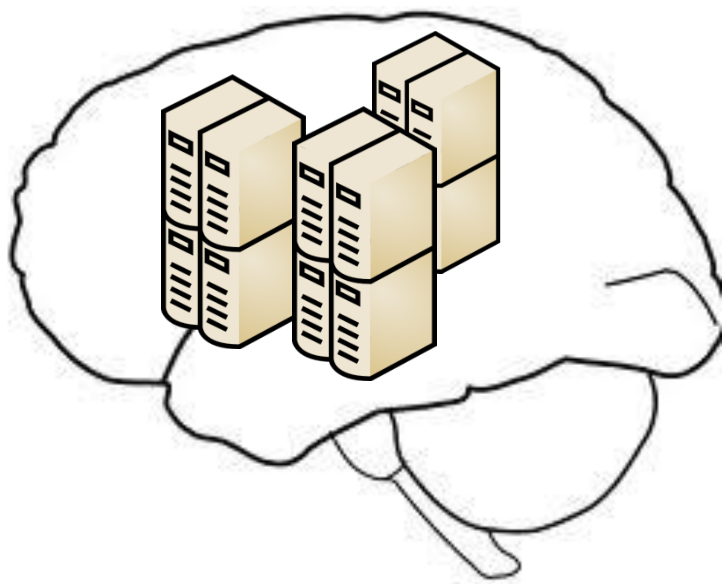


In-Memory Clusters

Mainak Ghosh and Hilfi Alkaff

Photo Courtesy: http://www.clker.com/

PACMan: Coordinated memory caching for parallel jobs

Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, Ion Stoica

Presenter: Mainak Ghosh

Content of this presentation is borrowed heavily from the original author's paper and presentation: https://www.usenix.org/system/files/conference/nsdi12/pacman.pdf

Paper In A Slide

Problem: Data intensive jobs in large clusters have large execution times.

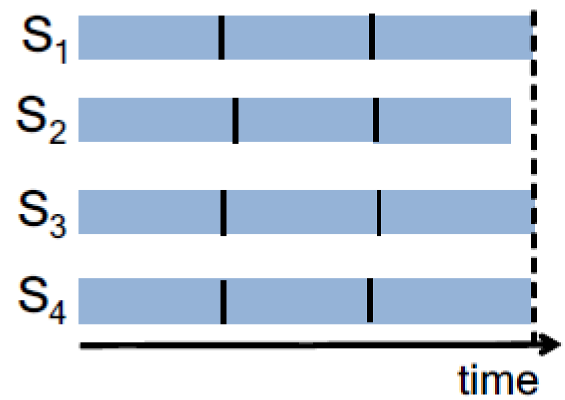
Key Observation:

- Jobs comprise of IO-intensive execution phases which run in parallel.
- Clusters have machines with large memory which are underutilized.

Strategy: In-memory caching of input data.

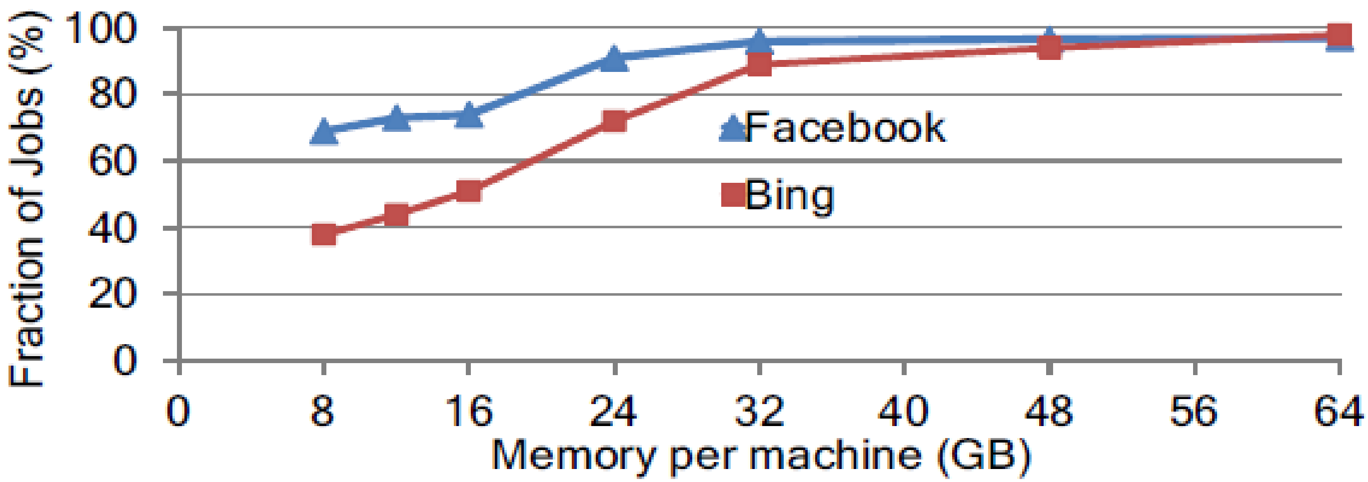
Terminology

- Task
- Wave
- Single Wave Job
- Multi Wave Job
- Completion Time
- Cluster Efficiency



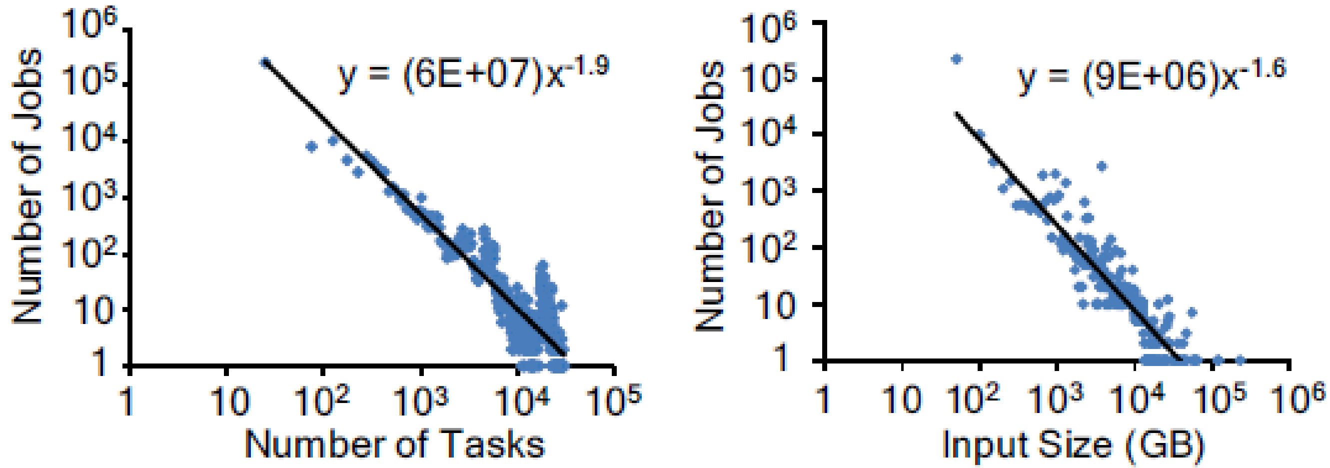
Goal: Reduce completion time and increase cluster efficiency

