MapReduce Online

Tyson Condie
UC Berkeley

Joint work with
Neil Conway, Peter Alvaro, and Joseph M. Hellerstein (UC Berkeley)
Khaled Elmeleegy and Russell Sears (Yahoo! Research)

MapReduce Programming Model

- Think data-centric
  - Apply a two step transformation to data sets
- **Map step:** *Map*(k1, v1) -> list(k2, v2)
  - Apply map function to input records
  - Assign output records to groups
- **Reduce step:** *Reduce*(k2, list(v2)) -> list(v3)
  - Consolidate groups from the map step
  - Apply reduce function to each group

MapReduce System Model

- Shared-nothing architecture
  - Tuned for massive data parallelism
  - Many maps operate on portions of the input
  - Many reduces, each assigned specific groups
- ⊙ Batch-oriented computations over massive data
  - Runtimes range in minutes to hours
  - Execute on tens to thousands of machines
  - Failures common (fault tolerance crucial)
- Fault tolerance via operator restart since …
  - Operators complete before producing any output
  - Atomic data exchange between operators

Life Beyond Batch

- MapReduce often used for analytics on *streams* of data that arrive *continuously*
  - Click streams, network traffic, web crawl data, …

- **Batch approach**: buffer, load, process
  - High latency
  - Hard to scale for real-time analysis
- **Online approach**: run MR jobs *continuously*
  - Analyze data as it arrives

Online Query Processing

- Two domains of interest (at massive scale):
  1. **Online aggregation**
     - Interactive data analysis (watch answer evolve)
  2. **Stream processing**
     - Continuous (real-time) analysis on data streams

- Blocking operators are a poor fit
  - Final answers only
  - No infinite streams
- Operators need to pipeline
  - BUT we must retain fault tolerance

A Brave New MapReduce World

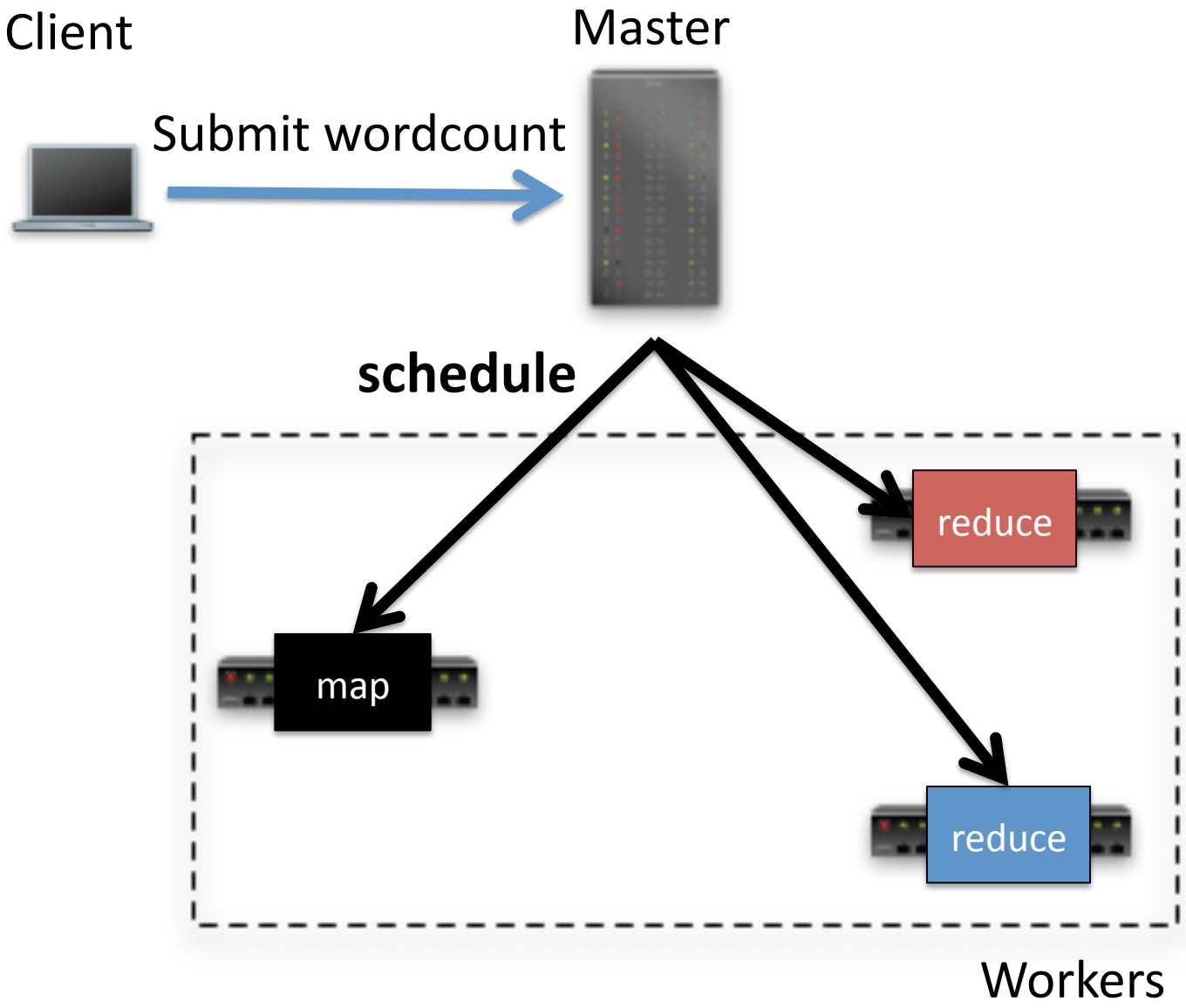
- Pipelined MapReduce
  - Maps can operate on infinite data **(Stream processing)**
  - Reduces can export early answers **(Online aggregation)**
- Hadoop Online Prototype (HOP)
  - Preserves Hadoop interfaces and APIs
  - Pipelining fault tolerance model

Outline

1. Hadoop Background
2. Hadoop Online Prototype (HOP)
3. Performance (blocking vs. pipelining)
4. Future Work

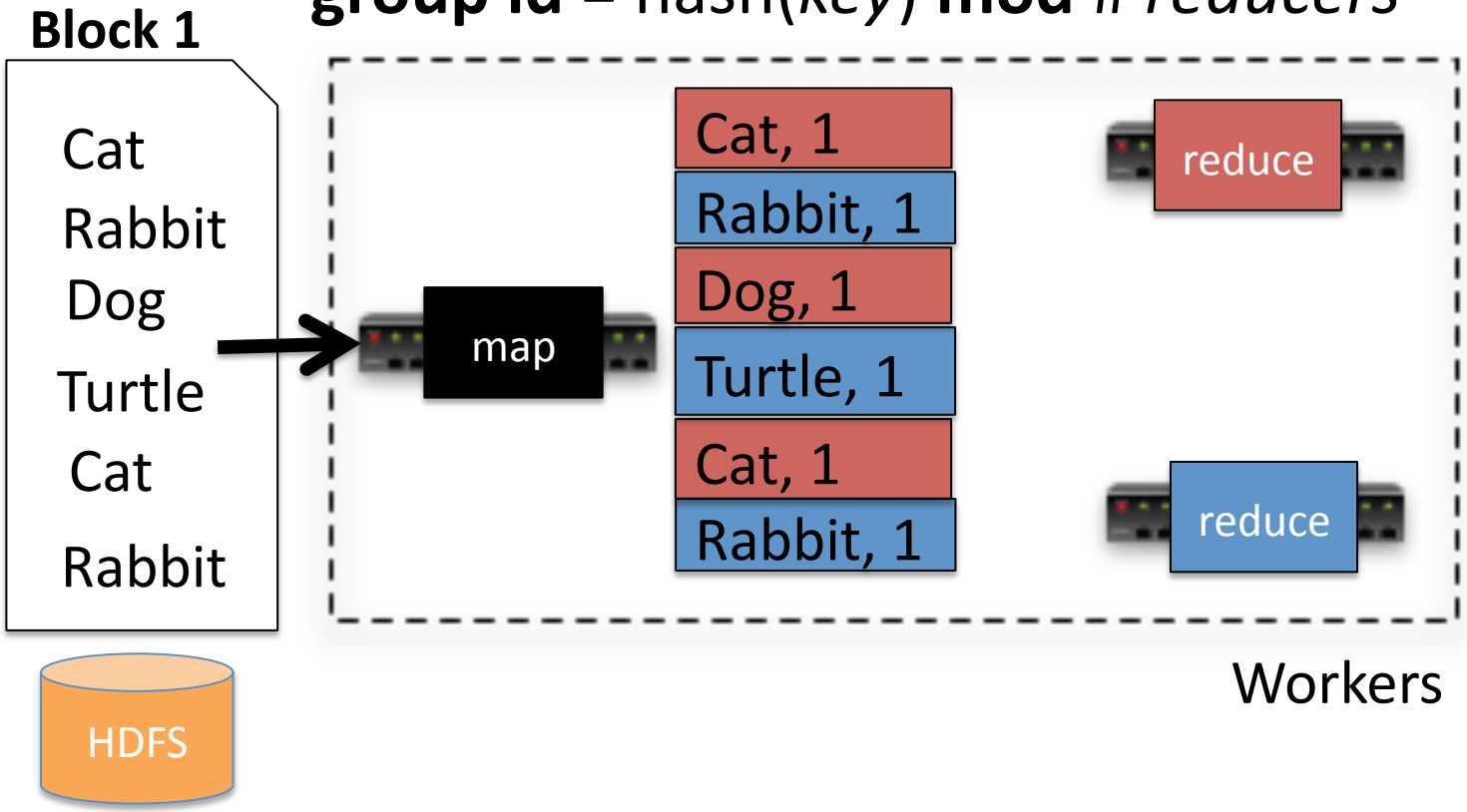
Wordcount Job

- Map step
  - Parse input into a series of words
  - For each *word*, output <*word*, 1>
- Reduce step
  - For each word, list of counts
  - Sum counts and output <*word*, *sum*>
- Combine step (**optional**)
  - Preaggregate map output
  - Same as the reduce step in wordcount


