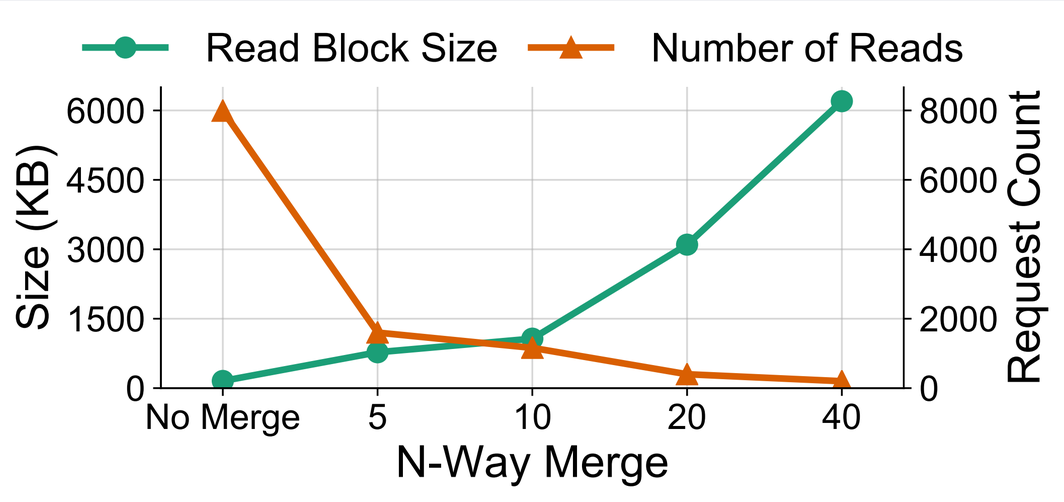


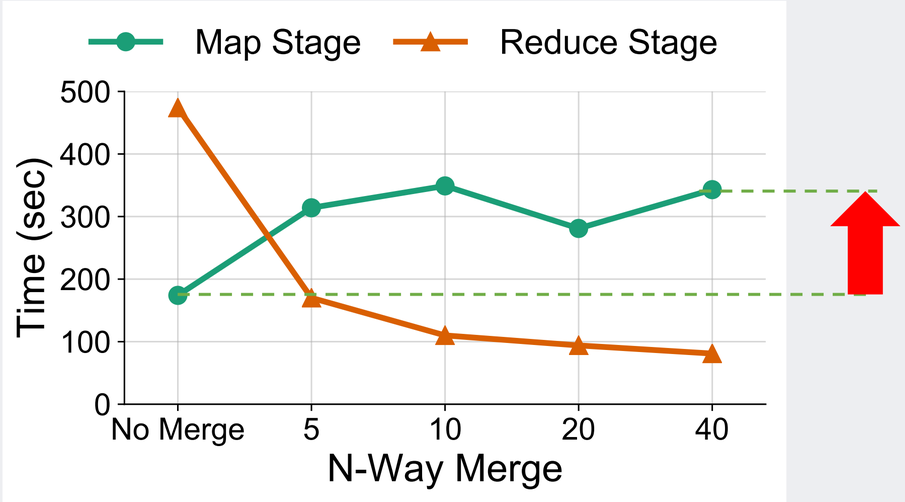
SOS: optimizing shuffle I/O



- SOS shuffle service: a long running instance on each physical node
- SOS scheduler: keeps track of shuffle files and issues merge requests

