IEMS5709
Advanced Topics in Information Processing - Big Data Systems and Information Processing
Spring 2016

MapReduce and Related Systems

Prof. Wing C. Lau
Department of Information Engineering
wclau@ie.cuhk.edu.hk

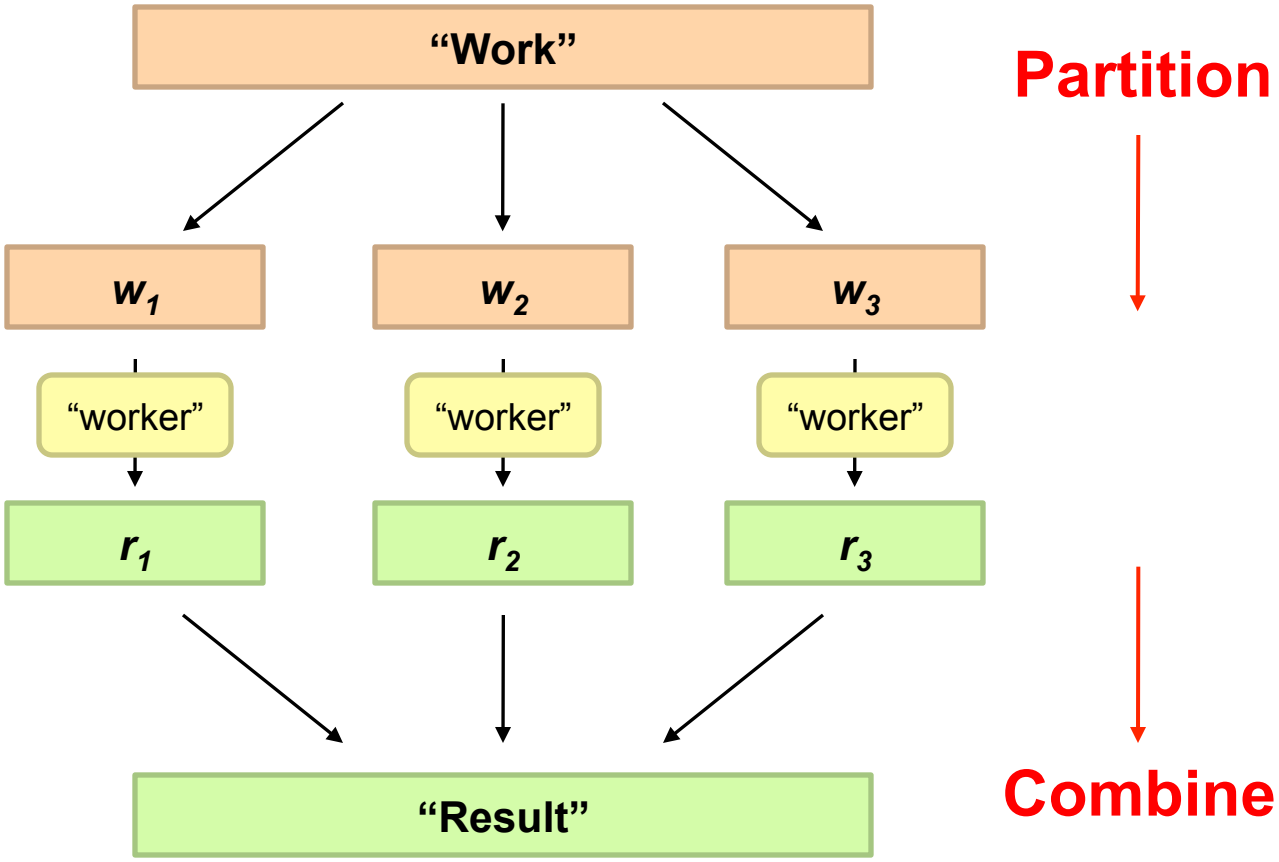
Acknowledgements

- The slides used in this chapter are adapted from the following sources:
  - "Data-Intensive Information Processing Applications," by Jimmy Lin, University of Maryland.

  - CS246 Mining Massive Data-sets, by Jure Leskovec, Stanford University.
  - "Data Management in the Cloud – Advanced Topics in Databases," by Saake, Schallehn, Mohammad of OvGU, Summer 2011.
  - Introduction to Advanced Computing Platform for Data Analysis, by Ruoming Jin, Kent University.
  - "Intro To Hadoop" in UCBerkeley i291 - Analyzing BigData with Twitter, by Bill Graham, Twitter.

How do we scale-up for Web-Scale Information Analytics ?

Divide and Conquer



Parallelization Challenges

- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers die?

What is the common theme of all of these problems?

Common Theme?

- Parallelization problems arise from:
  - Communication between workers (e.g., to exchange state)
  - Access to shared resources (e.g., data)
- Thus, we need a synchronization mechanism

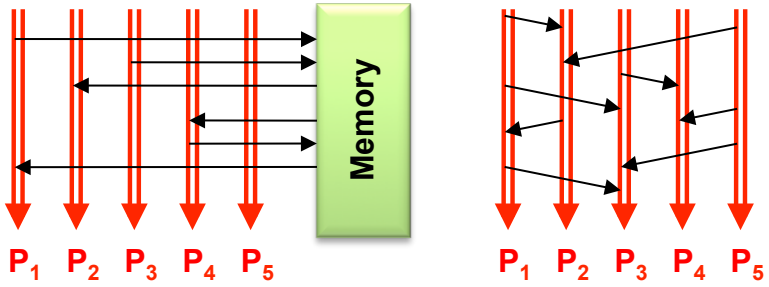
Source: Ricardo Guimarães Herrmann

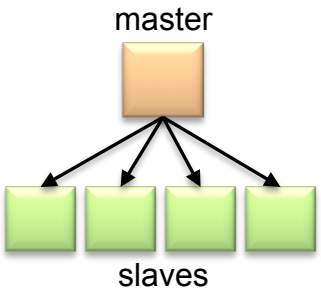
Managing Multiple Workers

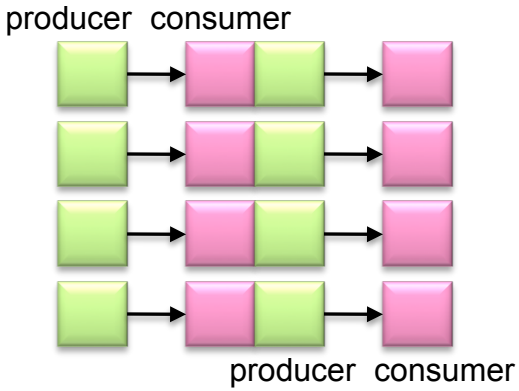
- Difficult because
  - We don't know the order in which workers run
  - We don't know when workers interrupt each other
  - We don't know the order in which workers access shared data
- Thus, we need:
  - Semaphores (lock, unlock)
  - Conditional variables (wait, notify, broadcast)
  - Barriers
- Still, lots of problems:
  - Deadlock, livelock, race conditions...
  - Dining philosophers, sleeping barbers, cigarette smokers...
- Moral of the story: be careful!

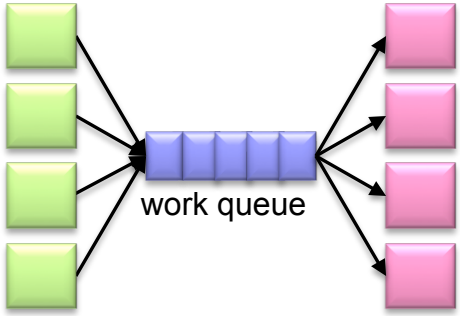
Current Tools

- Programming models
  - Shared memory (pthreads)
  - Message passing (MPI)
- Design Patterns
  - Master-slaves
  - Producer-consumer flows
  - Shared work queues





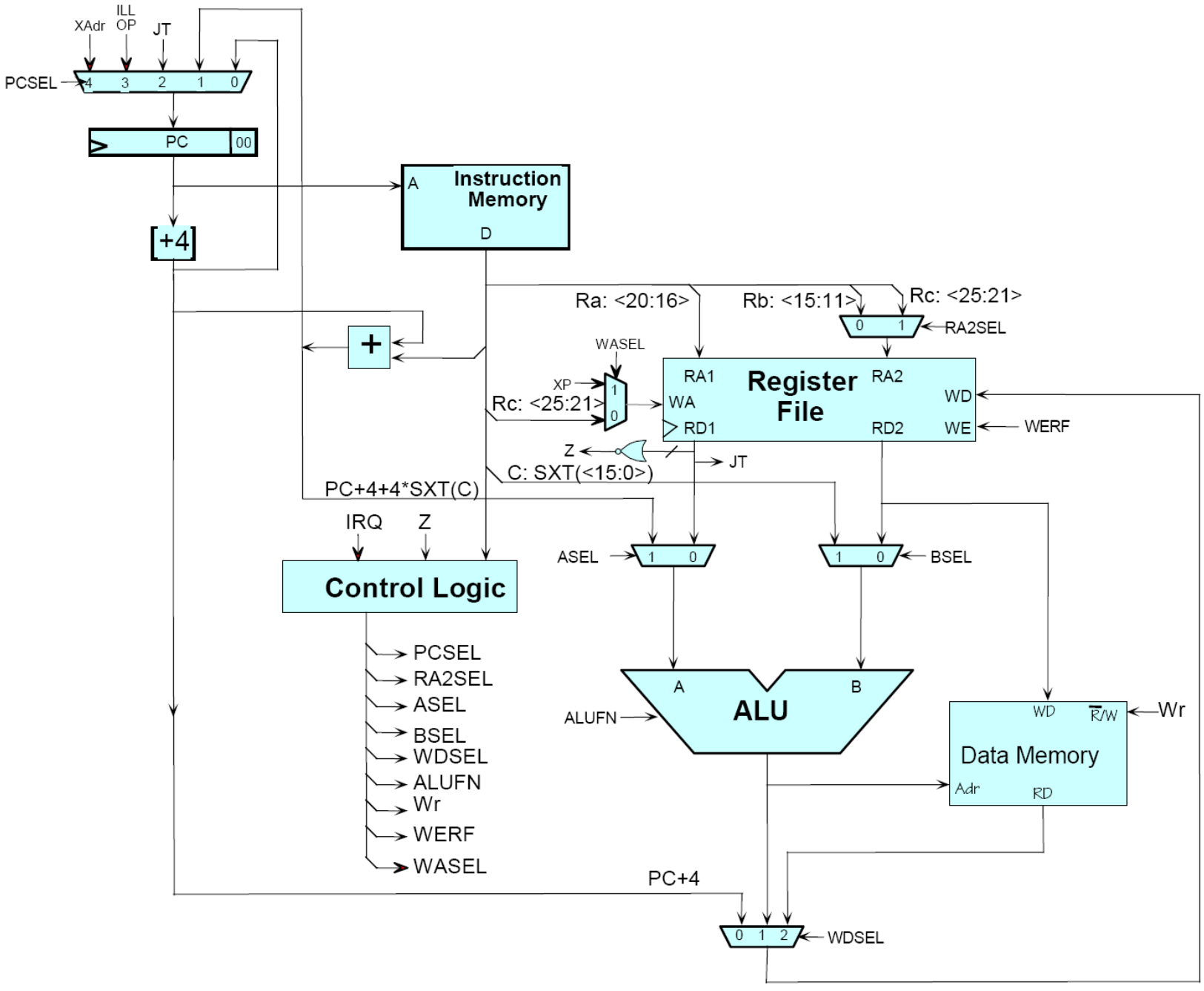




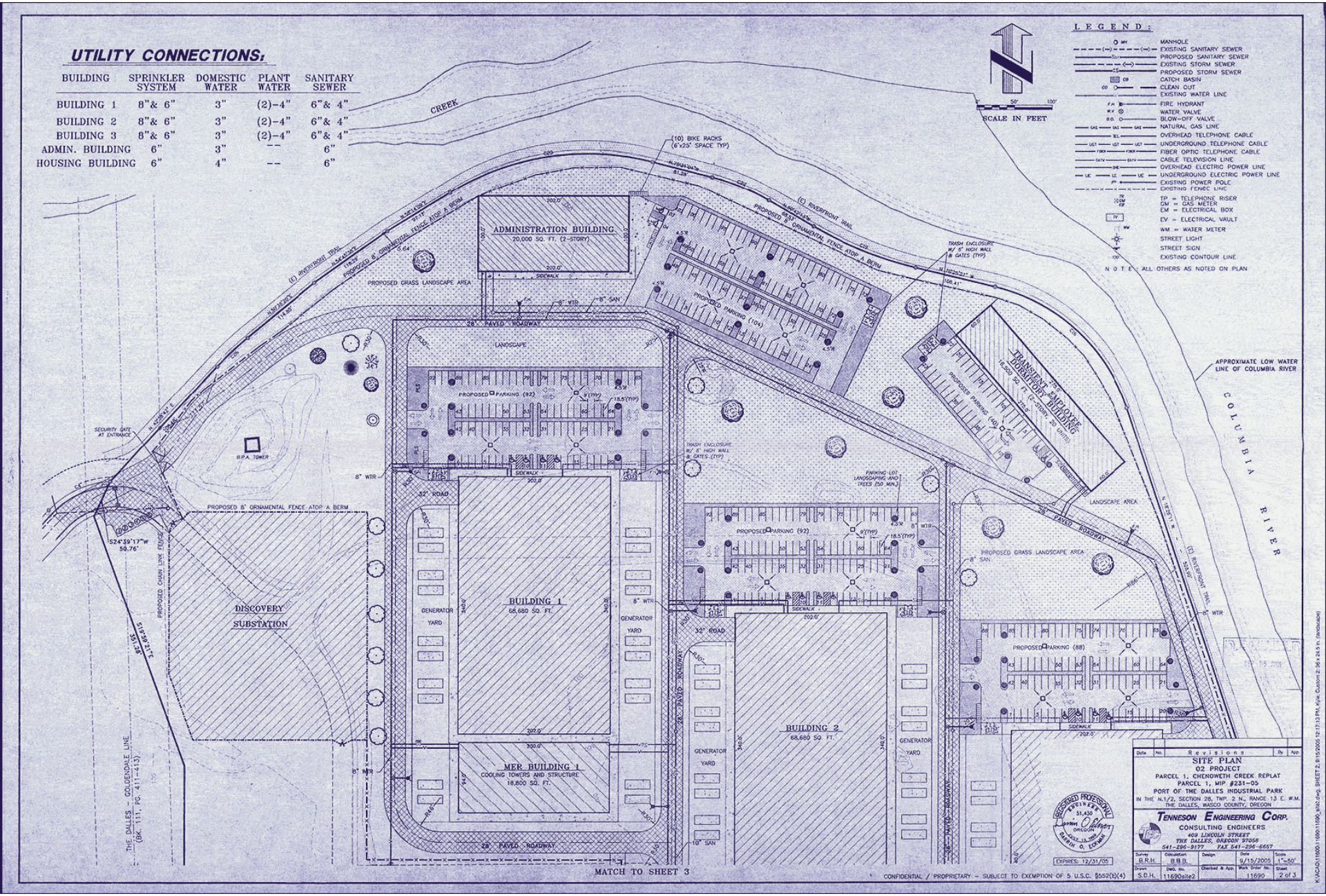
Where the rubber meets the road

- Concurrency is difficult to reason about
- Concurrency is even more difficult to reason about
  - At the scale of datacenters (even across datacenters)
  - In the presence of failures
  - In terms of multiple interacting services
- Not to mention debugging…
- The reality:
  - Lots of one-off solutions, custom code
  - Write you own dedicated library, then program with it
  - Burden on the programmer to explicitly manage everything

Source: Wikipedia (Flat Tire)


XAdr
ILL OP
JT
PCSEL
4 3 2 1 0
PC
00
+4
A
Instruction Memory
D
Ra: <20:16>
Rb: <15:11>
Rc: <25:21>
0 1
RA2SEL
+
WASEL
XP
Rc: <25:21>
1
0
RA1
WA
RD1
Register File
RA2
RD2
WD
WE
WERF
Z
JT
C: SXT(<15:0>)
PC+4+4*SXT(C)
IRQ
Z
ASEL
1 0
1 0
BSEL
Control Logic
PCSEL
RA2SEL
ASEL
BSEL
WDSEL
ALUFN
Wr
WERF
WASEL
A
B
ALUFN
ALU
WD
R/W
Wr
Data Memory
Adr
RD
PC+4
0 1 2
WDSEL


Instruction Memory
Register File
Control Logic
ALU
Data Memory



Source: Harper's (Feb, 2008)

What's the point?

- It's all about the right level of abstraction
  - The von Neumann architecture has served us well, but is no longer appropriate for the multi-core/cluster environment
- Hide system-level details from the developers
  - No more race conditions, lock contention, etc.
- Separating the *what* from *how*
  - Developer specifies the computation that needs to be performed
  - Execution framework ("runtime") handles actual execution

The datacenter *is* the computer!

“Big Ideas”

- Scale “out”, not “up”
  - Limits of SMP and large shared-memory machines
- Move processing to the data
  - Cluster have limited bandwidth
- Process data sequentially, avoid random access
  - Seeks are expensive, disk throughput is reasonable
- Seamless scalability
  - From the mythical man-month to the tradable machine-hour

Computational Model for Web-scale Information Processing: MapReduce

Google MapReduce

- Framework for parallel processing in large-scale shared-nothing architecture
- Developed initially (and patented) by Google to handle Search Engine's webpage indexing and page ranking in a more systematic and maintainable fashion
- Why NOT using existing Database (DB)/ Relational Database Management Systems (RDMS) technologies?

  Mismatch of Objectives

  - DB/ RDMS were designed for high-performance transactional processing to support hard guarantees on consistencies in case of MANY concurrent (often small) updates, e.g. ebanking, airline ticketing ; DB Analytics were "secondary" functions added on later ;
  - For Search Engines, the documents are never updated (till next Web Crawl) and they are Read-Only ; It is ALL about Analytics !
  - Import the webpages, convert them to DB storage format is expensive
  - The Job was simply too big for prior DB technologies !

Typical BigData Problem

- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

Map

Reduce

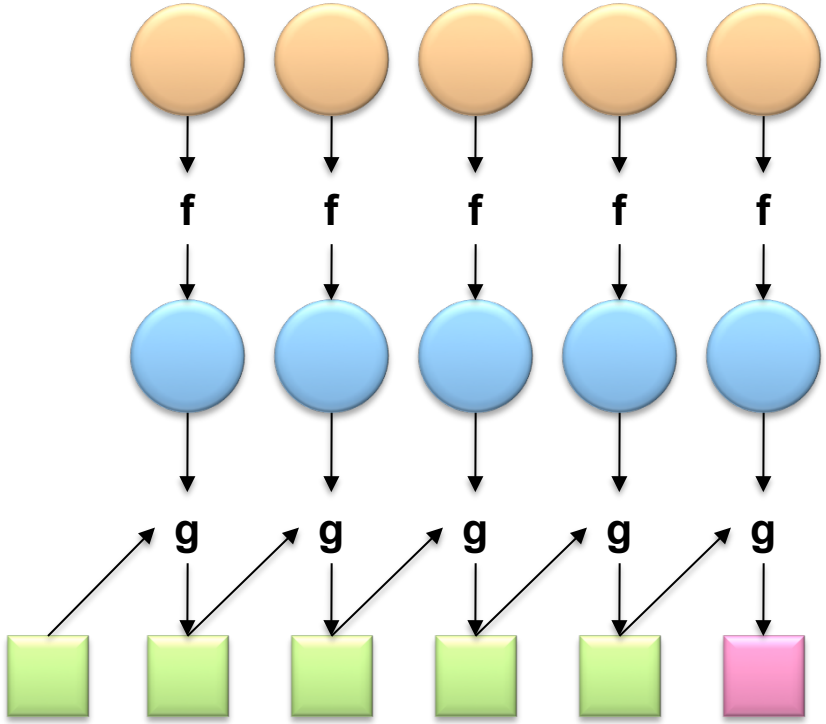
Key idea: provide a functional abstraction for these two operations

(Dean and Ghemawat, OSDI 2004)