Client
Master
Submit wordcount
schedule
reduce
map
reduce
Workers

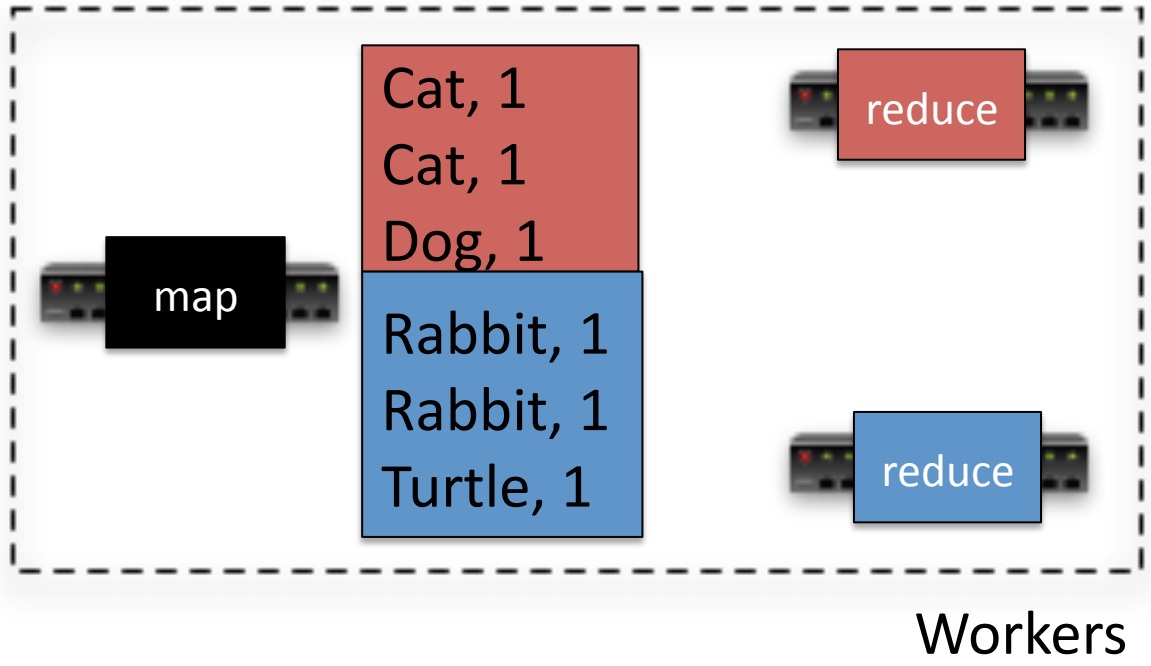
Map step

- Apply map function to the input block
- Assign a **group id (color)** to output records
- **group id** = hash(*key*) **mod** # *reducers*



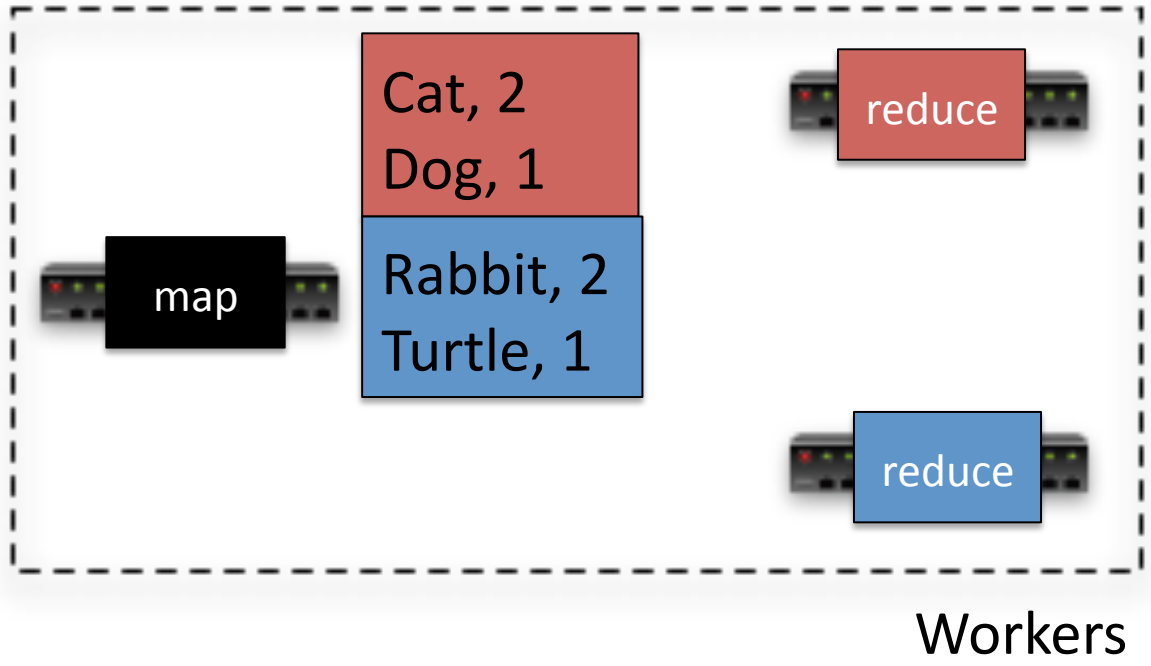
Group step (optional)

- Sort map output by **group id** and **key**



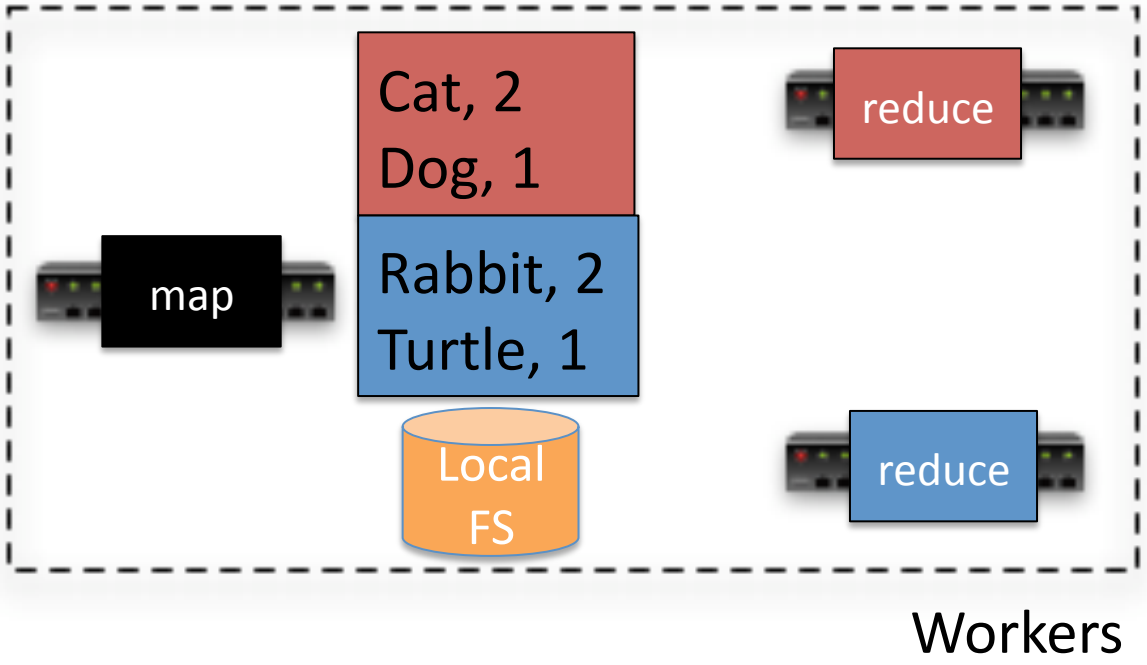
Combine step (optional)

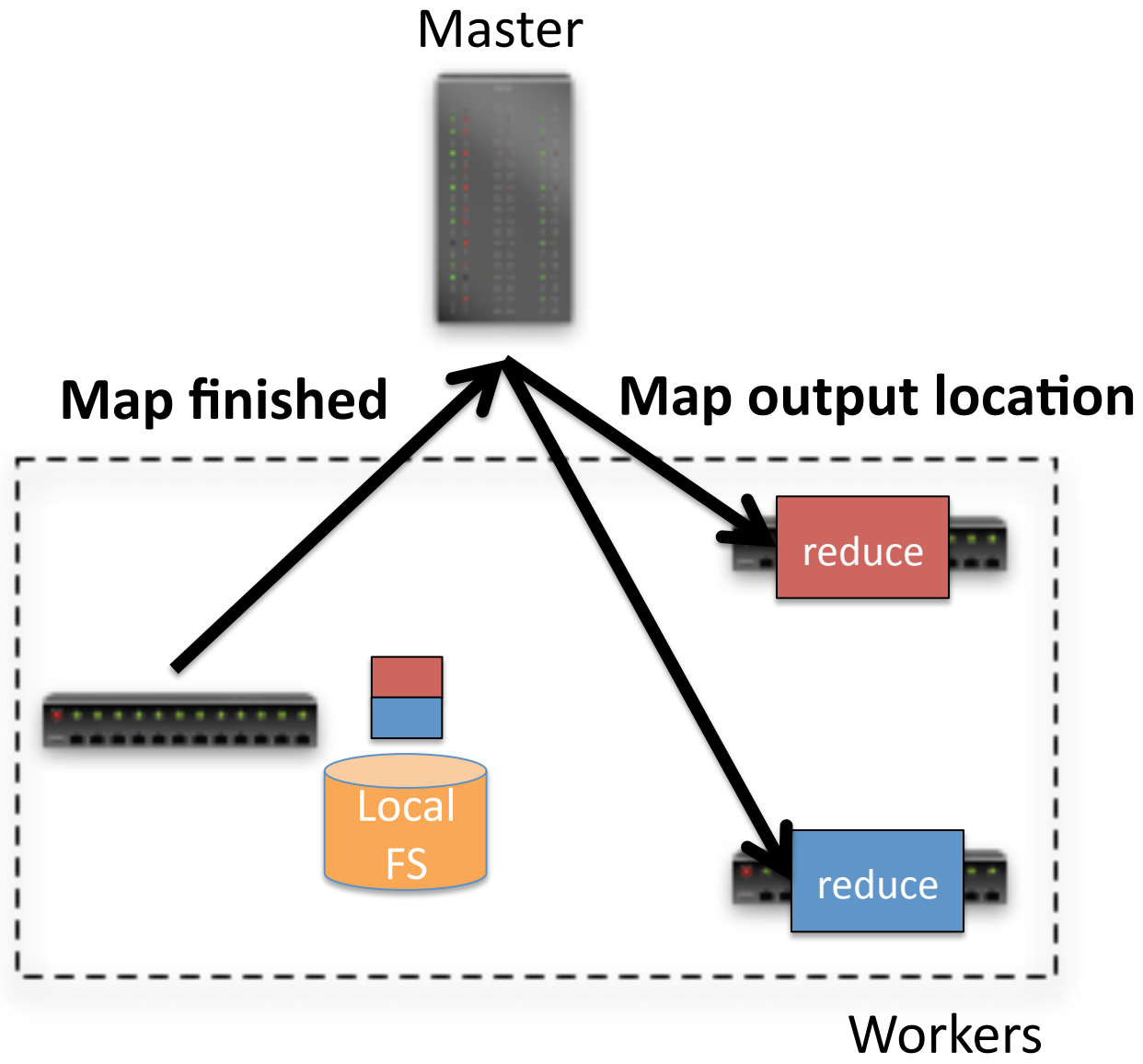
- Apply combiner function to map output
  - Usually reduces the output size



