MapReduce and Beyond

Steve Ko

Trivia Quiz: What's Common?

Data-intensive computing with MapReduce!

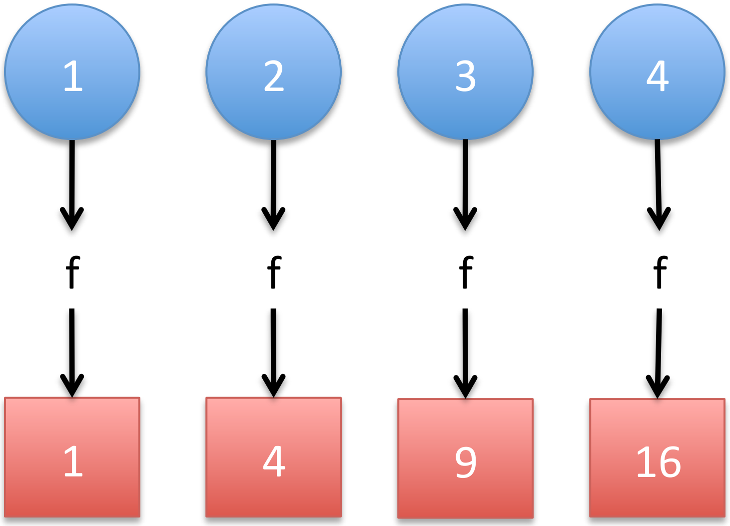
What is MapReduce?

- A system for processing large amounts of data
- Introduced by Google in 2004
- Inspired by map & reduce in Lisp
- OpenSource implementation: Hadoop by Yahoo!
- Used by many, many companies
  - A9.com, AOL, Facebook, The New York Times, Last.fm, Baidu.com, Joost, Veoh, etc.

Background: Map & Reduce in Lisp

- Sum of squares of a list (in Lisp)
- (**map** square ‘(1 2 3 4))
  - Output: (1 4 9 16)

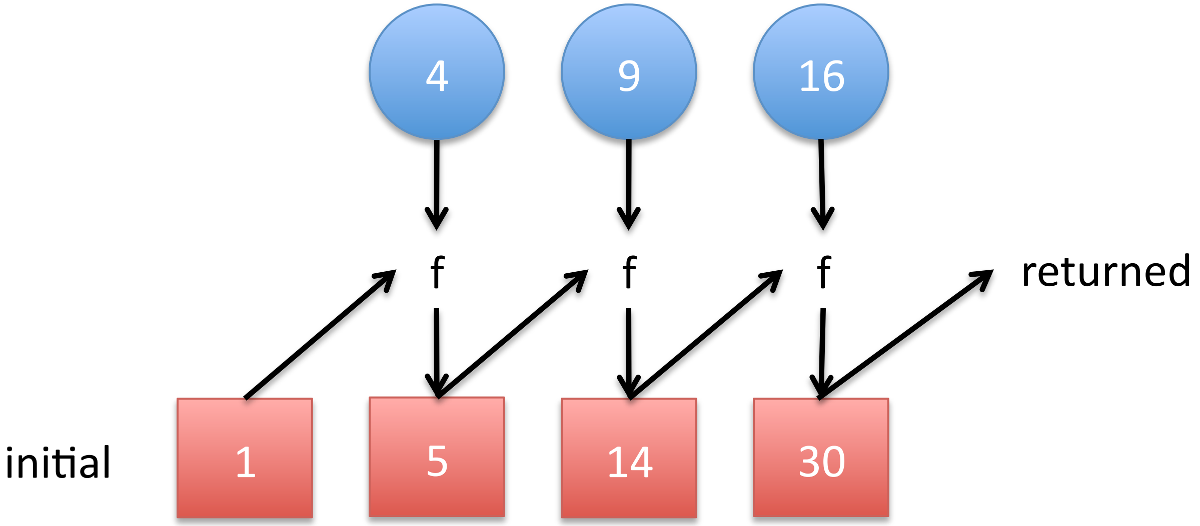
  [processes each record individually]



Background: Map & Reduce in Lisp

- Sum of squares of a list (in Lisp)
- **(reduce** + ‘(1 4 9 16))
  - (+ 16 (+ 9 (+ 4 1)))
  - Output: 30

  [processes set of all records in a batch]



Background: Map & Reduce in Lisp

- Map
  - processes each record individually
- Reduce
  - processes (combines) set of all records in a batch

What Google People Have Noticed

- Keyword search

Map Find a keyword in each web page individually, and if it is found, return the URL of the web page

Reduce Combine all results (URLs) and return it

- Count of the # of occurrences of each word

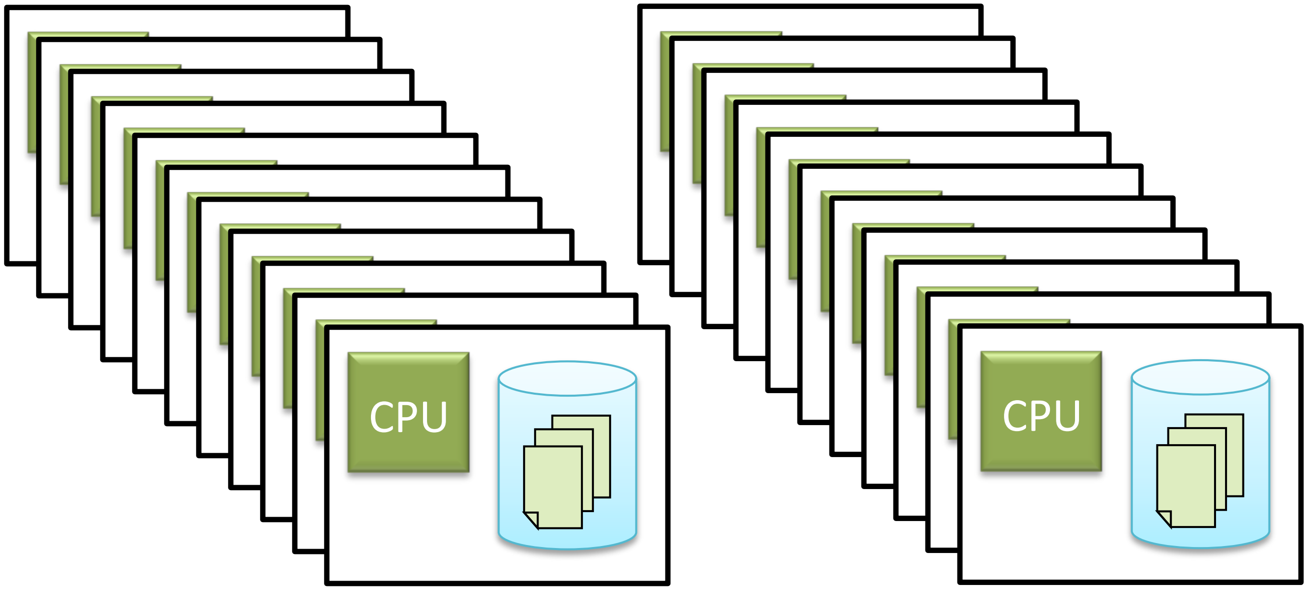
Map Count the # of occurrences in each web page individually, and return the list of <*word*, *#*>

Reduce For each word, sum up (combine) the count

- Notice the similarities?

What Google People Have Noticed

- Lots of storage + compute cycles nearby
- Opportunity
  - Files are distributed already! (GFS)
  - A machine can processes its own web pages (map)



Google MapReduce

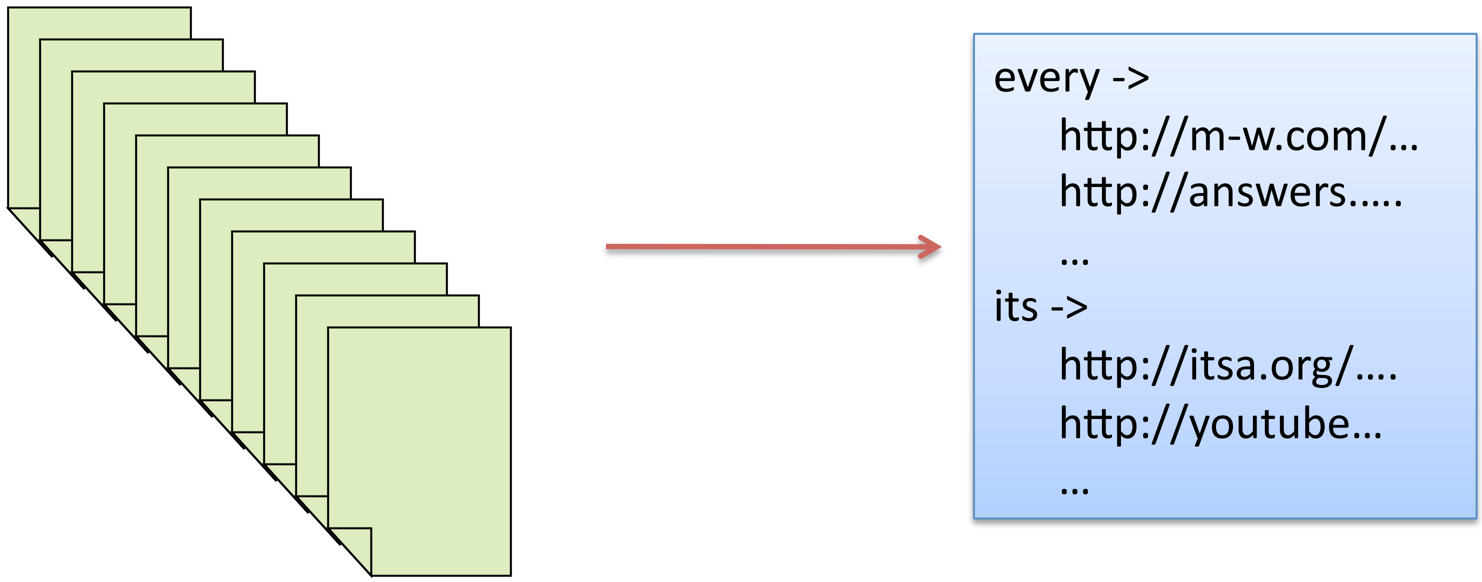
- Took the concept from Lisp, and applied to large-scale data-processing
- Takes two functions from a programmer (*map* and *reduce*), and performs three steps
- Map
  - Runs *map* for each file individually in parallel
- Shuffle
  - Collects the output from all *map* executions
  - Transforms the *map* output into the *reduce* input
  - Divides the *map* output into chunks
- Reduce
  - Runs *reduce* (using a *map* output chunk as the input) in parallel

Programmer's Point of View

- Programmer writes two functions – *map()* and *reduce()*
- The programming interface is fixed
  - map (in_key, in_value) ->
    list of (out_key, intermediate_value)
  - reduce (out_key, list of intermediate_value) ->
    (out_key, out_value)
- Caution: not exactly the same as Lisp

Inverted Indexing Example

- Word -> list of web pages containing the word

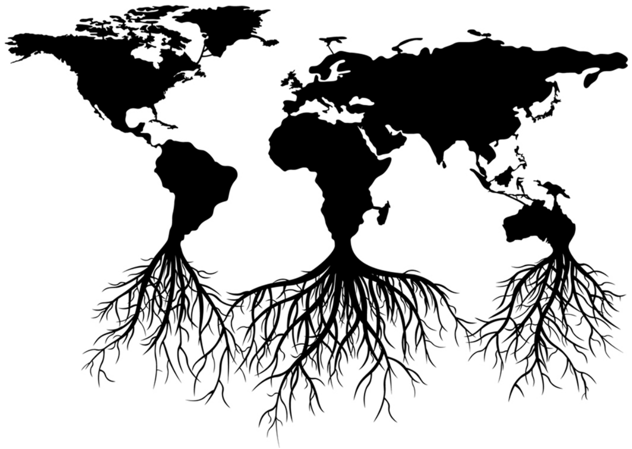


every ->
    http://m-w.com/…
    http://answers.....
    …
its ->
    http://itsa.org/....
    http://youtube…
    …

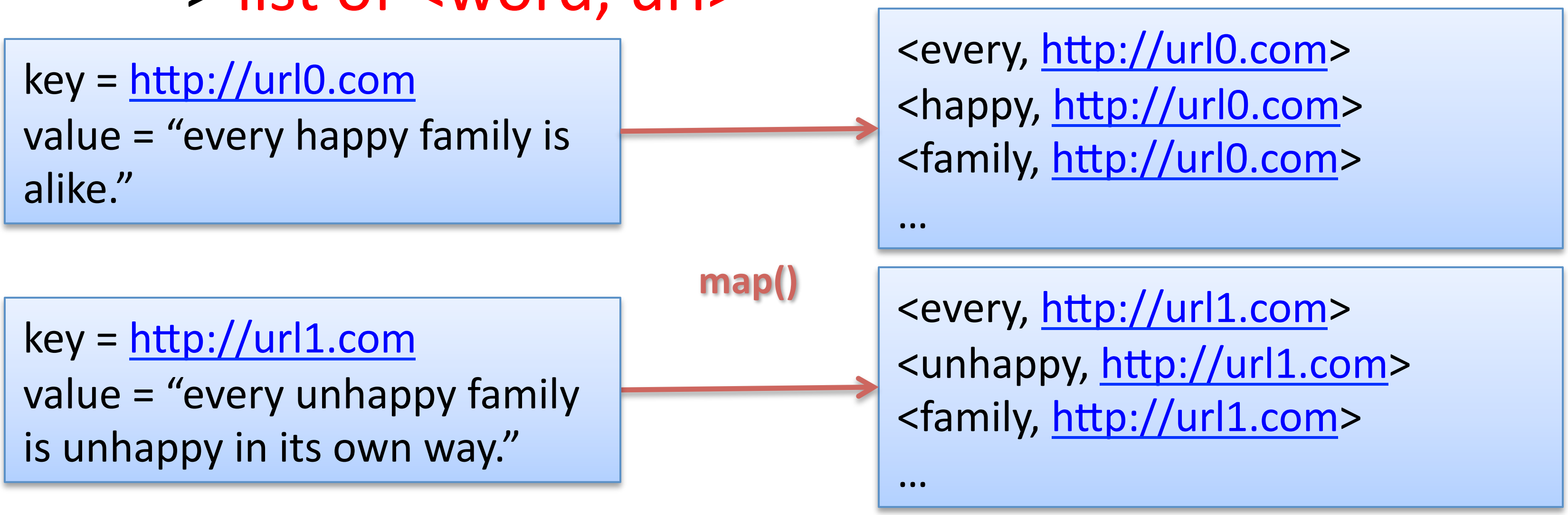
Input: web pages

Output: word-> urls

Map



- Interface
  - Input: <in_key, in_value> pair => <url, content>
  - Output: list of intermediate <key, value> pairs => list of <word, url>



key = http://url0.com
value = "every happy family is alike."