Roots in Functional Programming

Map

Fold



MapReduce

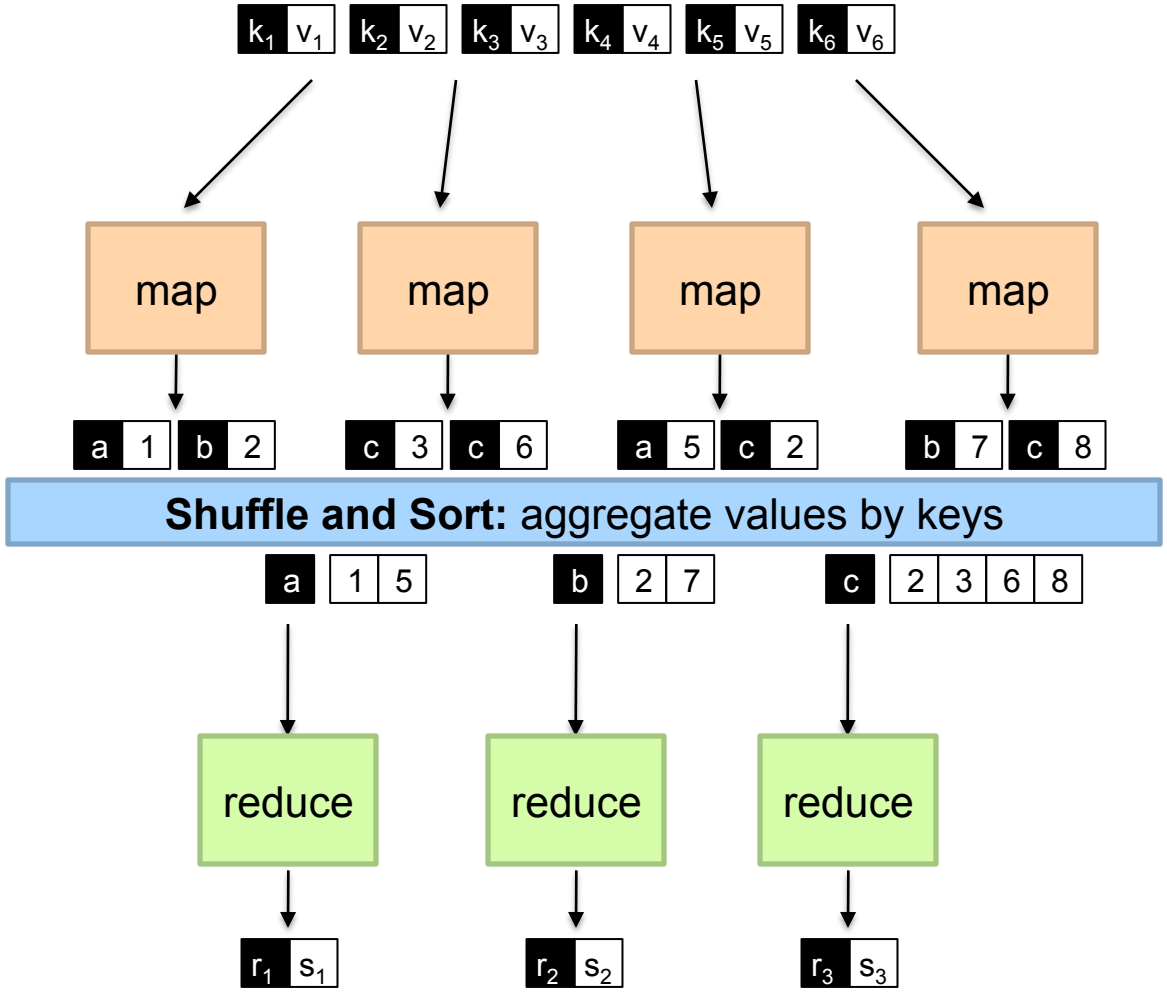
- Programmers specify two functions:

  map (k, v) → <k', v'>*

  reduce (k', v') → <k'', v''>*

  - All values with the same key are sent to the same reducer
  - <a,b>* means a list of tuples in the form of (a,b)

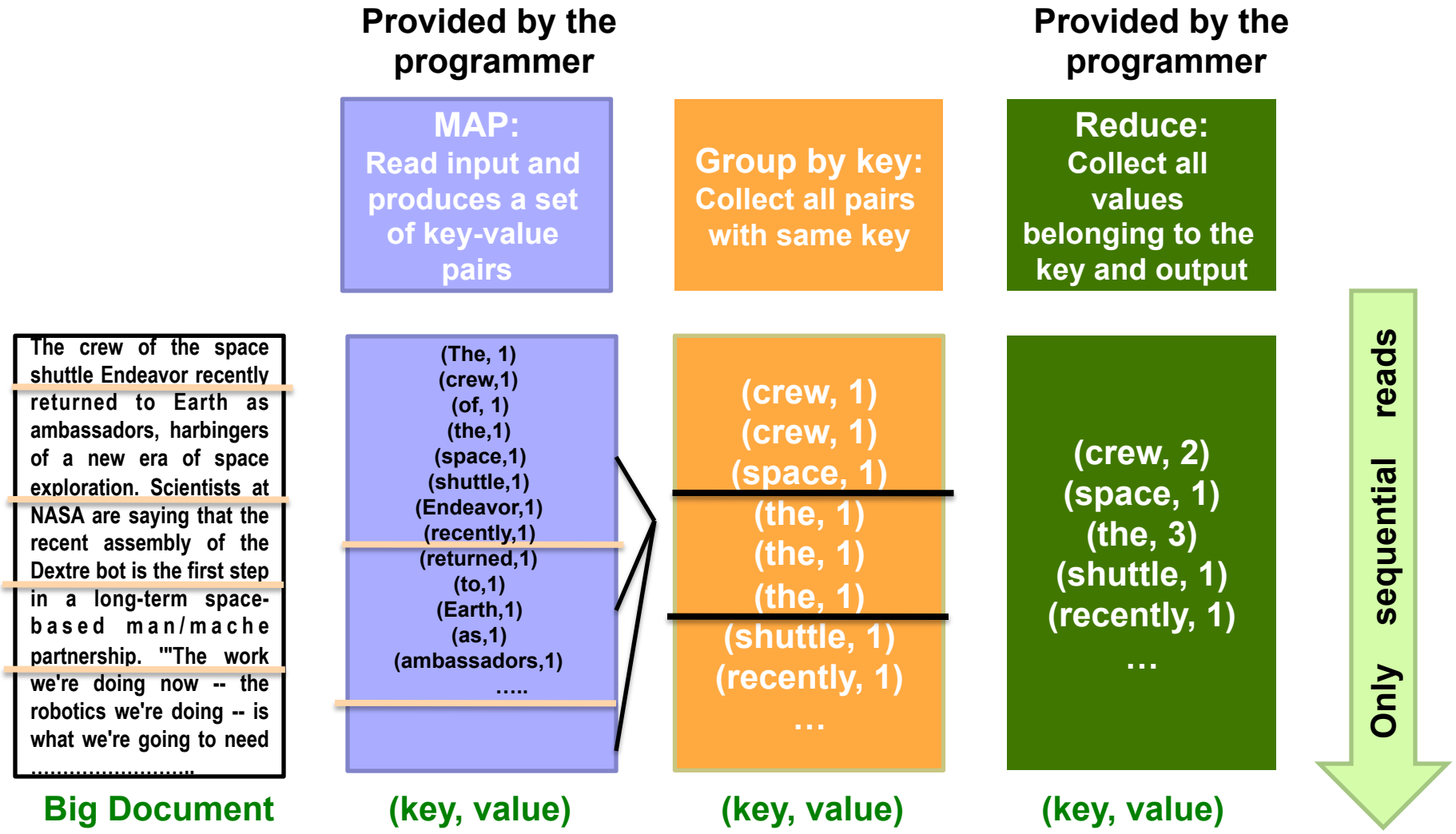
- The execution framework handles everything else…


k1 v1
k2 v2
k3 v3
k4 v4
k5 v5
k6 v6
map
map
map
map
a 1
b 2
c 3
c 6
a 5
c 2
b 7
c 8
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

"Hello World" Task for MapReduce: Word Counting

- Unix/Linux shell command to Count occurrences of words in a file named `doc.txt`:
  - **`words(doc.txt) | sort | uniq -c`**
    - where **`words`** takes a file and outputs the words in it, one per a line
    - **`"uniq"`** stands for unique, is a true Unix command ; see its manpage to find out what **`"uniq -c"`** does
- The above "Unix/Linux-shell command" captures the essence of **MapReduce**
  - Great thing is that it is naturally parallelizable

MapReduce: Word Counting



"Hello World": Pseudo-code for Word Count

```
Map(String docid, String text):
// docid: document name, i.e. the input key ;
// text:  text in the document, i.e. the input value
    for each word w in text:
        EmitIntermediate(w, 1);

Reduce(String term, Iterator<Int> Ivalues):
// term: a word, i.e. the intermediate key,  also happens to be the output key here ;
// Ivalues: an iterator over counts (i.e. gives the list of intermediate values from Map)
    int sum = 0;
    for each v in Ivalues:
        sum += v ;
    Emit(term, sum);
```

// The above is pseudo-code only ! True code is a bit more involved: needs to define how the input key/values are divided up and accessed, etc).

MapReduce

- Programmers specify two functions:

  map (k, v) → <k', v'>*

  reduce (k', v') → <k'', v''>*

  - All values with the same key are sent to the same reducer
- The execution framework handles everything else…

What's "everything else"?

MapReduce “Runtime”

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles “data distribution”
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed File System (later)

MapReduce

- Programmers specify two functions:

  map (k, v) → <k', v'>*

  reduce (k', v') → <k'', v''>*

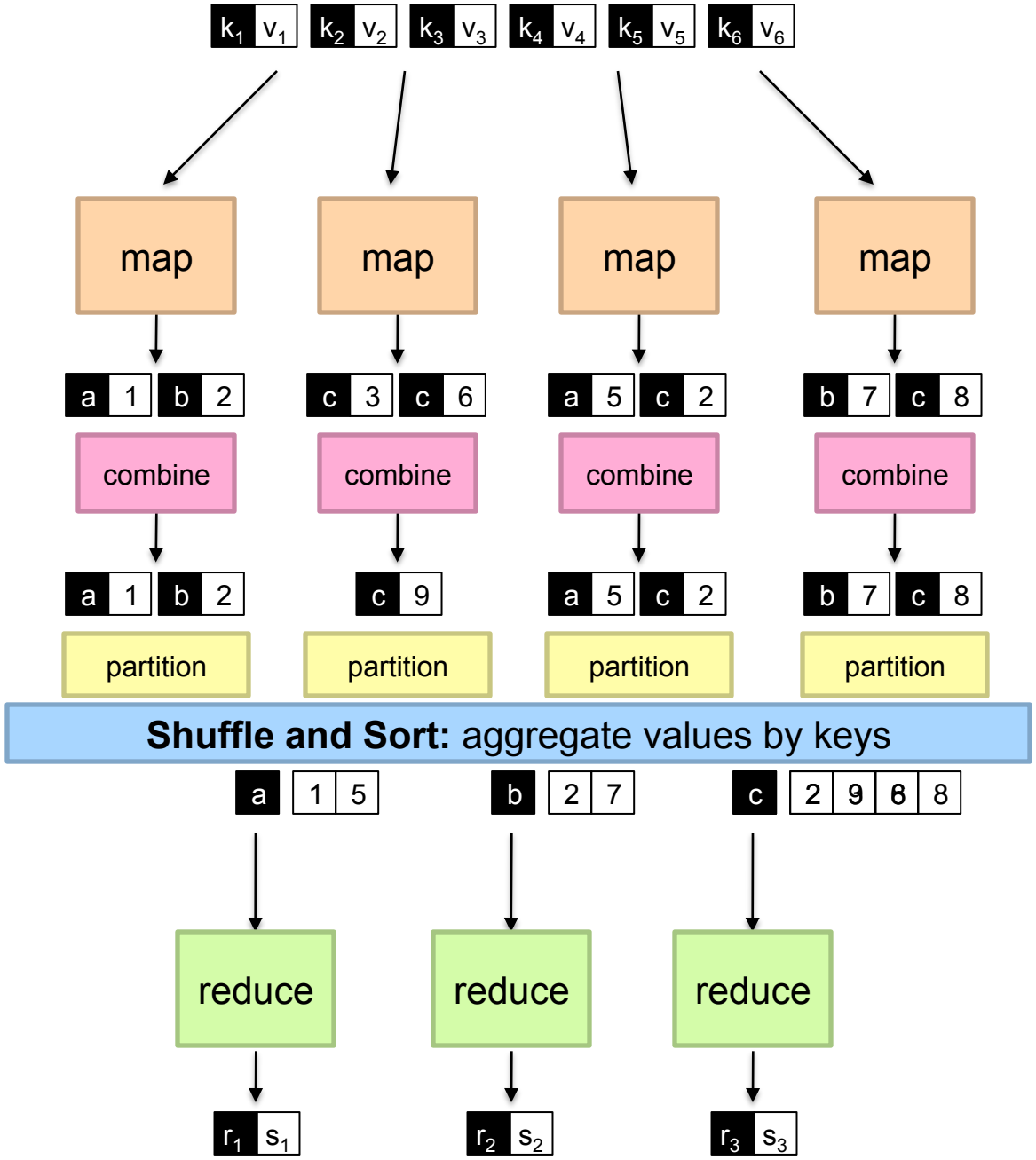
  - All values with the same key are reduced together
- The execution framework handles everything else…
- Not quite…usually, programmers also specify:

  partition (k', number of partitions) → partition for k'

  - Often a simple hash of the key, e.g., **hash(k') mod n**
  - Divides up key space for parallel reduce operations
  - **Sometimes useful to override the hash function:**
    - e.g., **hash(hostname(URL)) mod *R*** ensures URLs from a host end up in the same output file

  combine (k', v') → <k', v'>*

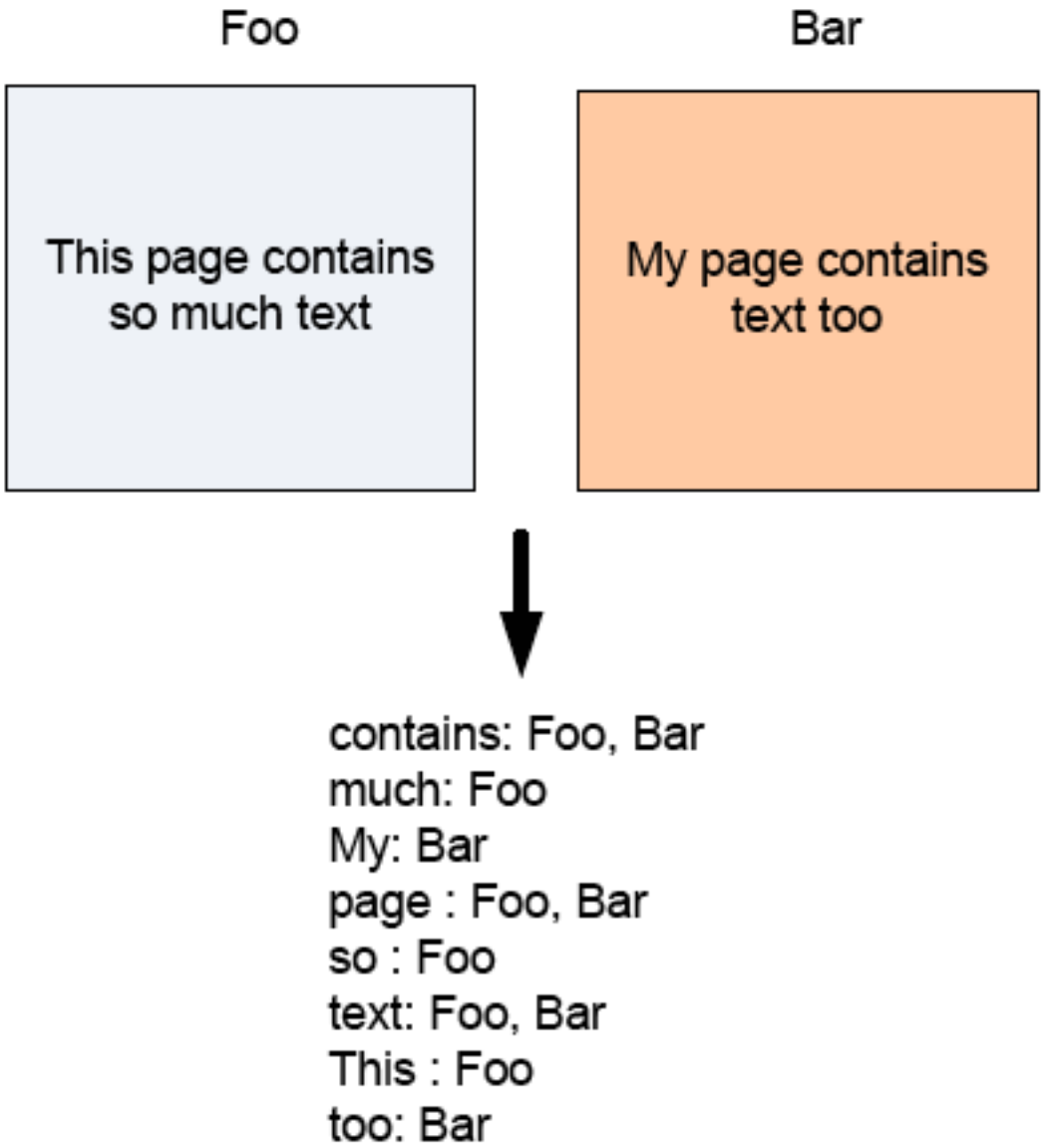
  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic
  - Works only if Reduce function is Commutative and Associative


k1 v1
k2 v2
k3 v3
k4 v4
k5 v5
k6 v6
map
map
map
map
a 1
b 2
c 3
c 6
a 5
c 2
b 7
c 8
combine
combine
combine
combine
a 1
b 2
c 9
a 5
c 2
b 7
c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

Two more details…

- Barrier between map and reduce phases
  - But we can begin copying intermediate data earlier
- Keys arrive at each reducer in sorted order
  - No enforced ordering *across* reducers

Example 2: Inverted Index (for a Search Engine)



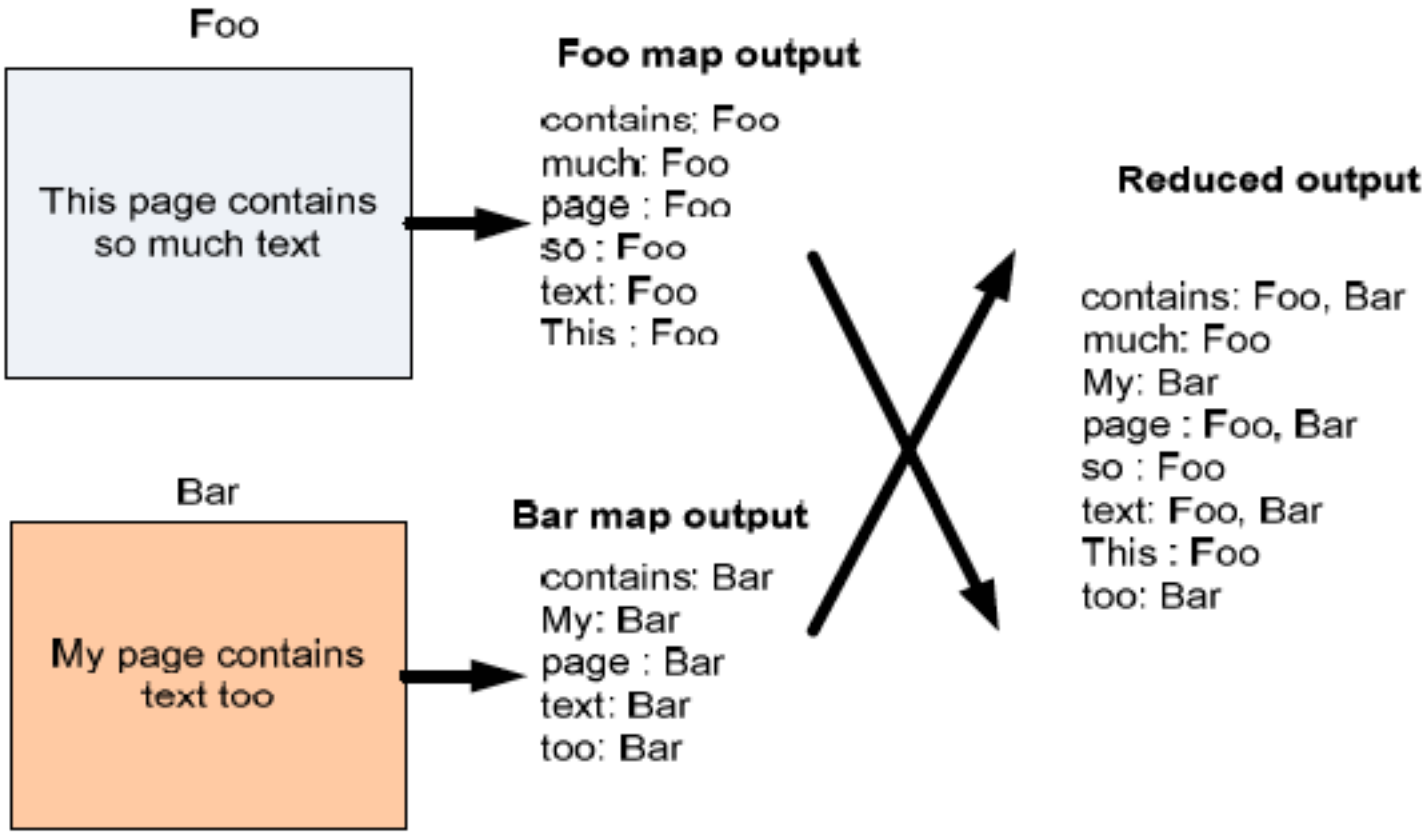
Inverted Index with MapReduce

- Mapper:
  - Key: PageName // URL of webpage
  - Value: Text // text in the webpage

```
foreach word w in Text
   EmitIntermediate(w, PageName)
```

- Reducer:
  - Key: word
  - Values: all URLs for word
  - … Just the Identity function