Commit step

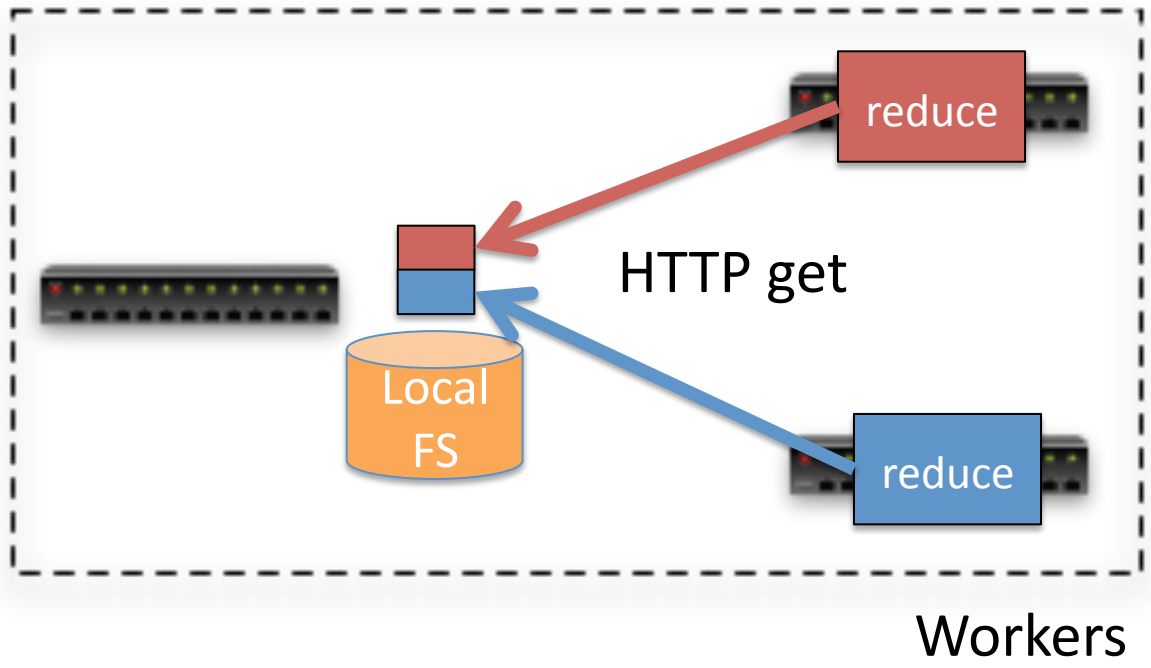
- Final output stored on local file system
- Register file location with TaskTracker




Master
Map finished
Map output location
reduce
Local
FS
reduce
Workers

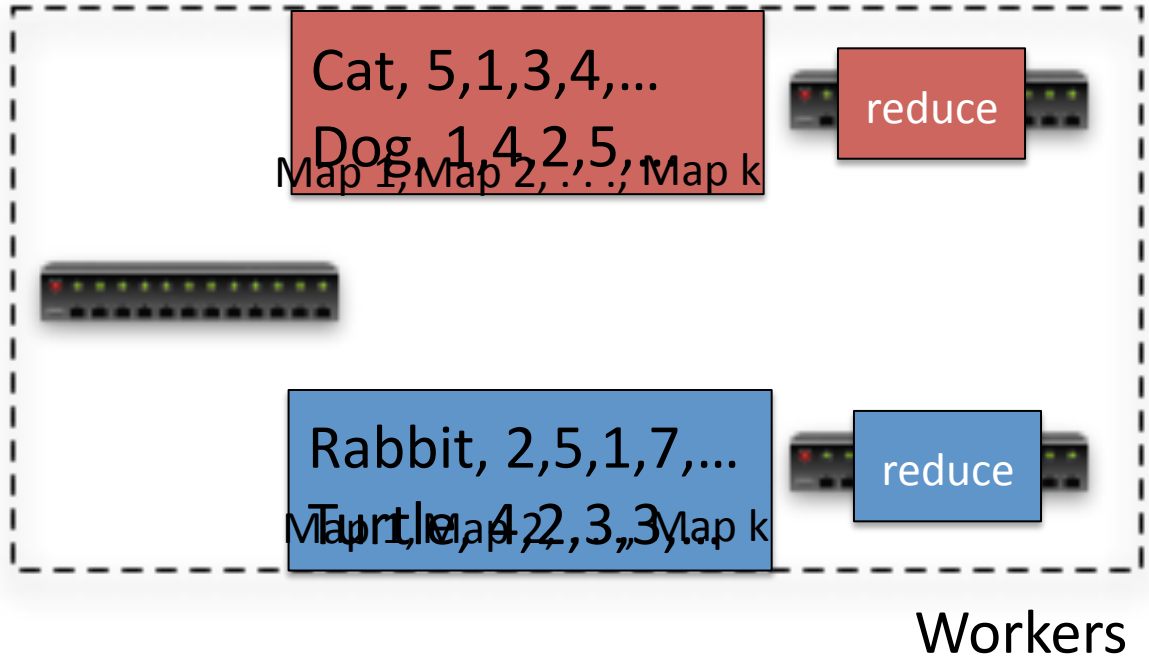
Shuffle step

- Reduce tasks **pull** data from map output locations



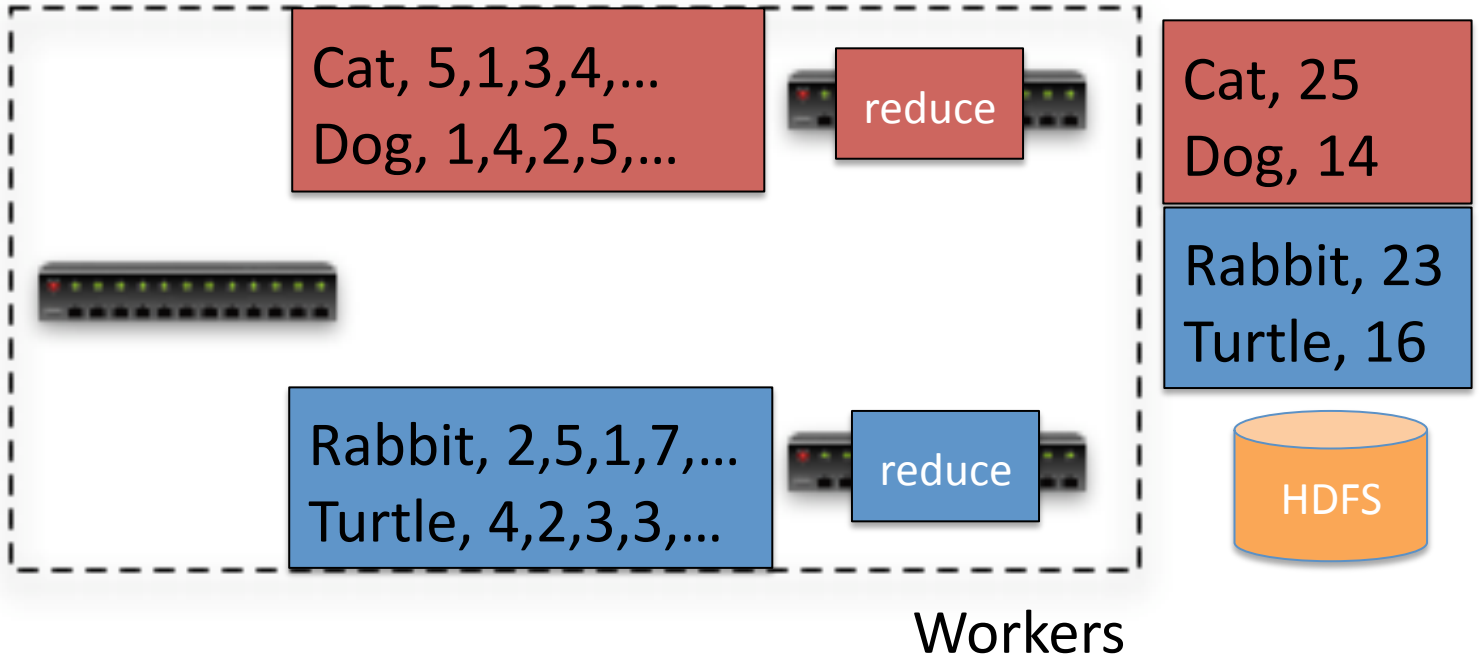
Group step (required)

- When <u>all sorted</u> runs are received
- merge-sort runs (optionally apply combiner)



Reduce step

- Call reduce function on each <key, list of values>
- Write final output to HDFS



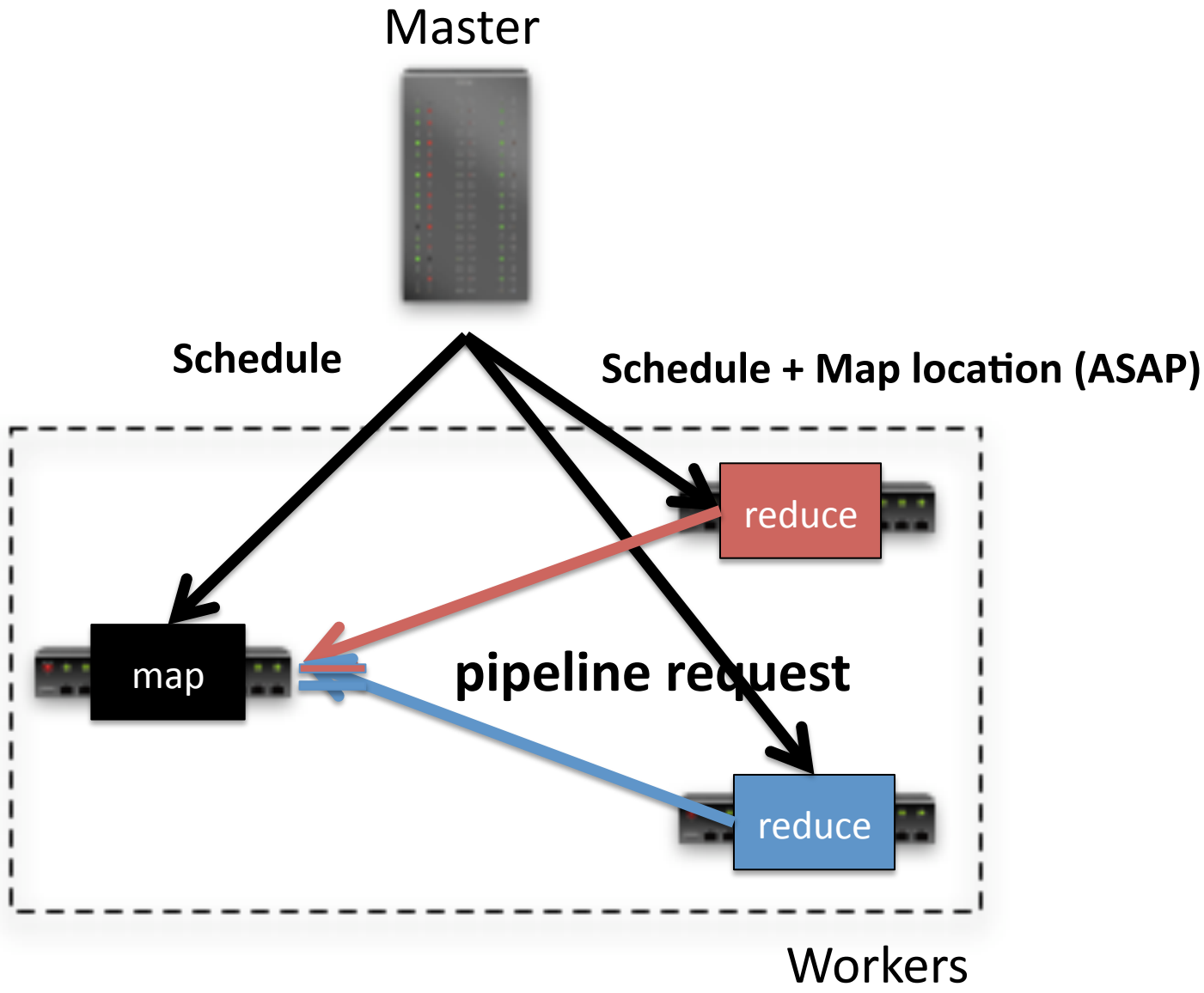
Outline

1. Hadoop MR Background
2. **Hadoop Online Prototype (HOP)**
   - **Implementation**
   - **Online Aggregation**
   - **Stream Processing (see paper)**
3. Performance (blocking vs. pipelining)
4. Future Work