Industry Speaks…



Large fraction of jobs fits the memory

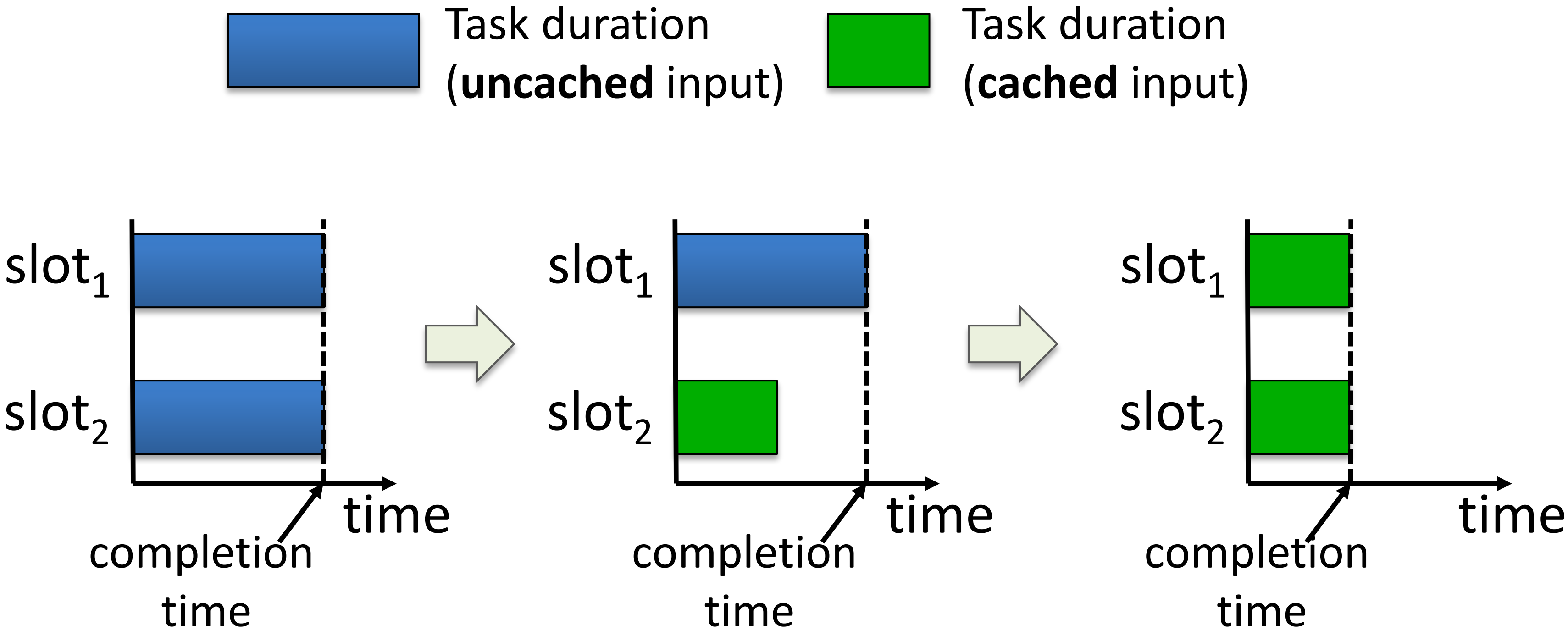
Industry Speaks…



(a) Number of tasks (b) Input Size

Large number of jobs have small number of task size and input file size

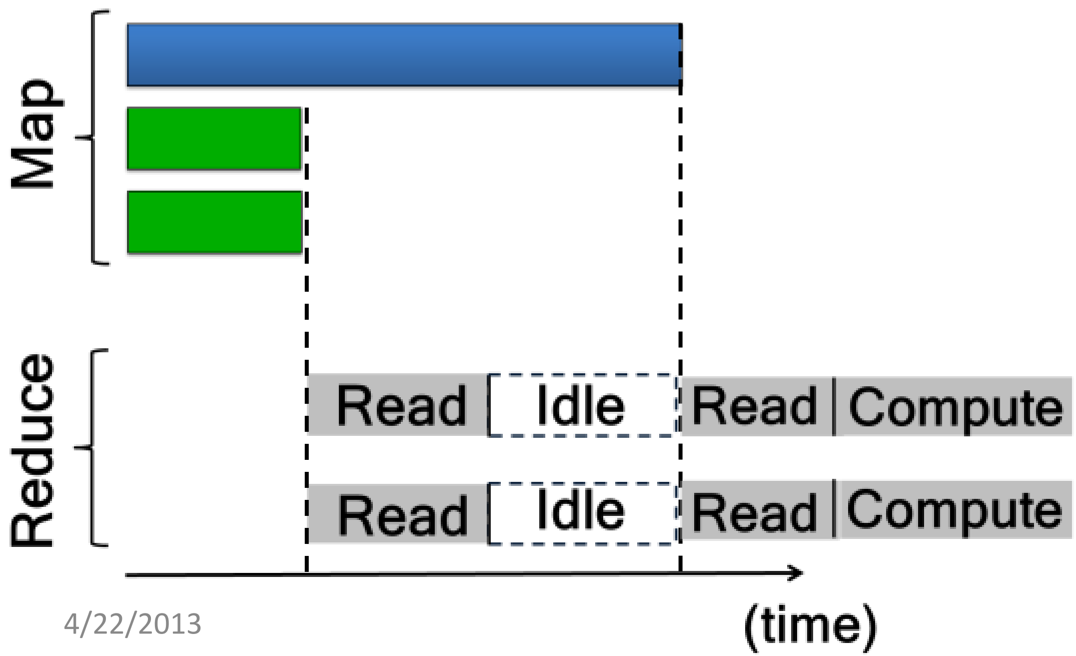
Is it enough?

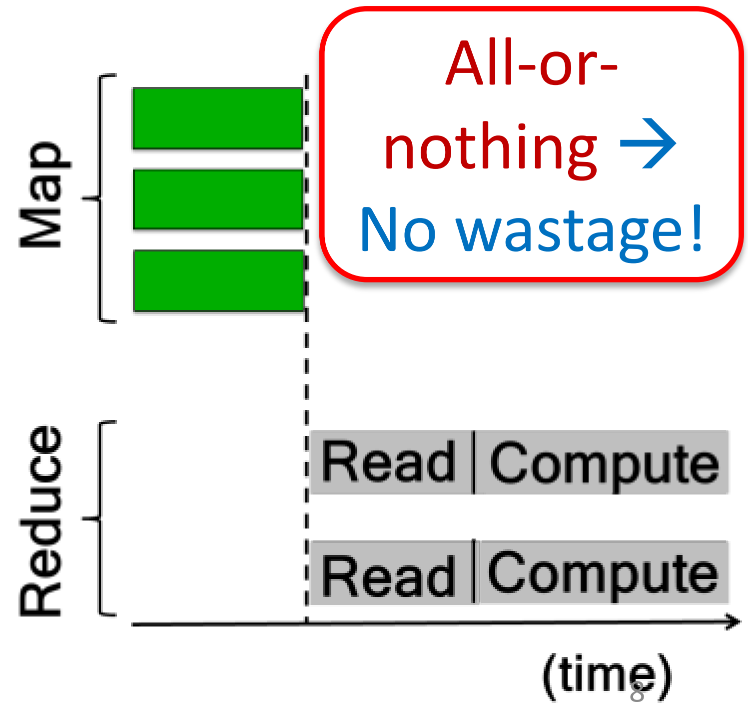


All-or-nothing: Unless all inputs are cached, there is no benefit

Cluster Efficiency?

- All-or-nothing property matters for utilization
- Tasks of different phases overlap
  - Reduce tasks start before all map tasks finish (to overlap communication)

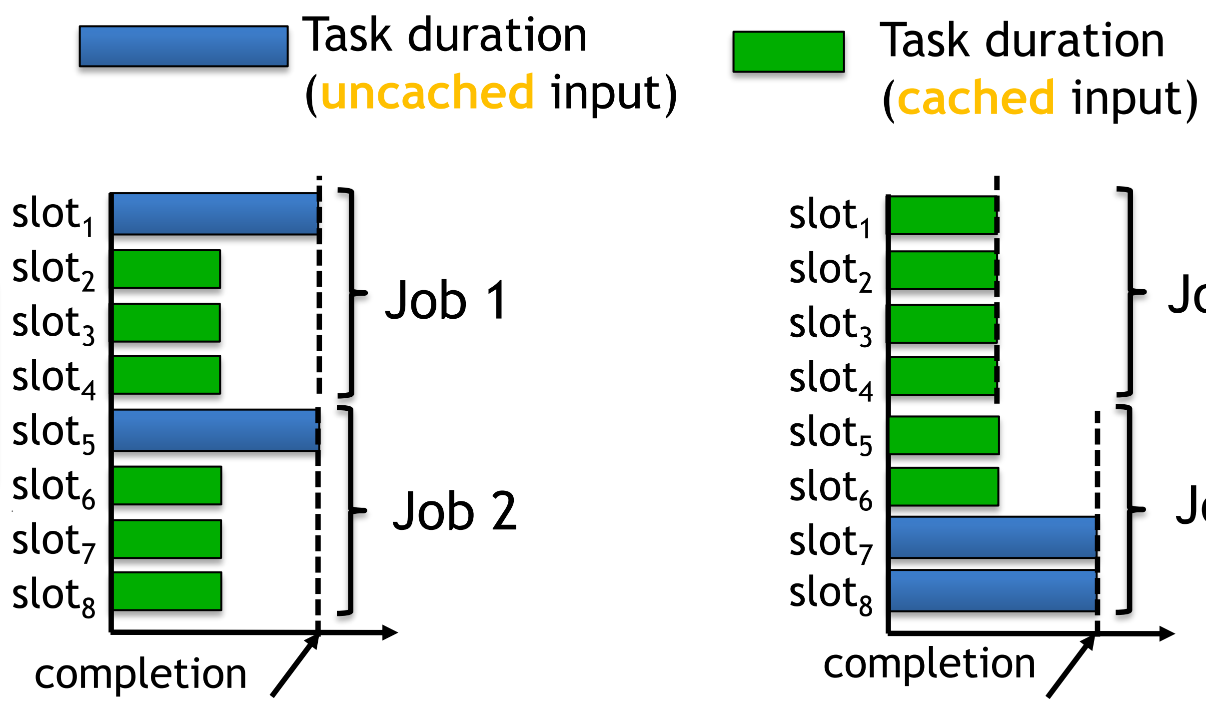




Cache Replacement Policy

- View at the granularity of a job's input (*file)*
- Focus evictions on incompletely cached waves– **Sticky Policy**

Without Sticky Policy



With Sticky Policy

Hit-ratio: 75%
No speed-up of jobs

Hit-ratio: 75%
Job 1 speeds up

Reduction in Completion Time

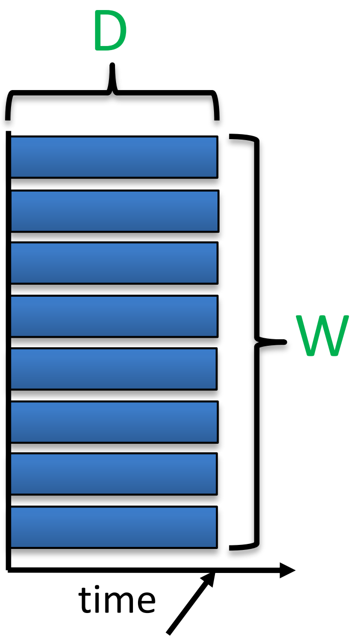
- Idealized model for job:
  - Wave-width for job: W
  - Frequency predicts future access: F
  - Data read is proportional to task length: D
  - Speedup factor for cached tasks: µ

- Cost of caching: W D
- Benefit of caching: µD F
- Benefit/cost: µF/W

LIFE: Favor Jobs with lesser wave width

Improvement in Utilization

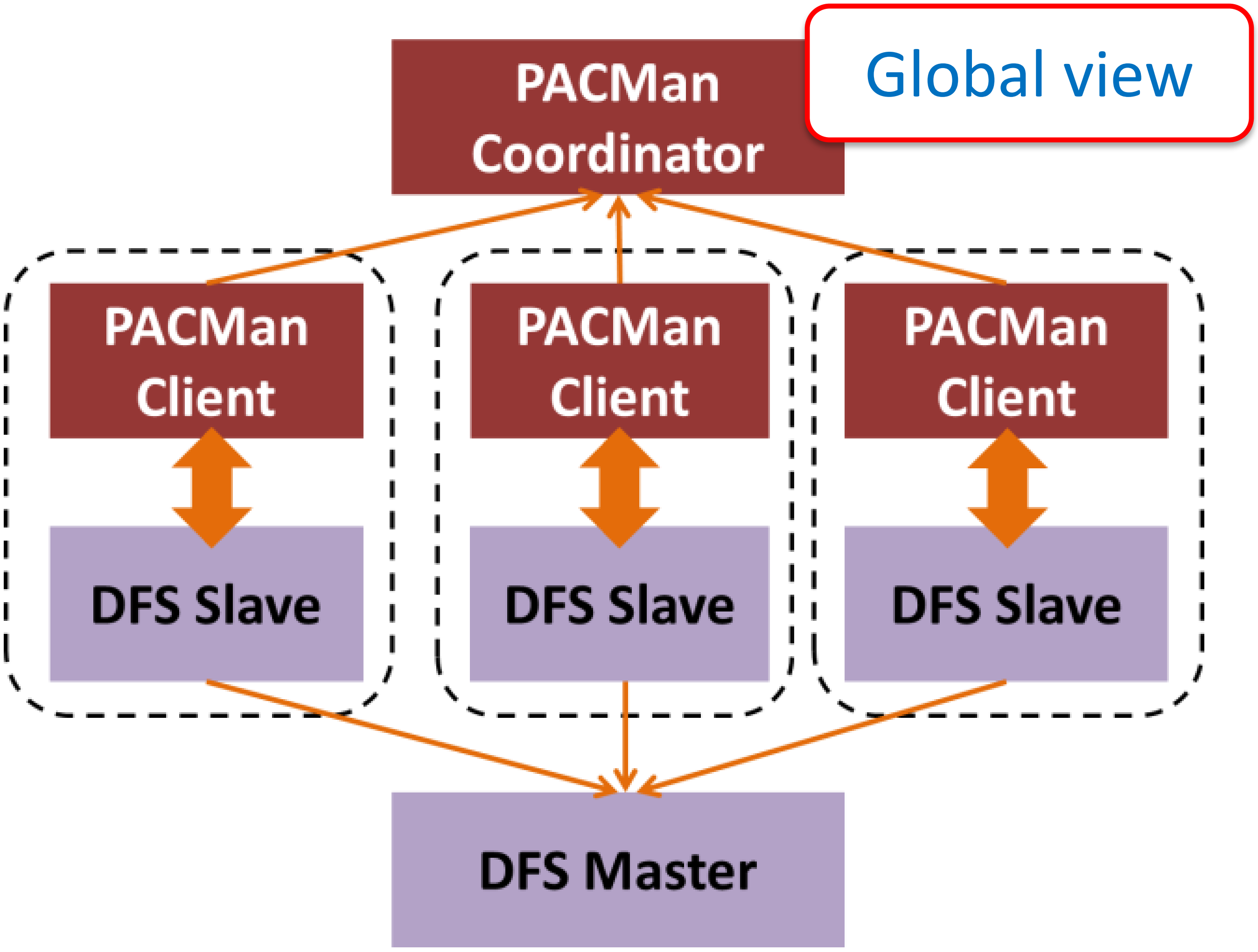
- Idealized model for job:
  - Wave-width for job: W
  - Frequency predicts future access: F
  - Data read is proportional to task length: D
  - Speedup factor for cached tasks: µ