Inverted Index Data flow w/ MapReduce



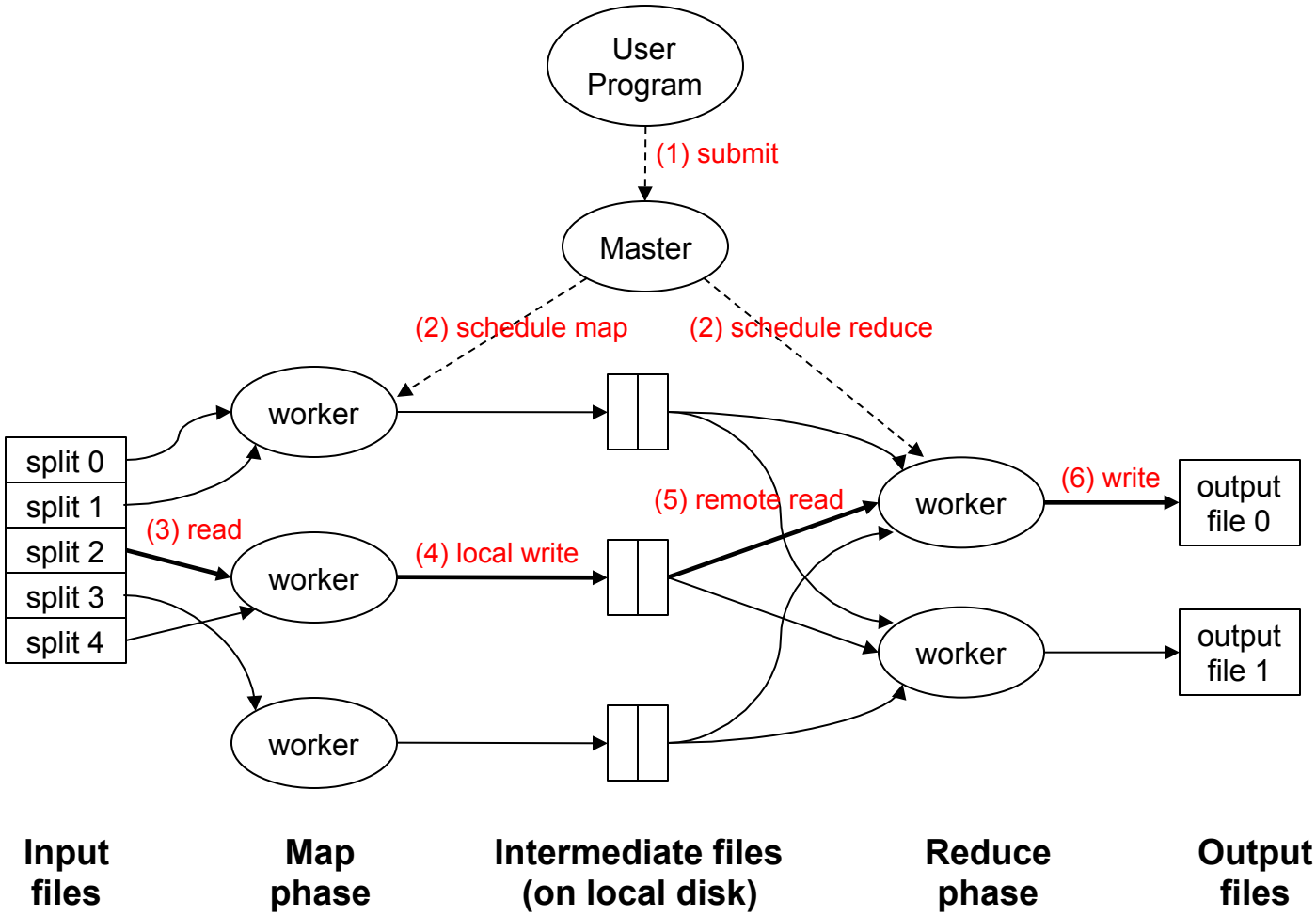
MapReduce can refer to…

- The programming model
- The execution framework (aka “runtime”)
- The specific implementation

Usage is usually clear from context!

MapReduce Implementations

- Google has a proprietary implementation in C++
  - Bindings in Java, Python
- Hadoop is an open-source implementation in Java
  - Development led by Yahoo, used in production
  - Now an Apache project
  - Rapidly expanding software ecosystem
- Lots of custom research implementations
  - For GPUs, cell processors, etc.


User Program
(1) submit
Master
(2) schedule map
(2) schedule reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disk)
Reduce phase
Output files

Data Flow

- **Input and final output are stored on a distributed file system (FS):**
  - Scheduler tries to schedule map tasks “close” to physical storage location of input data

- **Intermediate results are stored on local FS of Map and Reduce workers**

- **Output is often input to another MapReduce task**

Coordination: Master

- **Master node takes care of coordination:**
  - **Task status:** (idle, in-progress, completed)
  - **Idle tasks** get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its *R* intermediate files, one for each reducer
  - Master pushes this info to reducers

- Master pings workers periodically to detect failures

Dealing with Failures

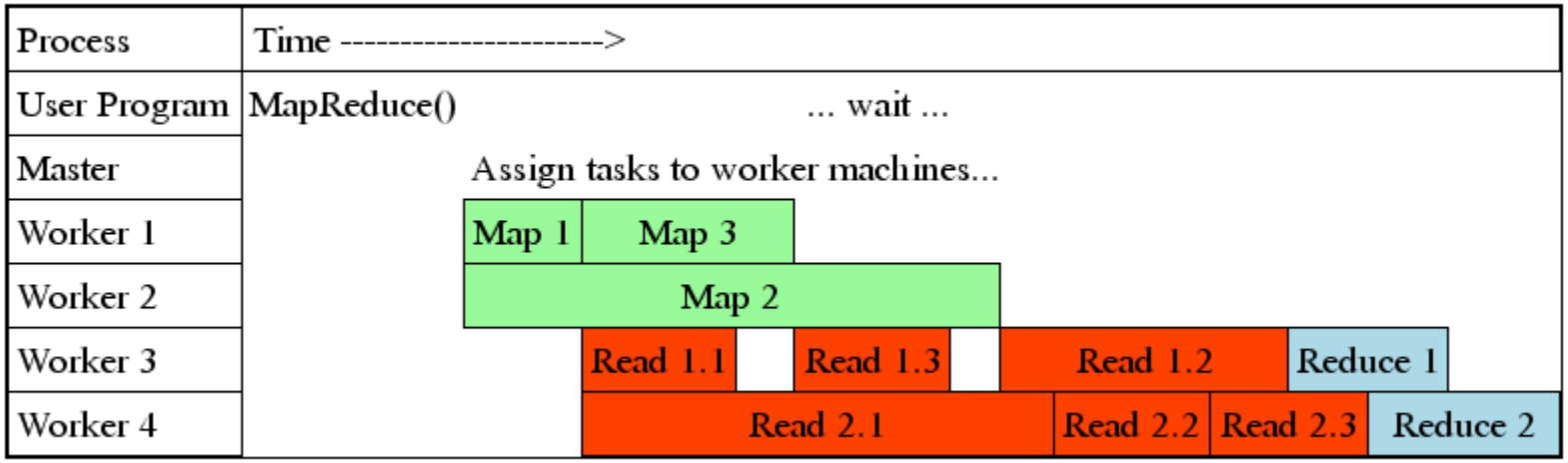
- **Map worker failure**
  - Map tasks completed (Why ??) or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- **Reduce worker failure**
  - Only in-progress tasks are reset to idle
  - Reduce task is restarted
- **Master failure**
  - MapReduce task is aborted and client is notified

How many Map and Reduce jobs?

- *M* map tasks, *R* reduce tasks
- **Rule of a thumb:**
  - Make *M* much larger than the number of nodes in the cluster
  - One DFS chunk (64 Mbyte each by default) per mapper is common
  - Improves dynamic load balancing and speeds up recovery from worker failures
- **Usually *R* is smaller than *M***
  - Because output is spread across *R* files

Task Granularity & Pipelining

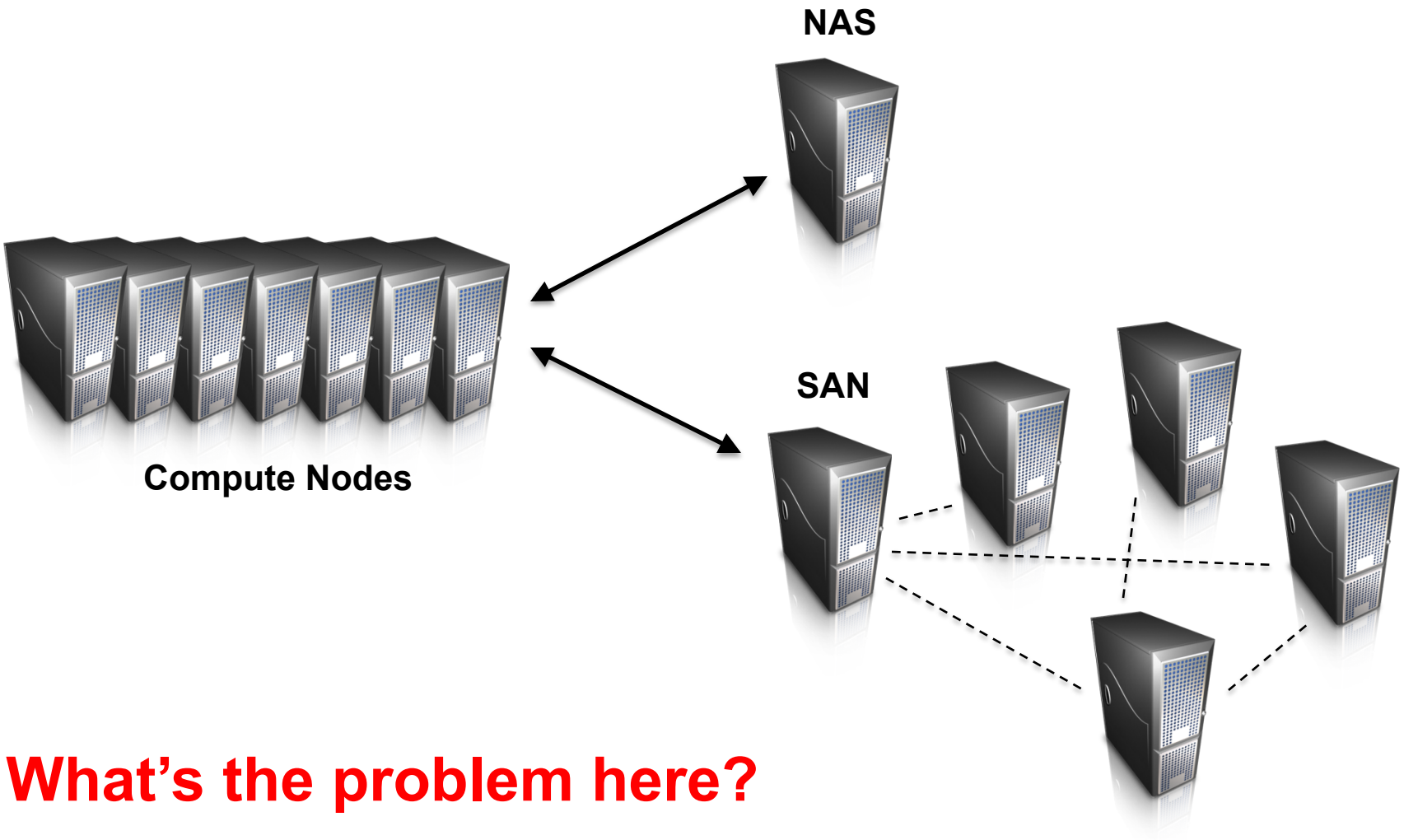
- **Fine granularity tasks:** # of map tasks >> machines
  - Minimizes time for fault recovery
  - Can do pipeline shuffling with map execution
  - Better dynamic load balancing
  - e.g. For 2000 processors, ***M*** = 200,000 ; ***R*** = 5000



Refinements: Backup Tasks

- **Problem**
  - Slow workers significantly lengthen the job completion time:
    - Other jobs on the machine
    - Bad disks
    - Weird things
- **Solution**
  - Near end of phase, spawn backup copies of tasks
    - Whichever one finishes first “wins”
- **Effect**
  - Dramatically shortens job completion time

How do we get data to the workers?



What's the problem here?

Distributed File System

- Don't move data to workers… move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- Why?
  - Not enough RAM to hold all the data in memory
  - Disk access is slow, but disk throughput is reasonable
- A distributed file system is the answer
  - GFS (Google File System) for Google's MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop
  - Non-starters
    - Lustre (high bandwidth, but no replication outside racks)
    - Gluster (POSIX, more classical mirroring, see Lustre)
    - NFS/AFS/whatever - doesn't actually parallelize

GFS: Assumptions

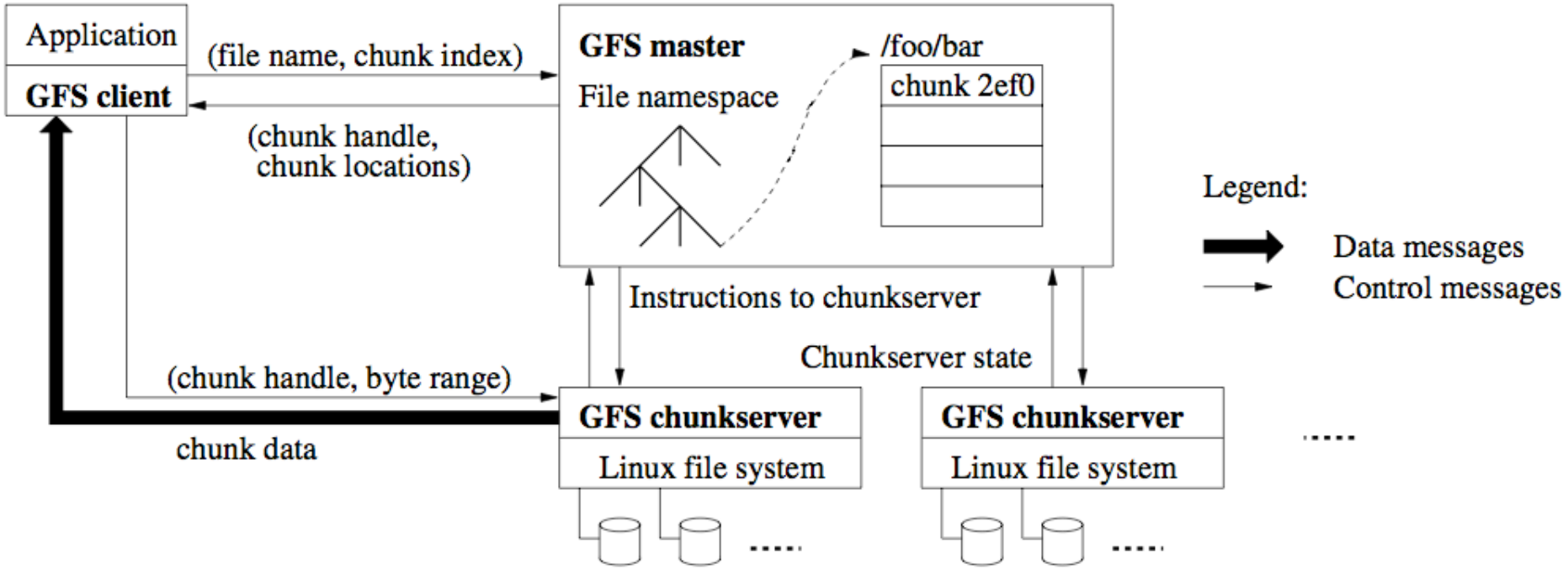
- Commodity hardware over “exotic” hardware
  - Scale “out”, not “up”
- High component failure rates
  - Inexpensive commodity components fail all the time
- “Modest” number of huge files
  - Multi-gigabyte files are common, if not encouraged
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads over random access
  - High sustained throughput over low latency

GFS: Design Decisions

- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ chunkservers
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large datasets, streaming reads
- Simplify the API
  - Push some of the issues onto the client (e.g., data layout)

HDFS = GFS clone (same basic ideas)

Google File System

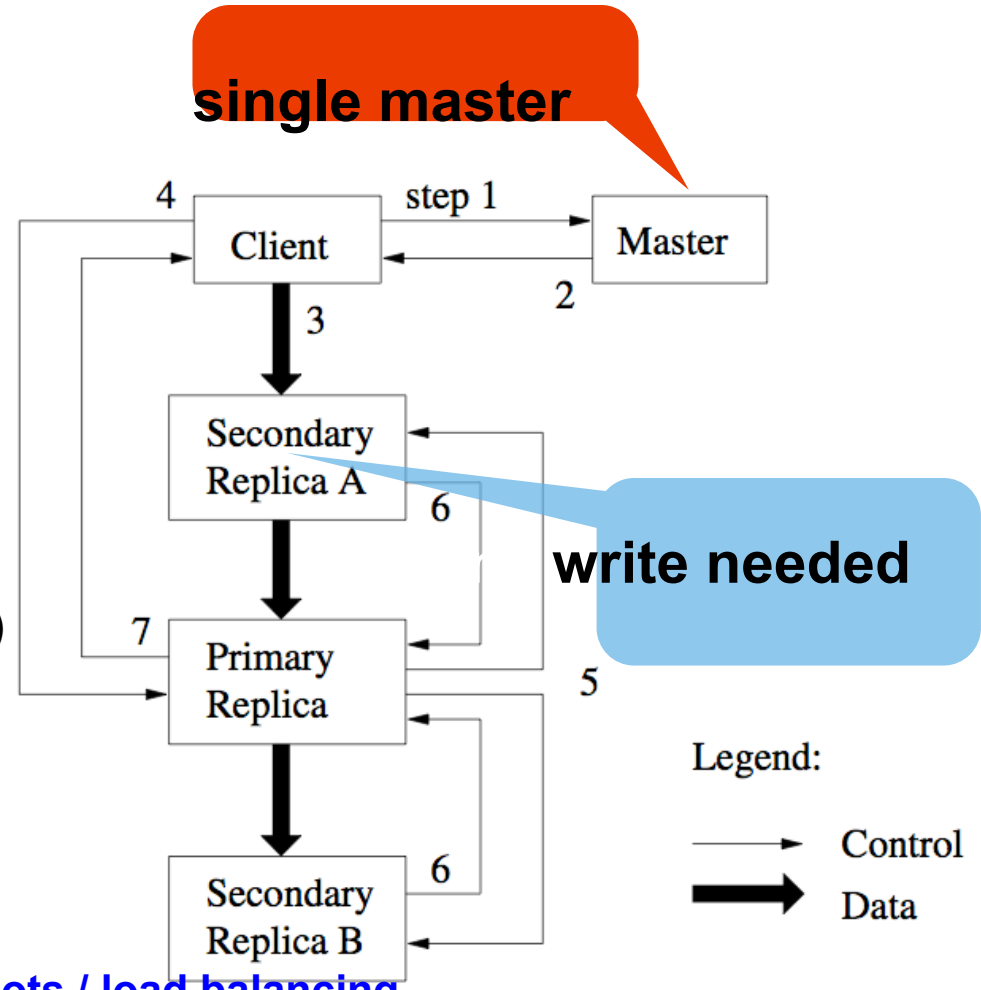


Ghemawat, Gobioff, Leung, 2003

- Chunk servers hold blocks of the file (64MB per chunk)
- Replicate chunks (chunk servers do this autonomously). More bandwidth and fault tolerance
- Master distributes, checks faults, rebalances (Achilles heel)
- Client can do bulk read / write / random reads

Google File System /HDFS

1. Client requests chunk from master
2. Master responds with replica location
3. Client writes to replica A
4. Client notifies primary replica
5. Primary replica requests data from replica A
6. Replica A sends data to Primary replica (same process for replica B)
7. Primary replica confirms write to client



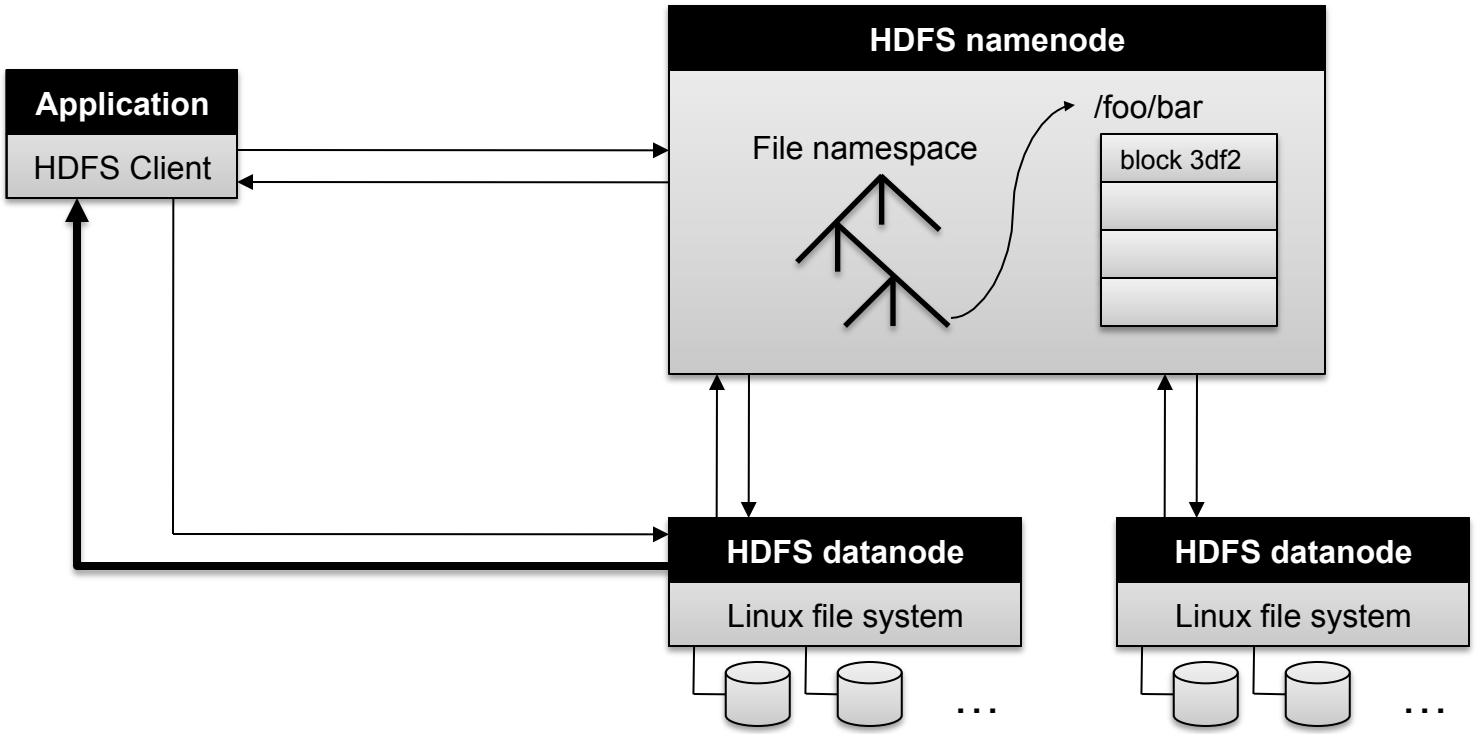
- Master ensures nodes are live
- Chunks are checksummed
- Can control replication factor for hotspots / load balancing
- Deserialize master state by loading data structure as flat file from disk (fast) ; *See Section 4.1 of GFS SOSP2003 paper for details*