<every, http://url0.com>
<happy, http://url0.com>
<family, http://url0.com>
…

key = http://url1.com
value = "every unhappy family is unhappy in its own way."

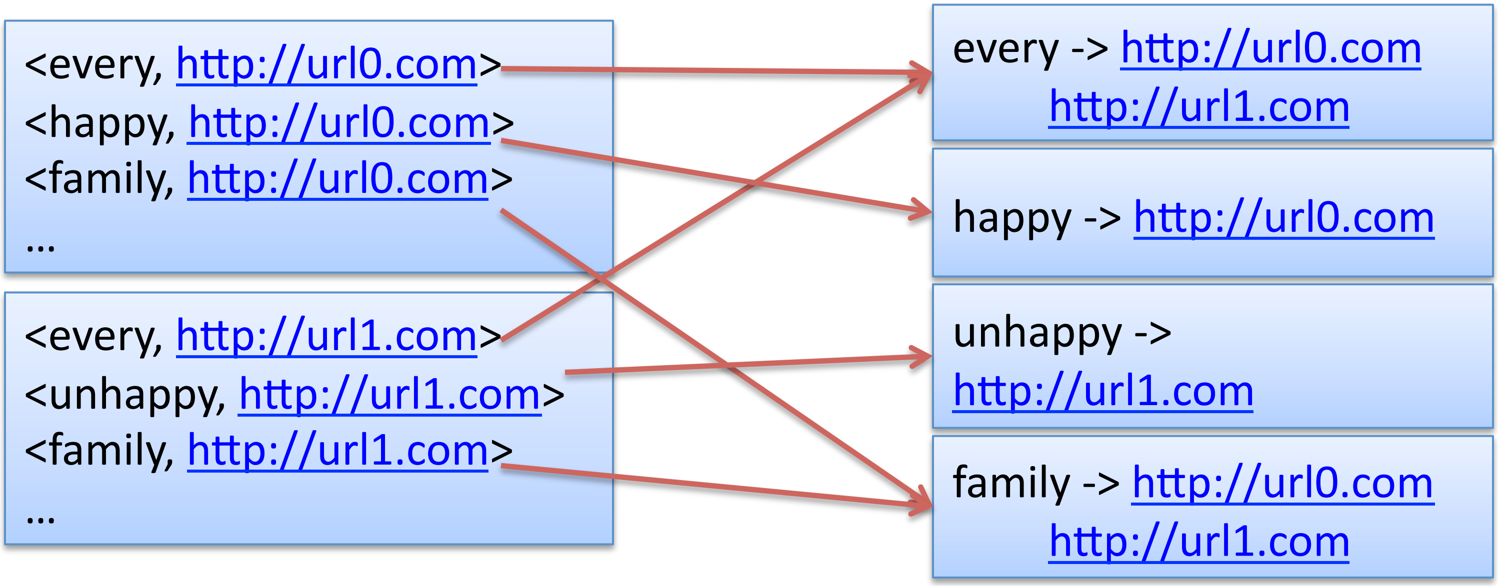
<every, http://url1.com>
<unhappy, http://url1.com>
<family, http://url1.com>
…

Map Input: <url, content>

Map Output: list of <word, url>

Shuffle

- MapReduce system
  - Collects outputs from all *map* executions
  - Groups all intermediate values by the same key



<every, http://url0.com>
<happy, http://url0.com>
<family, http://url0.com>
…

<every, http://url1.com>
<unhappy, http://url1.com>
<family, http://url1.com>
…

every -> http://url0.com
http://url1.com

happy -> http://url0.com

unhappy -> http://url1.com

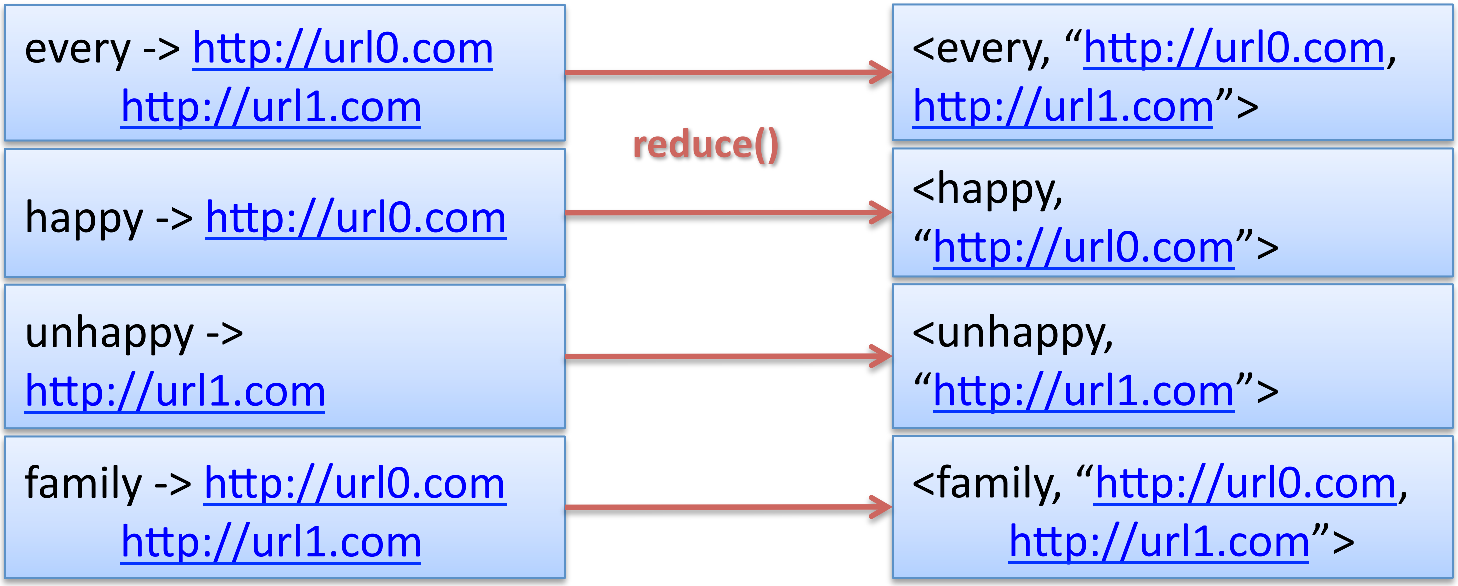
family -> http://url0.com
http://url1.com

Map Output: list of <word, url>

Reduce Input: <word, list of urls>

Reduce

- Interface
  - Input: <out_key, list of intermediate_value>
  - Output: <out_key, out_value>



every -> http://url0.com http://url1.com

happy -> http://url0.com

unhappy -> http://url1.com

family -> http://url0.com http://url1.com

<every, "http://url0.com, http://url1.com">

<happy, "http://url0.com">

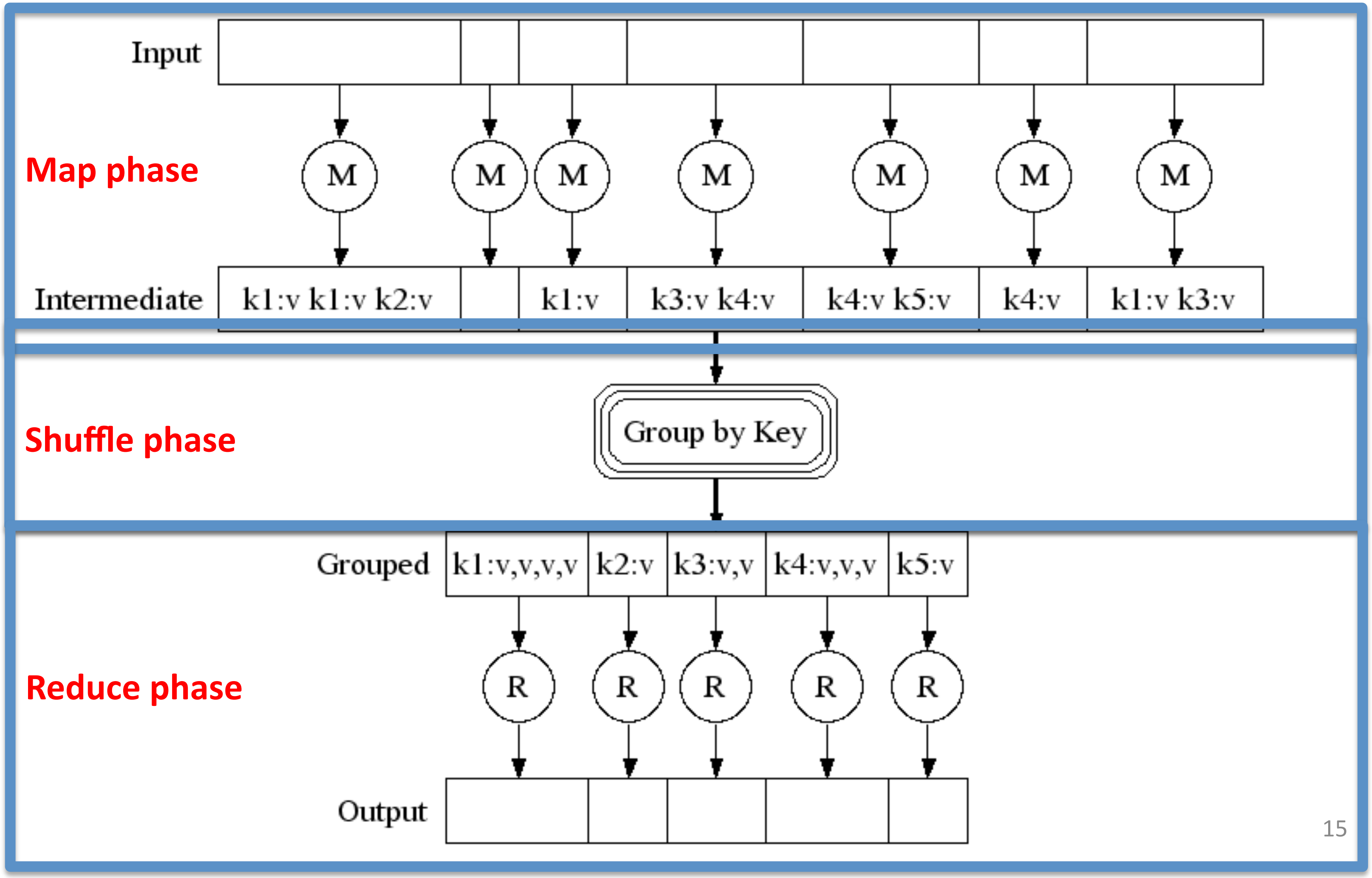
<unhappy, "http://url1.com">

<family, "http://url0.com, http://url1.com">

Reduce Input: <word, list of urls>

Reduce Output: <word, string of urls>

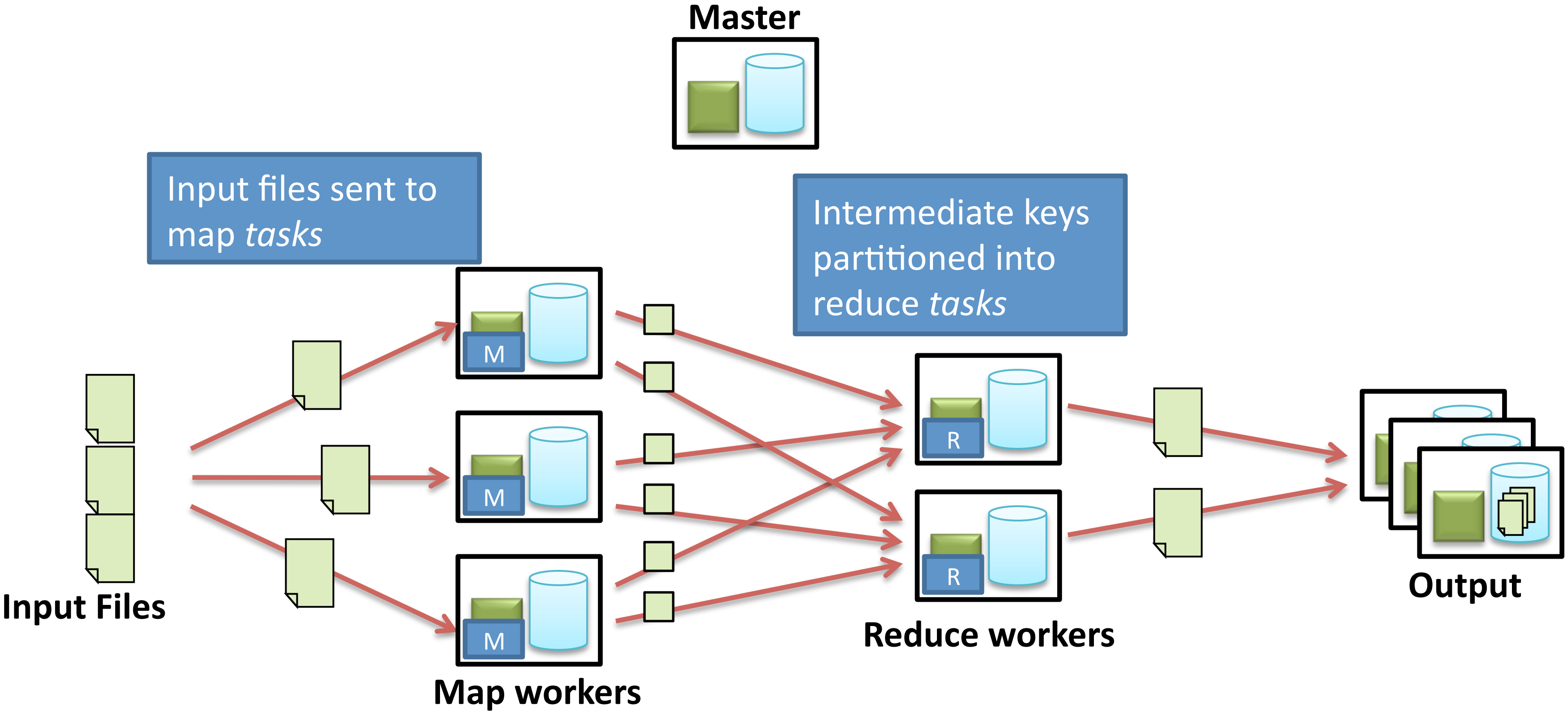
Execution Overview



Implementing MapReduce

- Externally for user
  - Write a map function, and a reduce function
  - Submit a job; wait for result
  - No need to know anything about the environment (Google: 4000 servers + 48000 disks, many failures)
- Internally for MapReduce system designer
  - Run map in parallel
  - Shuffle: combine map results to produce reduce input
  - Run reduce in parallel
  - Deal with failures