- Cost of caching: W D
- Benefit of caching: W µD F
- Benefit/cost: µF

LFU-F – Favor jobs with most recent accessed files

System Design



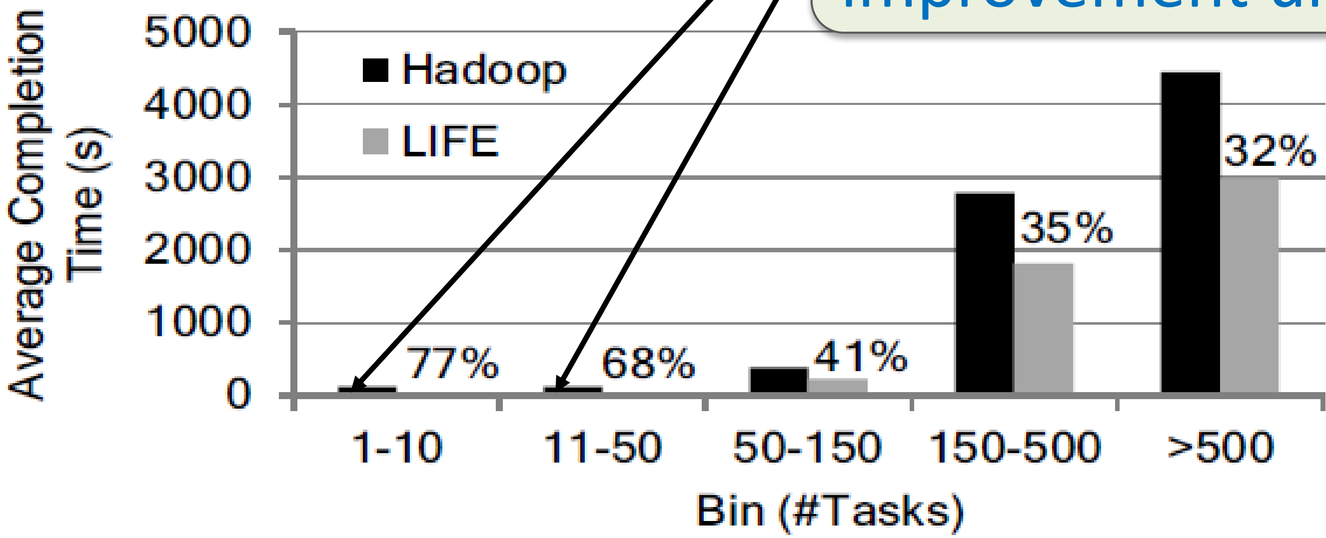
Evaluation Setup

- Workload derived from Facebook & Bing traces
  - FB: 3500 node Hadoop cluster, 375K jobs, 1 month
  - Bing: 1000's of nodes Dryad cluster, 200K jobs, 6 weeks

- Prototype in conjunction with HDFS
- Experiments on 100-node EC2 cluster
  - Cache of 20GB per machine

- Job Bins: Workload divided by number of map tasks they contained

Improvement in Completion Time



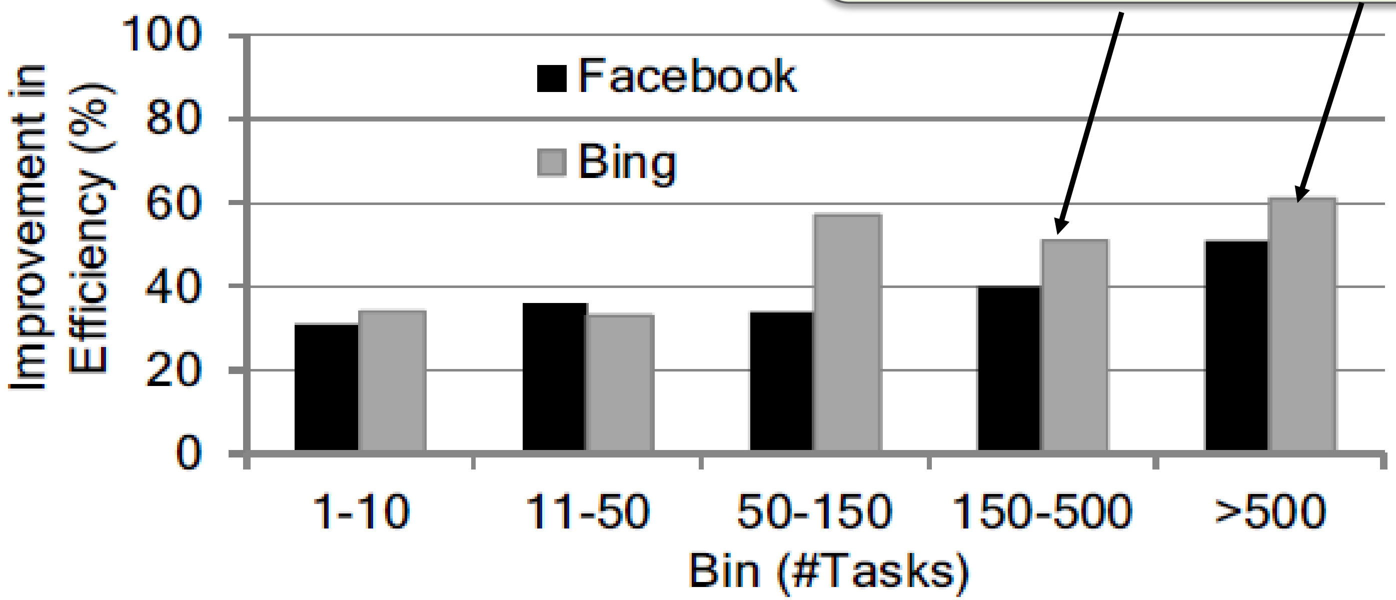
(a) Facebook Workload

small jobs: largest improvement under LIFE

Small jobs have lower wave-width

Improvement in Cluster Efficiency

large jobs: largest improvement under LFU-F



Large files are frequently accessed leading to lesser eviction under LFU-F

System Scalability Results

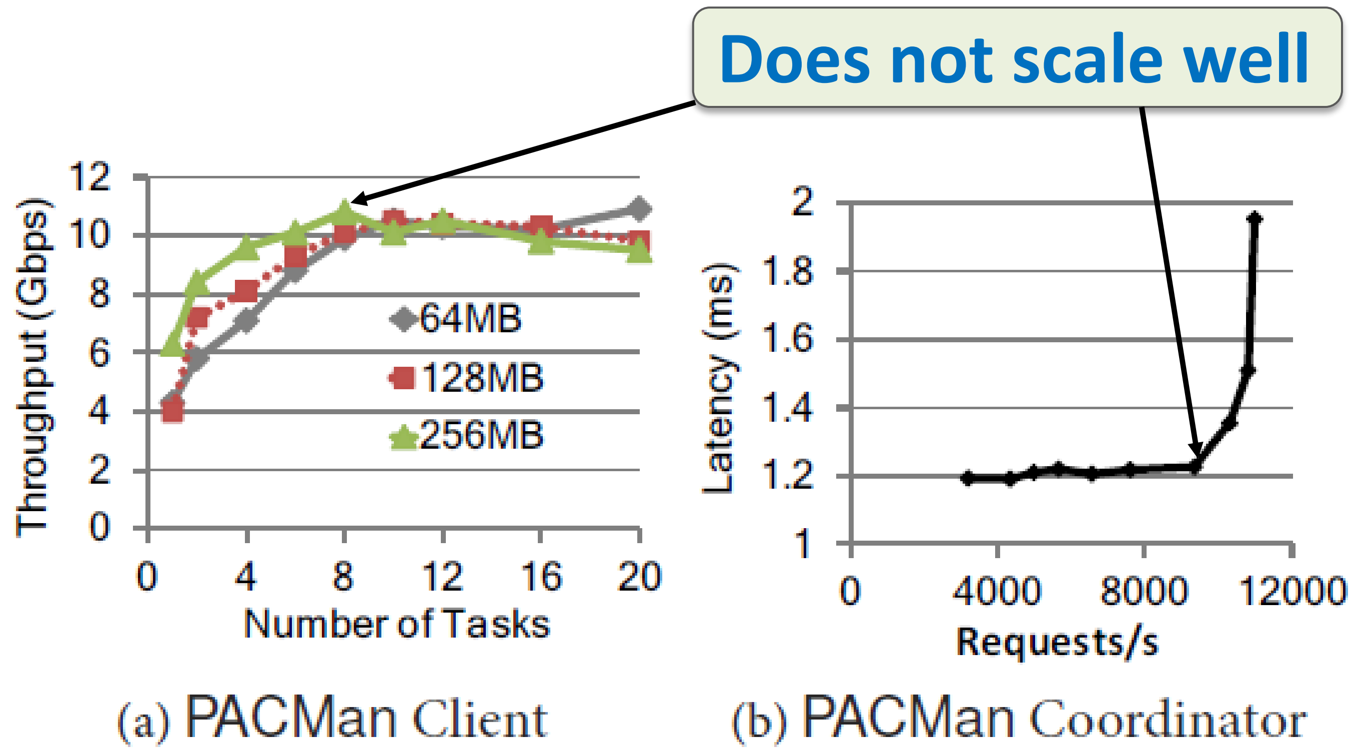


Figure 19: **Scalability. (a) Simultaneous tasks serviced by client, (b) Simultaneous client updates at the coordinator.**