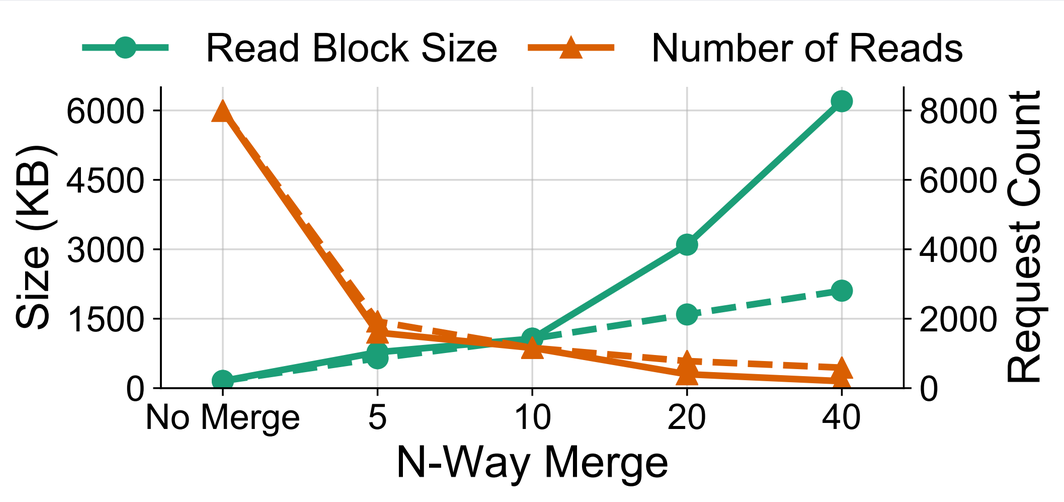
Results on synthetic workload (unoptimized)

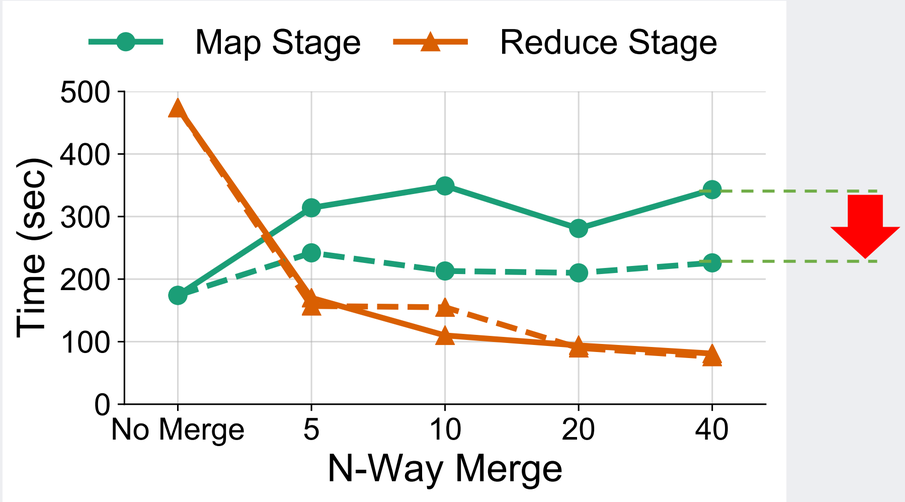




- SOS reduces number of fetch requests by **10x**
- Reduce stage **-393s**, map stage +169s → job completes **35% faster**

Best-effort merge: mixing merged and unmerged files





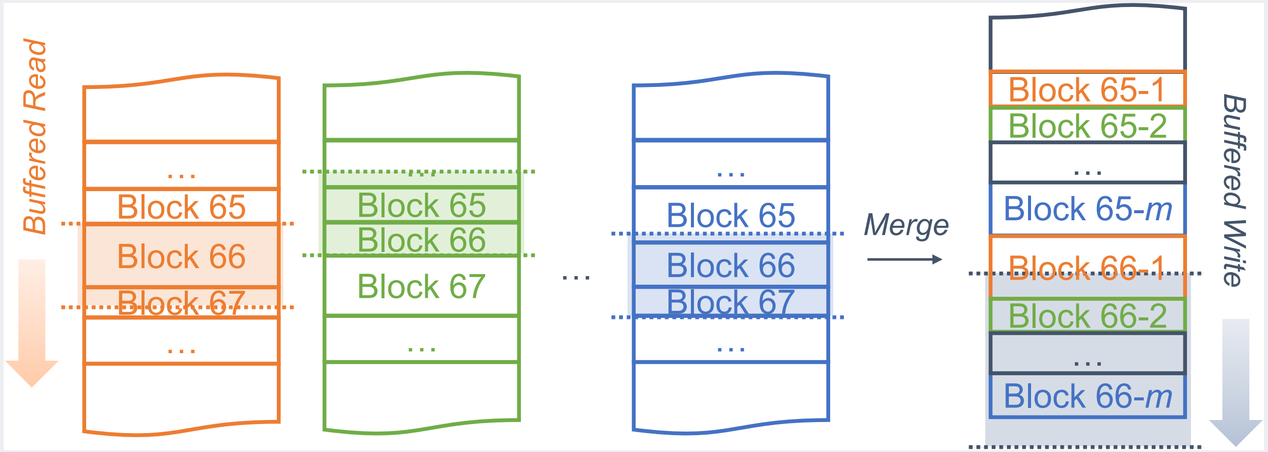
-------- -------- Best-effort merge (95%)