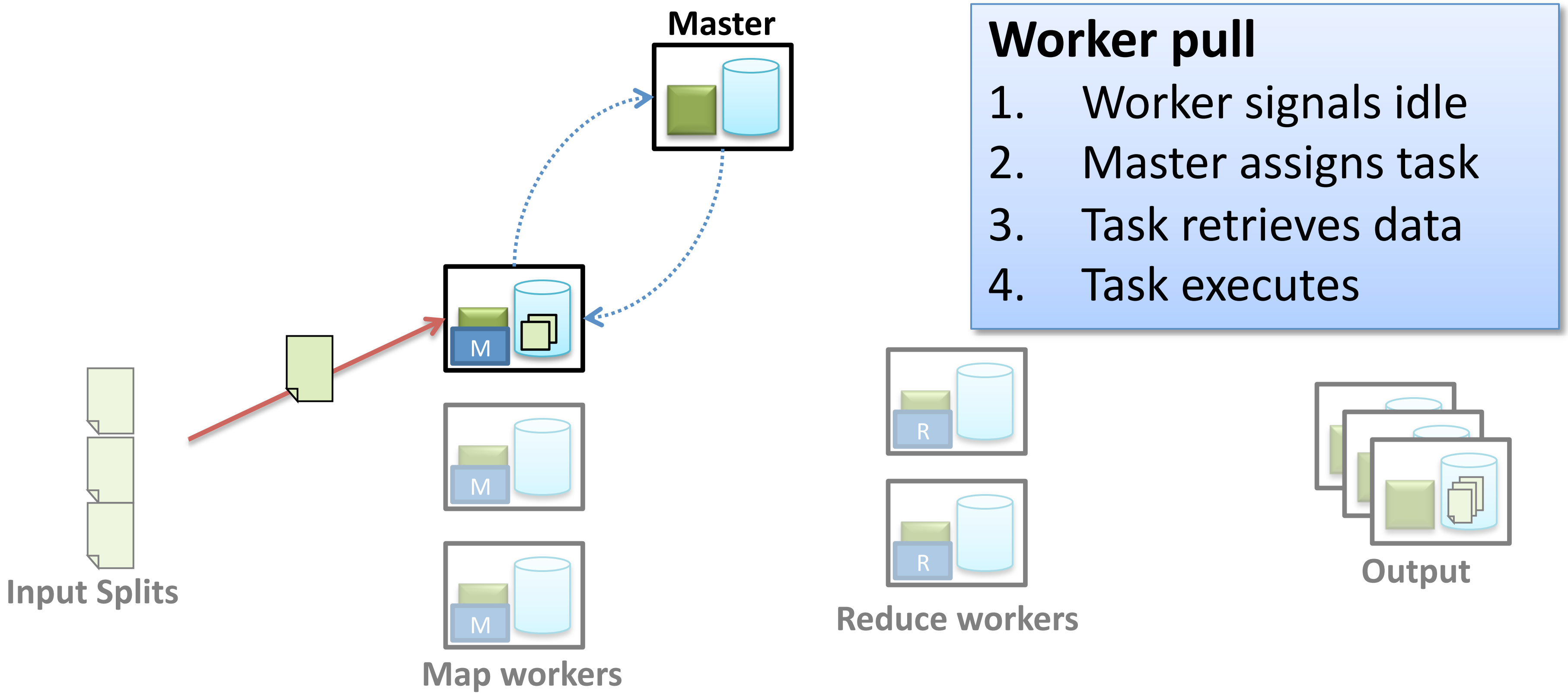
Execution Overview



Input files sent to map *tasks*

Intermediate keys partitioned into reduce *tasks*

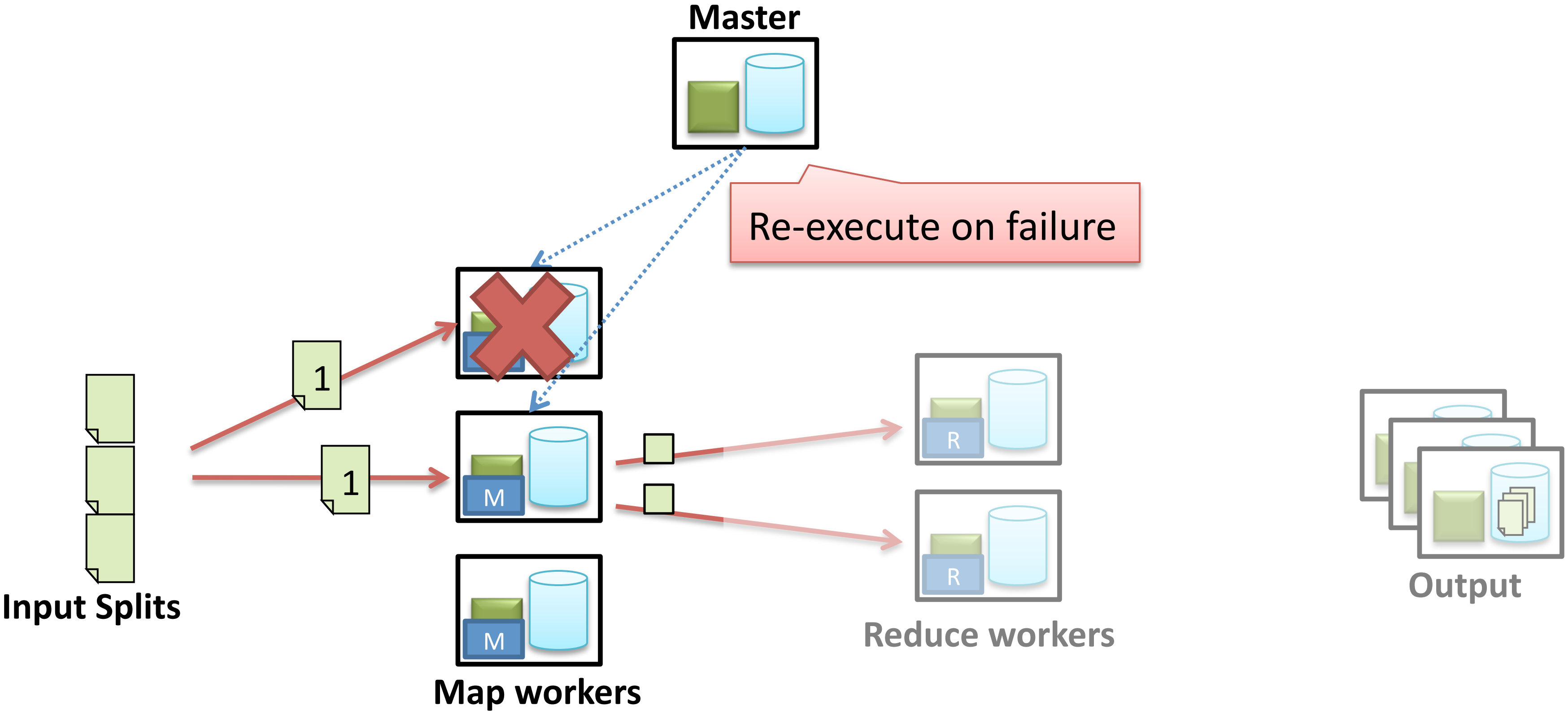
Task Assignment



Worker pull

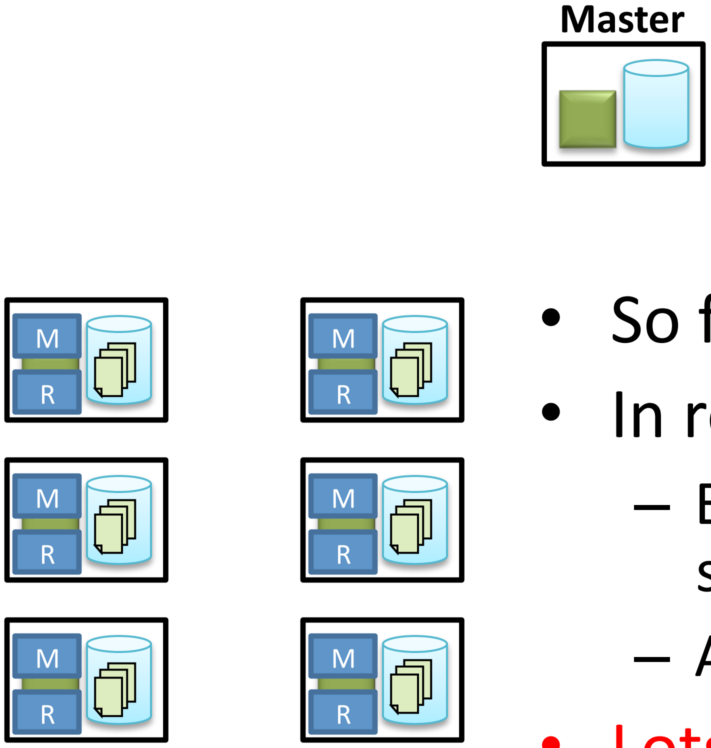
1. Worker signals idle
2. Master assigns task
3. Task retrieves data
4. Task executes

Fault-tolerance: re-execution



Re-execute on failure

Machines share roles



- So far, logical view of cluster
- In reality
  - Each cluster machine stores data
  - And runs MapReduce workers
- Lots of storage + compute cycles nearby

MapReduce Summary

- Programming paradigm for data-intensive computing
- Simple to program (for programmers)
- Distributed & parallel execution model
- The framework automates many tedious tasks (machine selection, failure handling, etc.)

Hadoop Demo

Beyond MapReduce

- As a programming model
  - Limited: only Map and Reduce
  - Improvements: Pig, Dryad, Hive, Sawzall, Map-Reduce-Merge, etc.
- As a runtime system
  - Better scheduling (e.g., LATE scheduler)
  - Better fault handling (e.g., ISS)
  - Pipelining (e.g., HOP)
  - Etc.

Making Cloud Intermediate Data Fault-Tolerant

*Steve Ko** (Princeton University),
Imranul Hoque (UIUC),
Brian Cho (UIUC),
and Indranil Gupta (UIUC)

* work done at UIUC

Our Position

- Intermediate data as a first-class citizen for dataflow programming frameworks in clouds

Our Position

- Intermediate data as a first-class citizen for dataflow programming frameworks in clouds
  - Dataflow programming frameworks

Our Position

- Intermediate data as a first-class citizen for dataflow programming frameworks in clouds
  - Dataflow programming frameworks
  - The importance of intermediate data

Our Position

- Intermediate data as a first-class citizen for dataflow programming frameworks in clouds
  - Dataflow programming frameworks
  - The importance of intermediate data
  - ISS (Intermediate Storage System)
    - Not to be confused with,

      International Space Station

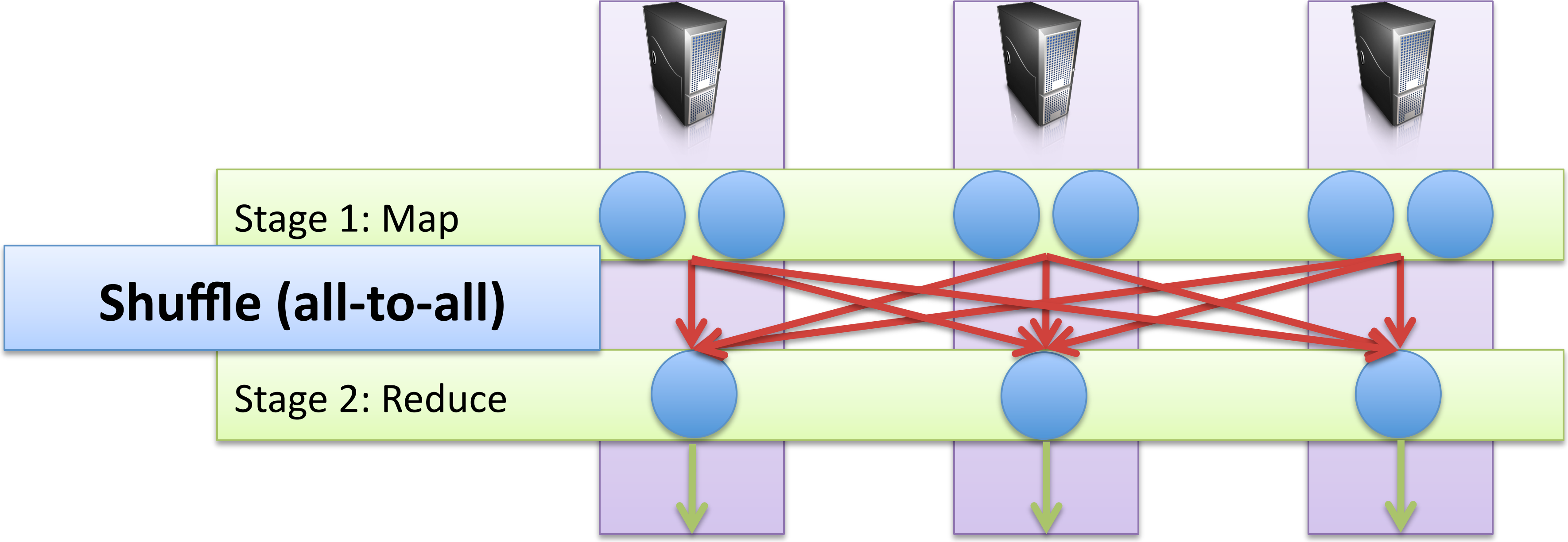
      IBM Internet Security Systems

Dataflow Programming Frameworks

- Runtime systems that execute dataflow programs
  - MapReduce (Hadoop), Pig, Hive, etc.
  - Gaining popularity for massive-scale data processing
  - Distributed and parallel execution on clusters
- A dataflow program consists of
  - Multi-stage computation
  - Communication patterns between stages