Hadoop Online Prototype (HOP)

- *Pipelining* between operators
  - Data **pushed** from producers to consumers
  - Data transfer scheduled concurrently with operator computation

- HOP API
  - ✓ No changes required to existing clients
    - Pig, Hive, Jaql still work
  - \+ Configuration for pipeline/block modes
  - \+ JobTracker accepts a series of jobs


Master
Schedule
Schedule + Map location (ASAP)
reduce
map
pipeline request
reduce
Workers

Pipelining Data Unit

- Initial design: pipeline *eagerly* (each record)
  - Prevents map side **group** and **combine** step
  - Map computation can block on network I/O

- Revised design: pipeline small sorted runs (spills)
  - **Task thread**: apply (map/reduce) function, buffer output
  - **Spill thread**: sort & combine buffer, spill to a file
  - **TaskTracker**: service consumer requests

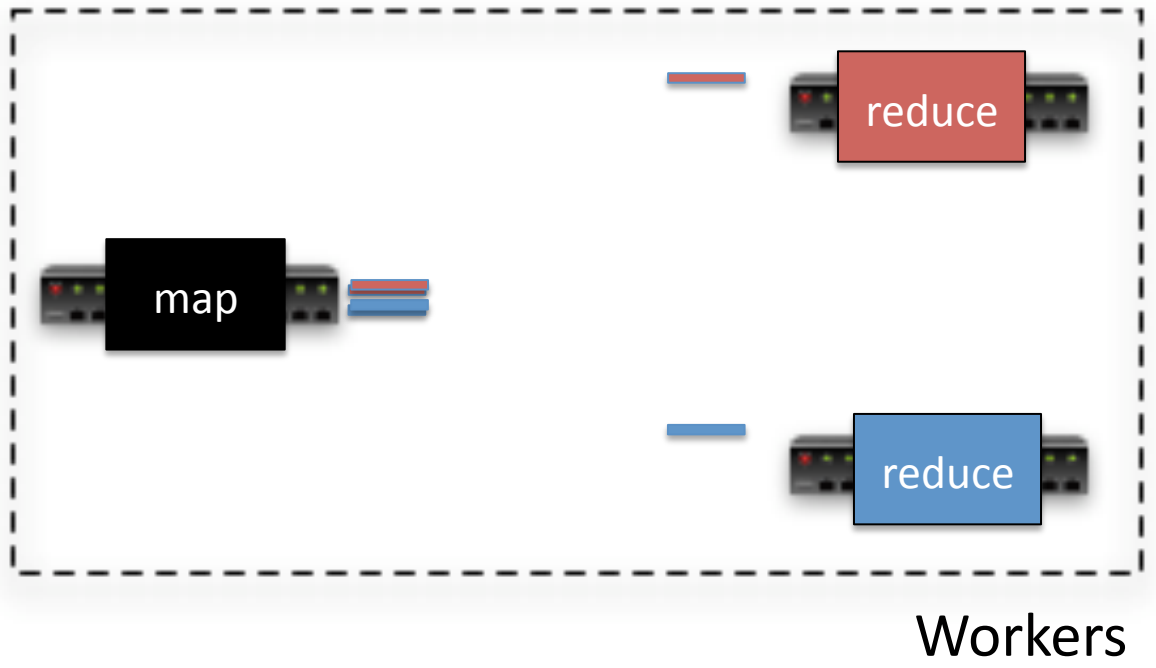
Simple Adaptive Policy

- **Halt** pipeline when …
  1. Unserviced spill files backup **OR**
  2. Effective combiner

- **Resume** pipeline by first …
  - merging & combining accumulated spill files into a single file
  - ➢ Map tasks adaptively take on more work

Pipelined shuffle step

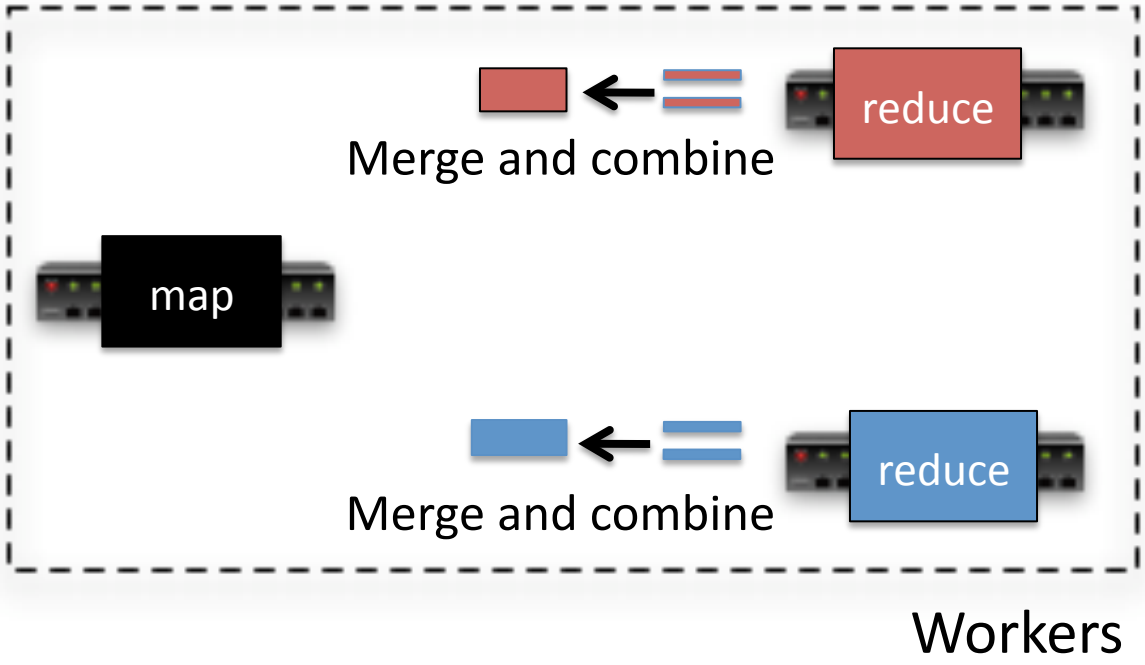
- Each map task can send multiple sorted runs



Pipelined shuffle step

- Each map task can send multiple sorted runs
- Reducers perform early group + combine during shuffle

  ➔ Also done in blocking but more so when pipelining

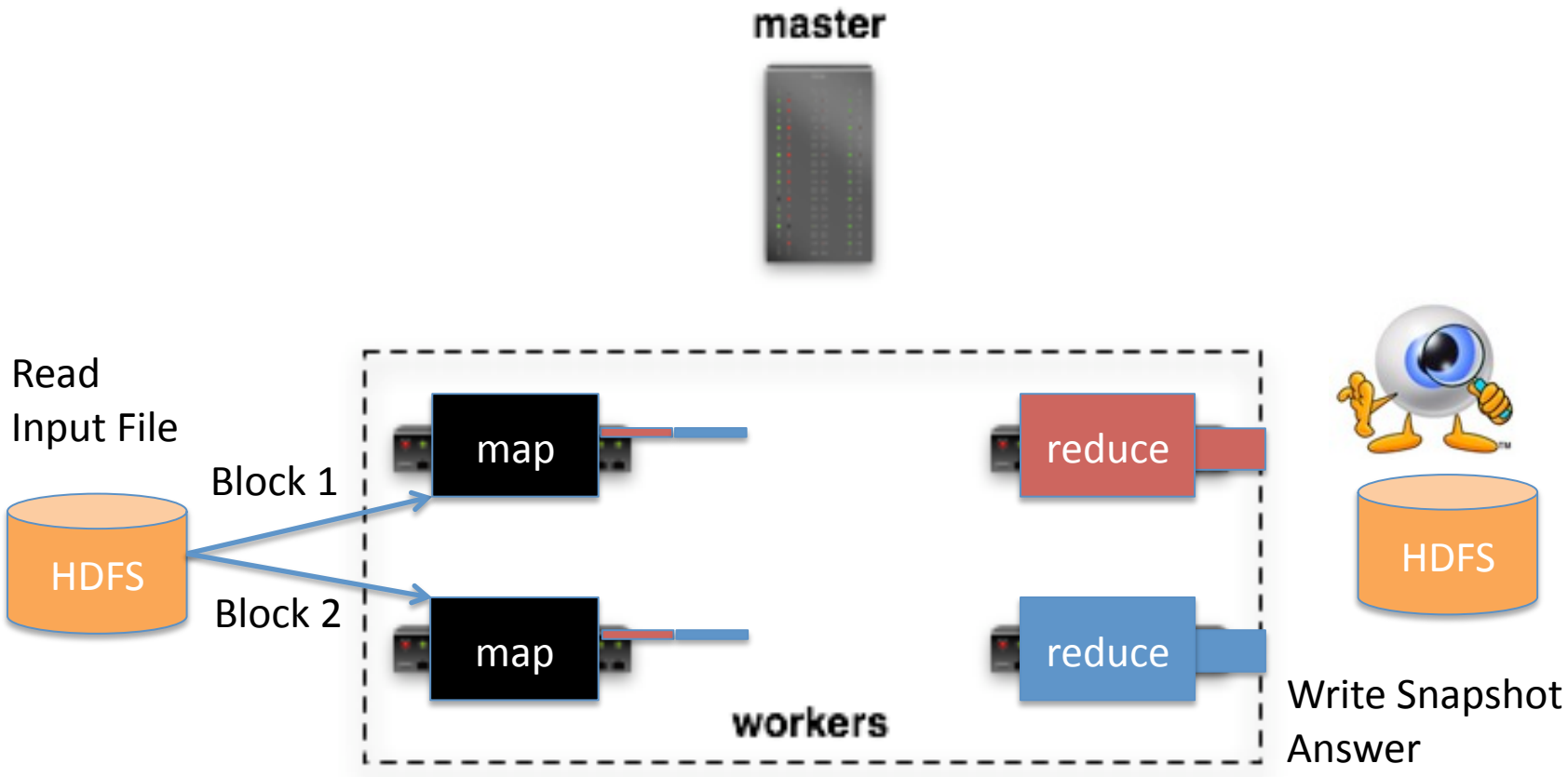


Pipelined Fault Tolerance (PFT)