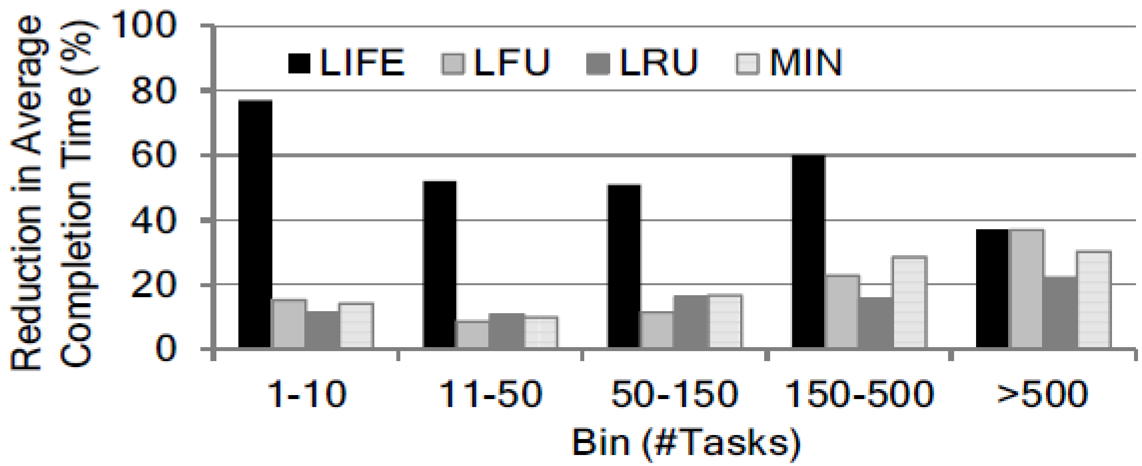
Summary

- All-or-nothing property of parallel jobs
  - Cache all of the inputs of a job
- PACMan: Coordinated Cache Management
  - Sticky policy: Evict from incomplete inputs
- LIFE for completion time, LFU-F for utilization
- Jobs are 53% faster, cluster utilization improves by 54%

Discussion

- Can Pacman handle graph computation systems like Pregel?
- Estimating wave width is hard for iterative computation?
- Pacman system does not scale that well.
- Piazza
  - Overhead of central coordinator
  - Experimental evaluation use only Facebook and Microsoft data
  - Job Priority not considered
  - Task dependency has not been studied or exploited

Comparison w/ State-of-the-Art



(a) Facebook Workload

Resilient Distributed Datasets

A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das,
Ankur Dave, Justin Ma, Murphy McCauley,
Michael Franklin, Scott Shenker, Ion Stoica

Presented by: Hilfi Alkaff
Content of this presentation is borrowed heavily from the original author's paper and presentation

Motivation

MapReduce greatly simplified "big data" analysis on large, unreliable clusters

But as soon as it got popular, users wanted more:

» More **complex**, multi-stage applications (e.g. iterative machine learning & graph processing)

» More **interactive** ad-hoc queries

Response: *specialized* frameworks for some of these apps (e.g. Pregel for graph processing)

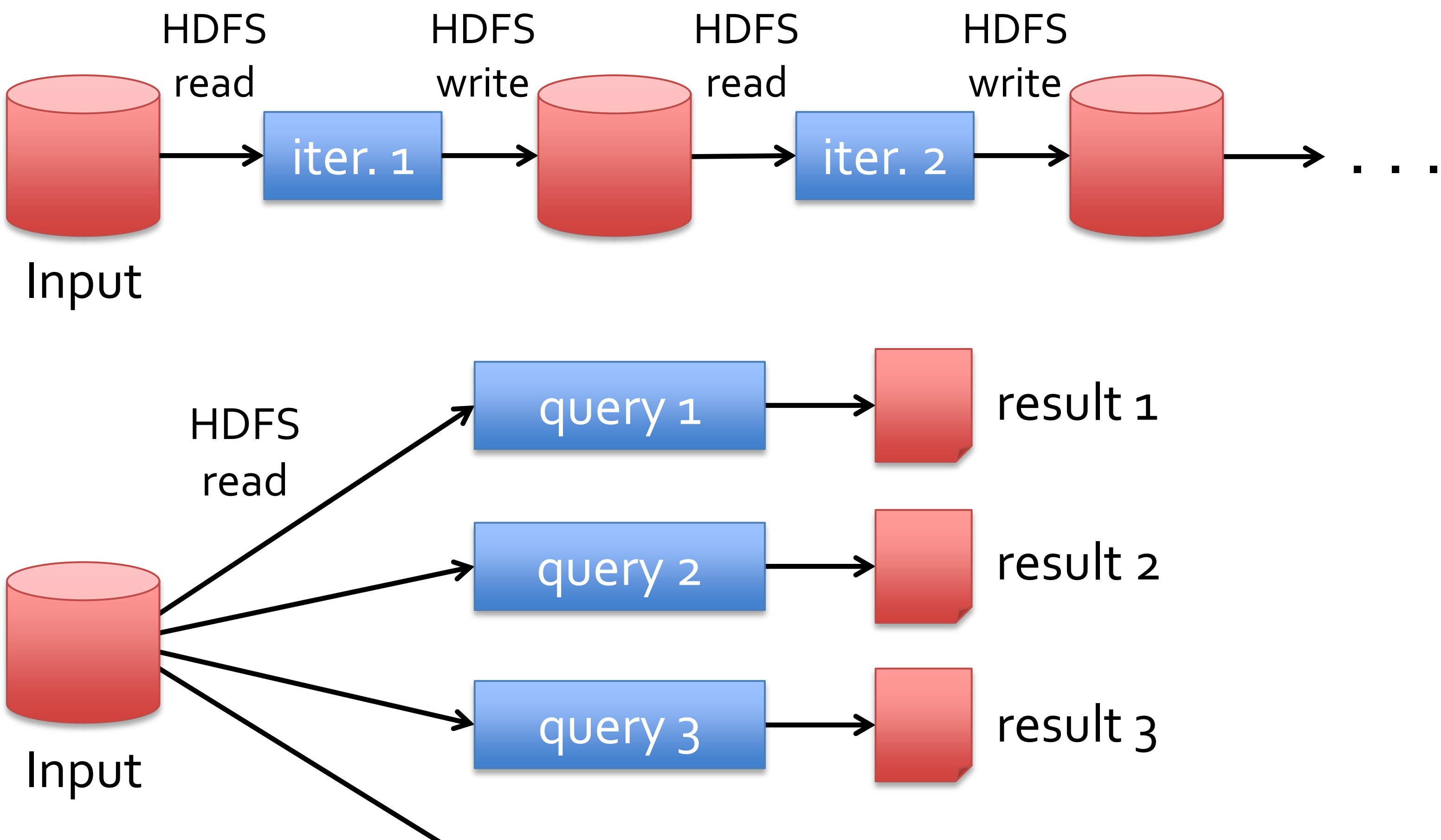
Motivation

Complex apps and interactive queries both need one thing that MapReduce lacks:

Efficient primitives for **data sharing**

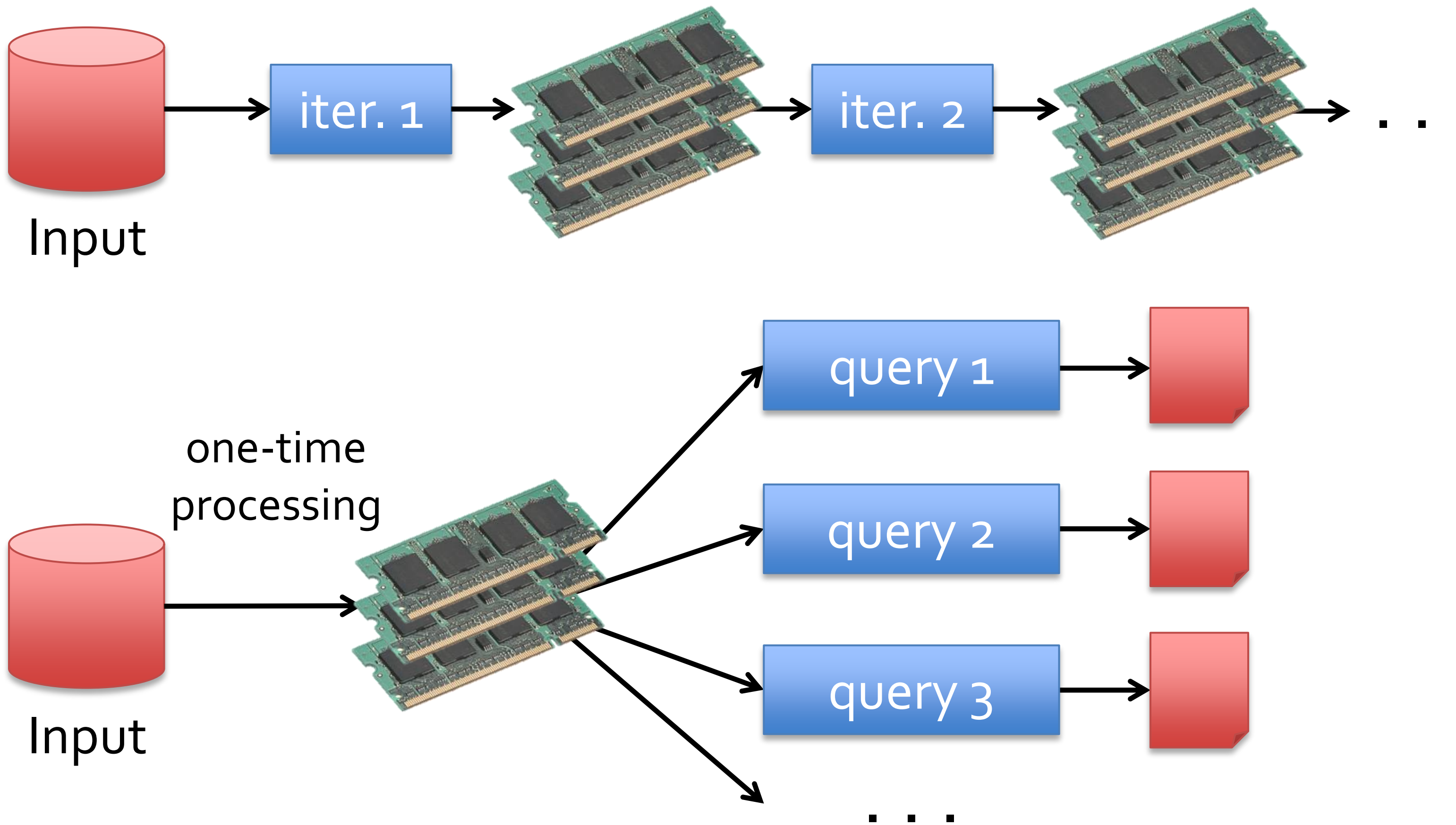
In MapReduce, the only way to share data across jobs is stable storage → slow!