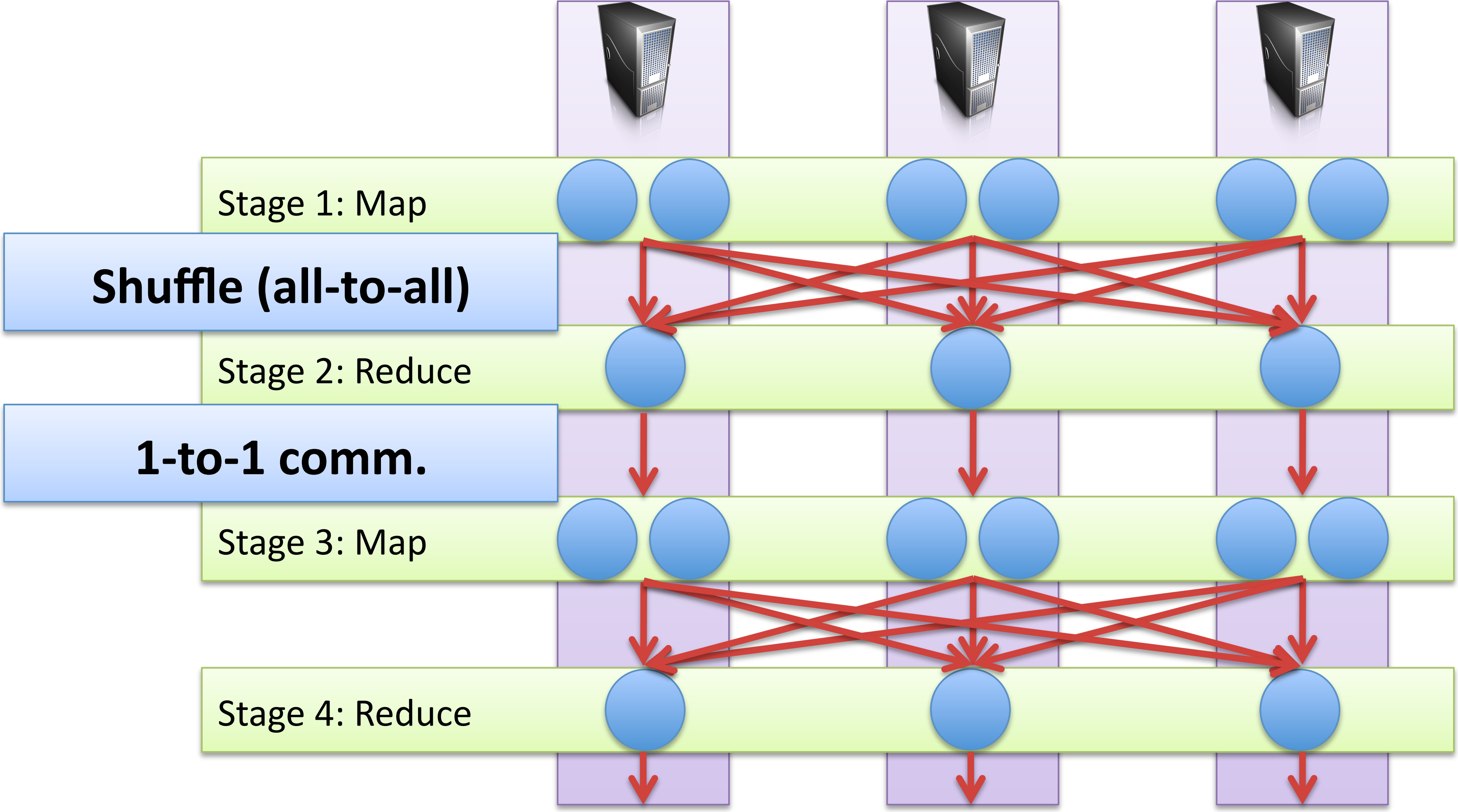
Example 1: MapReduce

- Two-stage computation with all-to-all comm.
  - Google introduced, Yahoo! open-sourced (Hadoop)
  - Two functions – Map and Reduce – supplied by a programmer
  - Massively parallel execution of Map and Reduce



Example 2: Pig and Hive

- Multi-stage with either all-to-all or 1-to-1



Usage

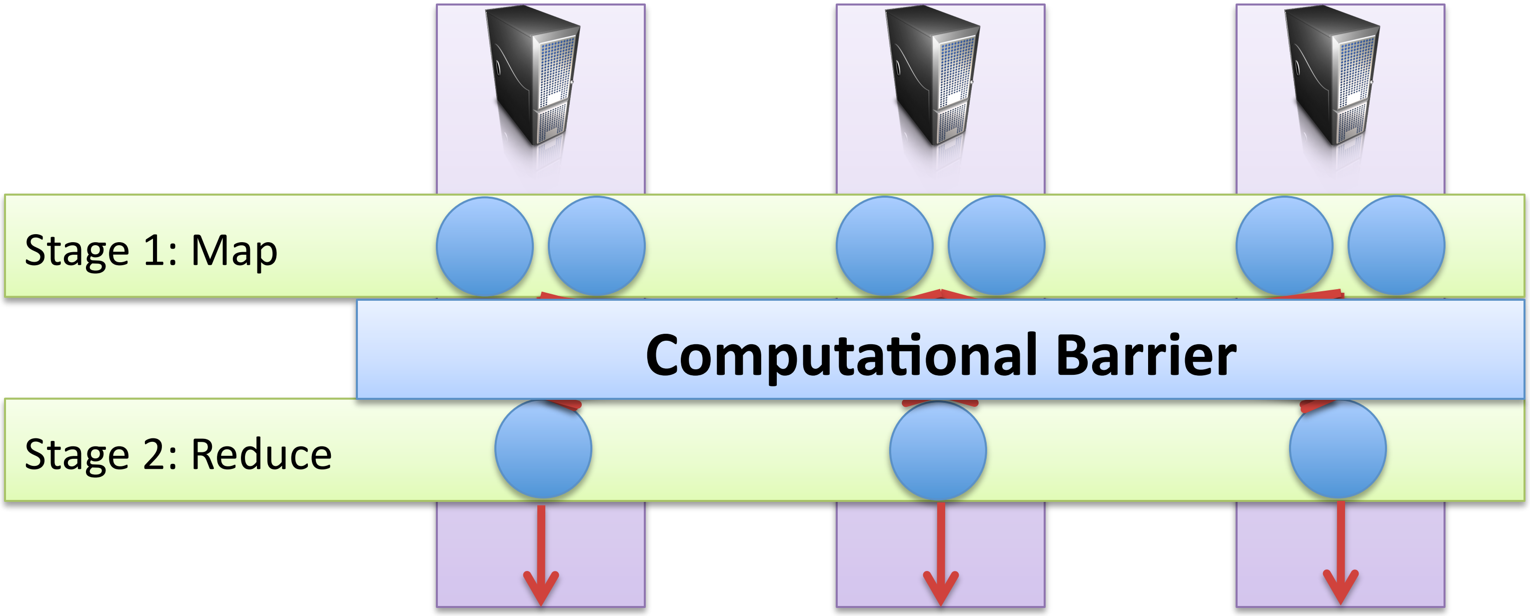
- Google (MapReduce)
  - Indexing: a chain of 24 MapReduce jobs
  - ~200K jobs processing 50PB/month (in 2006)
- Yahoo! (Hadoop + Pig)
  - WebMap: a chain of 100 MapReduce jobs
- Facebook (Hadoop + Hive)
  - ~300TB total, adding 2TB/day (in 2008)
  - 3K jobs processing 55TB/day
- Amazon
  - Elastic MapReduce service (pay-as-you-go)
- Academic clouds
  - Google-IBM Cluster at UW (Hadoop service)
  - CCT at UIUC (Hadoop & Pig service)

One Common Characteristic

- Intermediate data
  - Intermediate data? data between stages
- Similarities to traditional intermediate data [Bak91, Vog99]
  - E.g., .o files
  - Critical to produce the final output
  - Short-lived, written-once and read-once, & used-immediately
  - Computational barrier

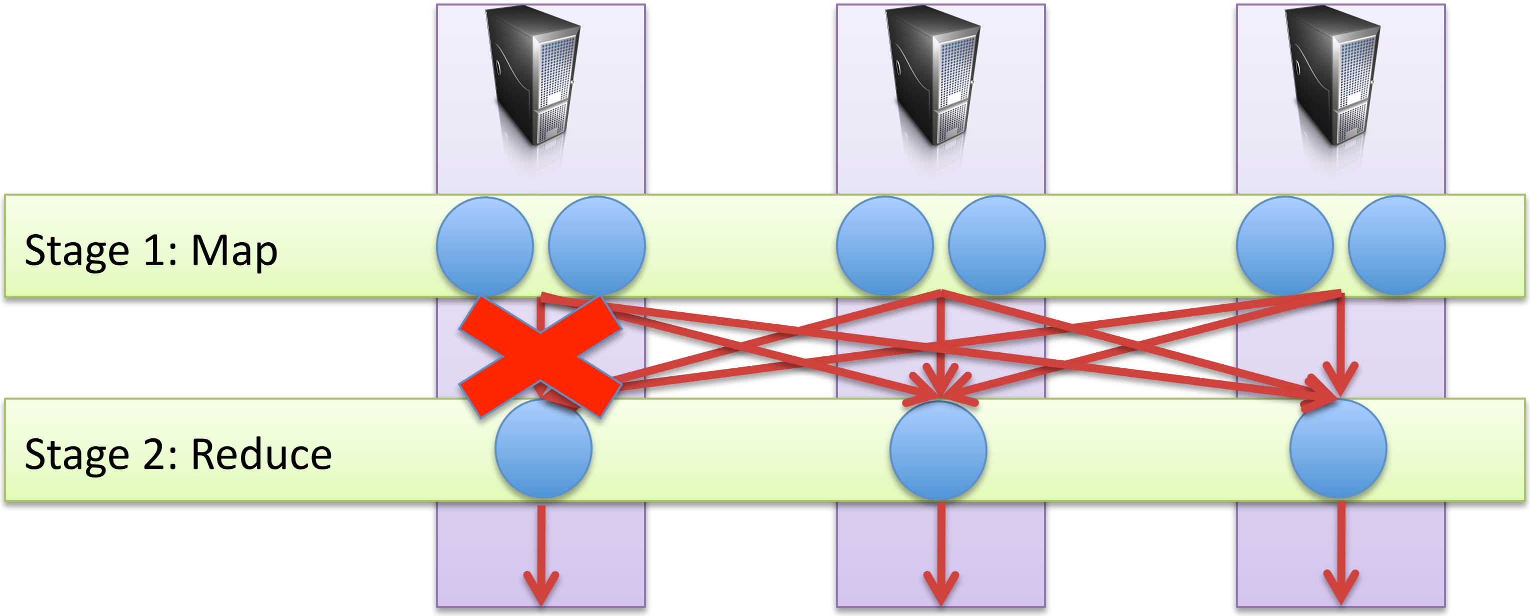
One Common Characteristic

- Computational Barrier



Why Important?

- Large-scale: possibly very large amount of intermediate data
- Barrier: Loss of intermediate data
  => the task can't proceed

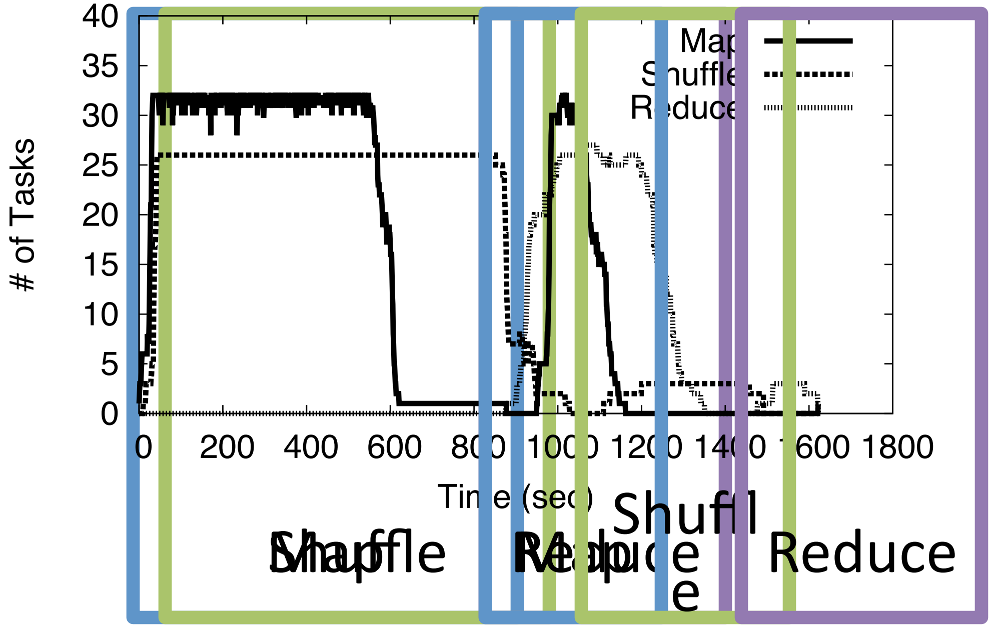


Failure Stats

- 5 average worker deaths per MapReduce job (Google in 2006)
- One disk failure in every run of a 6-hour MapReduce job with 4000 machines (Google in 2008)
- 50 machine failures out of 20K machine cluster (Yahoo! in 2009)