From GFS to HDFS

- Terminology differences:
  - GFS master = Hadoop namenode
  - GFS chunkservers = Hadoop datanodes
- Functional differences:
  - Initially, no file appends in HDFS (the feature has been added recently)
    - http://blog.cloudera.com/blog/2009/07/file-appends-in-hdfs/
    - http://blog.cloudera.com/blog/2012/01/an-update-on-apache-hadoop-1-0/
  - HDFS performance is (likely) slower

For the most part, we'll use the Hadoop terminology…

HDFS Architecture

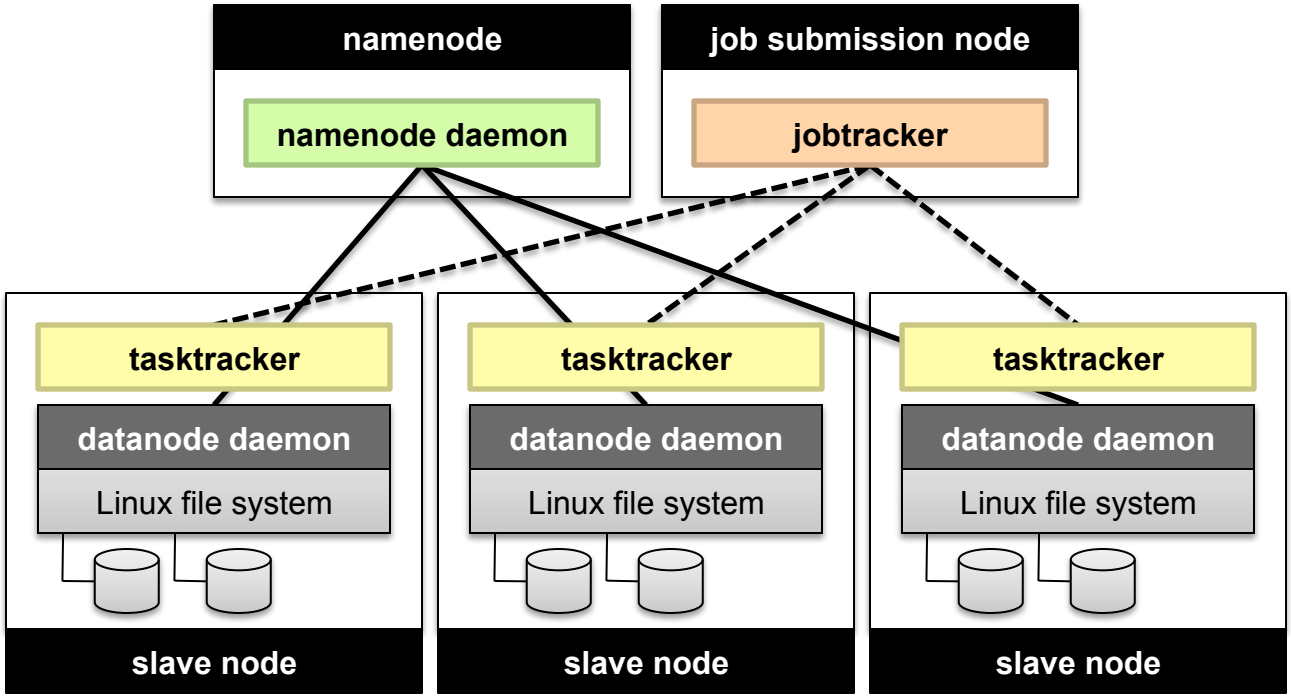


Adapted from (Ghemawat et al., SOSP 2003)

Namenode Responsibilities

- Managing the file system namespace:
  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.
- Coordinating file operations:
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - Periodic communication with the datanodes
  - Block re-replication and rebalancing
  - Garbage collection
- Namenode can be Archille's heel – Single point of failure or bottleneck of scalability for the entire FS:
  - Need to have a Backup Namenode HDFS (or Master in GFS)
  - Compared to the fully-distributed approach in Ceph

Putting everything together…



Sample Use of MapReduce

More MapReduce Example: Host size

- **Suppose we have a large web corpus**
- Look at the metadata file
  - Lines of the form: (URL, size, date, …)
- **For each host, find the total number of bytes**
  - That is, the sum of the page sizes for all URLs from that particular host
- **Other examples:**
  - Link analysis and graph processing
  - Machine Learning algorithms
  - More later in the course…

Another Example: Language Model

- **Statistical machine translation:**
  - Need to count number of times every 5-word sequence occurs in a large corpus of documents

- **With MapReduce:**
  - **Map:**
    - Extract (5-word sequence, count) from document
  - **Reduce:**
    - Combine the counts

Example: Join By Map-Reduce

- **Compute the natural join *R(A,B)* ⋈ *S(B,C)***
- *R* and *S* are each stored in files
- Tuples are pairs *(a,b)* or *(b,c)*

A	B
a_1	b_1
a_2	b_1
a_3	b_2
a_4	b_3

R

⋈

B	C
b_2	c_1
b_2	c_2
b_3	c_3

S

=

A	C
a_3	c_1
a_3	c_2
a_4	c_3

Map-Reduce Join

- **Use a hash function *h* from B-values to *1...k***
- **A Map process turns:**
  - Each input tuple *R(a,b)* into key-value pair *(b,(a,R))*
  - Each input tuple *S(b,c)* into *(b,(c,S))*
- **Map processes** send each key-value pair with key *b* to Reduce process *h(b)*
  - Hadoop does this automatically; just tell it what *k* is.
- Each **Reduce process** matches all the pairs *(b,(a,R))* with all *(b,(c,S))* and outputs *(a,b,c)*.

MapReduce is good for…

- *Embarrassingly Parallel* algorithms
- Summing, grouping, filtering, joining
- Off-line batch jobs on massive data sets
- Analyzing an entire large data set
  - New higher level languages/systems have been developed to further simplify data processing using MapReduce
    - Declarative description (NoSQL type) of processing task can be translated automatically to MapReduce functions
    - Control flow of processing steps (Pig)

MapReduce is OK, (and only ok) for…

- Iterative jobs (e.g. Graph algorithms like Pagerank)
  - Each iteration must read/write data to disk
  - I/O and latency cost of an iteration is high

MapReduce is NOT good for…

- Jobs that need shared state/ coordination
  - Tasks are shared-nothing
  - Shared-state requires scalable state store
- Low-latency jobs
- Jobs on small datasets
- Finding individual records

For some of these, we will introduce alternative computational models/ platforms, e.g. GraphLab, Spark, later in the course

Practical Limits of Hadoop1.0

- Scalability
  - Maximum Cluster Size – 4000 Nodes
  - Maximum Concurrent Tasks – 40000
  - Coarse synchronization in Job Tracker
- Single point of failure
  - Failure kills all queued and running jobs
  - Jobs need to be resubmitted by users
- Restart is very tricky due to complex state

Scalability/Flexibility Issues of the MapReduce/ Hadoop 1.0 Job Scheduling/Tracking

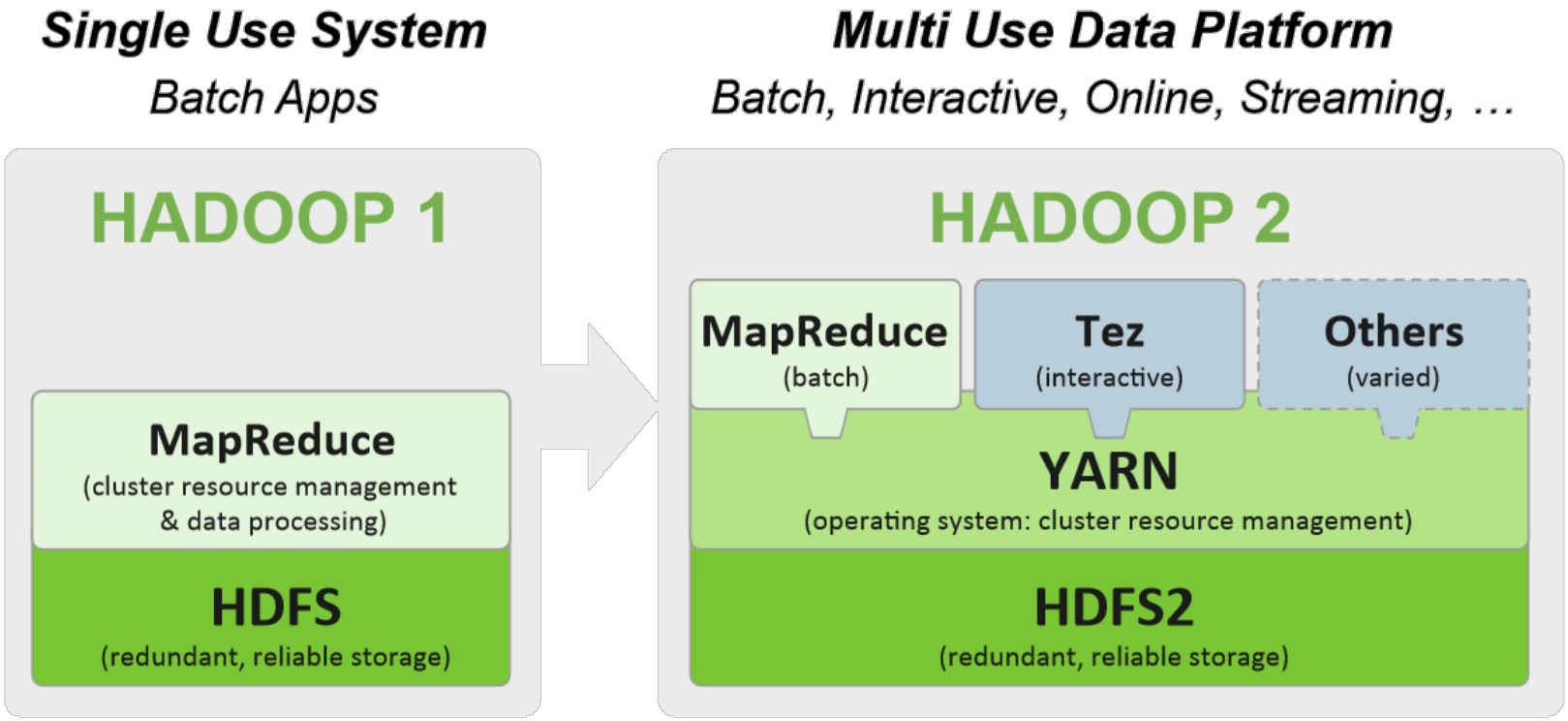
- The MapReduce Master node (or Job-tracker in Hadoop 1.0) is responsible to monitor the progress of ALL tasks of all jobs in the system and launch backup/replacement copies in case of failures
  - For a large cluster with many machines, the number of tasks to be tracked can be huge

  => Master/Job-Tracker node can become the performance bottleneck
- Hadoop 1.0 platform focuses on supporting MapReduce as its only computational model ; may not fit all applications
- Hadoop 2.0 introduces a new resource management/ job-tracking architecture, YARN [1], to address these problems

[1] V.K. Vavilapalli, A.C.Murthy, “Apache Hadoop YARN: Yet Another Resource Negotiator,” ACM Symposium on Cloud Computing 2013.

Resource Management Platforms for Big Data Processing Clusters

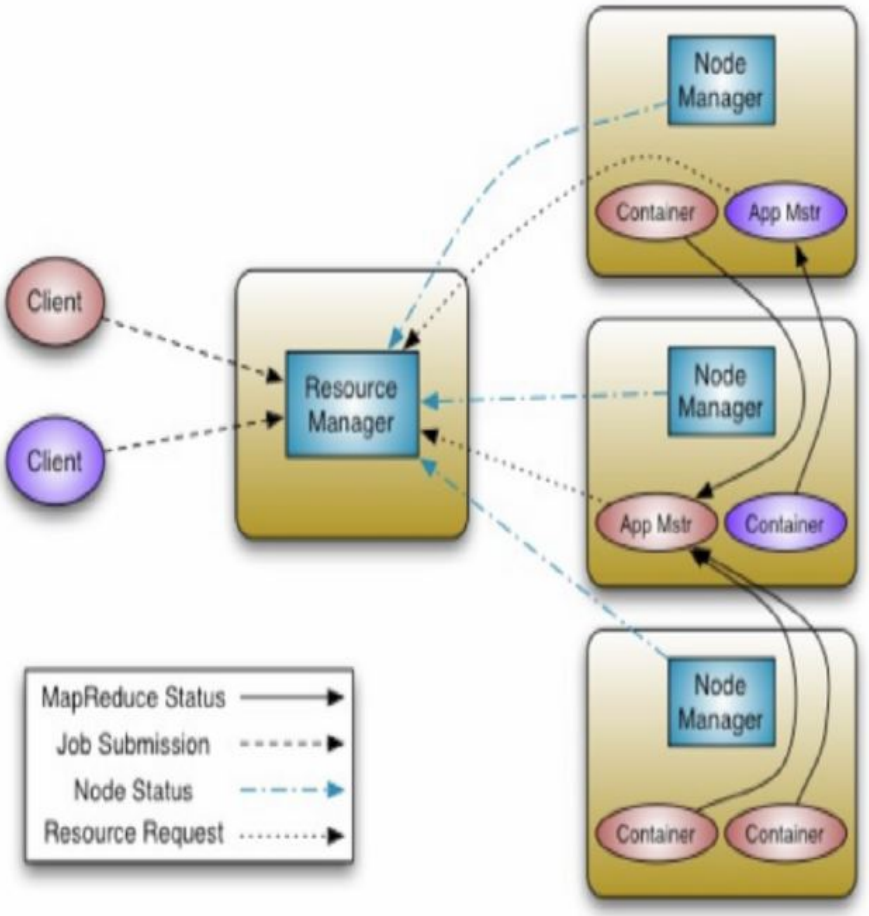
YARN for Hadoop 2.0



- YARN (Yet Another Resource Negotiator) provides a resource management platform for Cluster to support general Distributed/Parallel Applications/Frameworks beyond the MapReduce computational model.

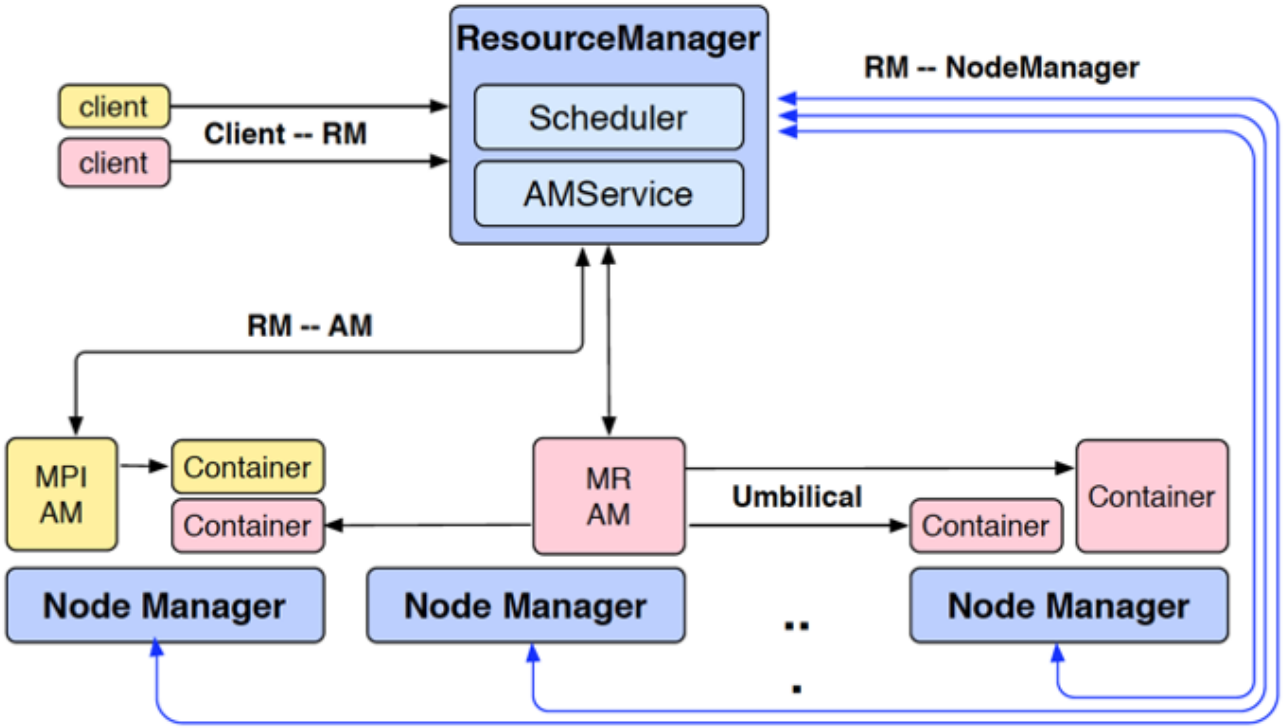
V. K. Vavilapalli, A. C. Murthy, “Apache Hadoop YARN: Yet Another Resource Negotiator”, in ACM Symposium on Cloud Computing (SoCC) 2013.

YARN Architectural Overview



- Scalability - Clusters of 6,000-10,000 machines
  - Each machine with 16 cores, 48G/96G RAM, 24TB/36TB disks
  - 100,000+ concurrent tasks
  - 10,000 concurrent jobs

Cluster Resource Management w/ YARN in Hadoop2.0

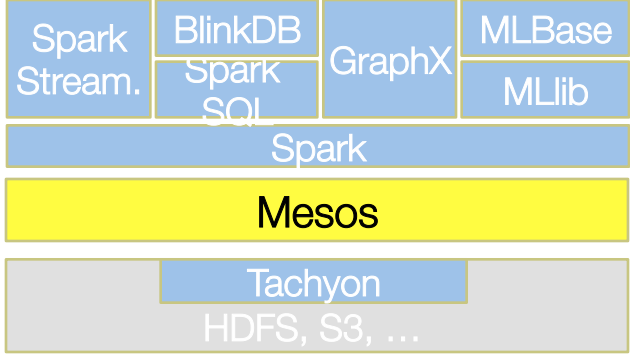


- Multiple frameworks (Applications) can run on top of YARN to share a Cluster, e.g. MapReduce is one framework (Application), MPI, or Storm are other ones.
- YARN splits the functions of JobTracker into 2 components: resource allocation and job-management (e.g. task-tracking/ recovery):
  - Upon launching, each Application will have its own Application Master (AM), e.g. MR-AM in the figure above is the AM for MapReduce, to track its own tasks and perform failure recovery if needed
  - Each AM will request resources from the YARN Resource Manager (RM) to launch the Application's jobs/tasks (Containers in the figure above) ;
  - The YARN RM determines resource allocation across the entire cluster by communicating with/ controlling the Node Managers (NM), one NM per each machine.