- Simple PFT design:
  - Reduce treats in-progress map output as *tentative*
  - **If** map dies **then** throw away its output
  - **If** map succeeds **then** accept its output

- Revised PFT design:
  - Spill files have deterministic boundaries and are assigned a sequence number
  - **Correctness**: Reduce tasks ensure spill files are idempotent
  - **Optimization**: Map tasks avoid sending redundant spill files

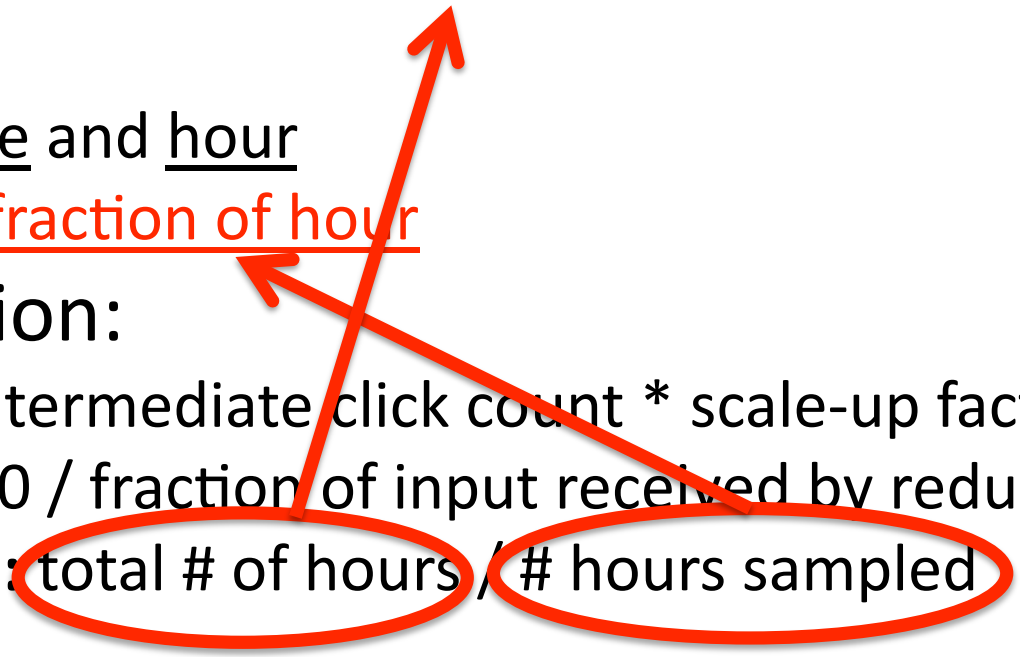
Online Aggregation

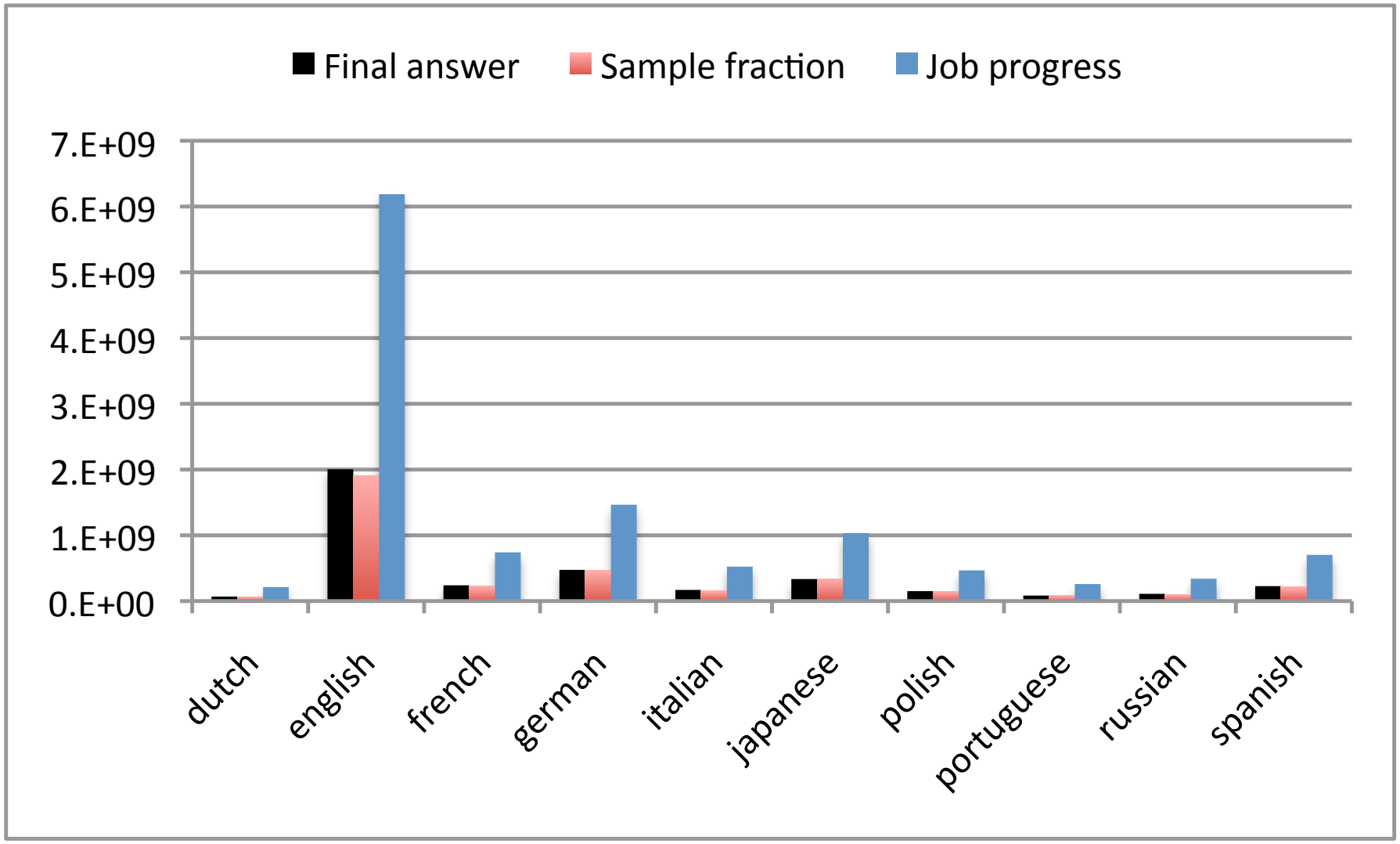


- Execute reduce task on intermediate data
  - Intermediate results published to HDFS

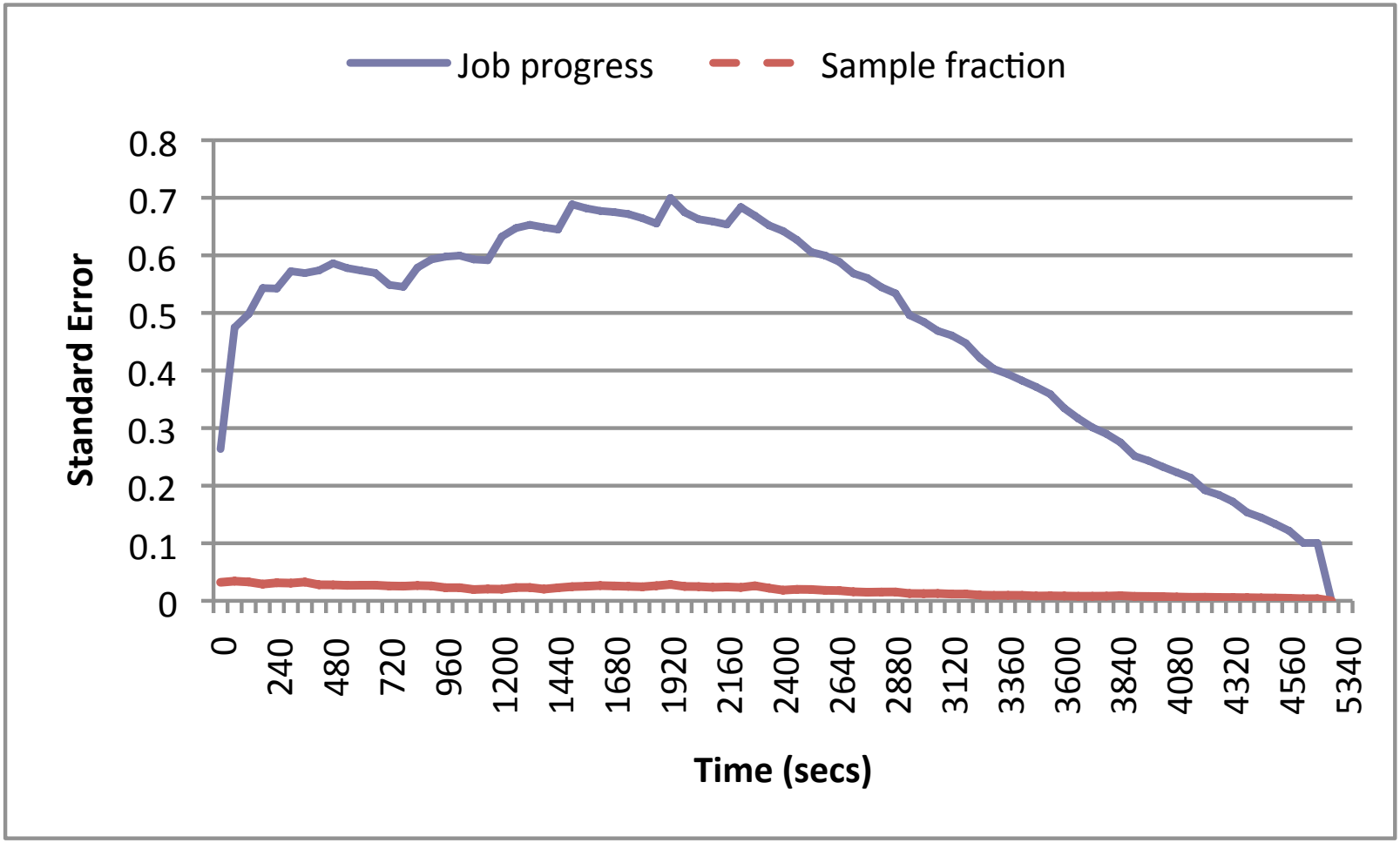
Example Approximation Query

- The data:
  - Wikipedia traffic statistics (1TB)
  - Webpage clicks/hour
  - 5066 compressed files (each file = 1 hour click logs)
- The query:
  - *group by* language and hour
  - *count* clicks and fraction of hour
- The approximation:
  - Final answer ≈ (intermediate click count * scale-up factor)
  1. **Job progress:** 1.0 / fraction of input received by reducers
  2. **Sample fraction:** total # of hours / # hours sampled





- Bar graph shows results for a single hour (1600)
  - Taken less than 2 minutes into a ~2 hour job!