Examples



Slow due to replication and disk I/O,
but necessary for fault tolerance

Goal: In-Memory Data Sharing



10-100× faster than network/disk, but how to get FT?

Challenge

How to design a distributed memory abstraction that is both **fault-tolerant** and **efficient**?

Challenge

Existing storage abstractions have interfaces based on *fine-grained* updates to mutable state

» RAMCloud, databases, distributed mem, Piccolo

Requires replicating data or logs across nodes for fault tolerance

» Costly for data-intensive apps

» 10-100x slower than memory write

Solution: Resilient Distributed Datasets (RDDs)

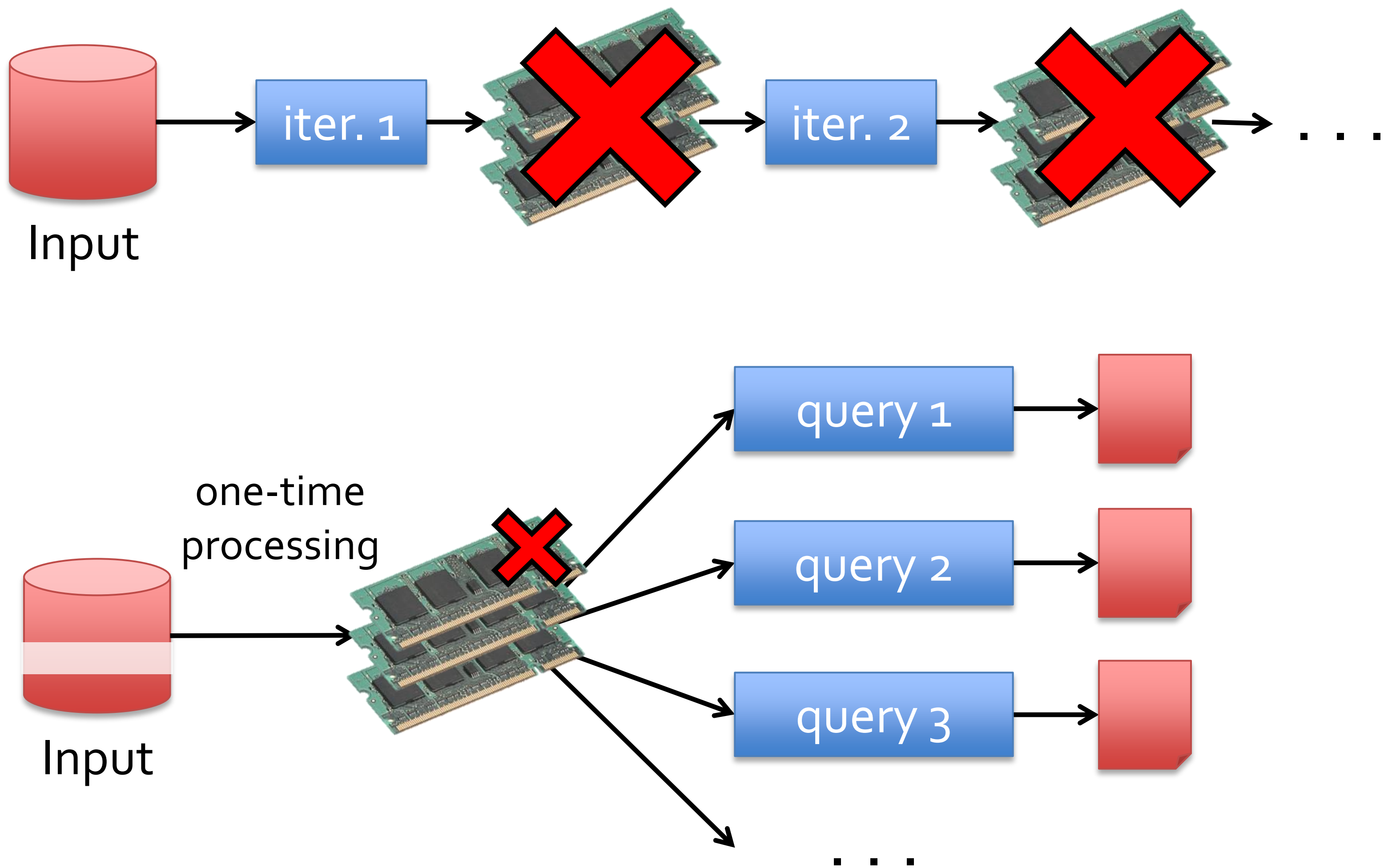
Restricted form of distributed shared memory

- » Immutable, partitioned collections of records
- » Can only be built through *coarse-grained* deterministic transformations (map, filter, join, …)

Efficient fault recovery using *lineage*

- » Log one operation to apply to many elements
- » Recompute lost partitions on failure
- » No cost if nothing fails

RDD Recovery



Generality of RDDs

Despite their restrictions, RDDs can express surprisingly many parallel algorithms

» These naturally *apply the same operation to many items*

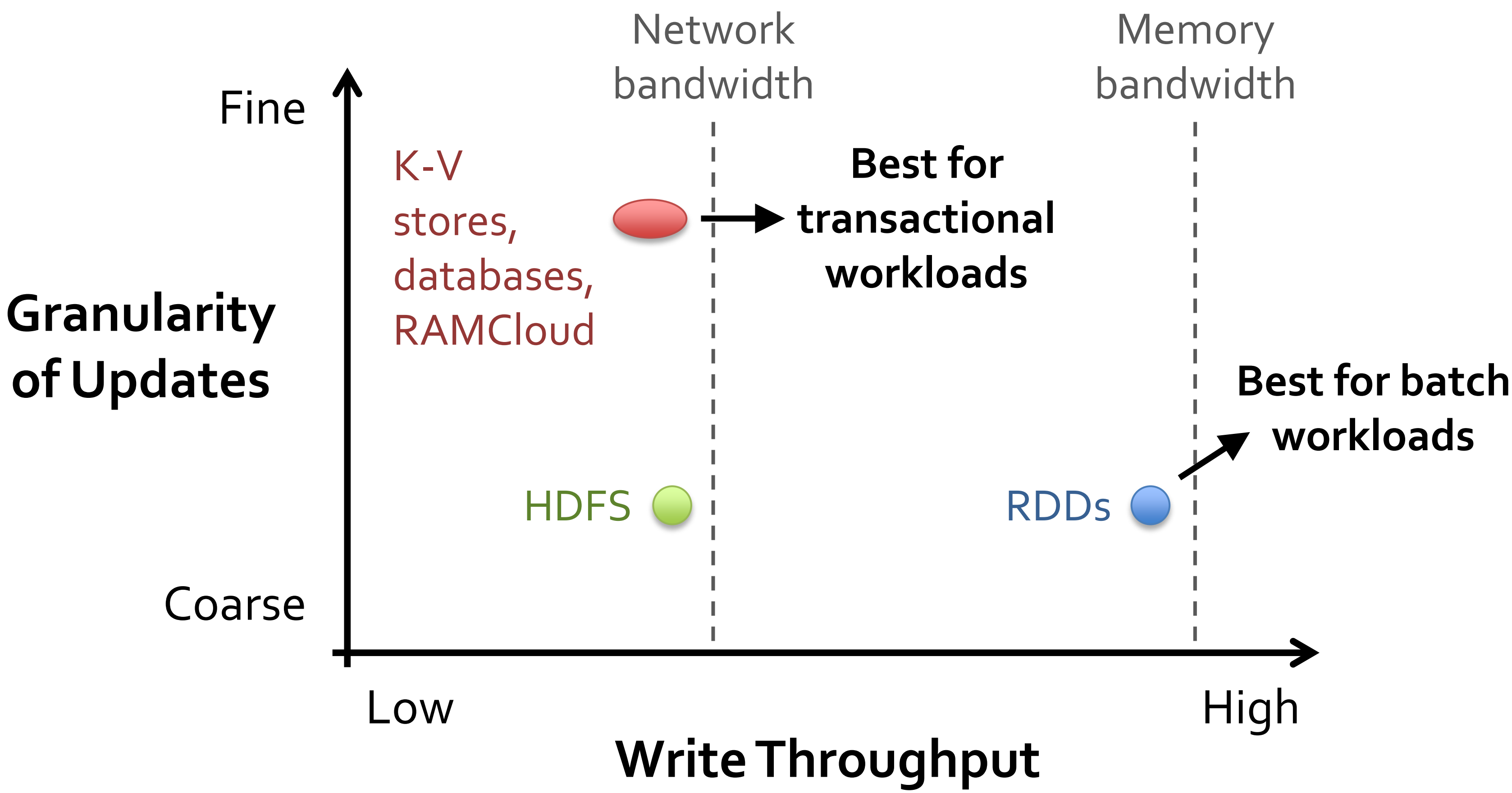
Unify many current programming models

» *Data flow models:* MapReduce, Dryad, SQL, …

» *Specialized models* for iterative apps: BSP (Pregel), iterative MapReduce (Haloop), bulk incremental, …

Support *new apps* that these models don't

Tradeoff Space



Spark Programming Interface

DryadLINQ-like API in the Scala language

Usable interactively from Scala interpreter

Provides:

- Resilient distributed datasets (RDDs)
- Operations on RDDs: *transformations* (build new RDDs), *actions* (compute and output results)
- Control of each RDD's *partitioning* (layout across nodes) and *persistence* (storage in RAM, on disk, etc)