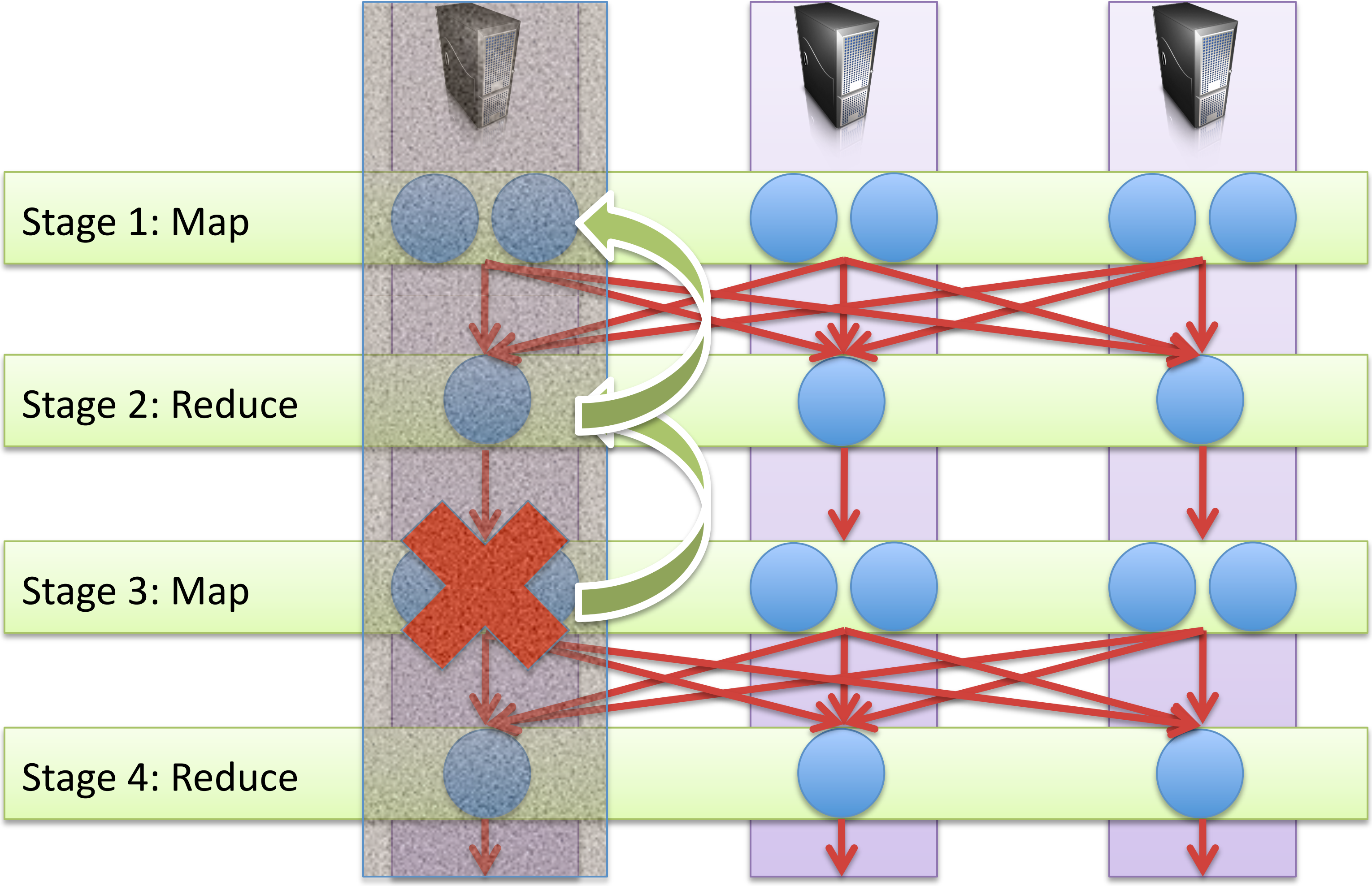
Hadoop Failure Injection Experiment

- Emulab setting
  - 20 machines sorting 36GB
  - 4 LANs and a core switch (all 100 Mbps)
- 1 failure after Map
  - Re-execution of Map-Shuffle-Reduce
- ~33% increase in completion time



Re-Generation for Multi-Stage

- Cascaded re-execution: expensive



Importance of Intermediate Data

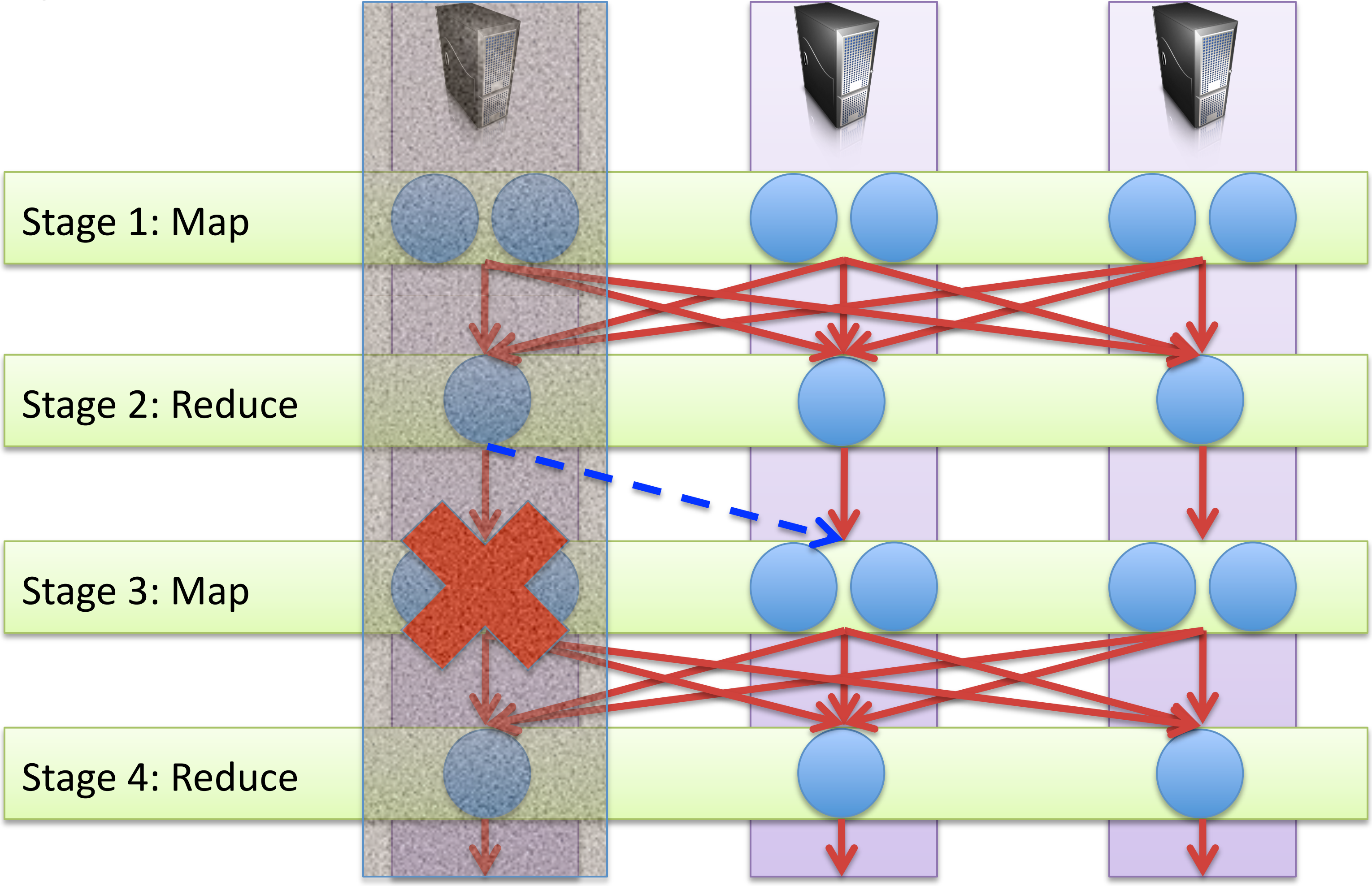
- Why?
  - (Potentially) a lot of data
  - When lost, very costly
- Current systems handle it themselves.
  - Re-generate when lost: can lead to expensive cascaded re-execution
- We believe that the storage can provide a better solution than the dataflow programming frameworks

Our Position

- Intermediate data as a first-class citizen for dataflow programming frameworks in clouds
  - ✓ Dataflow programming frameworks
  - ✓ The importance of intermediate data
  - ISS (Intermediate Storage System)
    - Why storage?
    - Challenges
    - Solution hypotheses
    - Hypotheses validation

Why Storage?

- Replication stops cascaded re-execution

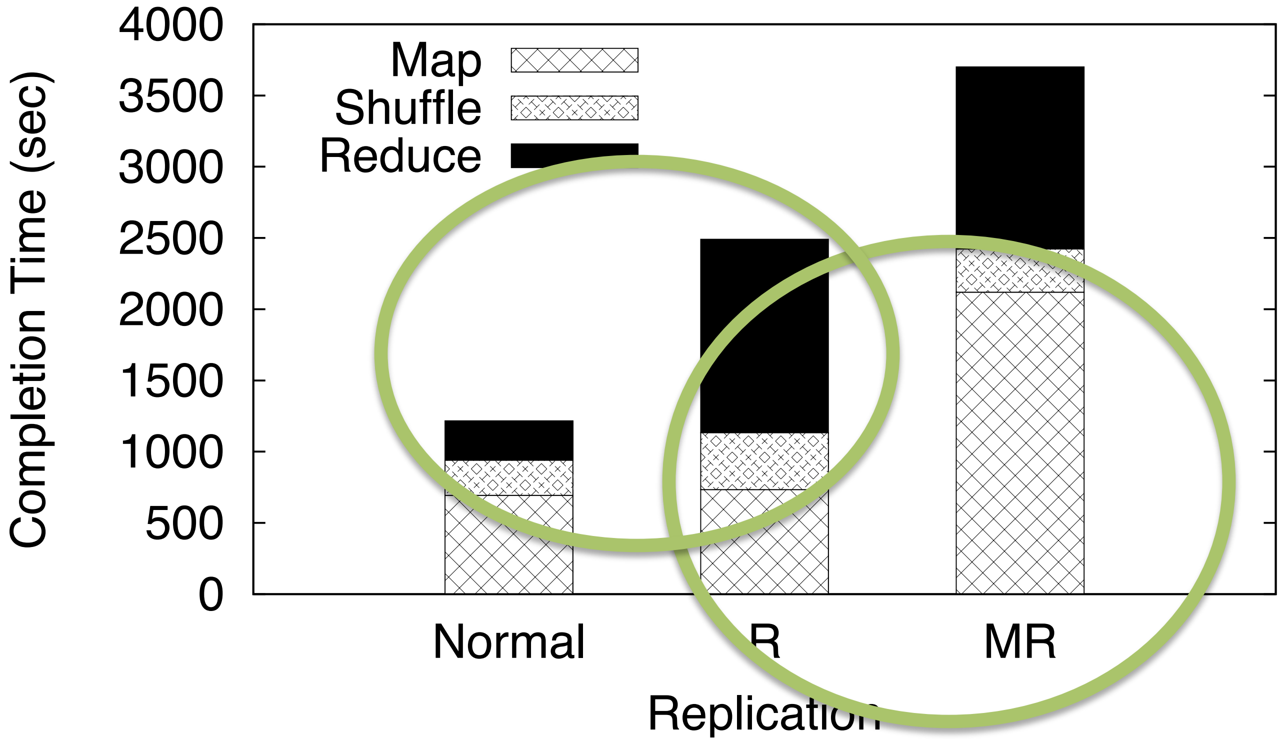


So, Are We Done?

- No!
- Challenge: minimal interference
  - Network is heavily utilized during Shuffle.
  - Replication requires network transmission too, and needs to replicate a large amount.
  - Minimizing interference is critical for the overall job completion time.
- HDFS (Hadoop Distributed File System): much interference

Default HDFS Interference

- Replication of Map and Reduce outputs (2 copies in total)



Background Transport Protocols

- TCP-Nice [Ven02] & TCP-LP [Kuz06]
  - Support background & foreground flows
- Pros
  - Background flows do not interfere with foreground flows (functionality)
- Cons
  - Designed for wide-area Internet
  - Application-agnostic
  - Not a comprehensive solution: not designed for data center replication
- Can do better!