- **Approximation error**: |*estimate* – *actual*| / *actual*
  - Job progress assumes hours are uniformly sampled
  - Sample fraction ≈ sample distribution of each hour

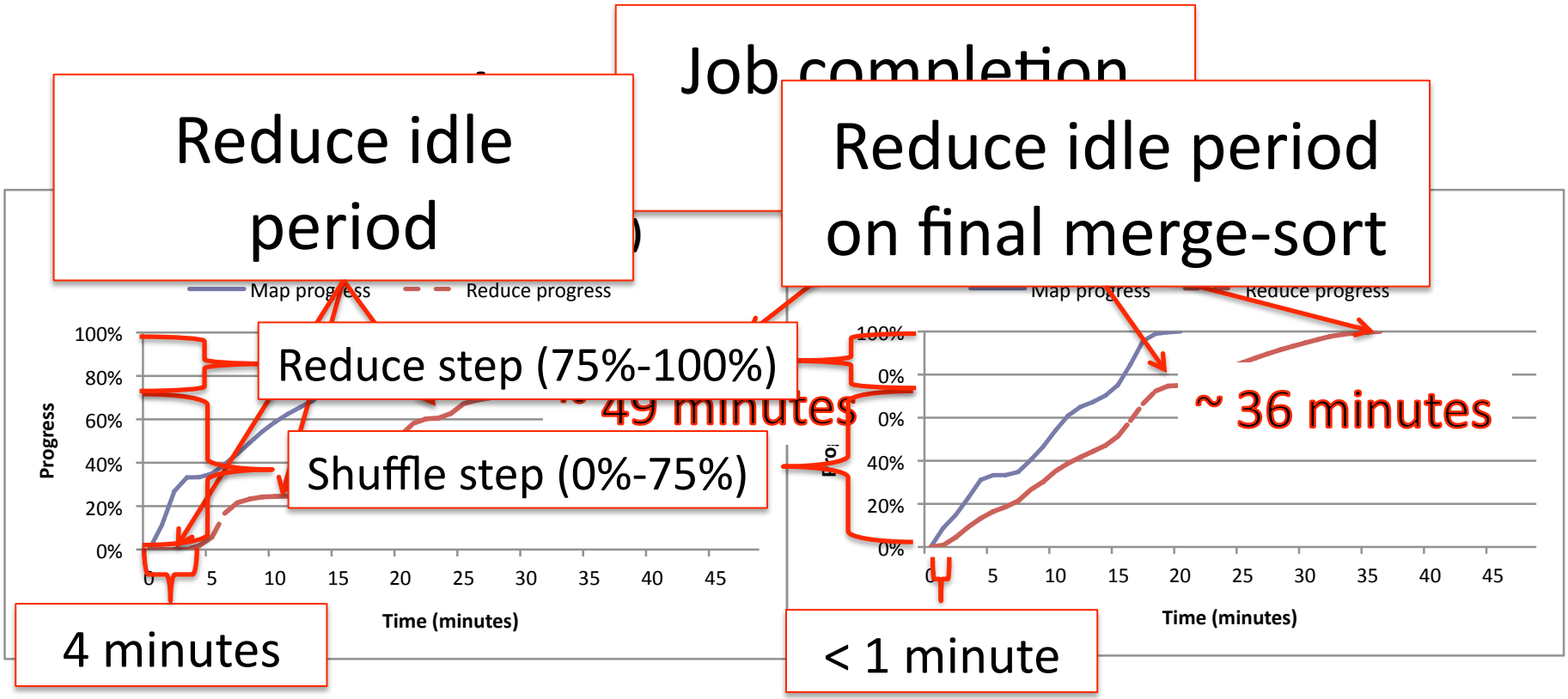
Outline

1. Hadoop MR Background
2. Hadoop Online Prototype (HOP)
3. **Performance (blocking vs. pipelining)**
   - **Does block size matter?**
4. Future Work

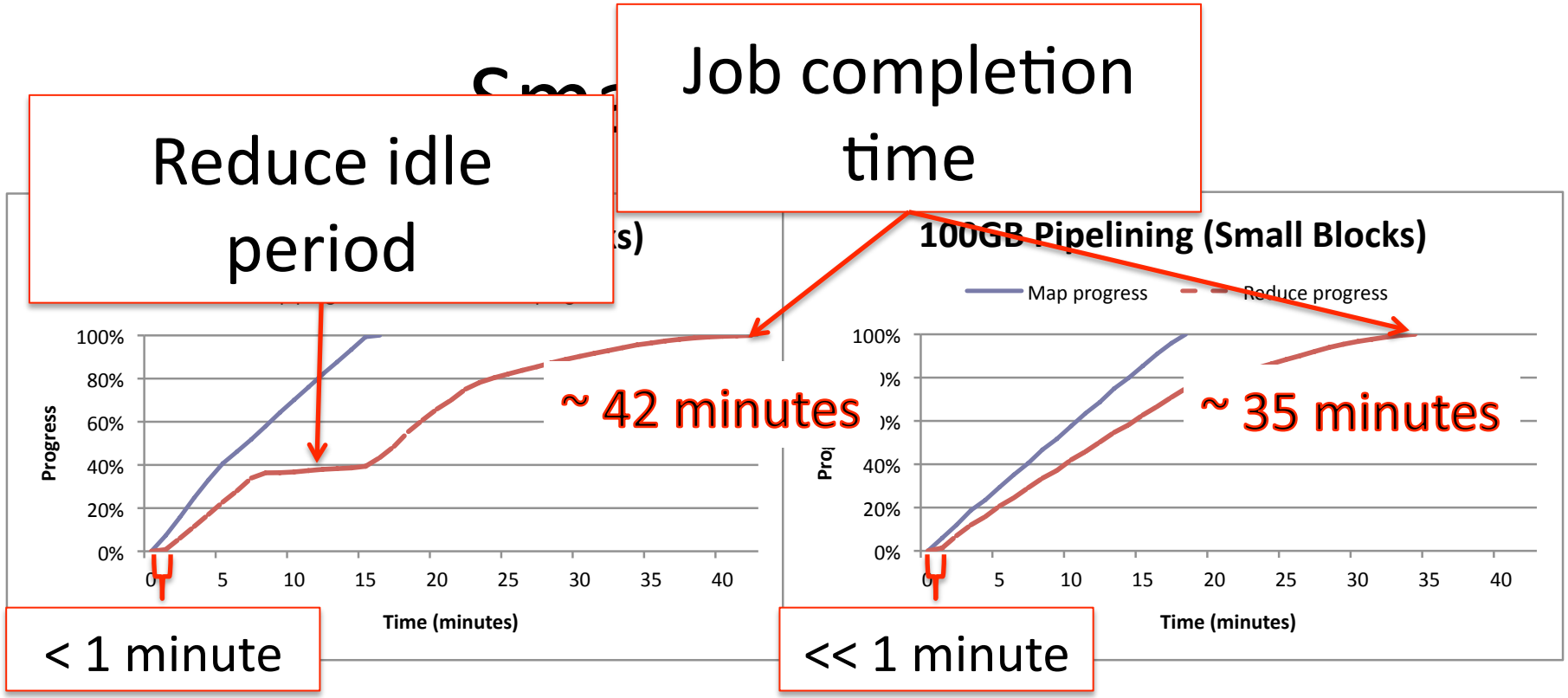
Large vs. Small Block Size

- Map input is a single block (*Hadoop default*)
  - Increasing block size => fewer maps with longer runtimes

- Wordcount on 100GB randomly generated words
  - 20 extra-large EC2 nodes: 4 cores, 15GB RAM
    - Slot capacity: 80 maps (4 per node), 60 reduces (3 per node)
  - Two jobs: large vs. small block size
    - Job 1 (large): 512MB (240 maps/blocks)
    - Job 2 (small): 32MB (3120 maps/blocks)
  - Both jobs hard coded to use 60 reduce tasks



- Poor CPU and I/O overlap
  - Especially in blocking mode
- Pipelining + adaptive policy less sensitive to block sizes
  - BUT incurs extra sorting between shuffle and reduce steps



- Improves CPU and I/O overlap
  - BUT idle periods still exist in blocking mode shuffle step
  - AND increases scheduler overhead (3120 maps)
  - AND increases HDFS (NameNode) memory pressure
- Adaptive policy finds the right degree of pipelined parallelism
  - Based on runtime dynamics (reducer load, network capacity, etc.)

Future Work

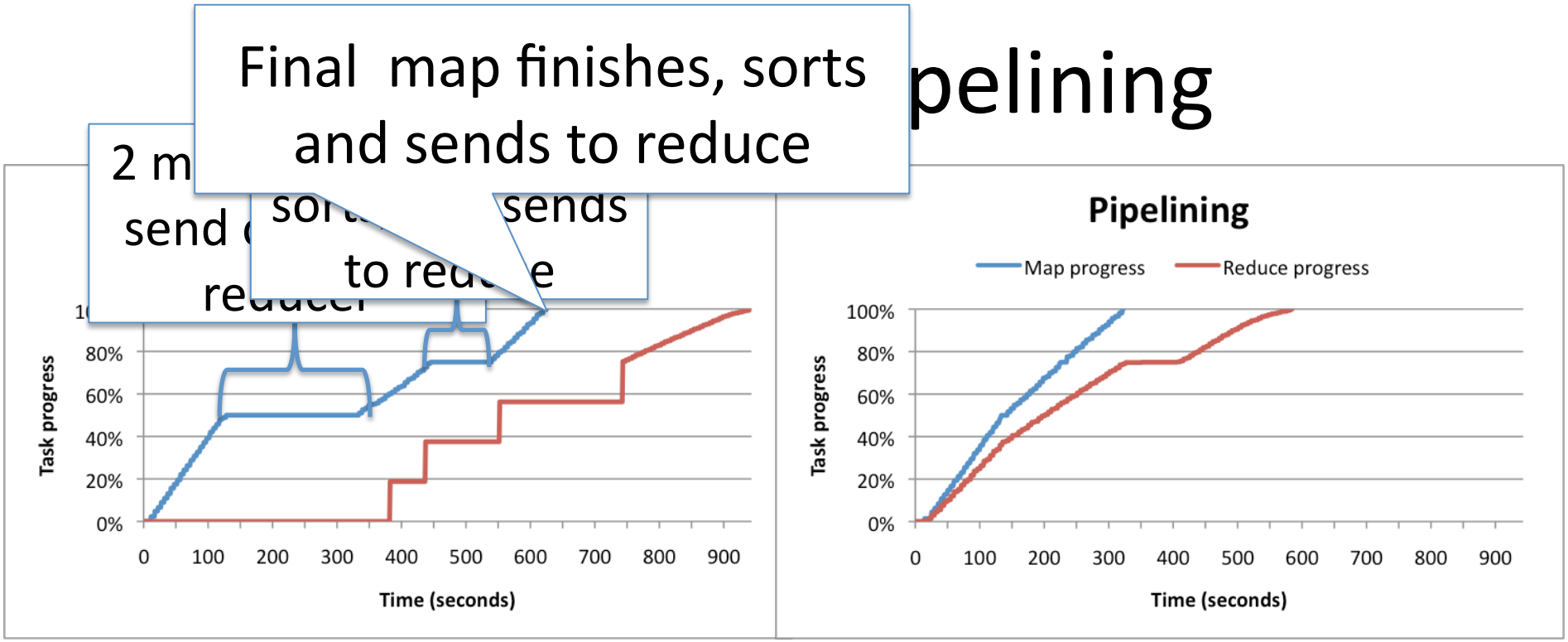
1. Blocking vs. Pipelining
   - Comprehensive performance study at scale
   - Hadoop optimizer
2. Online Aggregation
   - Random sampling of the input
   - Better UI for approximate results
3. Stream Processing
   - Better interface for window management
   - Support for high-level query languages

Thank you!