- Reduce stage **-393s**, map stage +52s → job completes **53% faster**
  - SOS finishes job with only ~50% of cluster resources!

Additional details

- Merge operation fault-tolerance
  - Handled by falling back to the unmerged files

- Efficient memory management
  - Merger read/write large buffers for performance and IO efficiency



3. Deployment and observed gains

Deployment

- Started staged rollout late last year
- Completed in April, running stably for over a month

SOS + zstd

- Rollout includes zstd compression with SOS
- Combined they produce a net gain in IO and Compute efficiency

SOS + zstd

- Rollout includes zstd compression with SOS
- Combined they produce a net gain in IO and Compute efficiency

	SOS	zstd	Net
Spill I/O			
Shuffle I/O			

SOS + zstd

- Rollout includes zstd compression with SOS
- Combined they produce a net gain in IO and Compute efficiency

	SOS	zstd	Net
Spill I/O	**Regression**	**Gain**	**Small Gain**
Shuffle I/O			

SOS + zstd

- Rollout includes zstd compression with SOS
- Combined they produce a net gain in IO and Compute efficiency

	SOS	zstd	Net
Spill I/O	Regression	Gain	Small Gain
Shuffle I/O	Gain	Small Gain	Gain

SOS + zstd

- Rollout includes zstd compression with SOS
- Combined they produce a net gain in IO and Compute efficiency

	SOS	zstd	Net
Spill I/O	Regression	Gain	Small Gain
Shuffle I/O	Gain	Small Gain	Gain

	SOS	zstd	Net
CPU time			
Reserved CPU time			

SOS + zstd

- Rollout includes zstd compression with SOS
- Combined they produce a net gain in IO and Compute efficiency

	SOS	zstd	Net
Spill I/O	Regression	Gain	Small Gain
Shuffle I/O	Gain	Small Gain	Gain

	SOS	zstd	Net
CPU time	No change	Small Regression	Small Regression
Reserved CPU time			

SOS + zstd

- Rollout includes zstd compression with SOS
- Combined they produce a net gain in IO and Compute efficiency

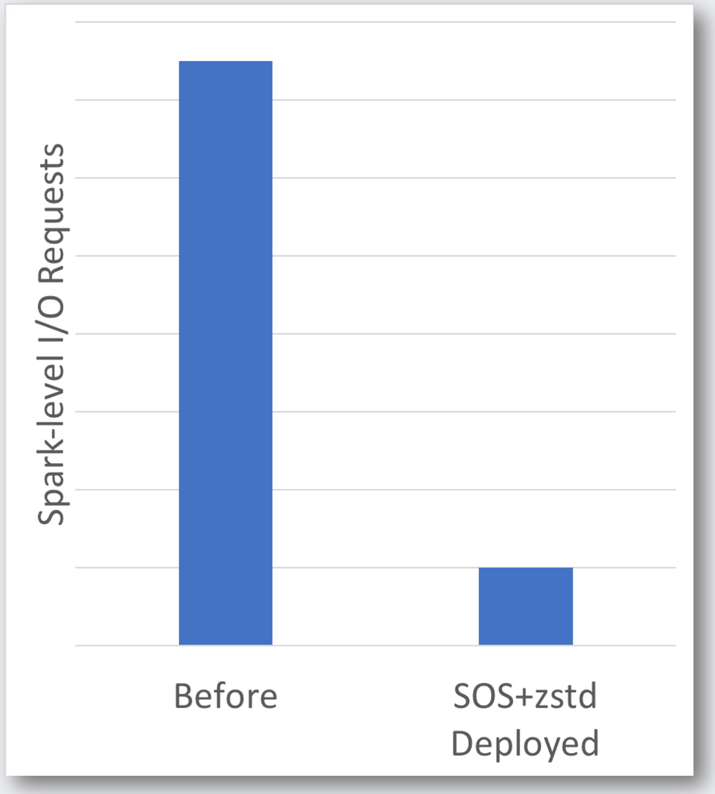
	SOS	zstd	Net
Spill I/O	Regression	Gain	Small Gain
Shuffle I/O	Gain	Small Gain	Gain