Our Position

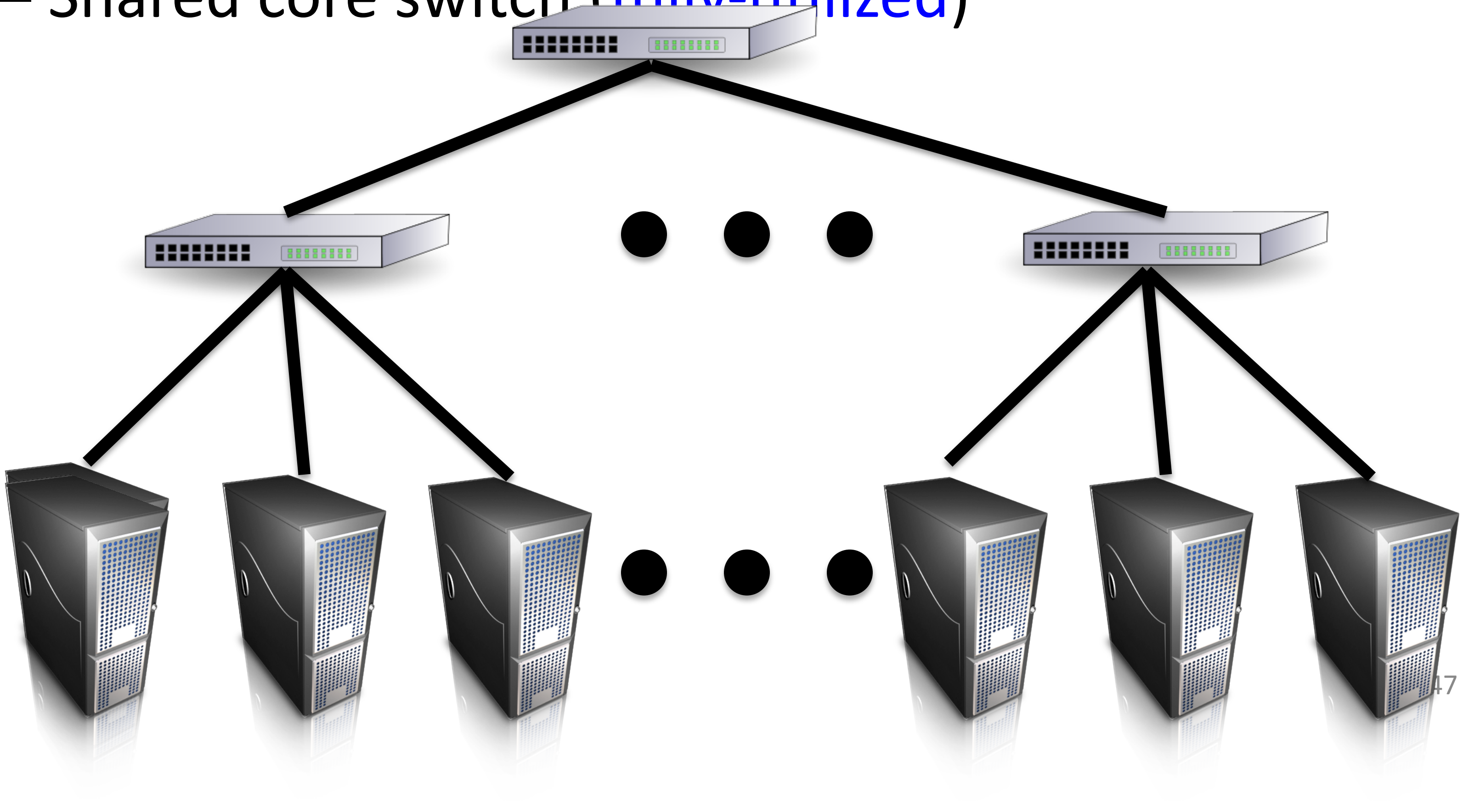
- Intermediate data as a first-class citizen for dataflow programming frameworks in clouds
  - ✓ Dataflow programming frameworks
  - ✓ The importance of intermediate data
  - ISS (Intermediate Storage System)
    - ✓ Why storage?
    - ✓ Challenges
    - Solution hypotheses
    - Hypotheses validation

Three Hypotheses

1. Asynchronous replication can help.
   - HDFS replication works synchronously.
2. The replication process can exploit the inherent bandwidth heterogeneity of data centers (next).
3. Data selection can help (later).

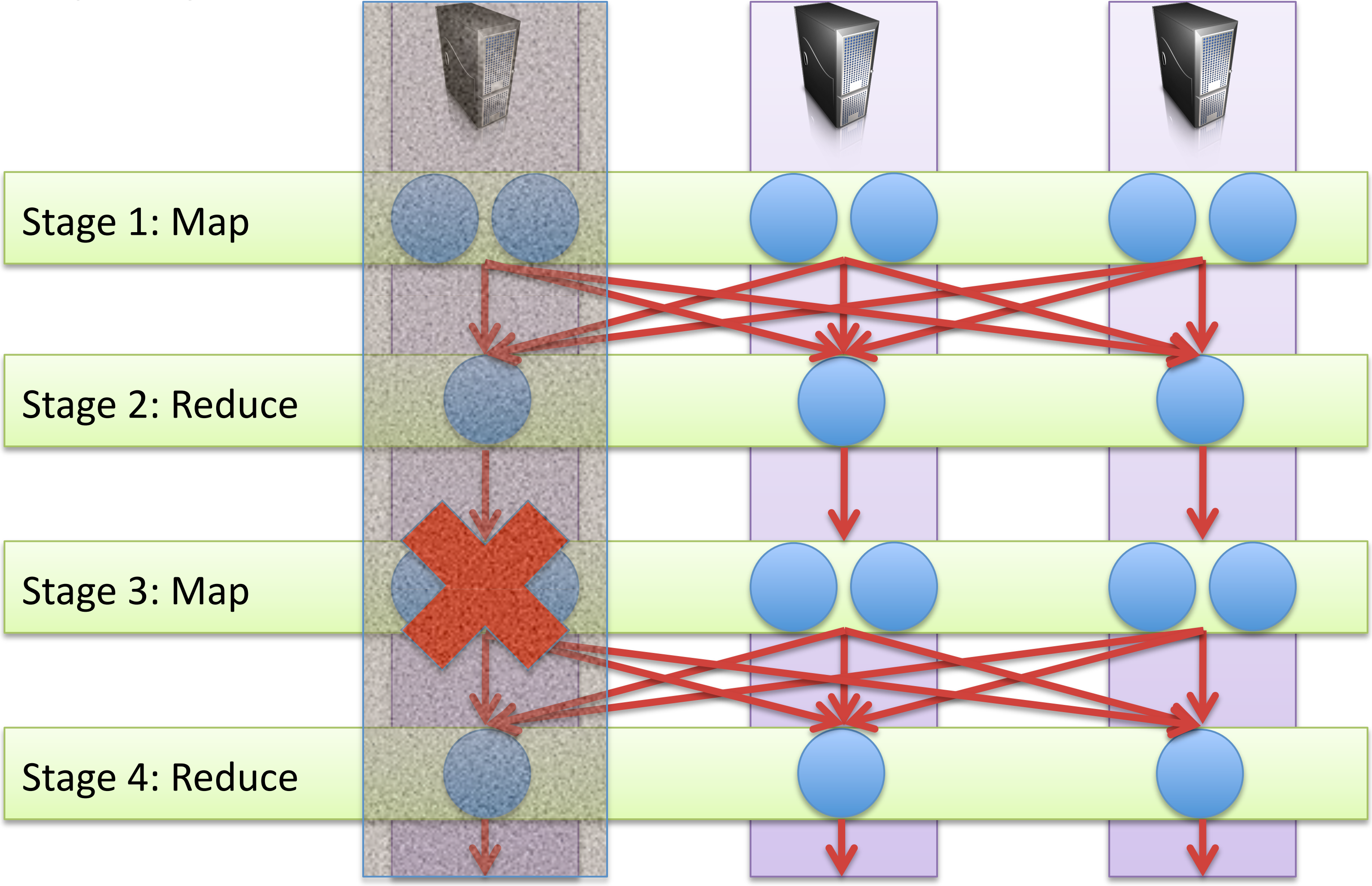
Bandwidth Heterogeneity

- Data center topology: hierarchical
  - Top-of-the-rack switches (under-utilized)
  - Shared core switch (fully-utilized)



Data Selection

- Only replicate locally-consumed data



Three Hypotheses

1. Asynchronous replication can help.
2. The replication process can exploit the inherent bandwidth heterogeneity of data centers.
3. Data selection can help.

- The question is not if, but how much.
- If effective, these become techniques.

Experimental Setting

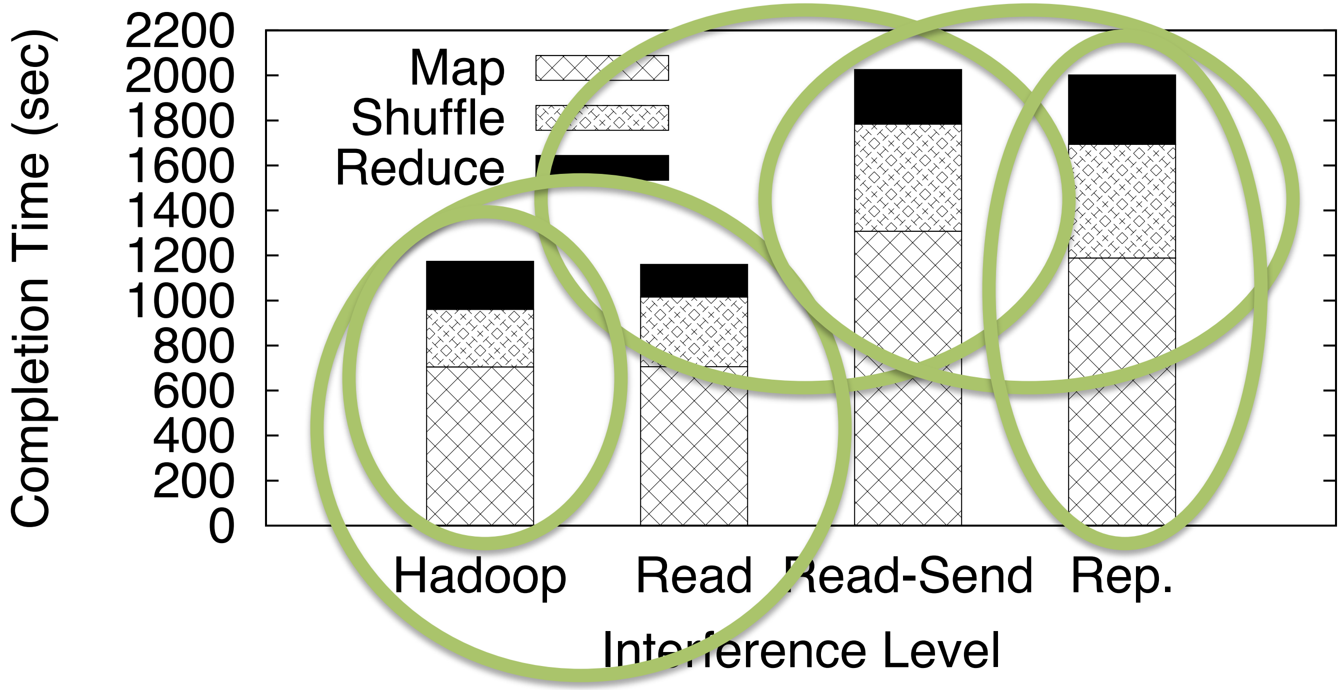
- Emulab with 80 machines
  - 4 X 1 LAN with 20 machines
  - 4 X 100Mbps top-of-the-rack switch
  - 1 X 1Gbps core switch
  - Various configurations give similar results.
- Input data: 2GB/machine, random-generation
- Workload: sort
- 5 runs
  - Std. dev. ~ 100 sec.: small compared to the overall completion time
- 2 replicas of Map outputs in total

Asynchronous Replication

- Modification for asynchronous replication
  - With an increasing level of interference
- Four levels of interference
  - Hadoop: original, no replication, no interference
  - Read: disk read, no network transfer, no actual replication
  - Read-Send: disk read & network send, no actual replication
  - Rep.: full replication

Asynchronous Replication

- Network utilization makes the difference
- Both Map & Shuffle get affected
  - Some Maps need to read remotely

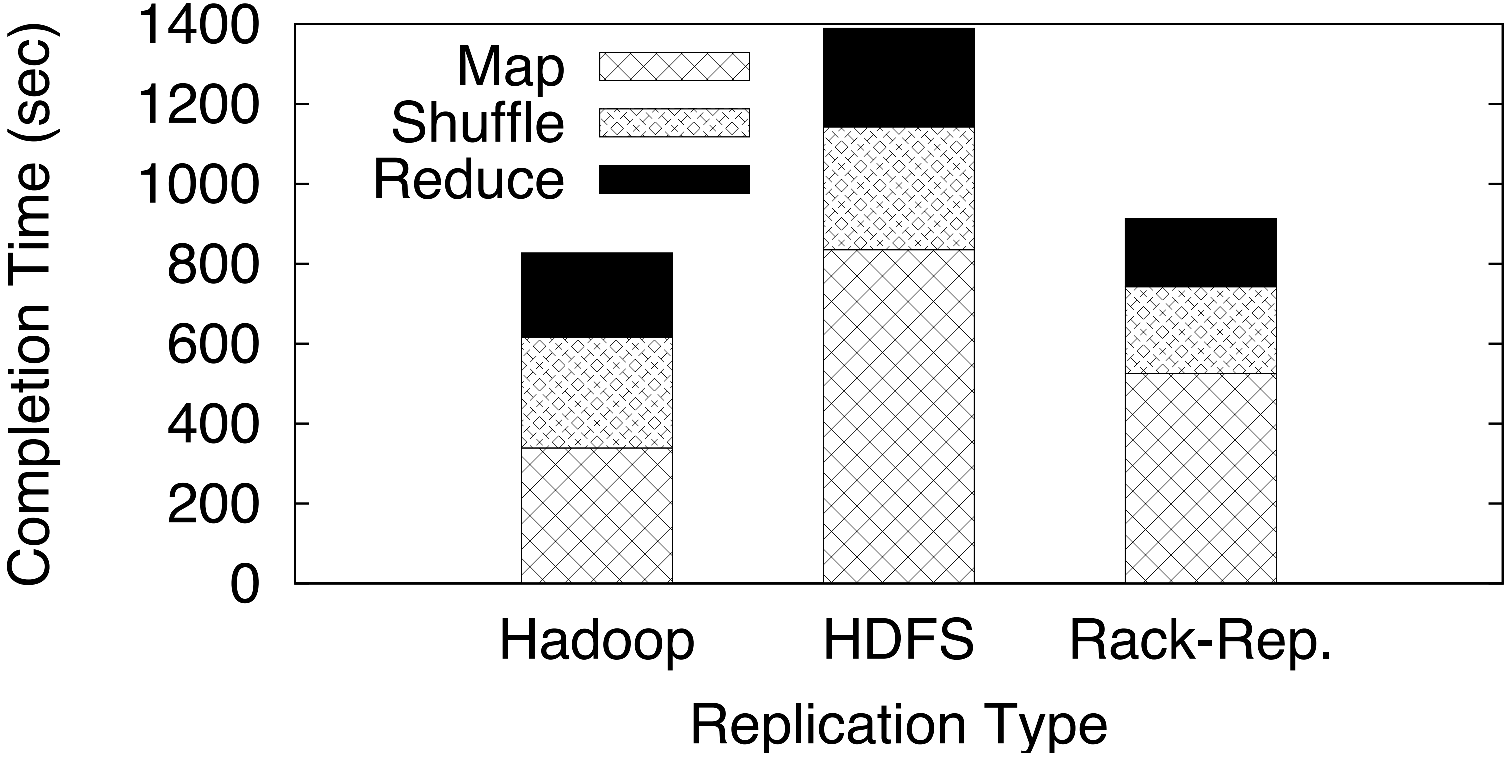


Three Hypotheses (Validation)

- ✓ Asynchronous replication can help, but still can’t eliminate the interference.
- The replication process can exploit the inherent bandwidth heterogeneity of data centers.
- Data selection can help.