Hadoop 2.0 vs. Hadoop1.0

- Hadoop 2.0 includes YARN's Multi-tenant Support for different Big Data Processing Frameworks
- YARN Fault Tolerance and Availability
  - Resource Manager
    - No single point of failure – state saved in ZooKeeper
    - Application Masters are restarted automatically on RM restart
  - Application Master
    - Optional failover via application-specific checkpoint
    - MapReduce applications pick up where they left off via state saved in HDFS
- Wire Compatibility
  - Protocols are wire-compatible
  - Old clients can talk to new servers
  - Rolling upgrades
- Besides YARN, Hadoop 2.0 also supports High Availability and Federation
  - High Availability takes away the Single Point of failure from HDFS Namenode and introduces the concept of the QuorumJournalNodes to sync edit logs between active and standby Namenodes
  - Federation allows multiple independent namespaces (private namespaces, or Hadoop as a service)

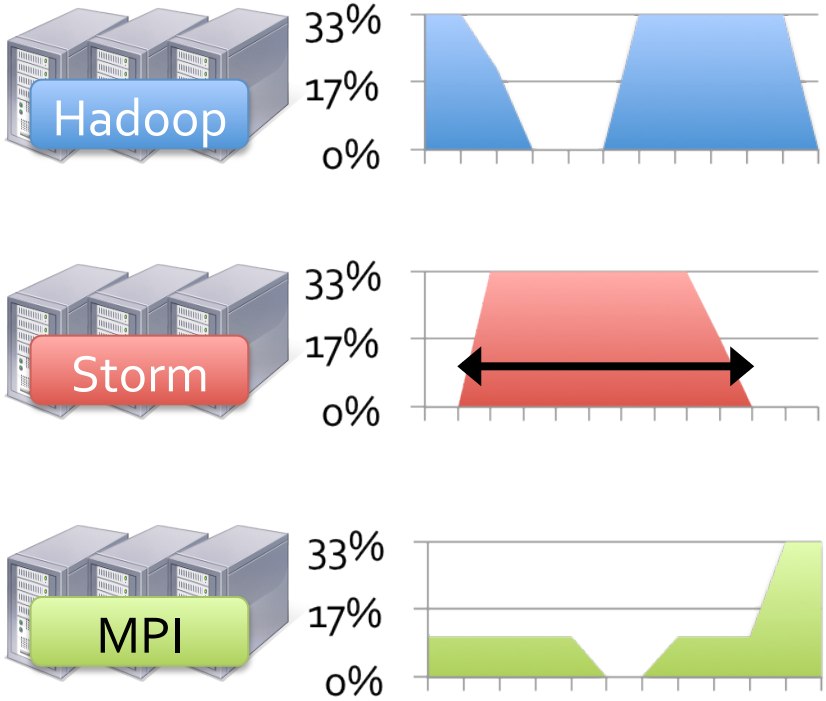
Apache Mesos
(http://mesos.apache.org)



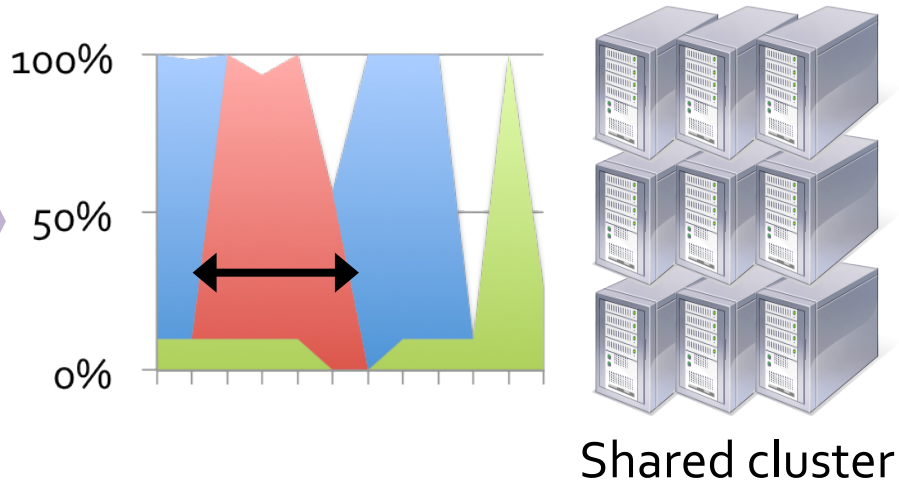
- Another competing Cluster Resource Management software
- Enable multiple frameworks to share same cluster resources (e.g., MapReduce, Storm, Spark, HBase, etc)
- Originated from UCBerkeley's BDAS project ;
  - B. Hindman et al, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center", Usenix NSDI 2011.
- Hardened via Twitter's large scale in-house deployment
  - 6,000+ servers,
  - 500+ engineers running jobs on Mesos
- Third party Mesos schedulers
  - AirBnB's Chronos ; Twitter's Aurora
- Mesospehere: startup to commercialize Mesos

Motivation of Mesos

Previously: Static partitioning of a cluster among different big data processing frameworks



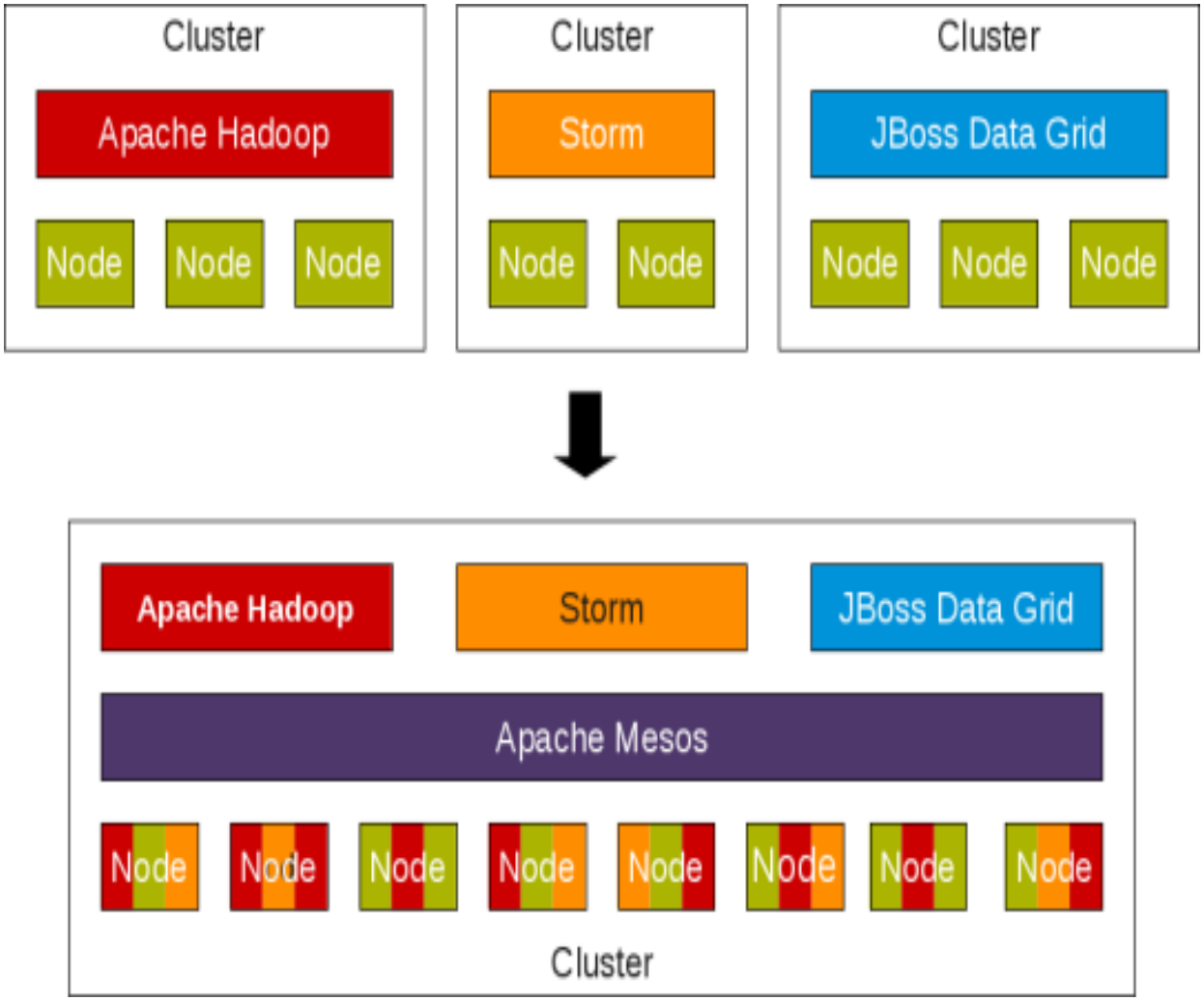
Mesos aims to achieve dynamic sharing of cluster across different frameworks



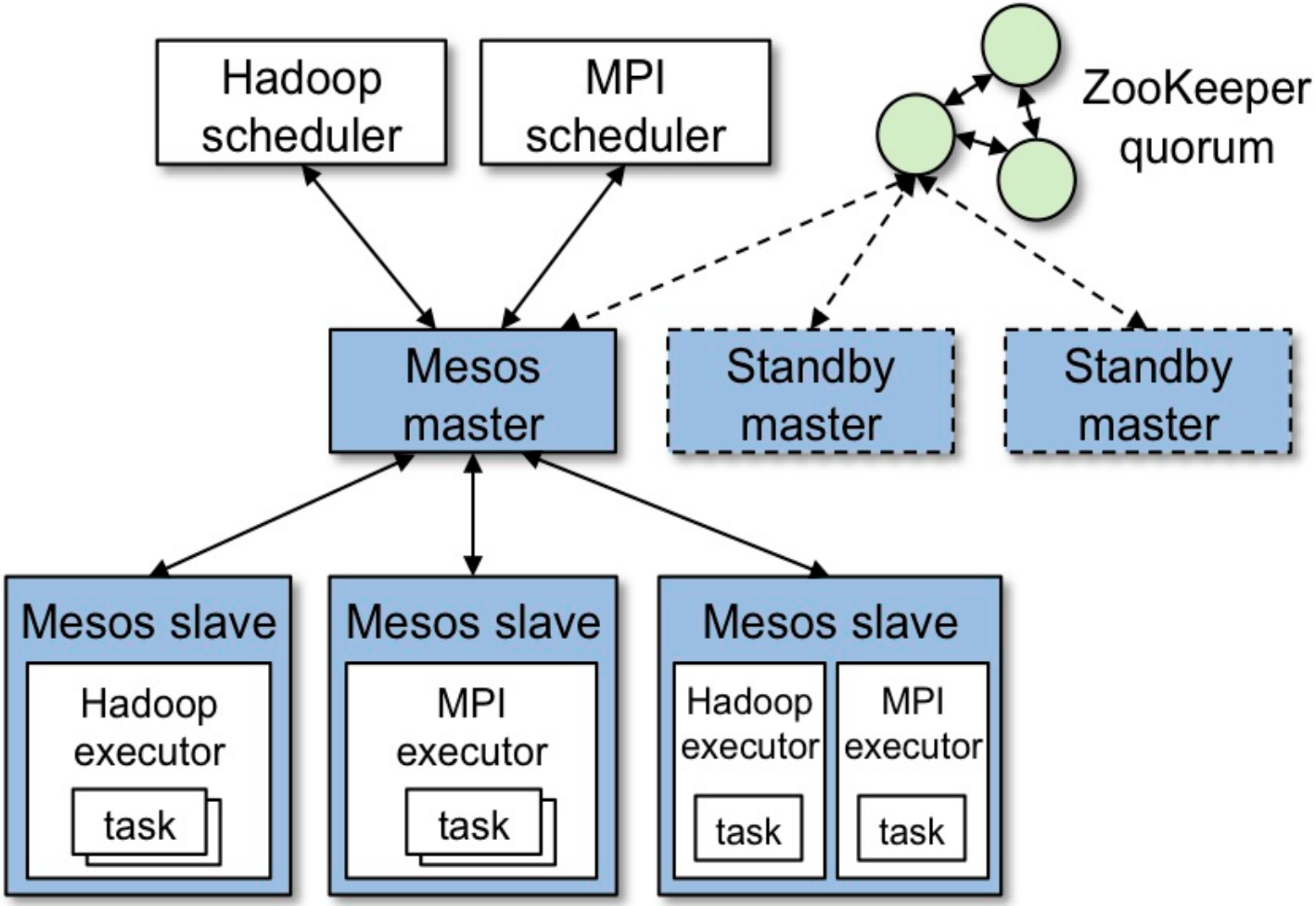
- Hard to fully utilize machines (e.g., X GB RAM & Y CPUs)
- Hard to scale elastically (to take advantage of statistical multiplexing)
- Hard to deal with failures

Mesos as a Data-Center "Kernel"

- Mesos provides a Node Abstraction of the entire Cluster
- Mesos is a common resource sharing layer over which diverse frameworks can run



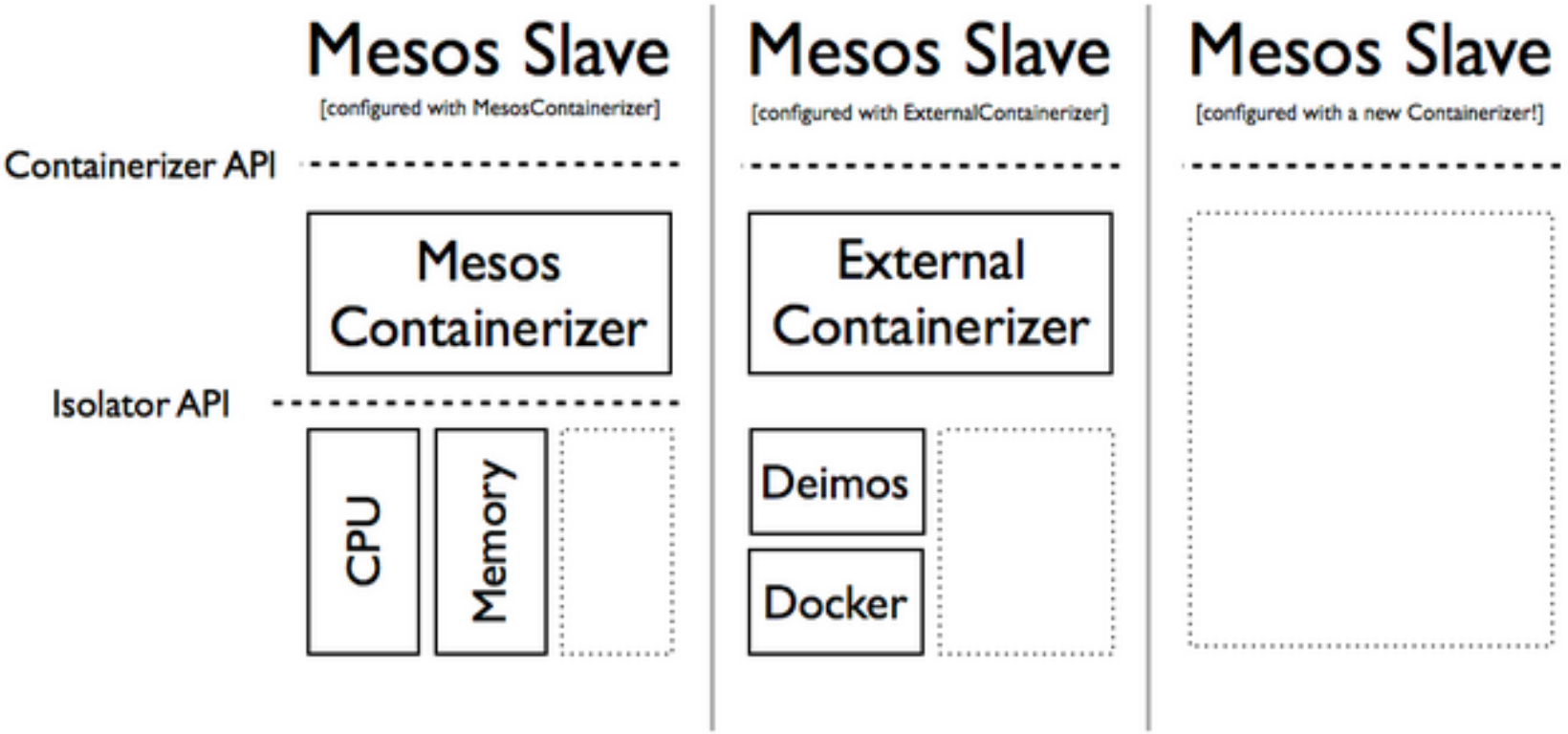
System Architecture of Mesos



Framework Isolation

- Mesos uses OS isolation mechanisms, such as Linux containers and Solaris projects
- Containers currently support CPU, memory, IO and network bandwidth isolation
- Not perfect, but much better than no isolation

Mesos' use of Container Technology

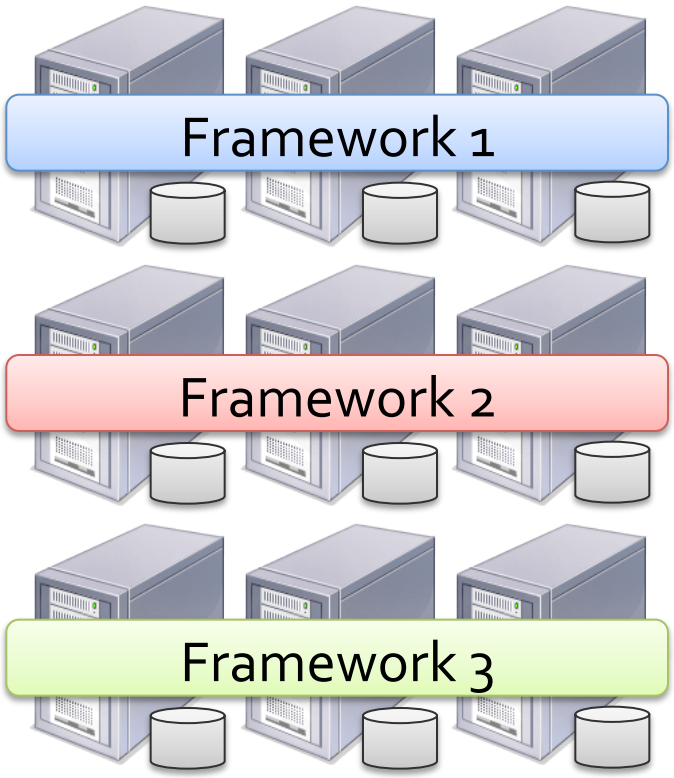


Design Elements

- Fine-grained sharing:
  - Allocation at the level of *tasks* within a job
  - Improves utilization, latency, and data locality
- Resource offers:
  - Simple, scalable application-controlled scheduling mechanism

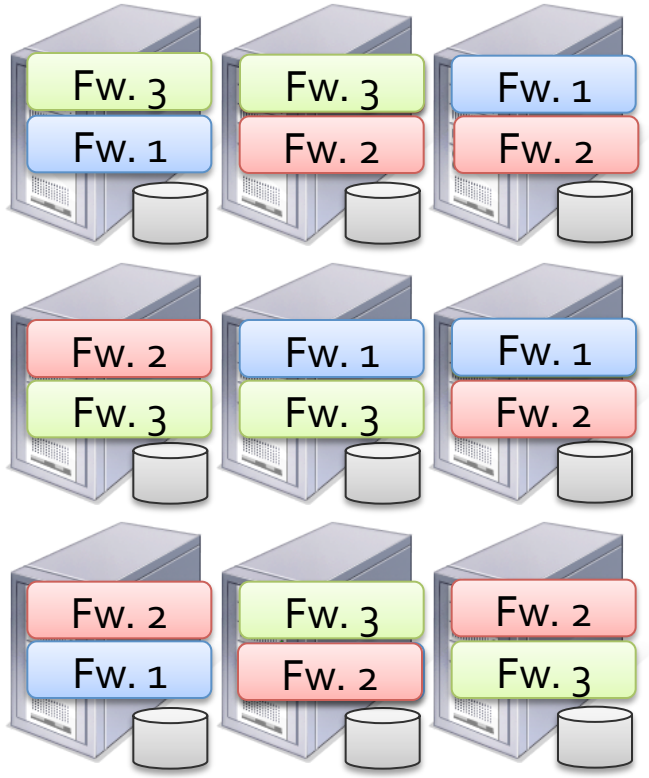
Element 1: Fine-Grained Sharing

Coarse-Grained Sharing (HPC):



Storage System (e.g. HDFS)

Fine-Grained Sharing (Mesos):



Storage System (e.g. HDFS)

+ Improved utilization, responsiveness, data locality

Element 2: Resource Offers

- Option: Global scheduler
  - Frameworks express needs in a specification language, global scheduler matches them to resources

+ Can make optimal decisions

- – Complex: language must support all framework needs

– Difficult to scale and to make robust

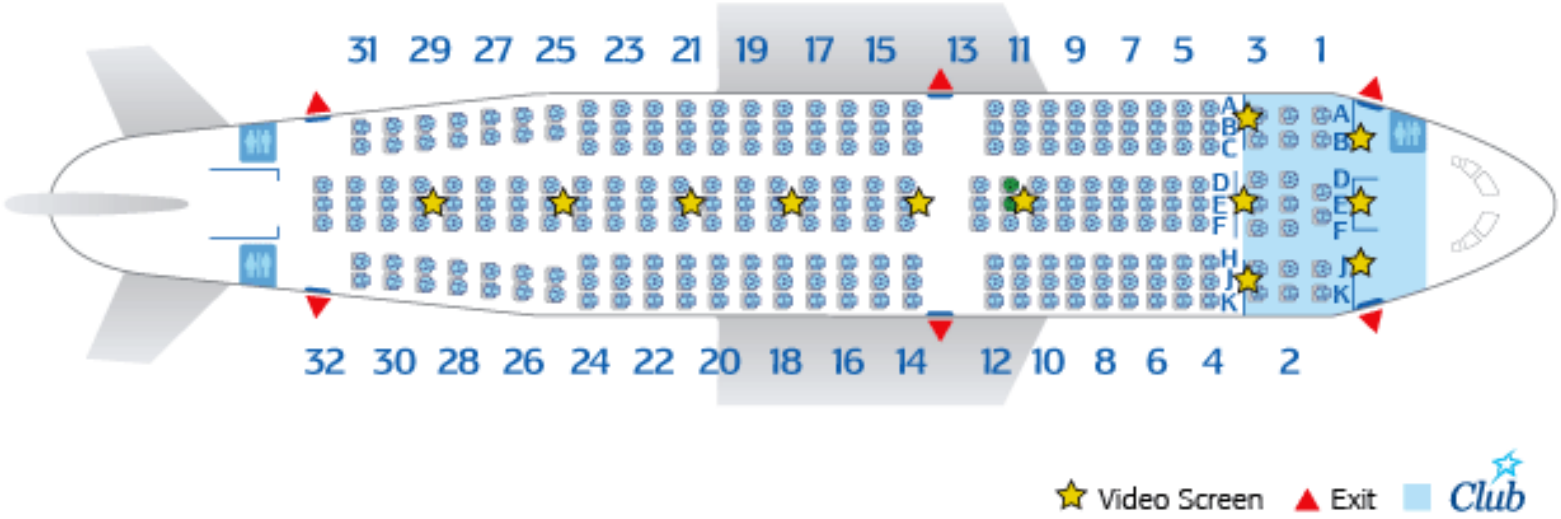
– Future frameworks may have unanticipated needs