	SOS	zstd	Net
CPU time	No change	Small Regression	Small Regression
Reserved CPU time	Gain	No change	Gain

IO Gains: Request-level

Spark-level I/O requests: number of application-level R/W requests made

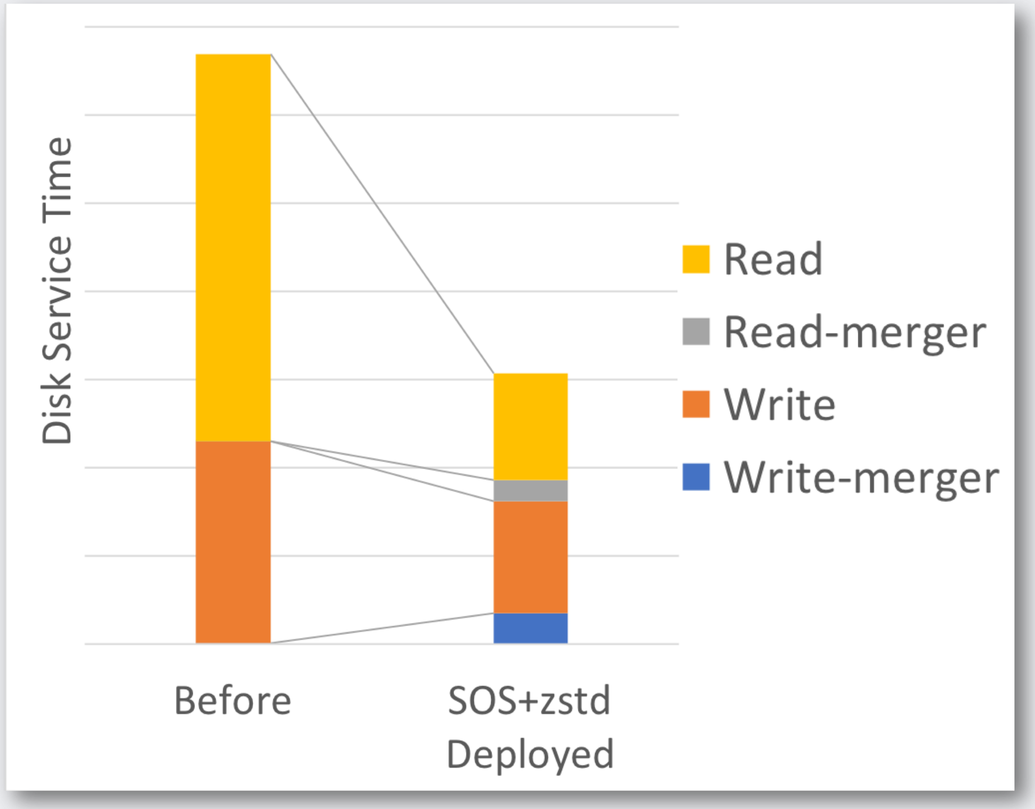
- **7.5x less**