Spark Operations

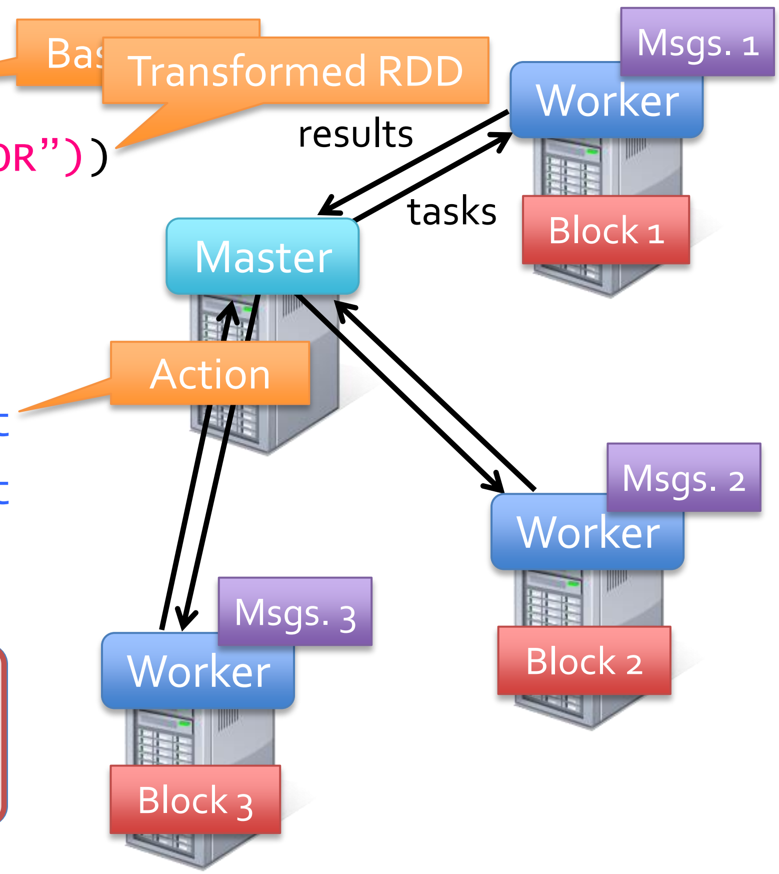
Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a result to driver program)	collect reduce count save lookupKey	

Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
messages.persist()

messages.filter(_.contains("foo")).count
messages.filter(_.contains("bar")).count
```

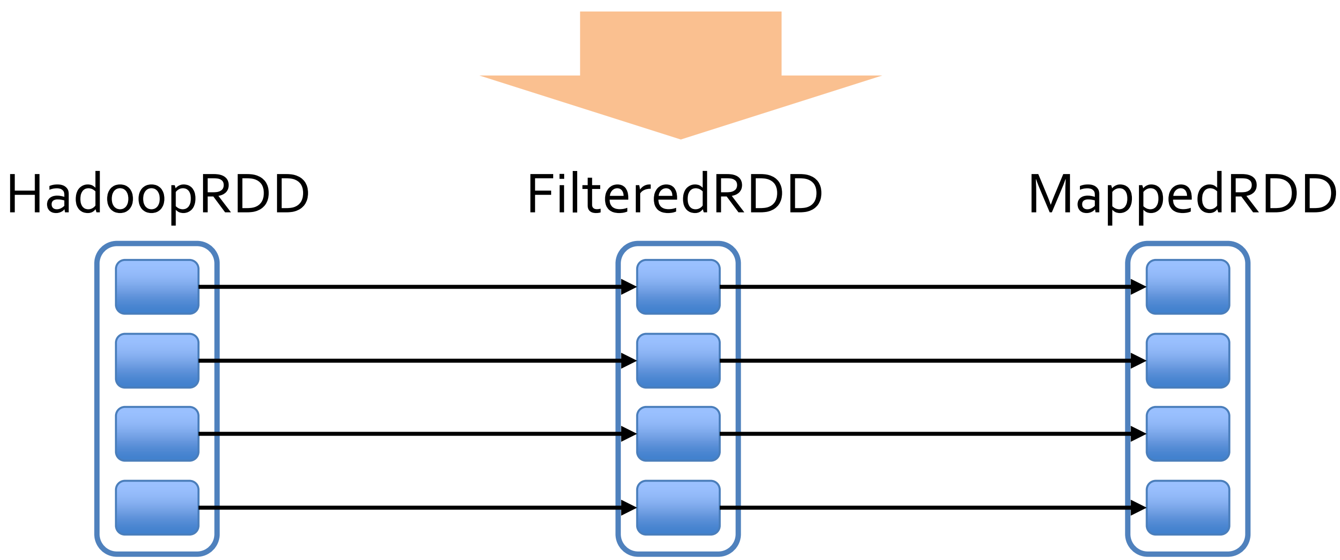


Result: scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

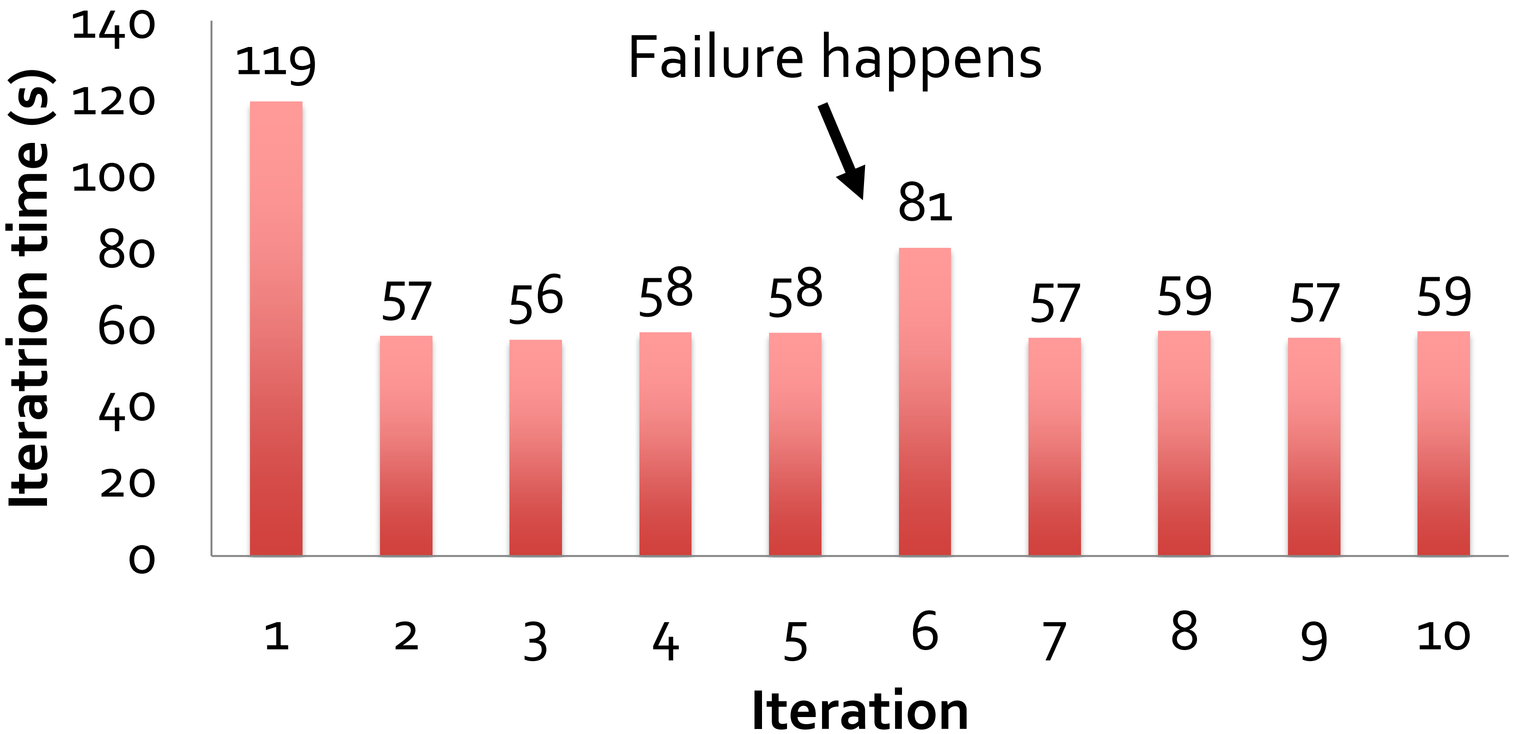
Fault Recovery

RDDs track the graph of transformations that built them (their *lineage*) to rebuild lost data

E.g.:
```
messages = textFile(...).filter(_.contains(“error”))
                        .map(_.split(‘\t’)(2))
```



Fault Recovery Results



Example: PageRank

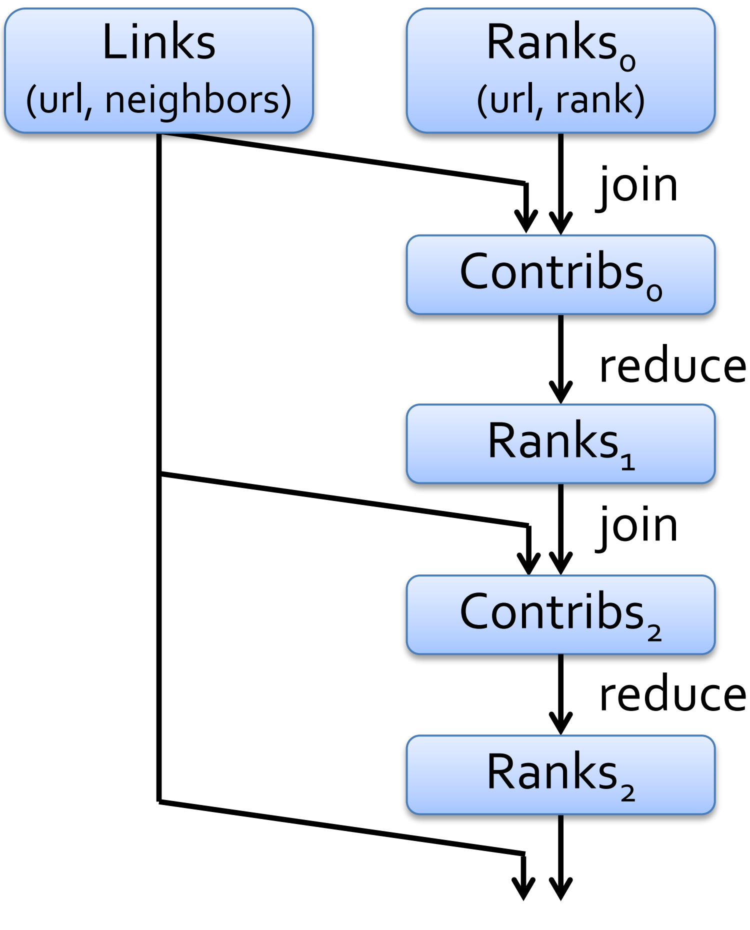
1. Start each page with a rank of 1
2. On each iteration, update each page's rank to

$$\Sigma_{i \in neighbors} \text{rank}_i \,/\, |\text{neighbors}_i|$$