Element 2: Resource Offers

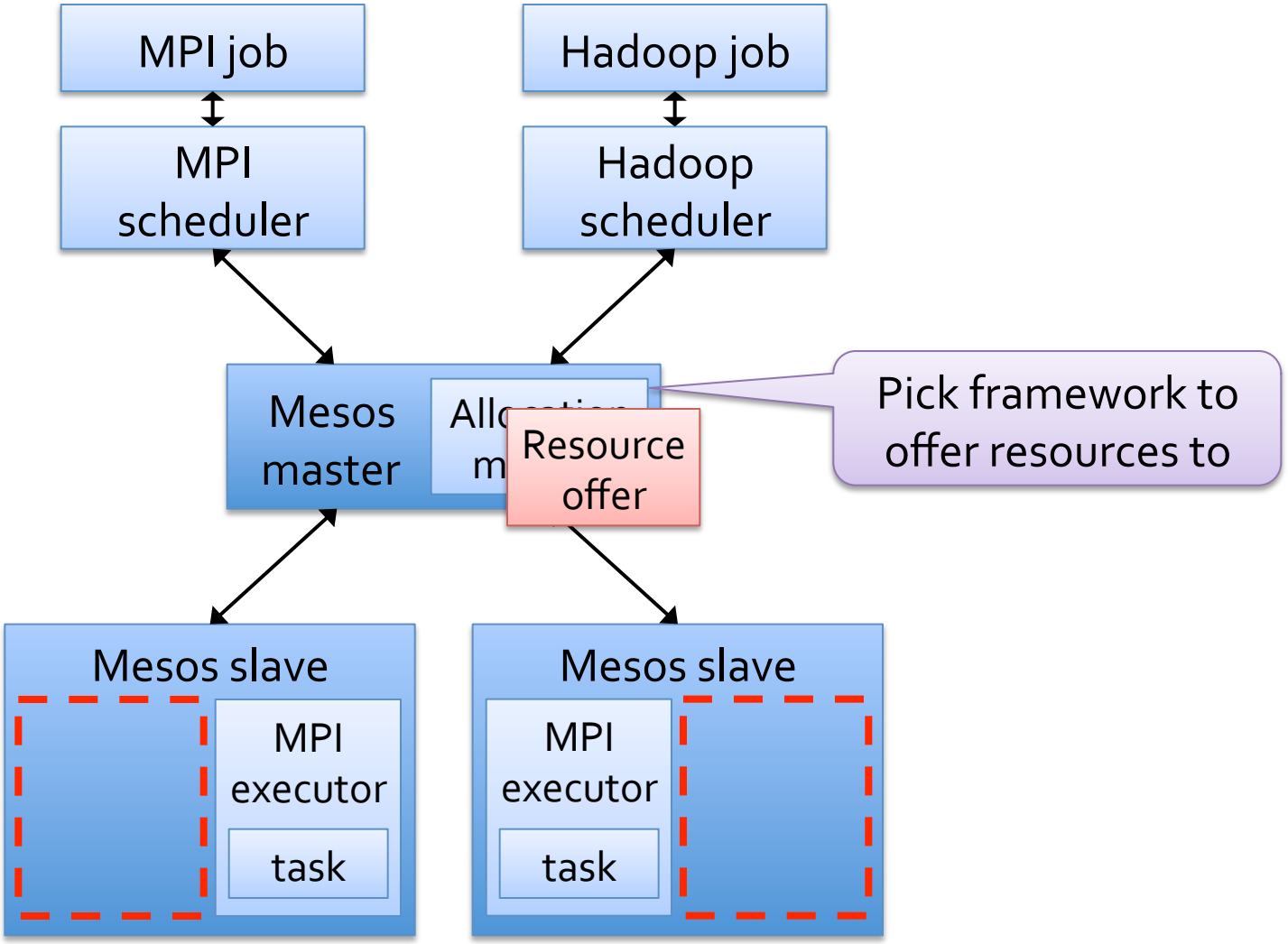
- Mesos: Resource offers
  - Offer available resources to frameworks, let them pick which resources to use and which tasks to launch

  \+ Keeps Mesos simple, lets it support future frameworks

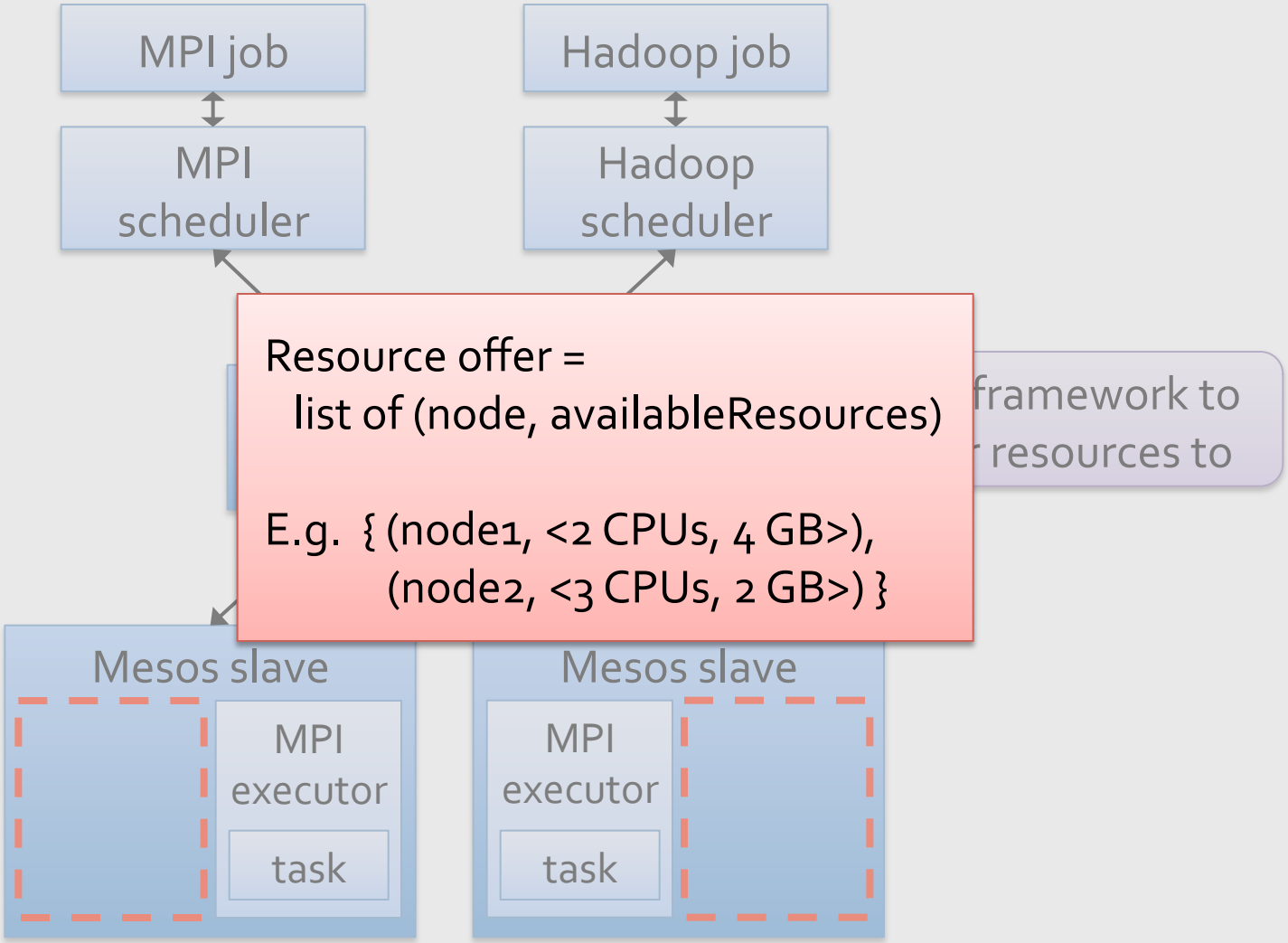
  – Decentralized decisions might not be optimal



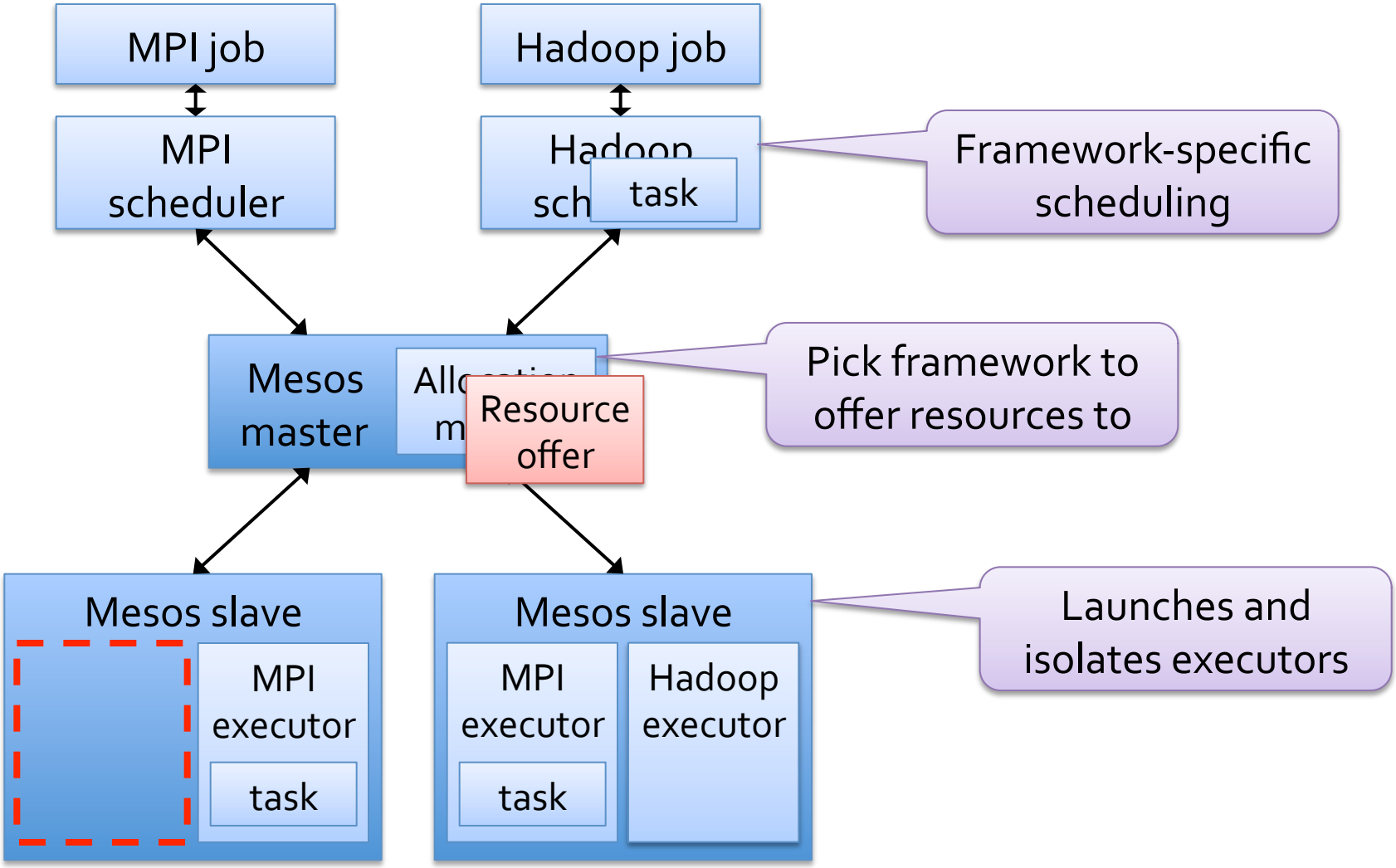
Mesos Architecture



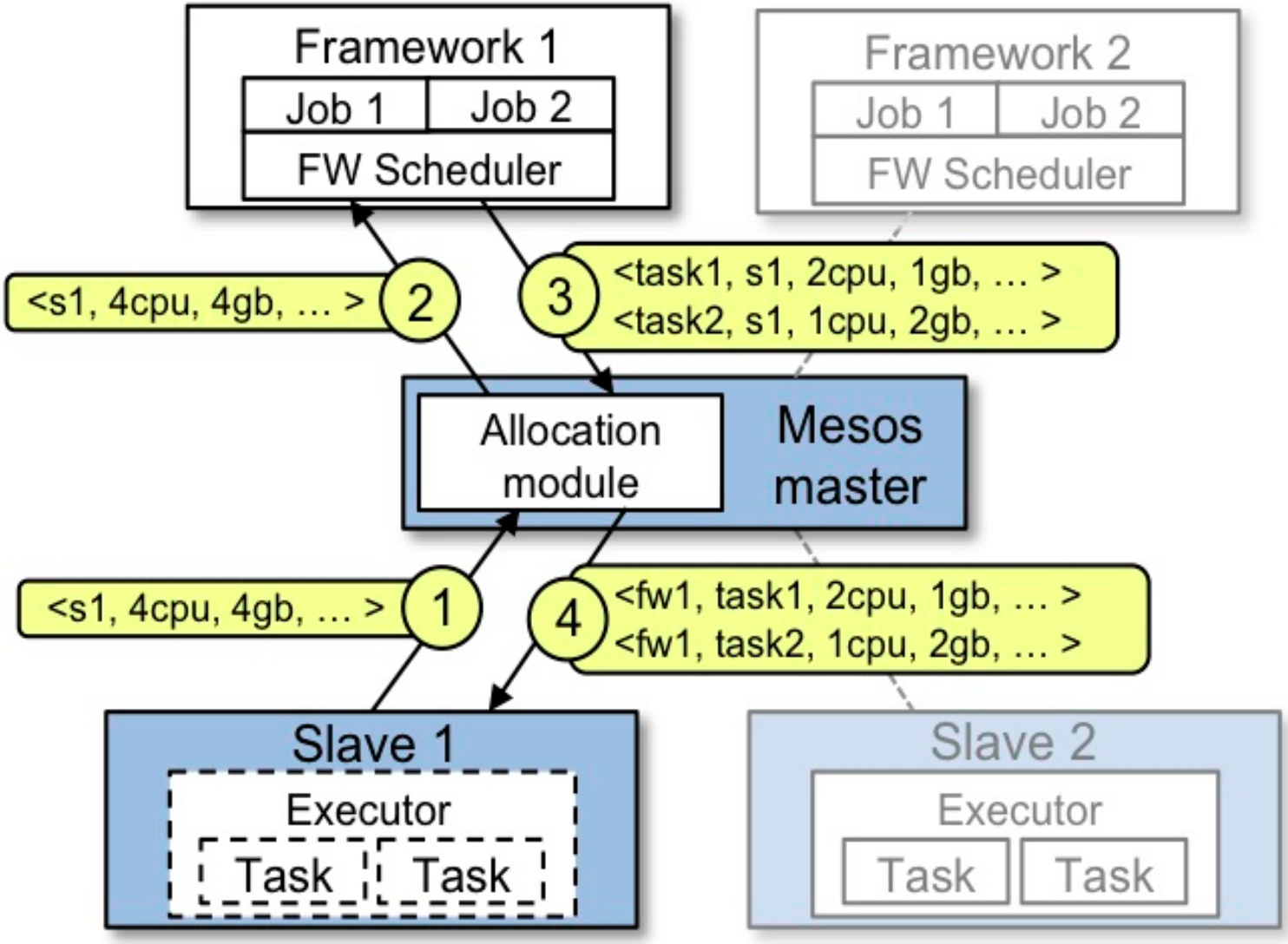
Mesos Architecture



Mesos Architecture



Another Resource Offering Example



Optimization: Filters

- Let frameworks short-circuit rejection by providing a predicate on resources to be offered
  - » E.g. "nodes from list L" or "nodes with > 8 GB RAM"
  - » Could generalize to other hints as well
- Ability to reject still ensures *correctness* when needs cannot be expressed using filters

Revocation

- Mesos allocation modules can revoke (kill) tasks to meet organizational SLOs
- Framework given a grace period to clean up
- "Guaranteed share" API lets frameworks avoid revocation by staying below a certain share

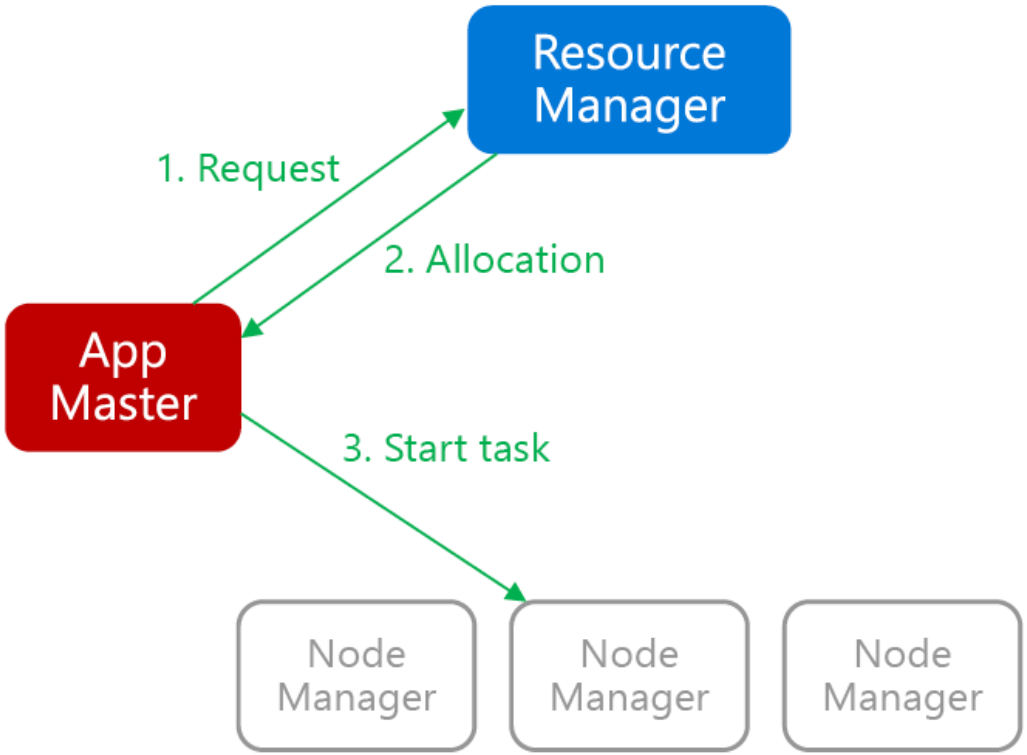
Mesos Implementation Statistics

- 20,000 lines of C++
- Master failover using ZooKeeper
- Frameworks ported: Hadoop1.0, MPI, Storm, etc
- New specialized framework: Spark, for iterative jobs
  (up to 20× faster than Hadoop)
- Open source under Apache license

Other Schedulers/ Resource Management Platforms for Big Data Processing Clusters

Approach 1: Centralized Resource Management

[YARN, Mesos, Omega, Borg]