IO Gains: Disk-level

Disk service time: time spent on disks in the storage system

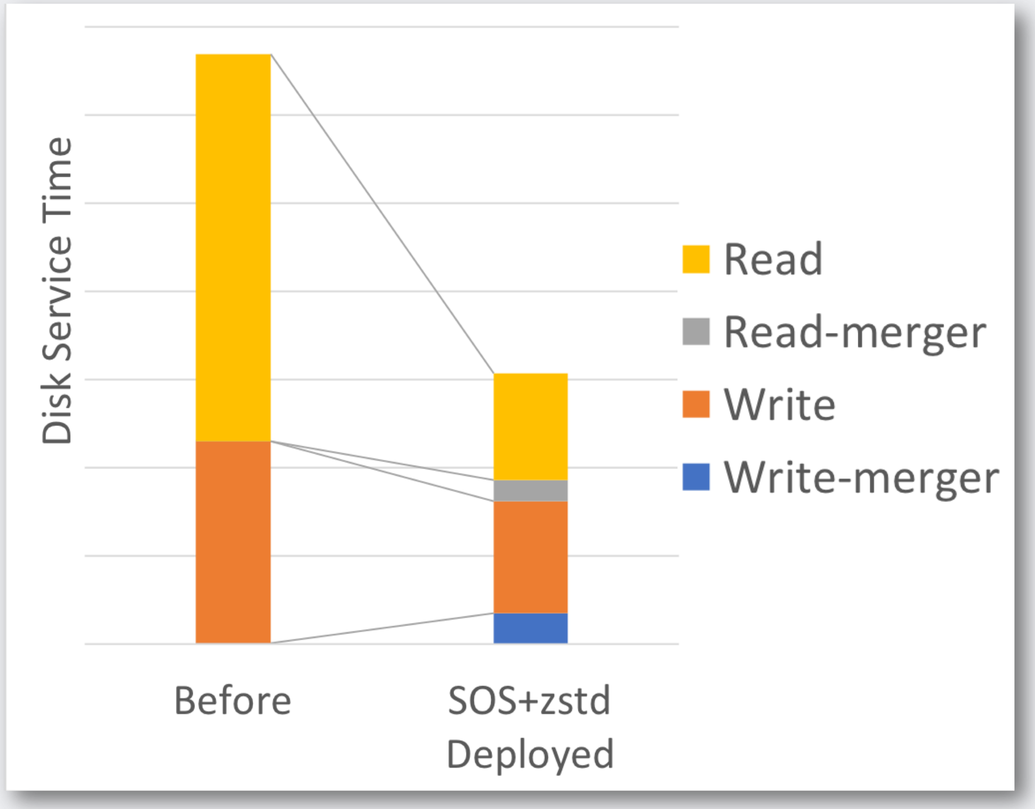
- **2x more efficient**



IO Gains: Disk-level

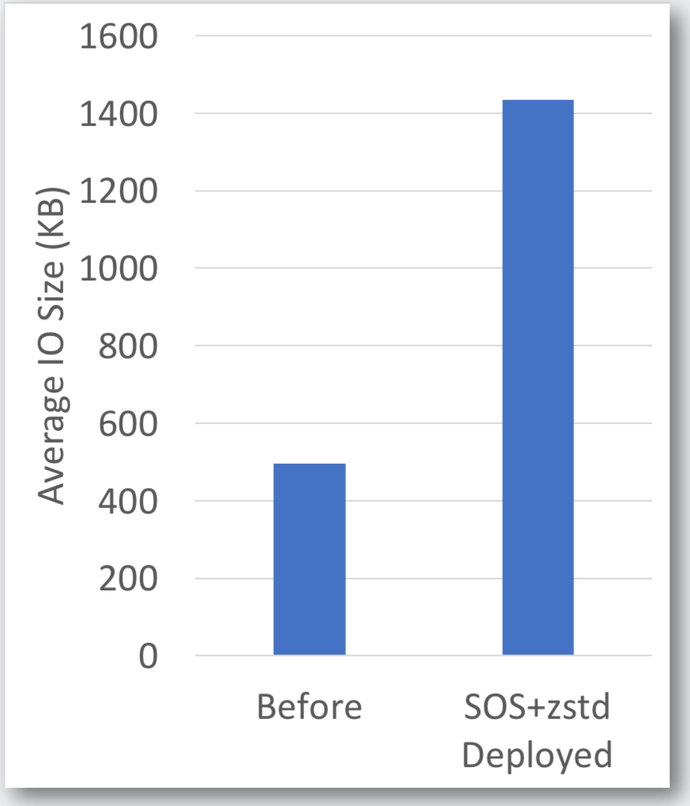
Disk service time: time spent on disks in the storage system