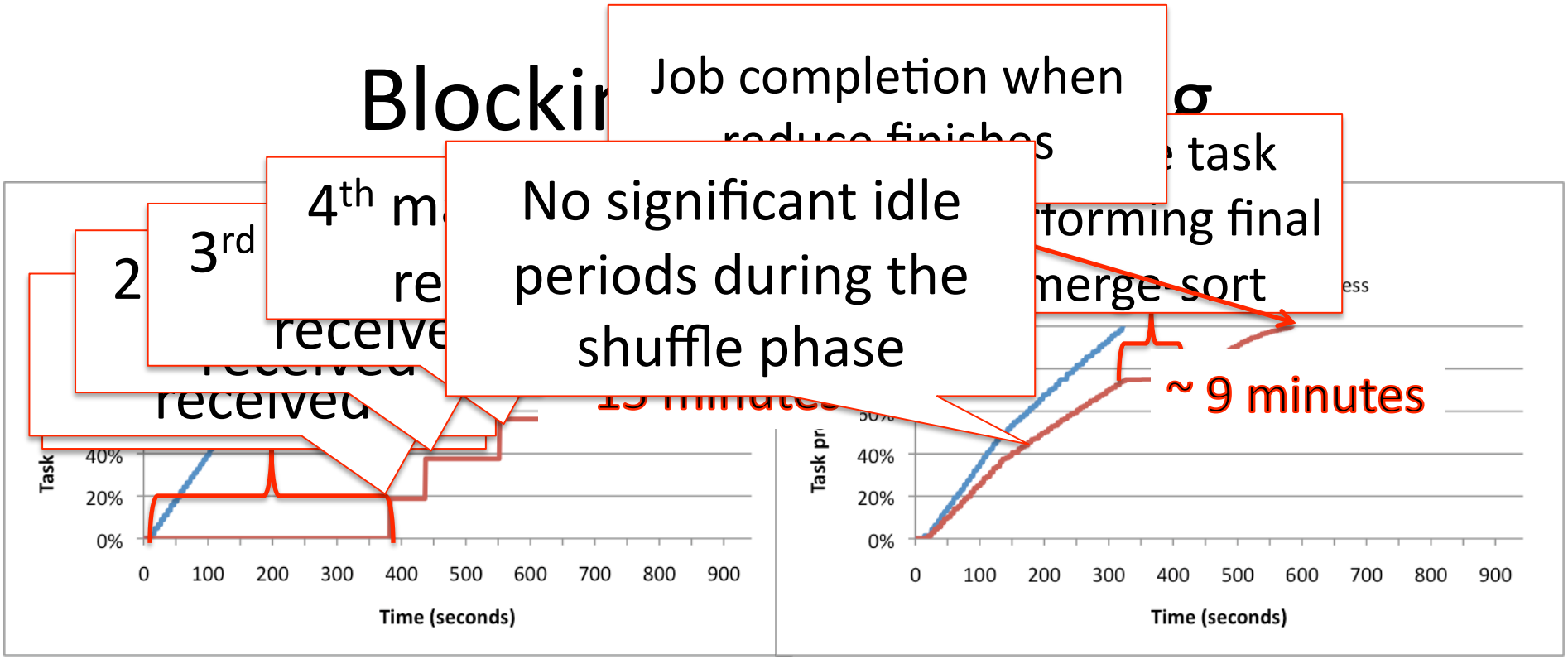
More information: http://boom.cs.berkeley.edu

HOP code: http://code.google.com/p/hop/

pelining



- Simple wordcount on two (small) EC2 nodes
  1. Map machine: 2 map slots
  2. Reduce machine: 2 reduce slots
- Input 2GB data, 512MB block size
  - So job contains 4 maps and (a hard-coded) 2 reduces



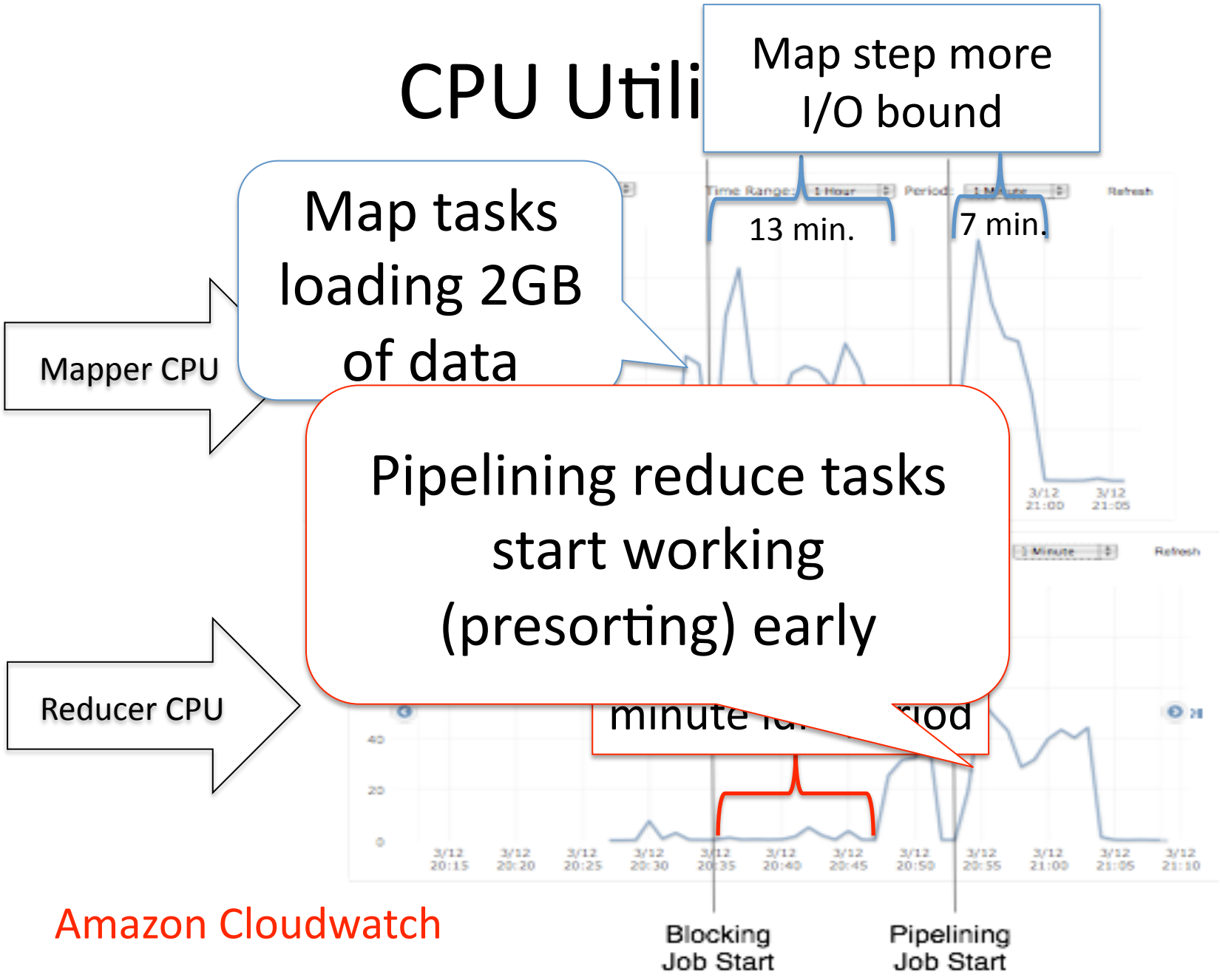
- Simple wordcount on two (small) EC2 nodes
  1. Map machine: 2 map slots
  2. Reduce machine: 2 reduce slots
- Input 2GB data, 512MB block size
  - So job contains 4 maps and (a hard-coded) 2 reduces

Recall in blocking mode …

- Operators block
  - Poor CPU and I/O overlap
  - Reduce task idle periods

- Only the final answer is fetched
  - So more data is fetched resulting in…
  - Network traffic spikes
  - Especially when a group of maps finish