Rack-Level Replication

- Rack-level replication is effective.
  - Only 20~30 rack failures per year, mostly planned (Google 2008)

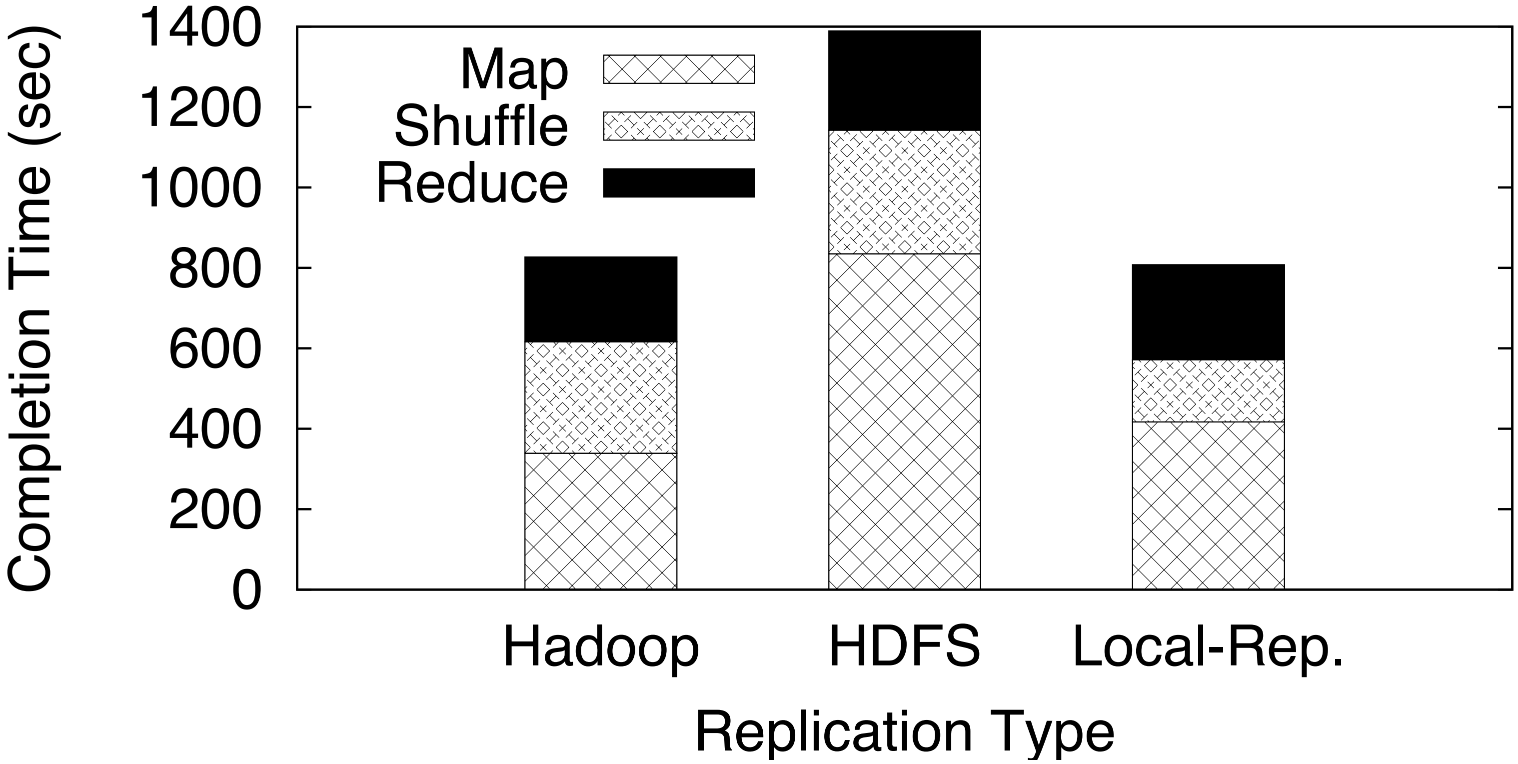


Three Hypotheses (Validation)

- ✓ Asynchronous replication can help, but still can’t eliminate the interference
- ✓ The rack-level replication can reduce the interference significantly.
- Data selection can help.

Locally-Consumed Data Replication

- It significantly reduces the amount of replication.



Three Hypotheses (Validation)

- ✓ Asynchronous replication can help, but still can’t eliminate the interference
- ✓ The rack-level replication can reduce the interference significantly.
- ✓ Data selection can reduce the interference significantly.

ISS Design Overview

- Implements asynchronous rack-level selective replication (all three hypotheses)
- Replaces the Shuffle phase
  - MapReduce does not implement Shuffle.
  - Map tasks write intermediate data to ISS, and Reduce tasks read intermediate data from ISS.
- Extends HDFS (next)

ISS Design Overview

- Extends HDFS
  - iss_create()
  - iss_open()
  - iss_write()
  - iss_read()
  - iss_close()
- Map tasks
  - iss_create() => iss_write() => iss_close()
- Reduce tasks
  - iss_open() => iss_read() => iss_close()

Performance under Failure

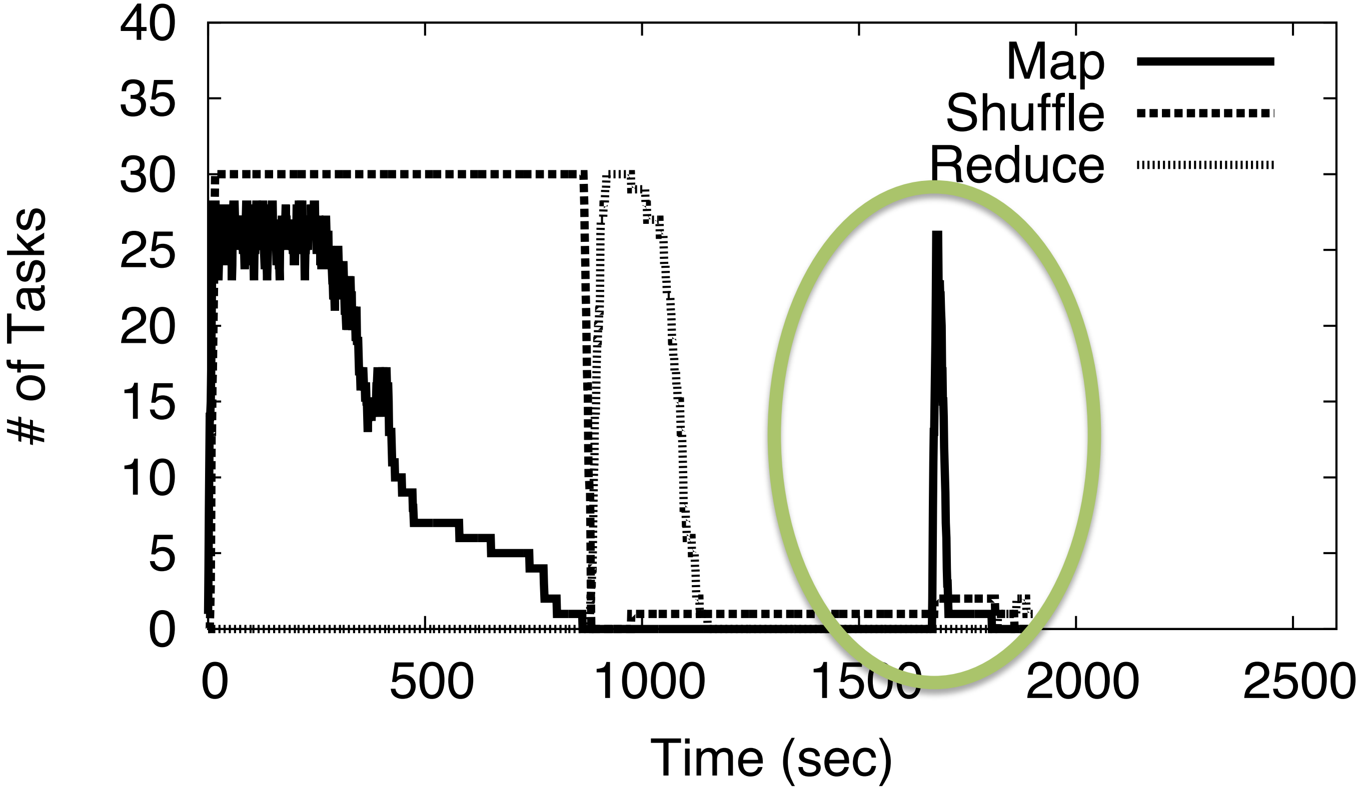
- 5 scenarios
  - Hadoop (no rep) with one permanent machine failure
  - Hadoop (reduce rep=2) with one permanent machine failure
  - ISS (map & reduce rep=2) with one permanent machine failure
  - Hadoop (no rep) with one transient failure
  - ISS (map & reduce rep=2) with one transient failure

Summary of Results

- Comparison to no failure Hadoop
  - One failure ISS: 18% increase in completion time
  - One failure Hadoop: 59% increase
- One failure Hadoop vs. One failure ISS
  - 45% speedup