- **2x more efficient**



Average IO Size: average size of IO request at the disks

- **2.5x increase**



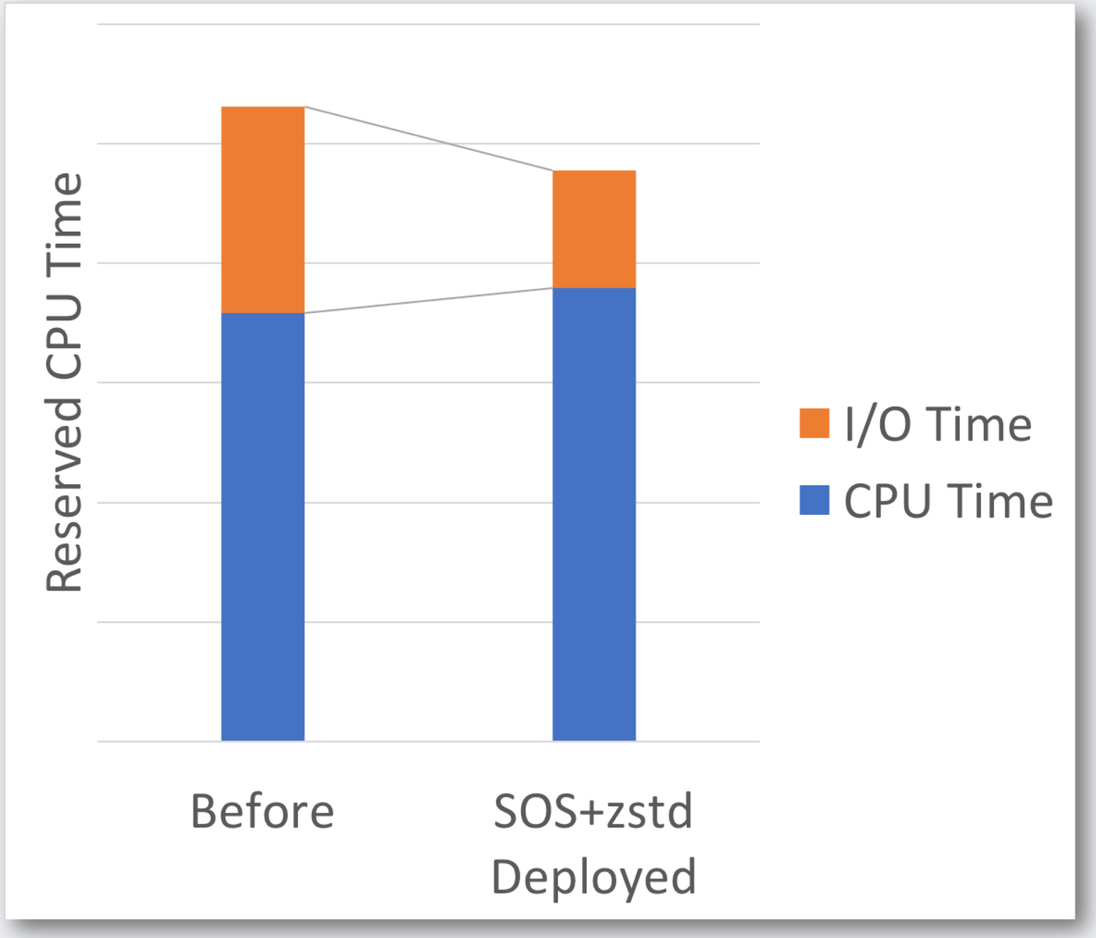
Compute Gains

Reserved CPU time: resources allocated for Spark executors

Total **10% Gain**

- CPU time: time spent using CPU
  → **5% Regression**
- I/O time: time spent waiting (not using CPU)
  → **75% Gain**

Currently working on increasing these gains



4. Summary

Summary

1) Shuffle at large scale induces large fragmented shuffle I/Os
2) SOS provides a solution to optimize these I/Os
3) SOS deployed and running stably at Facebook scale
4) Observed gains of 2x more efficient I/O which translates to 10% more efficient compute
5) Plan to contribute back to Apache Spark

Questions?

facebook