CPU Utili



Amazon Cloudwatch

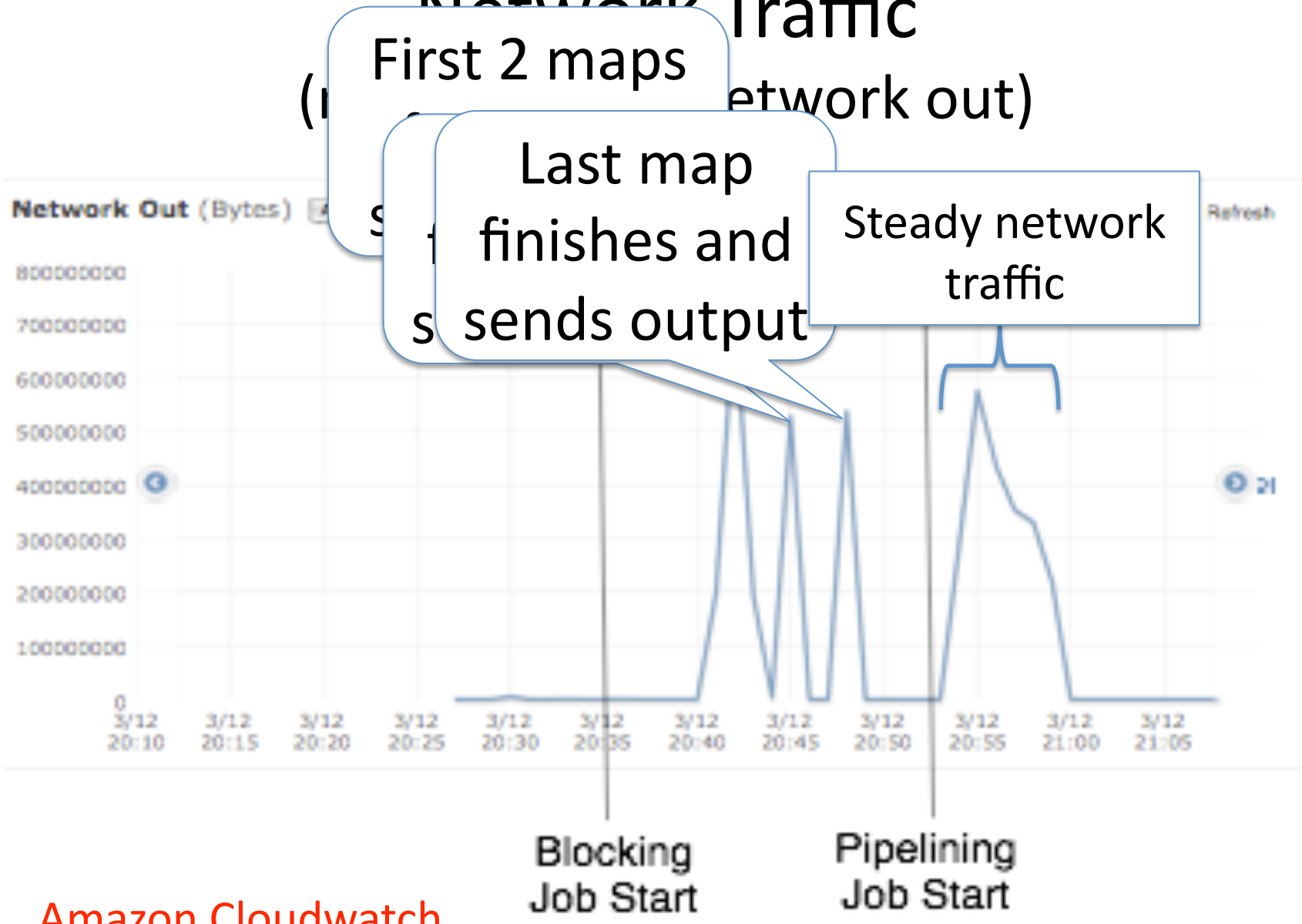
Recall in blocking mode …

- Operators block
  - Poor CPU and I/O overlap
  - Reduce task idle periods

- Only the final answer is fetched
  - So more data is fetched at once resulting in…
  - Network traffic spikes
  - Especially when a group of maps finish

Network Traffic

(n etwork out)



Amazon Cloudwatch

Benefits of Pipelining

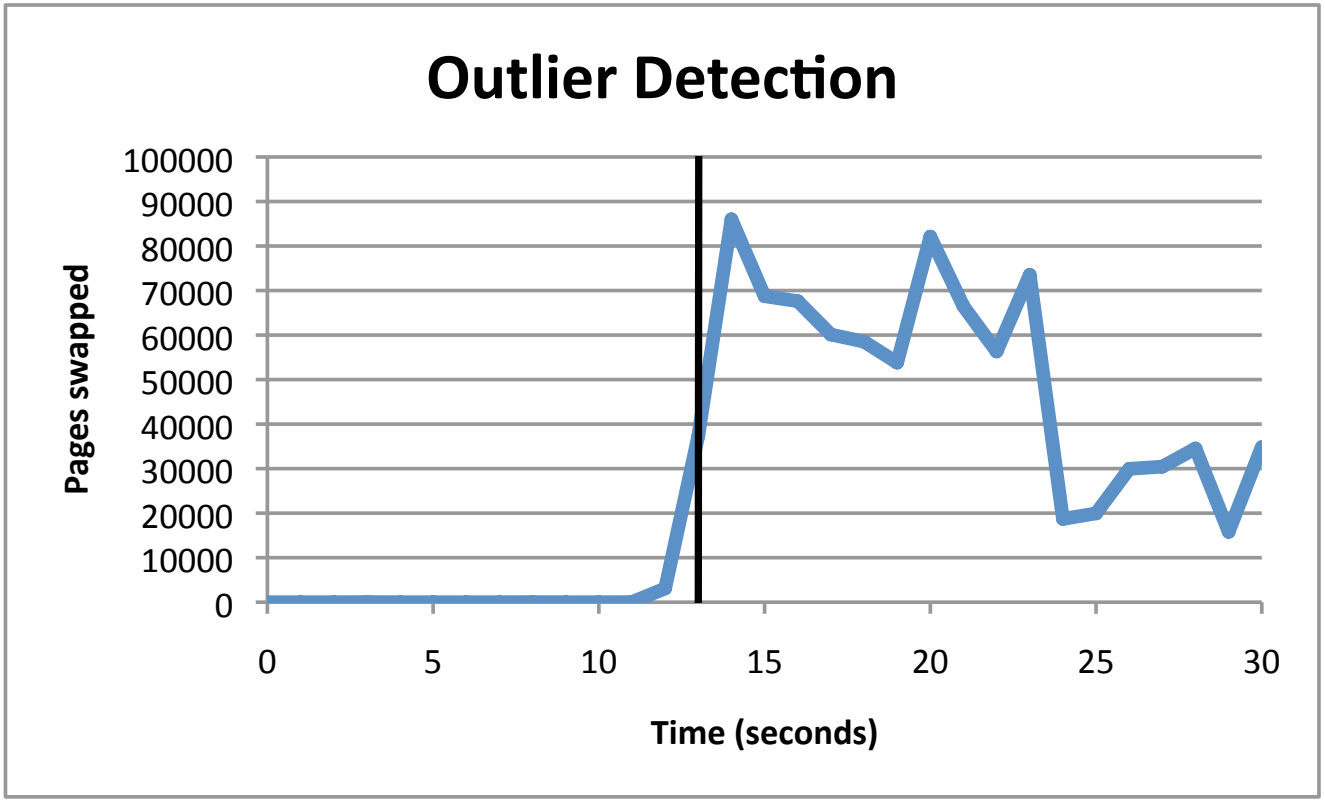
- Online aggregation
  - An early view of the result from a running computation
  - Interactive data analysis (you say when to stop)
- Stream processing
  - Tasks operate on infinite data streams
  - Real-time data analysis
- Performance? Pipelining can …
  - Improve CPU and I/O overlap
  - Steady network traffic (fewer load spikes)
  - Improve cluster utilization (reducers do more work)

Stream Processing

- Map and reduce tasks run *continuously*
  - *Scheduler*: wait for required slot capacity
- Map tasks stream spill files
  - Input taken from arbitrary source
    - MapReduce job, TCP socket, log files, etc.
  - Garbage collection handled by system
- Window management done at reducer
  - Reduce output is an infinite series of windowed results
  - Window boundary based on time, record counts, etc.

Real-time Monitoring System

- Use MapReduce to monitor MapReduce
  - Economy of Mechanism
- **Agents** monitor machines
  - Implemented as a continuous map task
  - Record statistics of interest (/proc, log files, etc.)
- **Aggregators** group agent-local statistics
  - Implemented as reduce tasks
  - Aggregate statistics along machine, rack, datacenter
  - Reduce windows: 1, 5, and 15 second load averages

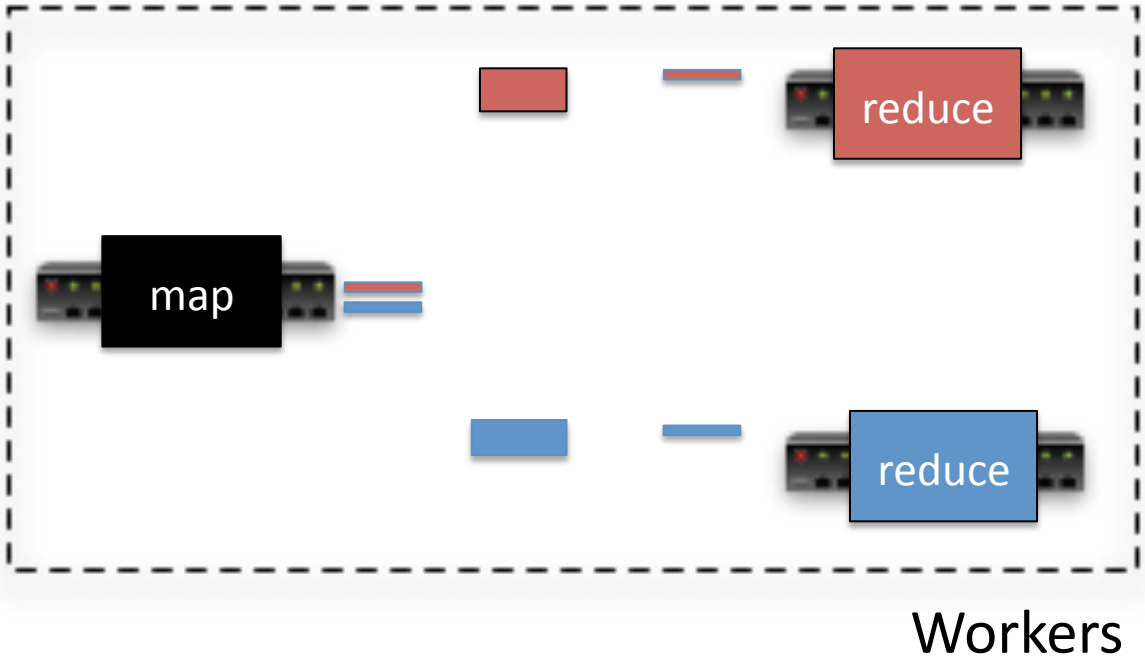


- Monitor /proc/vmstat for swapping
  - Alert triggered after some threshold
- Alert reported around a second after passing threshold
  - Faster than the (~5 second) TaskTracker reporting interval
  - ? Feedback loop to the JobTracker for better scheduling

Pipelined shuffle step

- Each map task can send multiple sorted runs
- Reducers perform early group + combine during shuffle

  ➔ Also done in blocking but more so when pipelining



Hadoop Architecture

- Hadoop MapReduce
  - Single master node (**JobTracker**), many worker nodes (**TaskTrackers**)
  - Client submits a *job* to the JobTracker
  - JobTracker splits each job into *tasks* (map/reduce)
  - Assigns tasks to TaskTrackers on demand
- Hadoop Distributed File System (HDFS)
  - Single name node, many data nodes
  - Data is stored as fixed-size (e.g., 64MB) blocks
  - HDFS typically holds map input and reduce output

Performance

- Why block?
  - Effective combiner
  - Reduce step is a bottleneck
- Why pipeline?
  - Improve cluster utilization
  - Smooth out network traffic