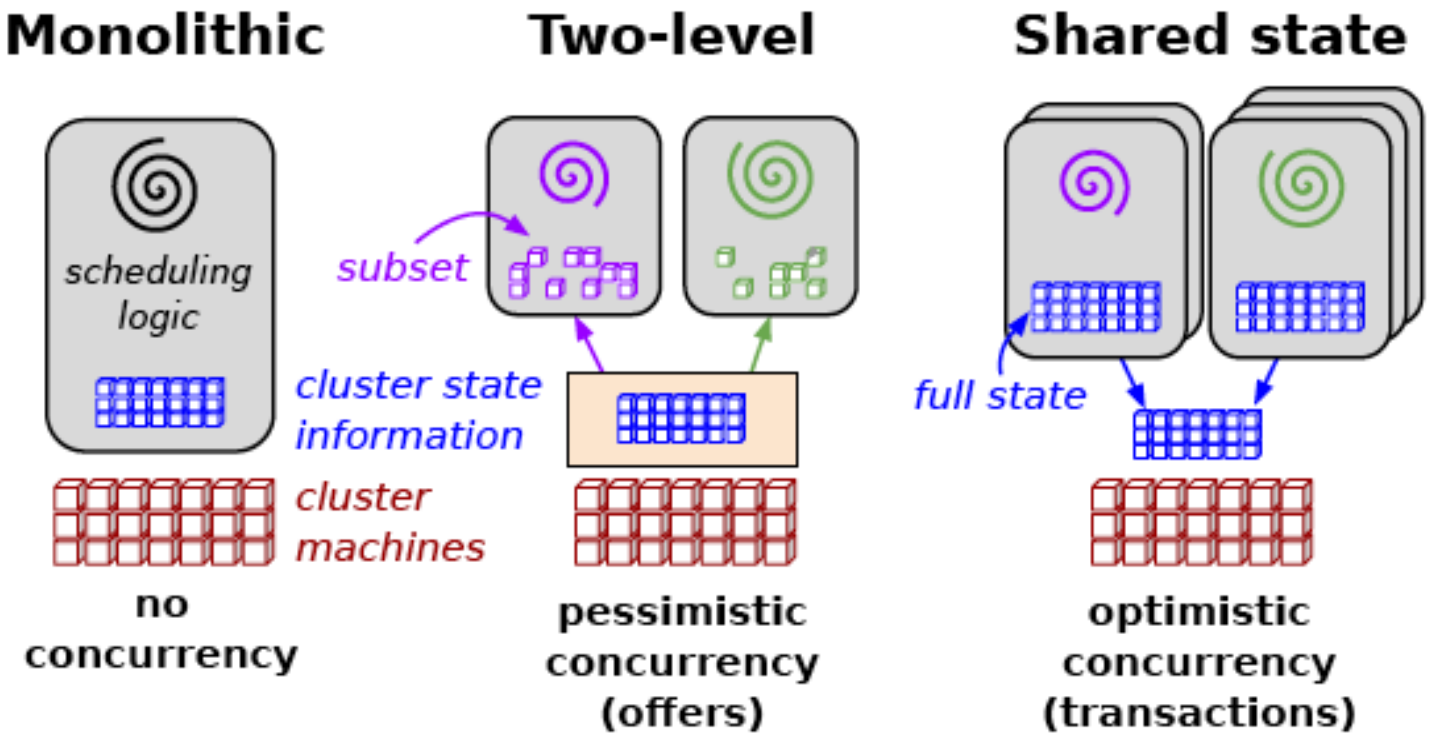
- All scheduling decisions go through the central RM
- The RM resolves all conflicts and guarantees resources to applications

M. Schwarzkopf, A. Konwinski, M.Abd-El-Malek, J. Wilkes, “Omega: flexible, scalable schedulers for large compute clusters,” Eurosys 2013

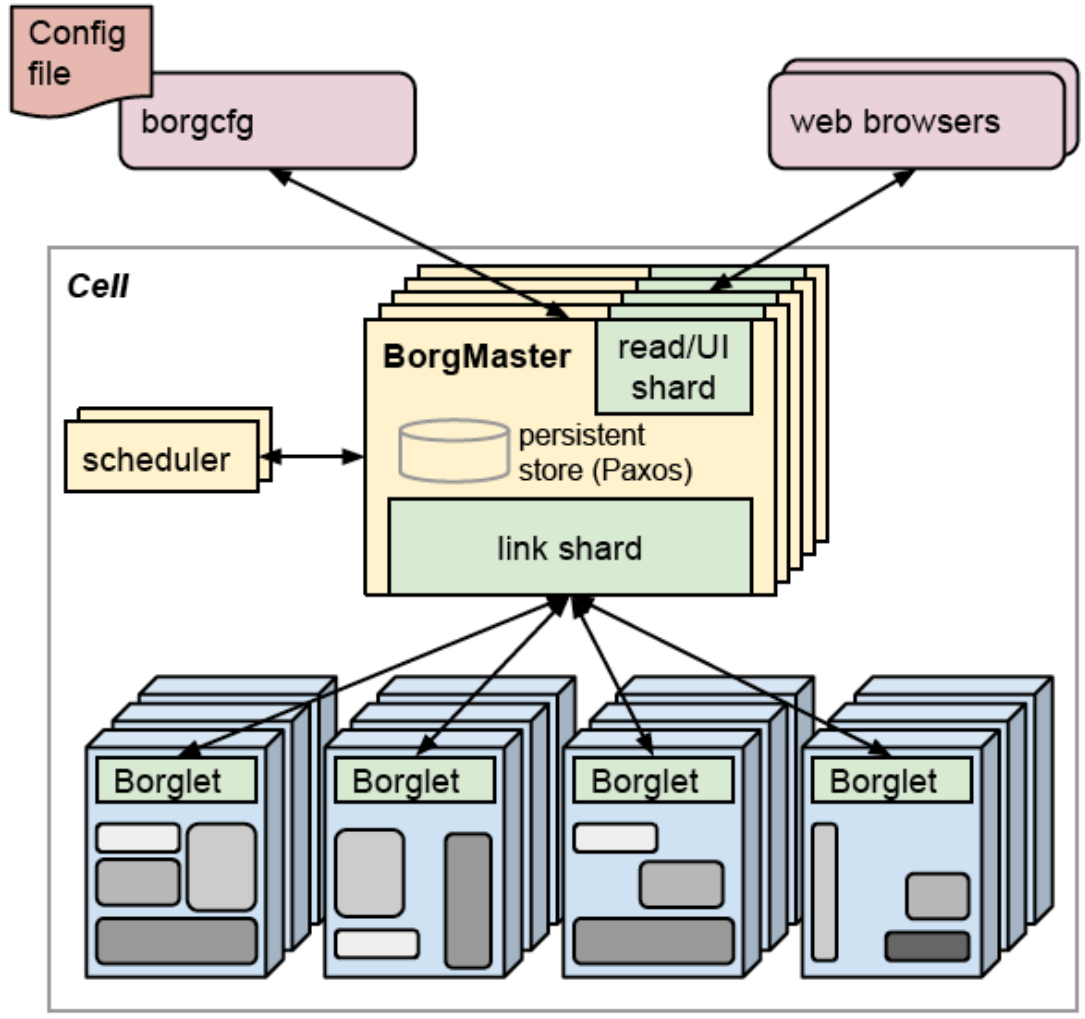
A. Verma, L. Pedrosa, “Large-scale cluster management at Google with Borg”, Eurosys 2015

Design Options for Centralized Resource Management:

Monolithic [Hadoop1.0] vs.Two-level [Mesos] vs. Shared-state [Omega, Borg]



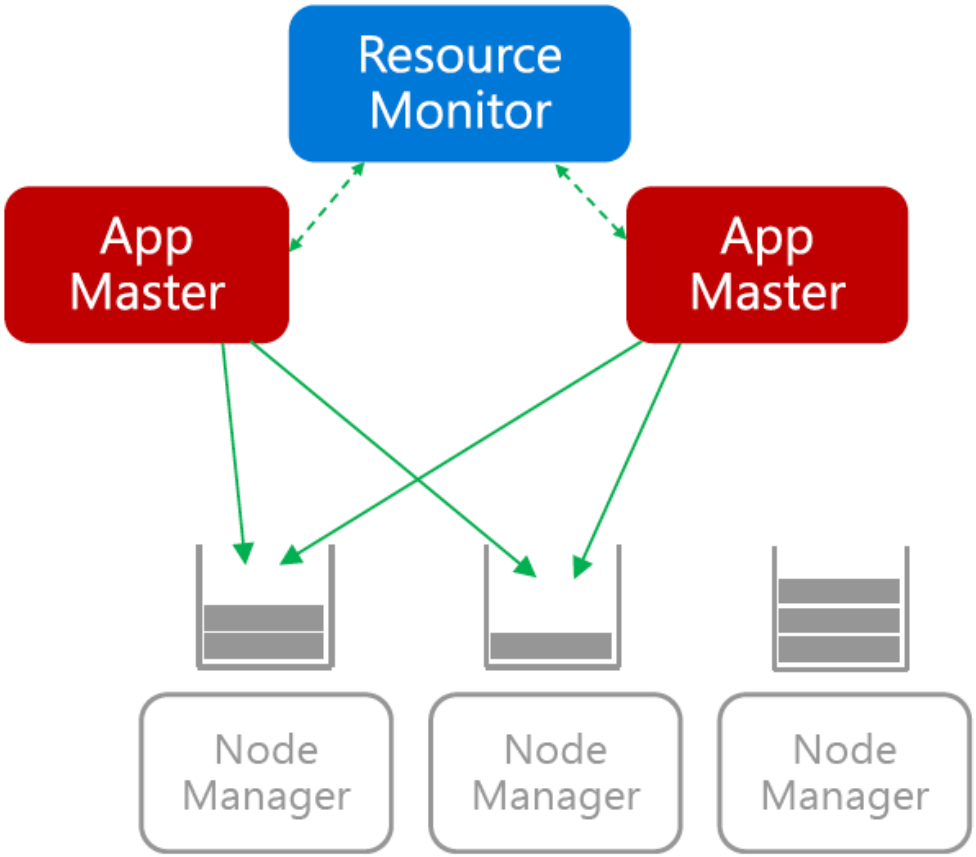
High-level Architecture of Google's Borg



A. Verma, L. Pedrosa, "Large-scale cluster management at Google with Borg", Eurosys 2015

Approach 2: Distributed Resource Management

[Apollo, Sparrow]

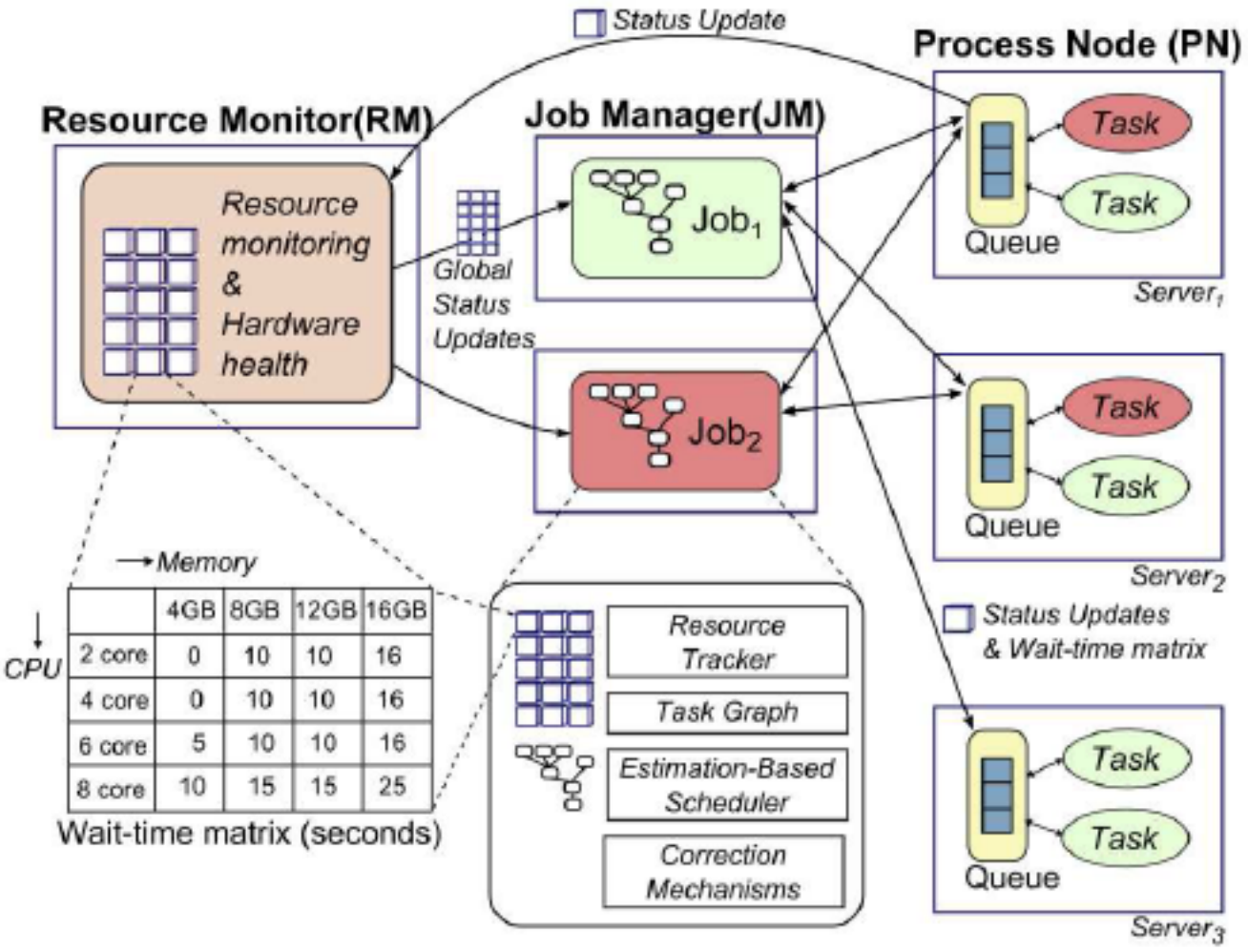


- AMs queue tasks directly to NMs
- Loose coordination through the Resource Monitor

K. Ousterhout et al, “Sparrow: Distributed, Low Latency Scheduling”, ACM SOSP 2013

E. Boutin et al, “Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing”, Usenix OSDI 2014

High-level Distributed Resource Management Architecture of Microsoft's Apollo



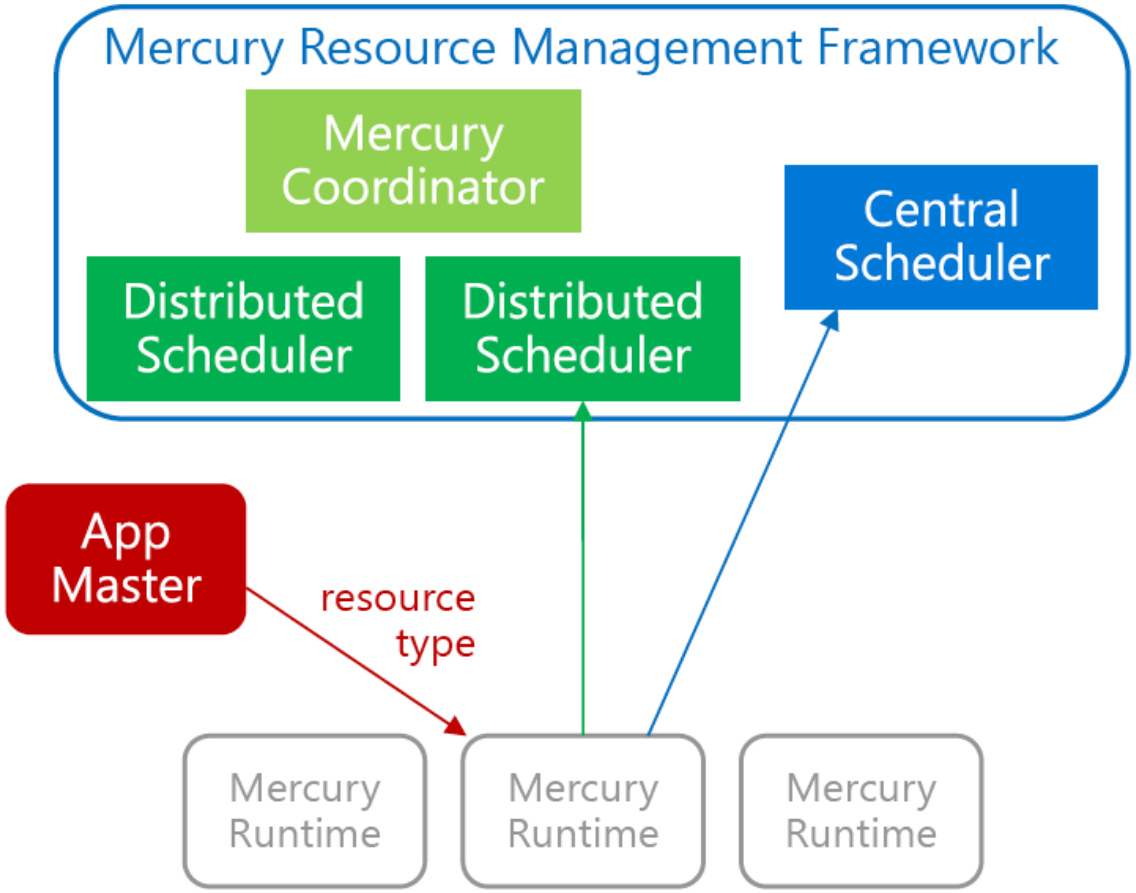
	4GB	8GB	12GB	16GB
2 core	0	10	10	16
4 core	0	10	10	16
6 core	5	10	10	16
8 core	10	15	15	25

E. Boutin et al, "Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing", OSDI 2014

Centralized vs. Distributed Resource Management

	Centralized	Distributed
Workload heterogeneity	✓	
Task placement	✓	
Enforcing scheduling invariants	✓	
Allocation latency		✓
Slot utilization		✓
Scalability		✓

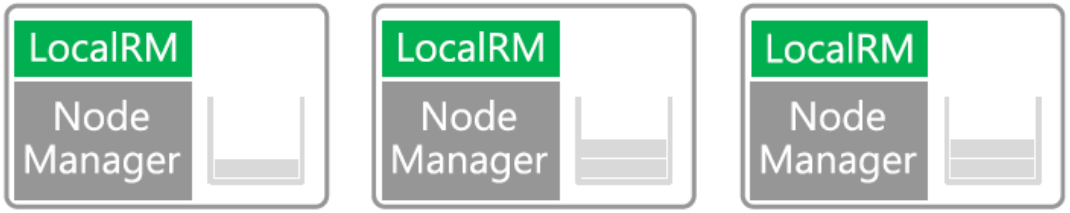
Approach 3: Hybrid (Distributed and Centralized) Resource Management in Microsoft's Mercury



- Two types of schedulers
- Central scheduler
  Scheduling policies/guarantees
  Slow(er) decisions
- Distributed schedulers
  Fast/low-latency decisions
- AM specifies resource *type*

K. Karanasos et al, "Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters", Usenix ATC 2015

Mercury Architecture over YARN



Overview of YARN extensions

- LocalRM (distributed scheduling)
- Queuing of (QUEUEABLE) containers at the NMs
- Framework policies
- Application policies for determining container type per task

Operations and Implementation of Mercury

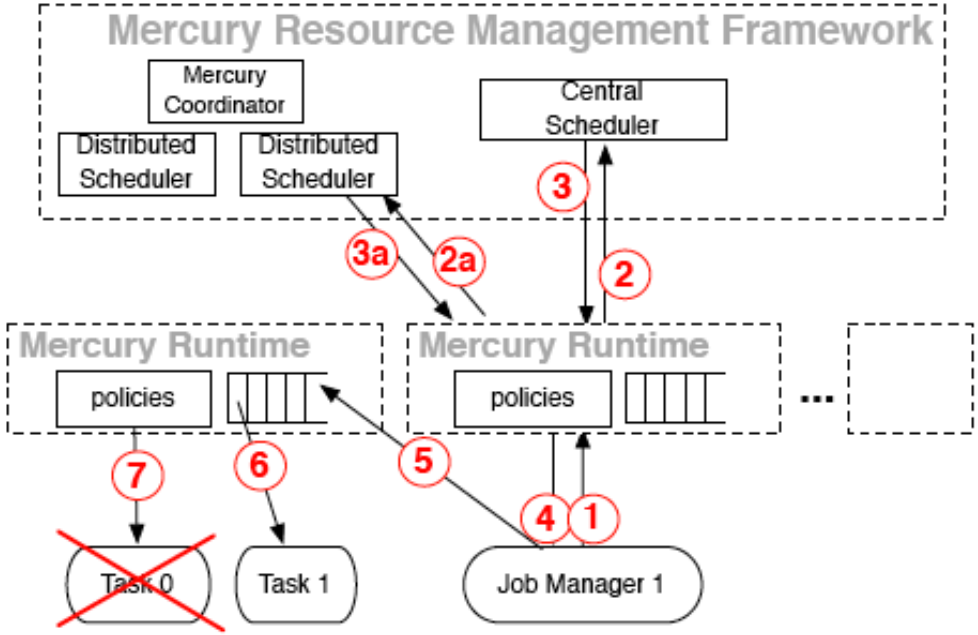


Figure 3: Mercury resource management lifecycle.

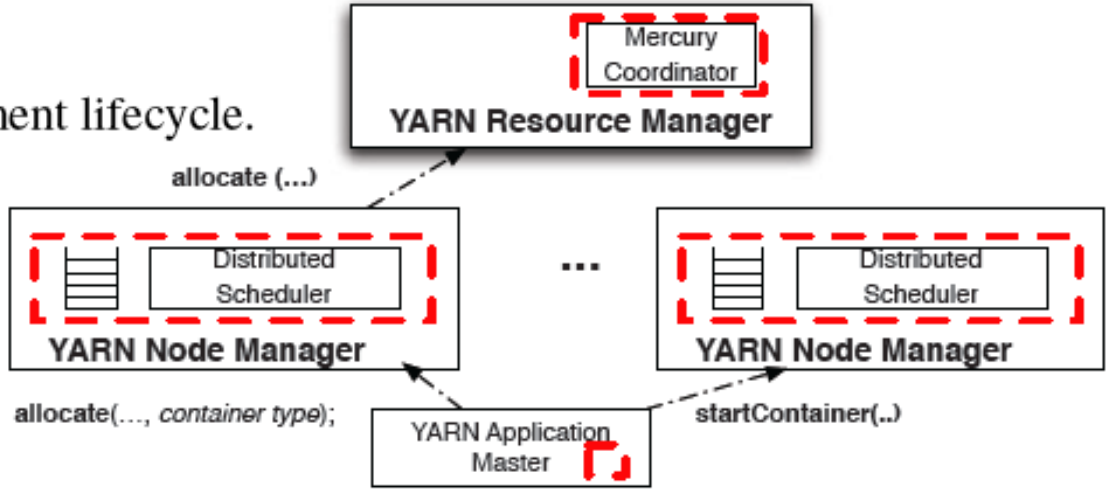


Figure 4: *Mercury implementation*: dashed boxes show Mercury modules and APIs as YARN extensions.