```
links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  ranks = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }.reduceByKey(_ + _)
}
```

Optimizing Placement



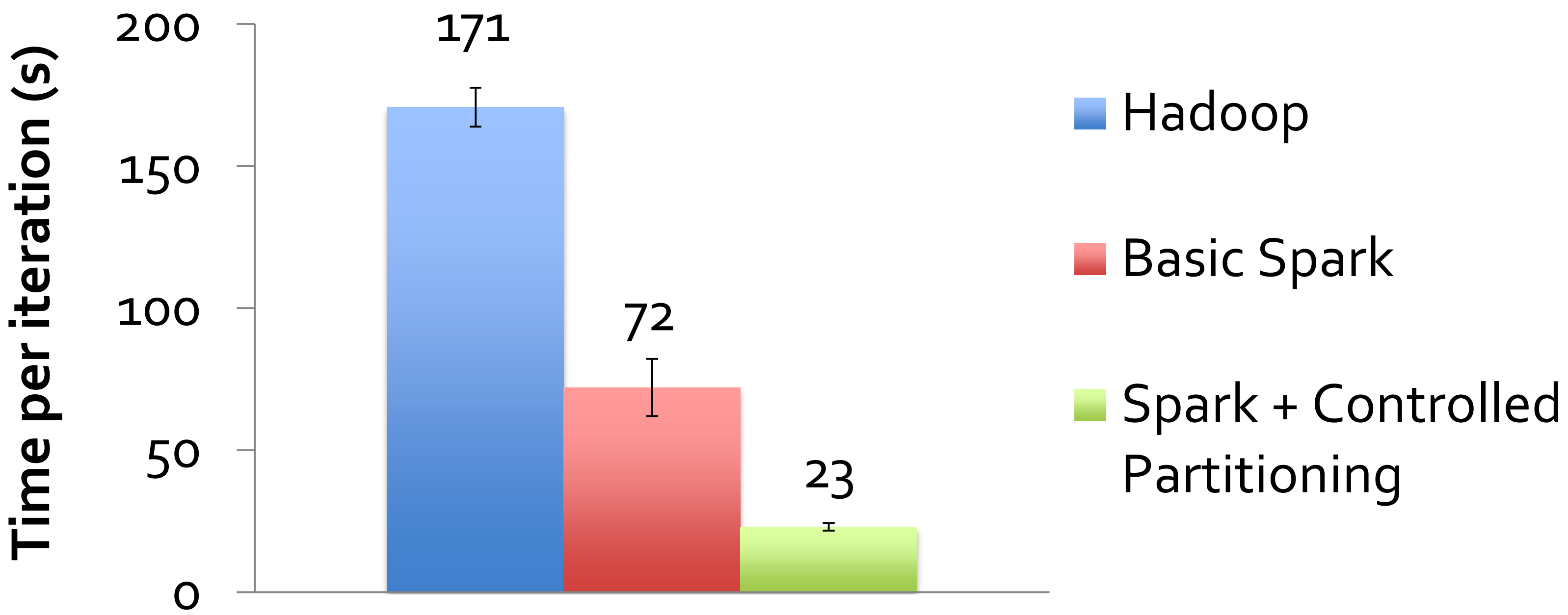
`links` & `ranks` repeatedly joined

Can *co-partition* them (e.g. hash both on URL) to avoid shuffles

Can also use app knowledge, e.g., hash on DNS name

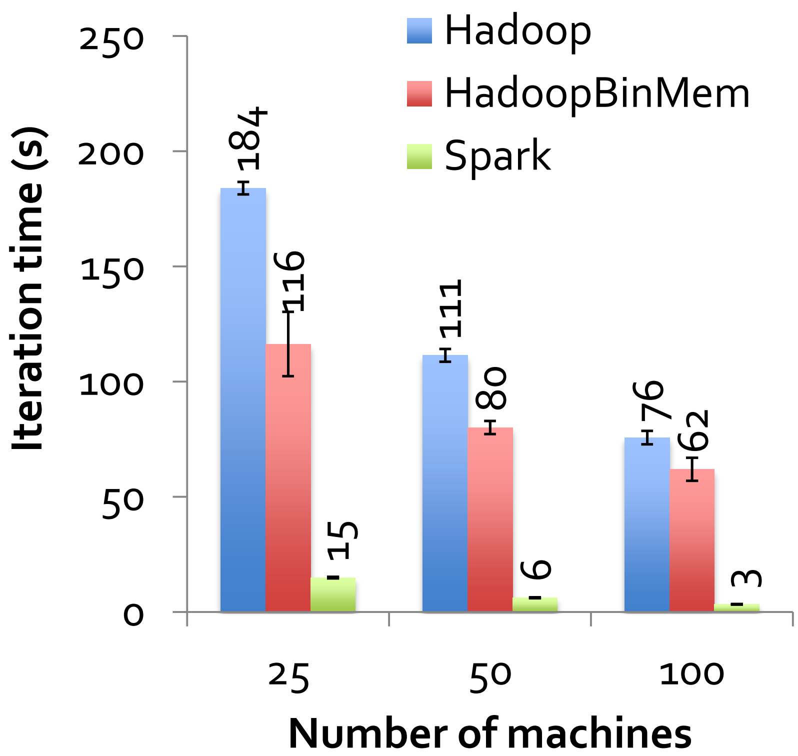
```
links = links.partitionBy(
          new URLPartitioner())
```

PageRank Performance

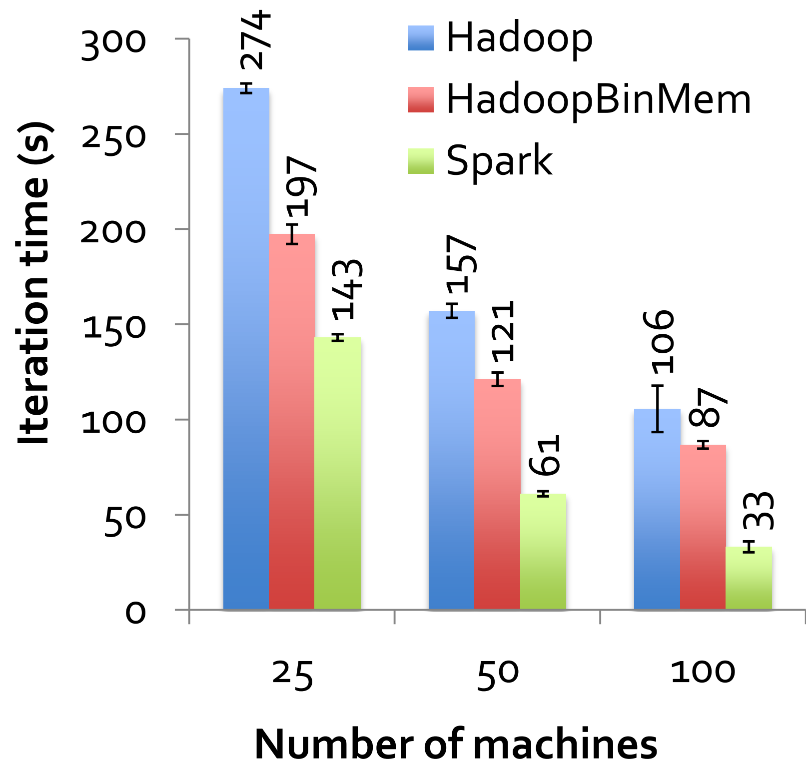


Scalability

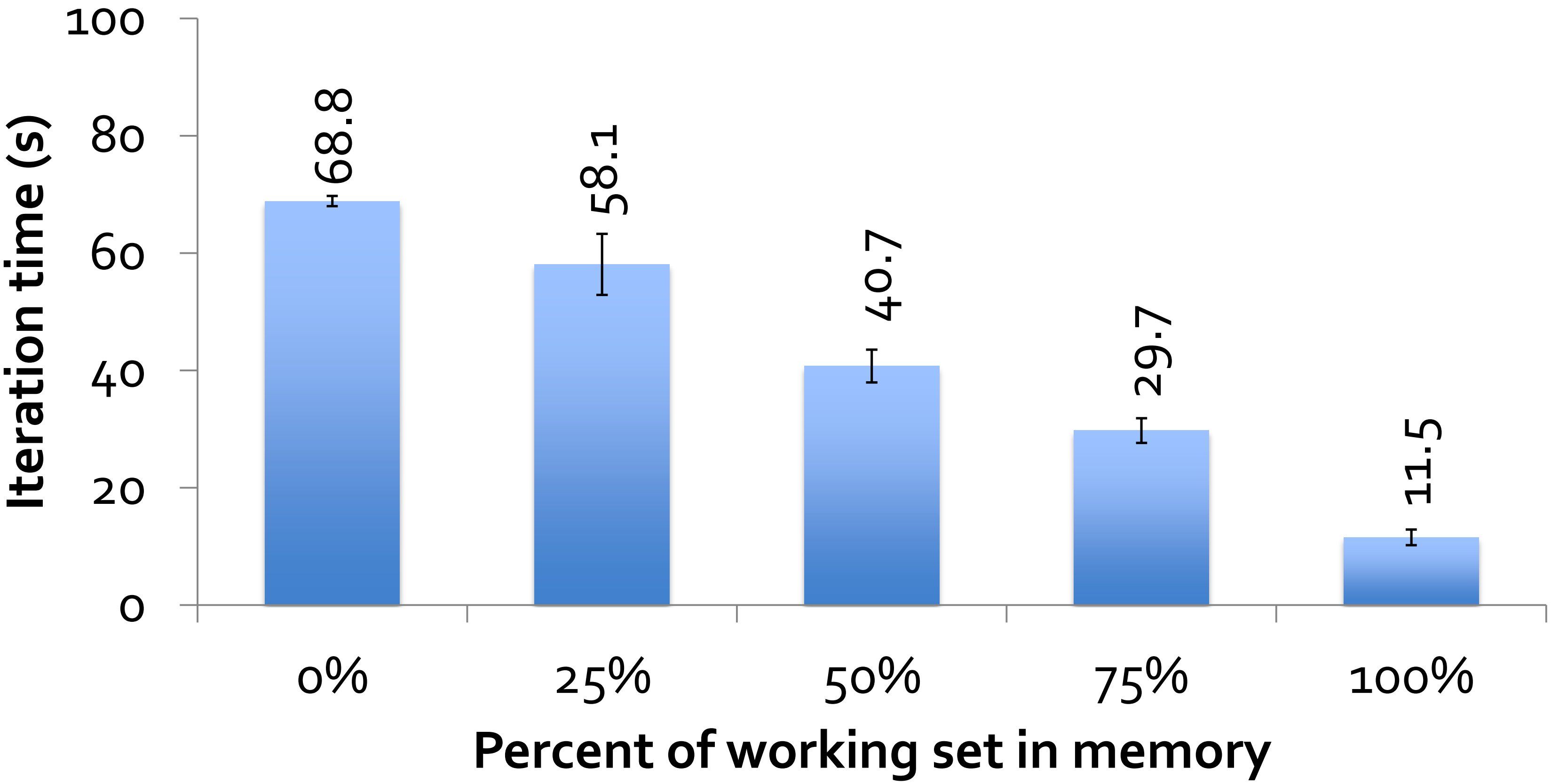
Logistic Regression



K-Means



Behavior with Insufficient RAM



Stuff

- Express many existing parallel models
  - Pregel (200 LOC), Iterative Map Reduce (200 LOC), SQL
  - Apps could efficiently intermix these models
- Used by **5+** companies, **3+** applications projects at Berkeley
  - Conviva, FourSquare, MobileMillenium
- Runs on Mesos [NSDI 11] to share clusters w/ Hadoop
- No changes to Scala language or compiler
  - Reflection + bytecode analysis to correctly ship code
- Open-sourced at: www.spark-project.org

Aftermath

- Concept of priority for different jobs
- Which data to kick out?