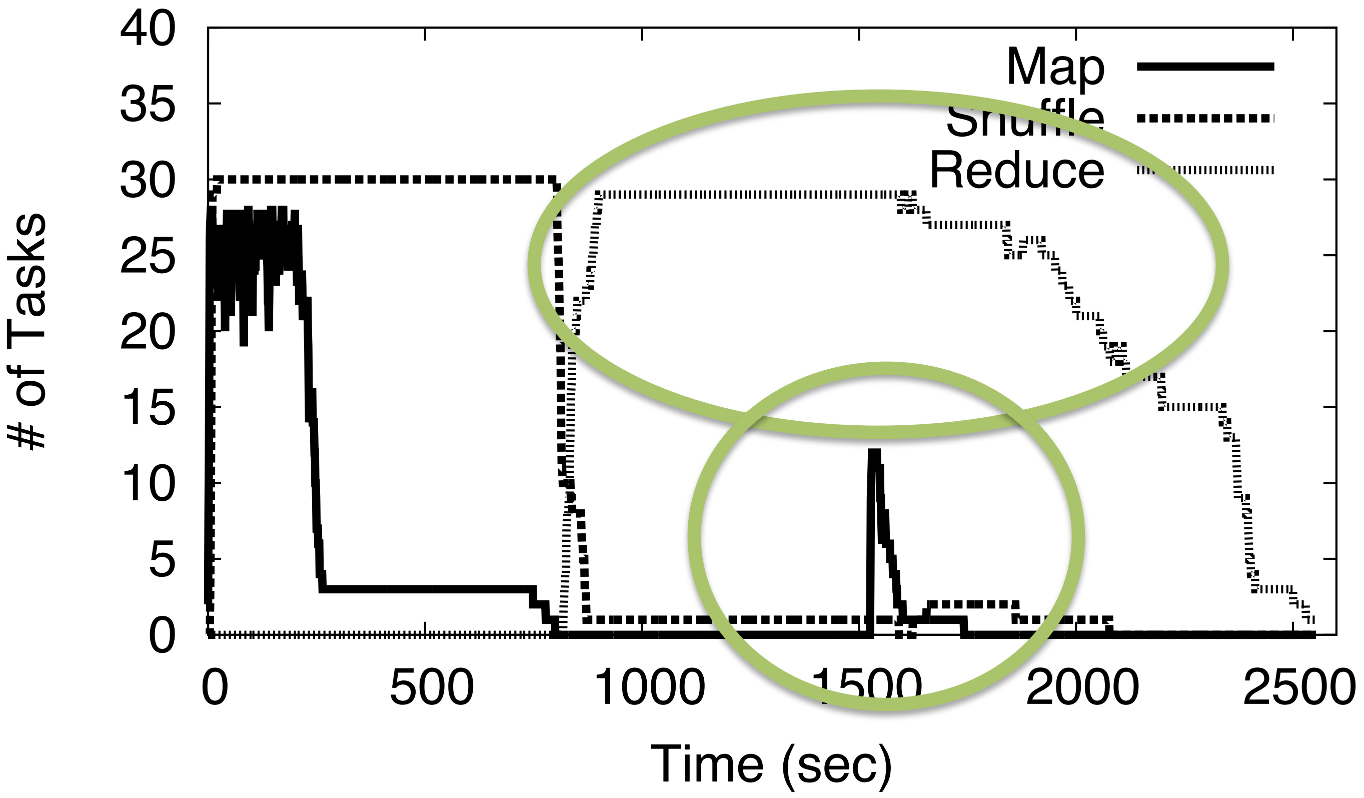
Performance under Failure

- Hadoop (rep=1) with one machine failure



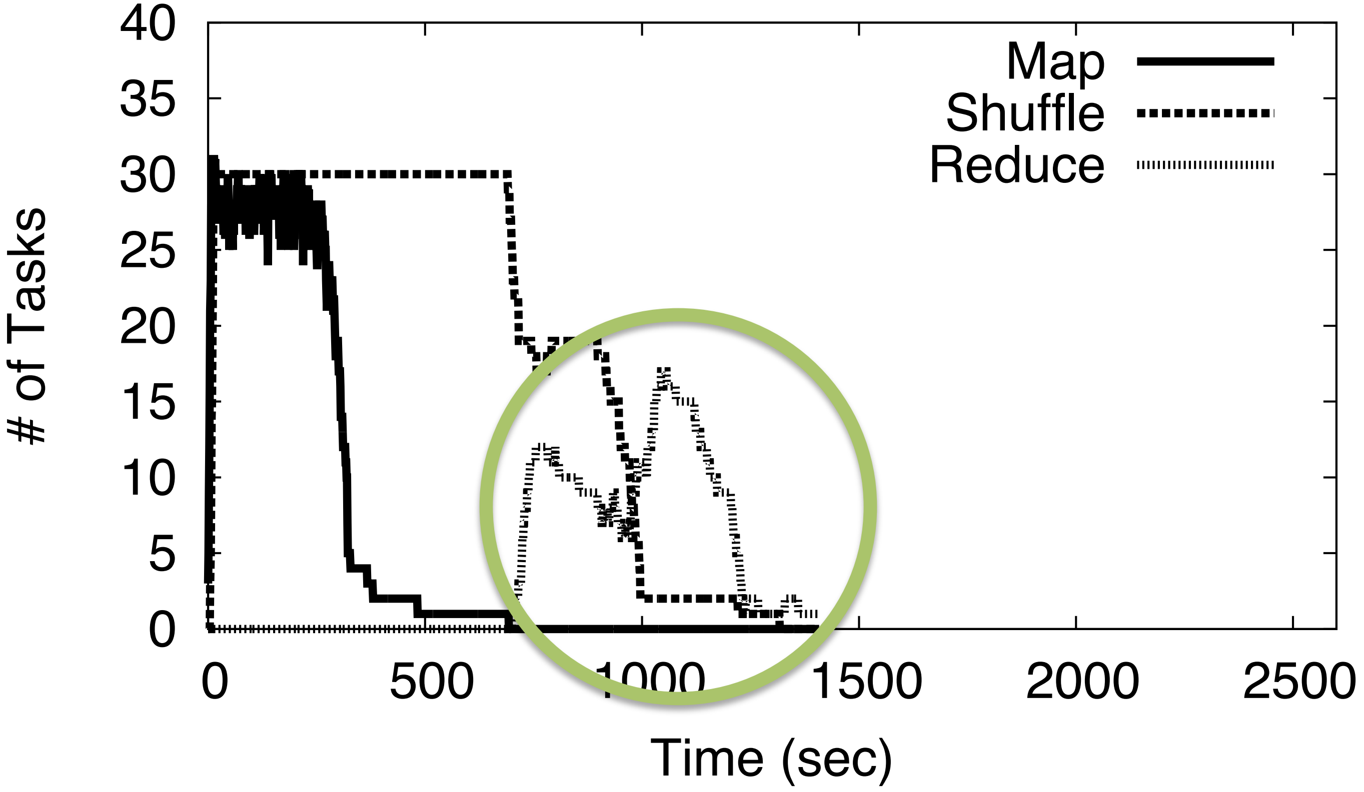
Performance under Failure

- Hadoop (rep=2) with one machine failure



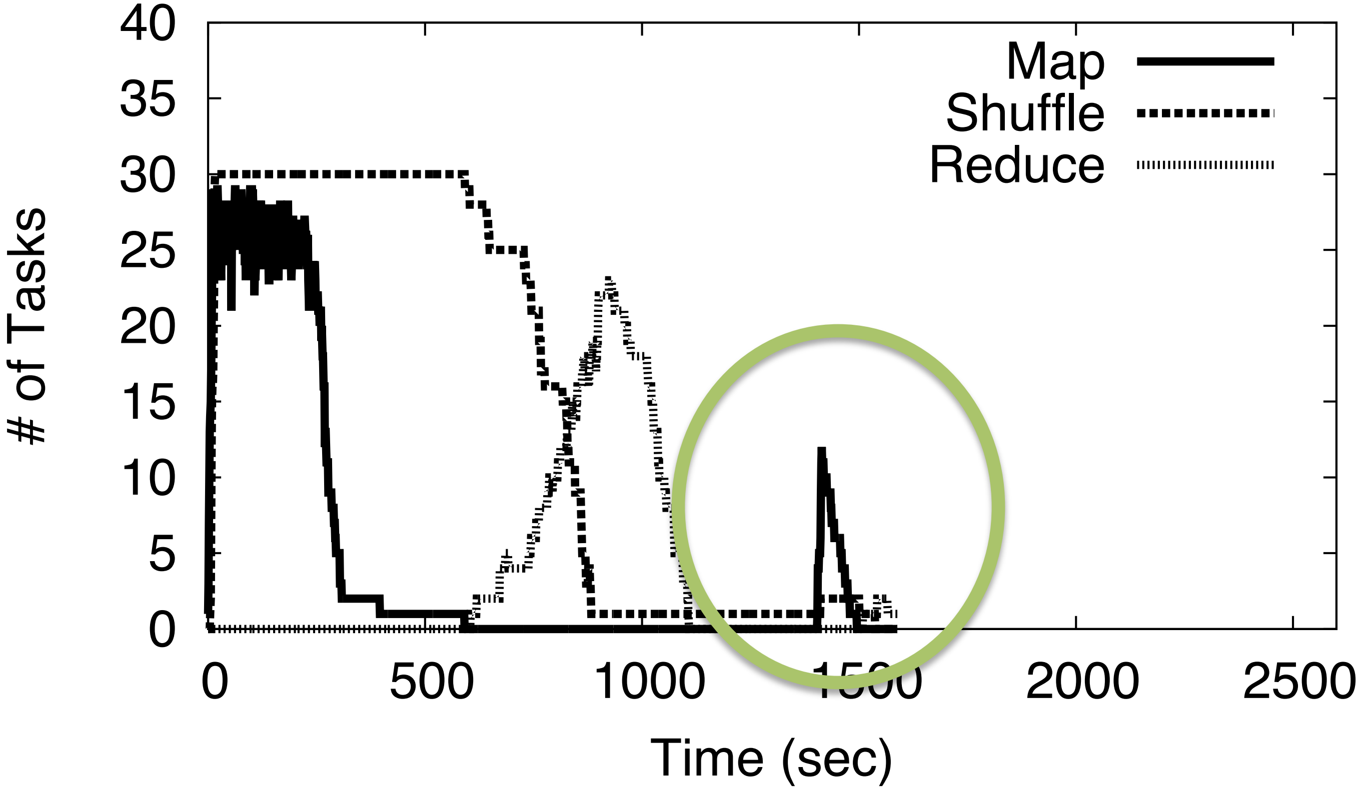
Performance under Failure

- ISS with one machine failure



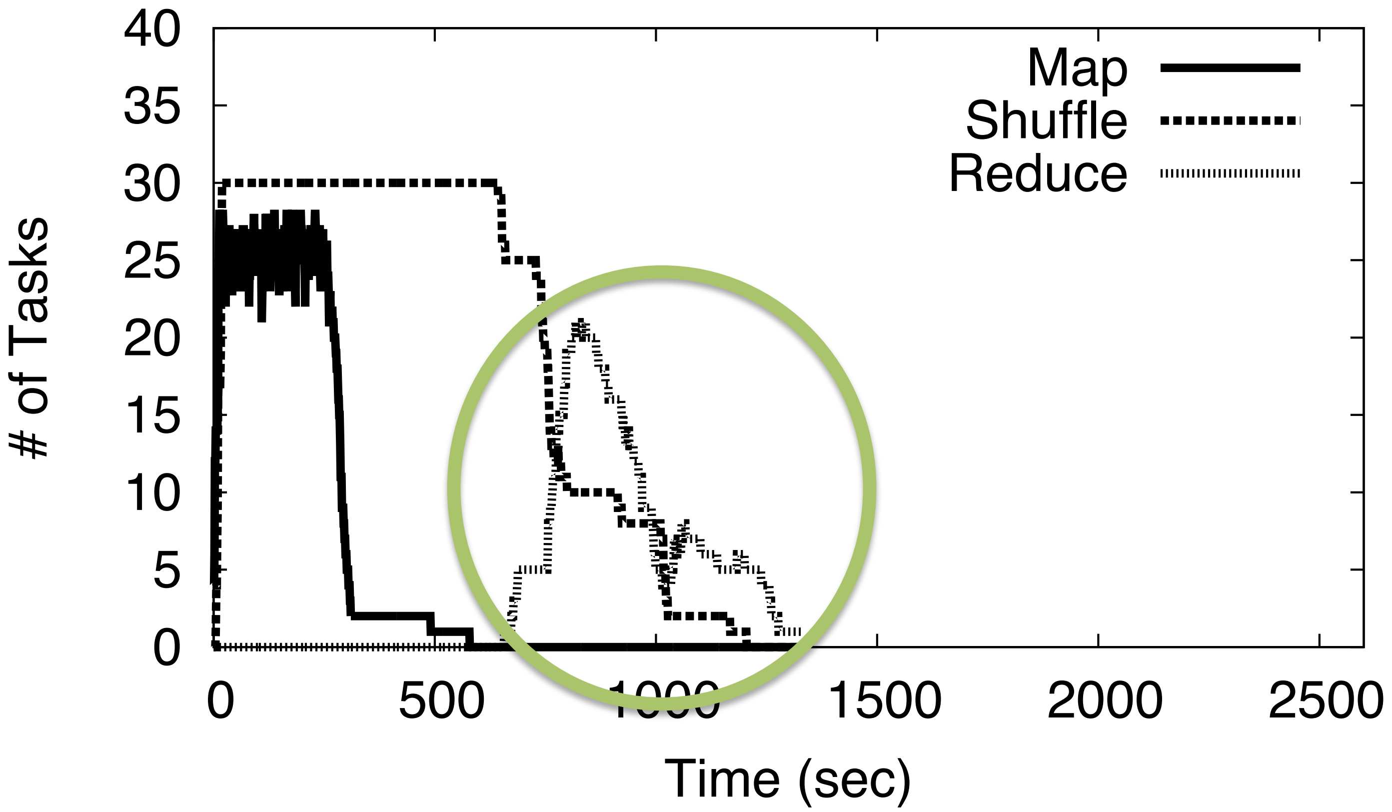
Performance under Failure

- Hadoop (rep=1) with one transient failure



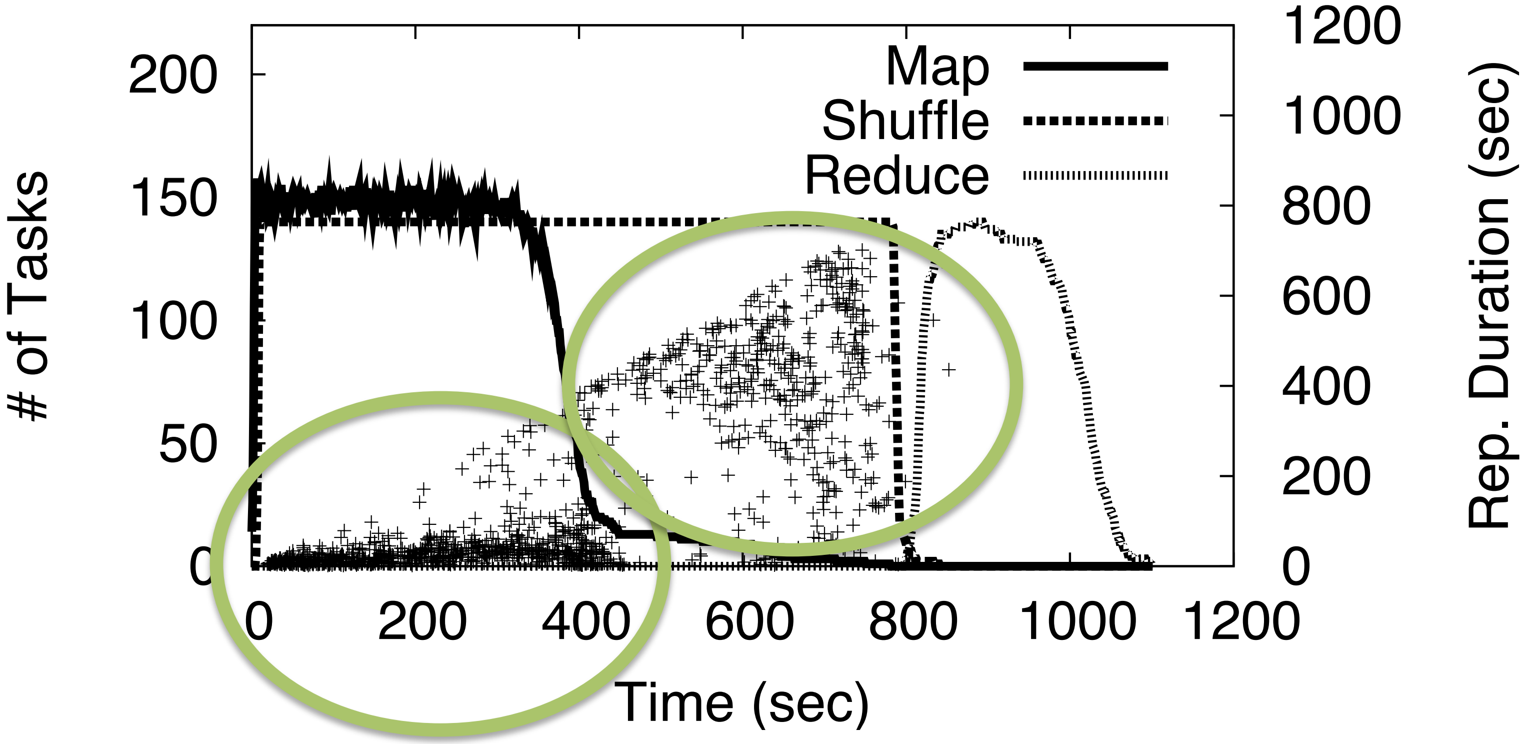
Performance under Failure

- ISS-Hadoop with one transient failure



Replication Completion Time

- Replication completes before Reduce
  - ‘+’ indicates replication time for each block



Summary

- Our position
  - Intermediate data as a first-class citizen for dataflow programming frameworks in clouds
- Problem: cascaded re-execution
- Requirements
  - Intermediate data availability (scale and dynamism)
  - Interference minimization (efficiency)
- Asynchronous replication can help, but still can't eliminate the interference
- The rack-level replication can reduce the interference significantly.
- Data selection can reduce the interference significantly.
- Hadoop & Hadoop + ISS show comparable completion times.

References

- [Vog99] W. Vogels. File System Usage in Windows NT 4.0. In SOSP, 1999.
- [Bak91] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. SIGOPS OSR, 25(5), 1991.
- [Ven02] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: A Mechanism for Background Transfers. In OSDI, 2002.
- [Kuz06] A. Kuzmanovic and E. W. Knightly. TCP-LP: Low-Priority Service via End-Point Congestion Control. IEEE/ACM TON, 14(4): 739–752, 2006.