  - Currently just LRU
- Do we need to store data back to storage if job is too long? When?
- Spark Streaming [HotCloud '12]

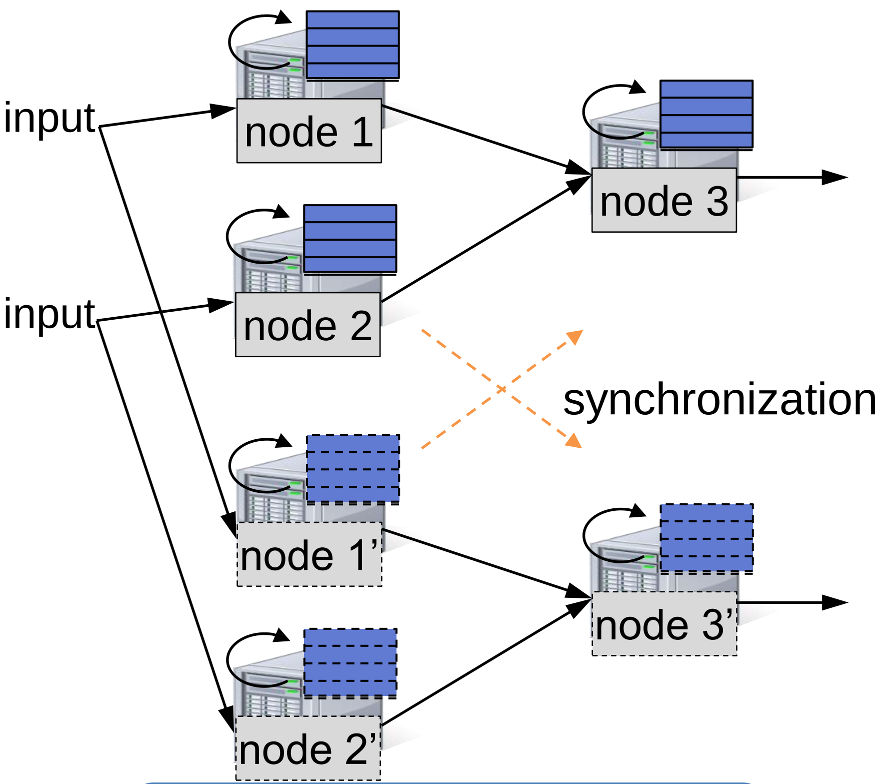
Conclusion

- RDDs offer a simple and efficient programming model for a broad range of applications
- Leverage the coarse-grained nature of many parallel algorithms for low-overhead recovery
- Best suited for batch applications

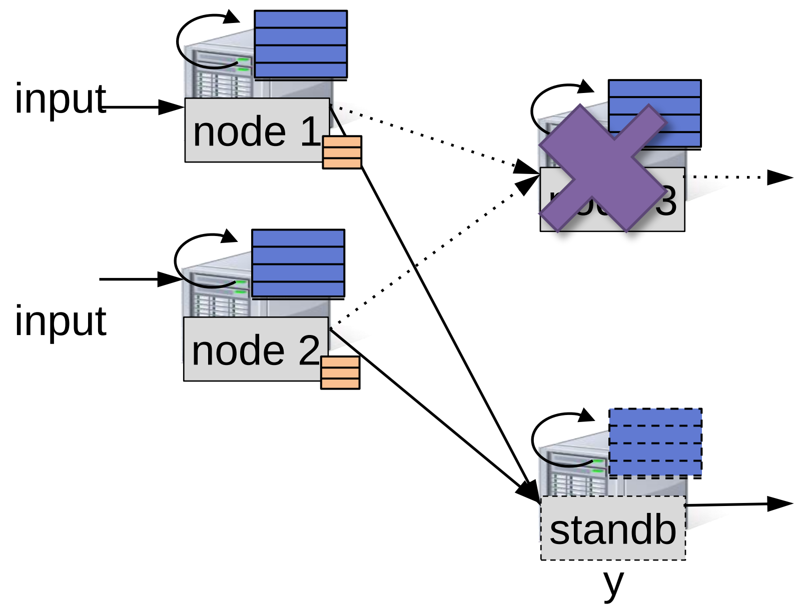
Backup Slides

Traditional Streaming Systems

Fault tolerance via **replication** or **upstream backup**:

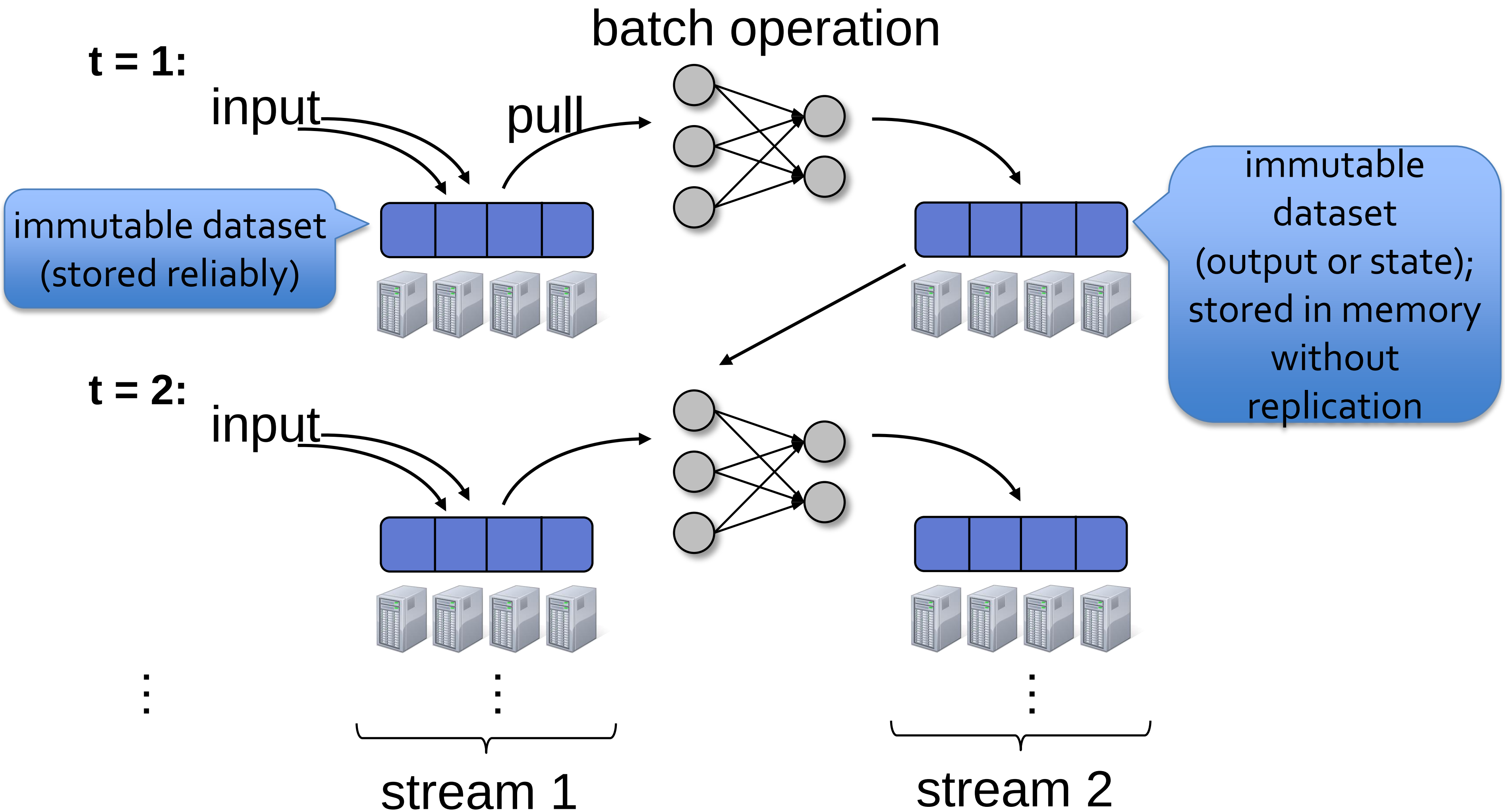


Fast recovery, but 2x hardware cost



Only need 1 standby, but slow to recover

Discretized Stream Processing



Related Work

RAMCloud, Piccolo, GraphLab, parallel DBs

» Fine-grained writes requiring replication for resilience

Pregel, iterative MapReduce

» Specialized models; can't run arbitrary / ad-hoc queries

DryadLINQ, FlumeJava

» Language-integrated "distributed dataset" API, but cannot share datasets efficiently *across* queries

Nectar [OSDI 10]

» Automatic expression caching, but over distributed FS

PacMan [NSDI 12]

» Memory cache for HDFS, but writes still go to network/disk