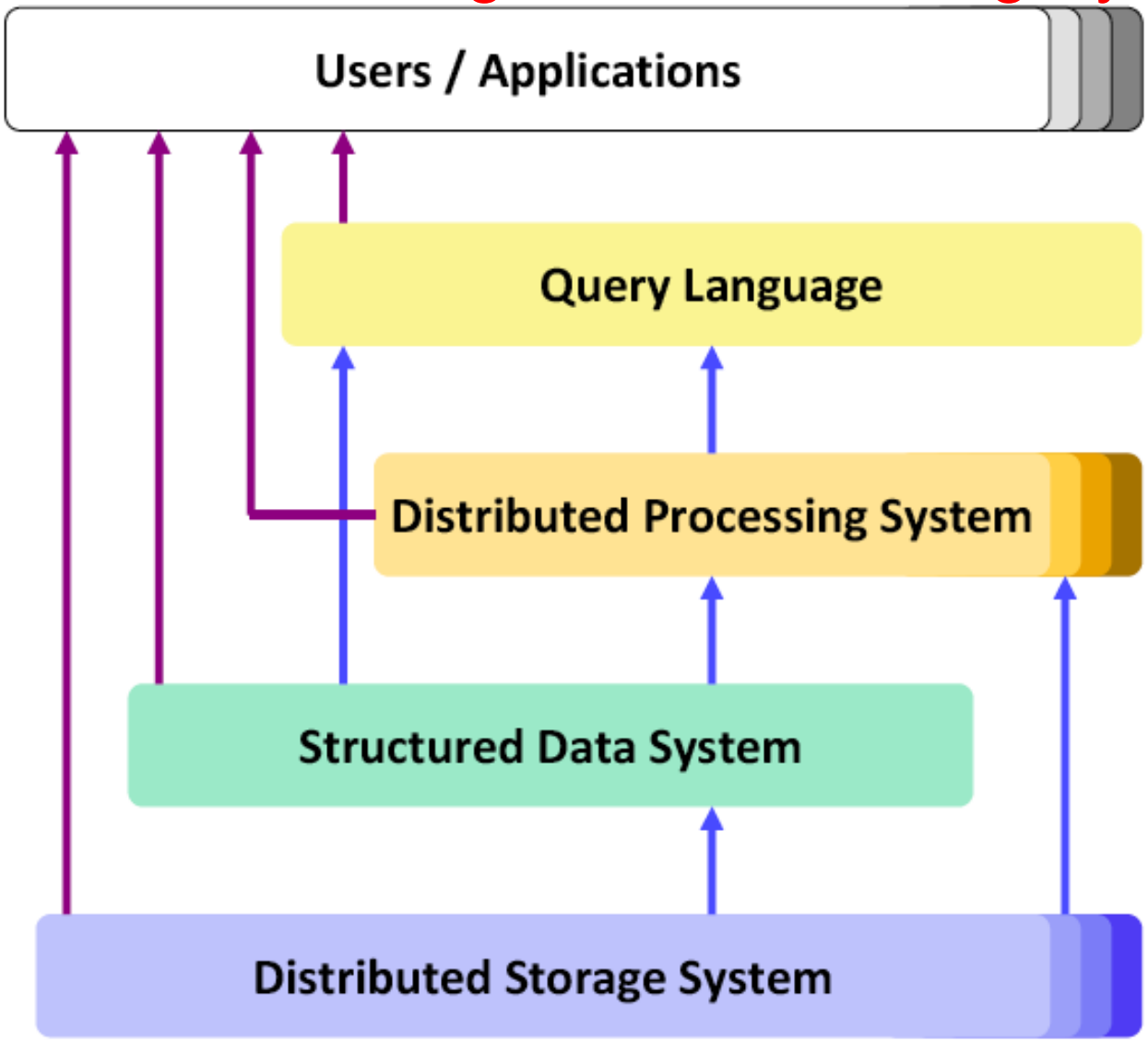
Comparisons of Recent Resource Management Platforms for Clusters

Resource Management Platform for Clusters	Scheduling/Resource Sharing paradigm	Scalability	Multiple Programming Frameworks/ Multi-tenant Support
Hadoop 1.0	Centralized	Limited but OK	No
YARN in Hadoop 2.0	Centralized	Good	Yes
Mesos	Centralized (Two-level) via Resource Offers to Individual Frameworks	Good	Yes
Apollo	Distributed and Loosely Coordinated (via Expected Resource Wait-Time matrix)	Very Good	Yes
Borg, Omega	Centralized per-cell BorgMaster which allows multiple // schedulers to performs optimistic-concurrent allocation followed by checking	Very Good	Yes
Mercury	Hybrid (Centralized and Distributed scheduling for Big and Small jobs respectively)	Very Good	Yes

Besides the Computational Model:

Typical Architecture for Big Data Processing Systems

Typical Architecture of Cloud Computing/ Big Data Processing Systems

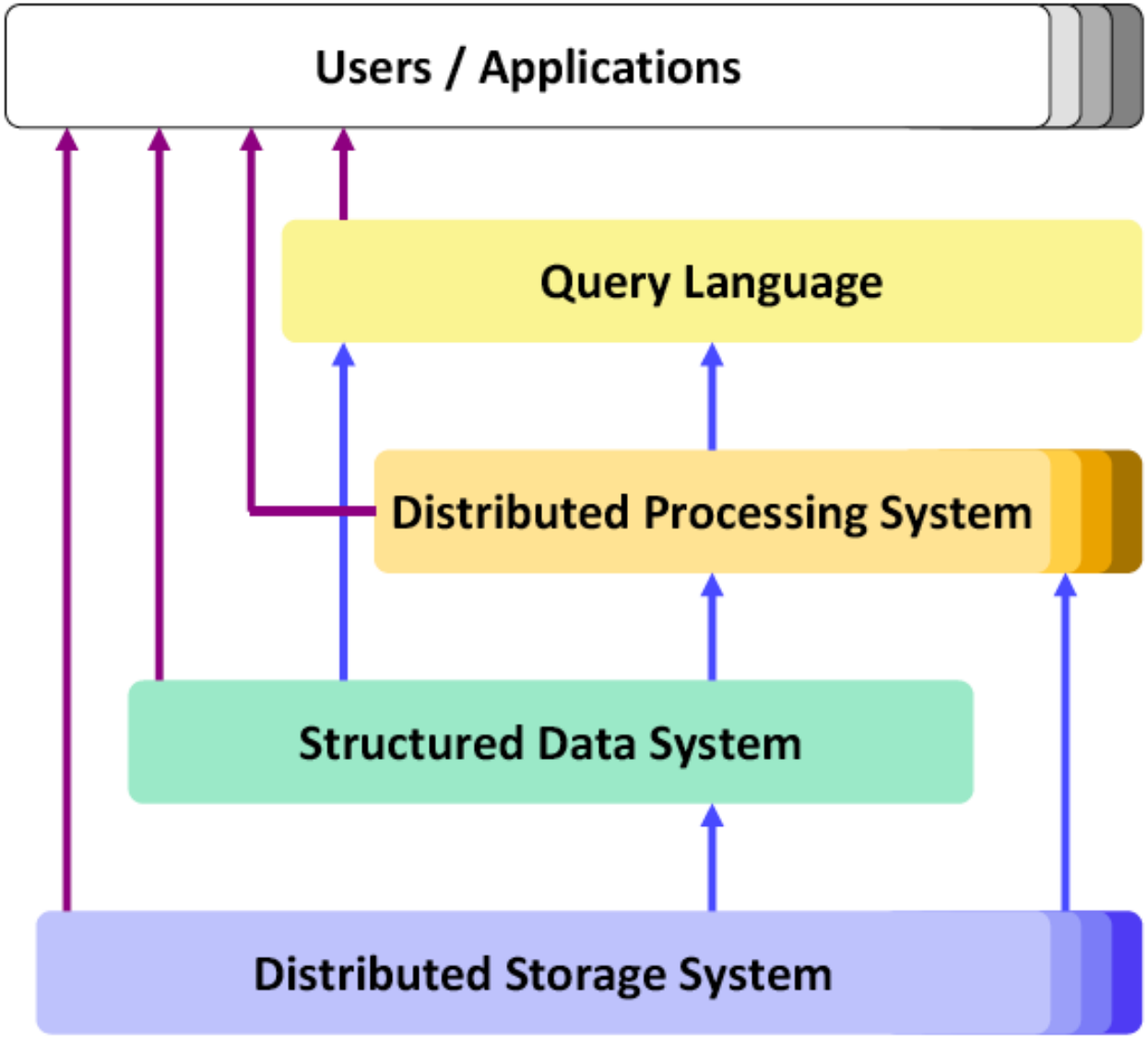


Applications may run as services on virtual / distributed servers within the Cloud themselves

Frameworks to support programming for massive parallel processing (shared nothing) with dedicated servers

Massively distributed storage (shared nothing) on dedicated servers, heavy replication

Typical Architecture: Different Component Systems for various Services and Functionalities



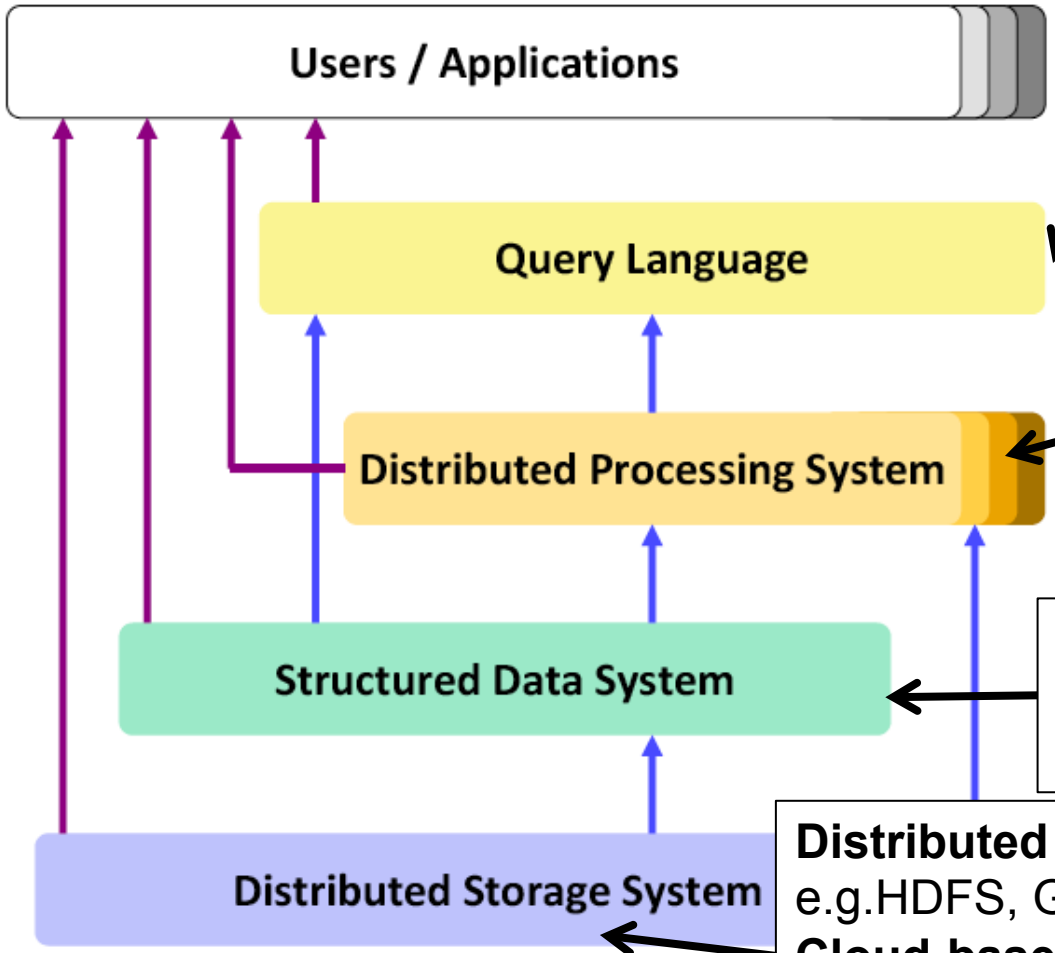
High-level language for accessing data and controlling processing, but typically not SQL!

Performance for complex operations (SQL-style joins and grouping, data analysis, etc.)

Simple but flexible data model (mostly key/value pairs), basic access operations (lookup-API)

Performance for data accesses, fault-tolerance, availability, scalability

Typical Architecture: Different Component Systems for various Services and Functionalities



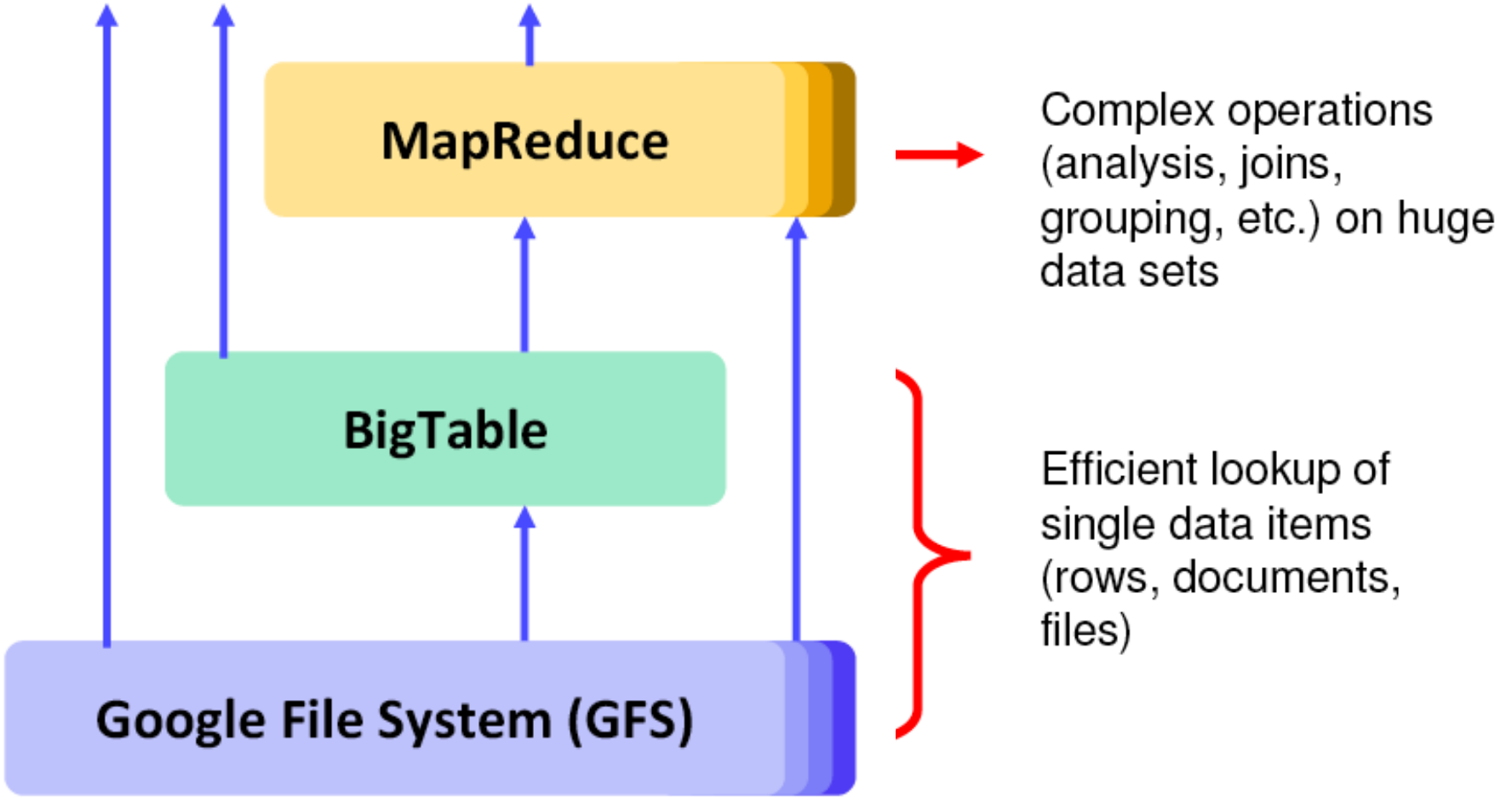
e.g. HiveQL of Hive (Facebook), BigSQL (IBM), Apache Drill, Cloudera Impala ; Pig (Yahoo); Spark SQL, DryadLINQ, other NoSQL query languages (NoSQL = Not-only-SQL)

e.g. Hadoop/ MapReduce, GraphLab (CMU/UWash), Spark (Berkeley), Storm/Heron (Twitter), Dryad (Microsoft), TeZ, Pregel/ Giraph (Apache), Flink

e.g. BigTable(Google)/ Hbase(open) Cassandra(Facebook), SimpleDB, DynamoDB (Amazon)

Distributed FileSystems:
e.g.HDFS, GFS, ceph
Cloud-based Data-Store Service/System:
e.g. Amazon S3, EBS, OpenStack Swift , Amazon Dynamo <key,value> store

Architecture Sample 1: The Google-way (Initially)



Google's BigTable

- Fast and Large-scale (PB range) Database Management System (DBMS) for Google applications and services
- Data Model = Sparse, Distributed Multi-dimensional Sorted Map,
  - Think of it as a Distributed, Super-Large Spreadsheet split over a huge cluster of servers
  - Adjacent rows grouped to form a Tablet, which is hosted in the same server
  - Row range for each Tablet is dynamically partitioned for Load-balancing
  - Read/Write under a single Row key are atomic
- Distributed, persistent lock/ name service from Chubby
- Rely on GFS to store data and logs
- Commonly used as Data Input source and Output target for MapReduce programs ;

BigTable Data Model

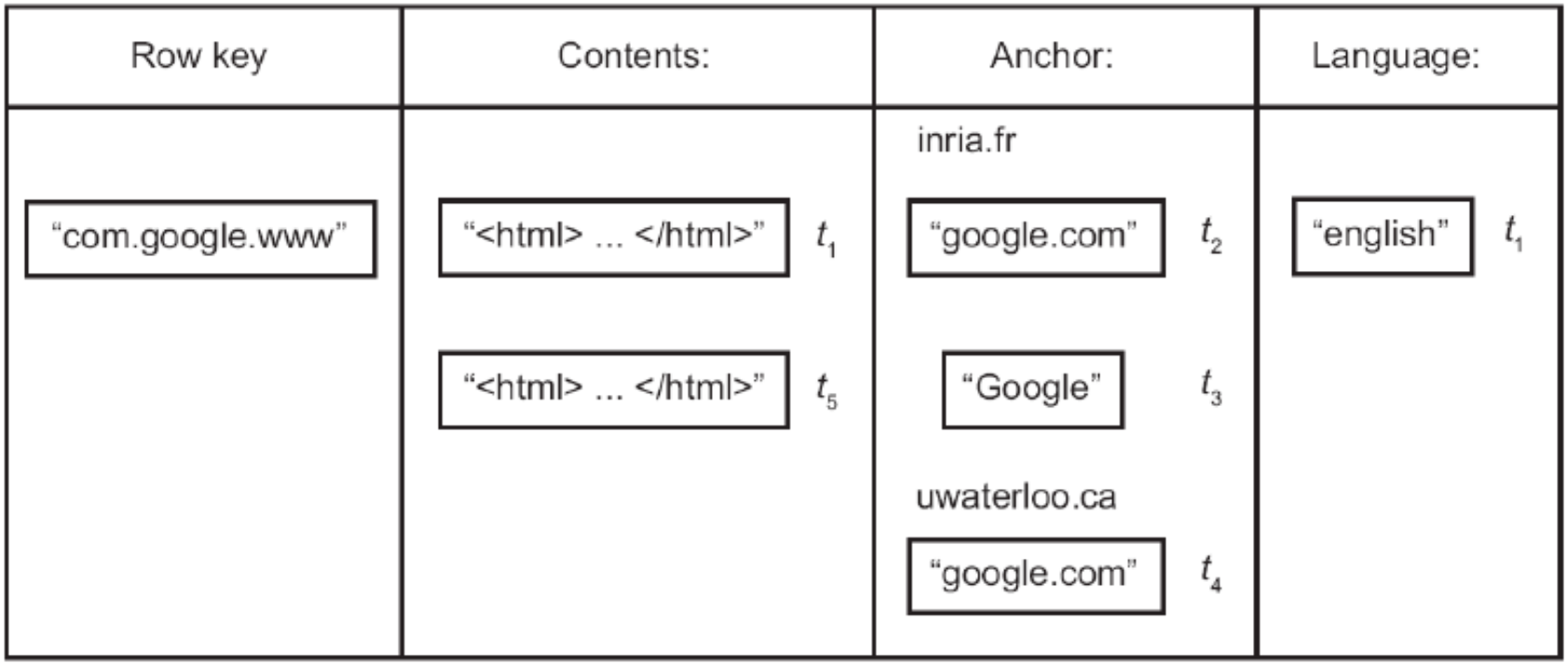


Fig. 18.7 Example of a Bigtable Row

From [ÖzsuValduriez2011]

More detailed coverage on BigTable in the later part of this course !

Google Applications using BigTable

- Google Maps
- Google Book Search
- Google Earth
- Google Analytics
- Blogger.com
- YouTube
- Gmail
- …

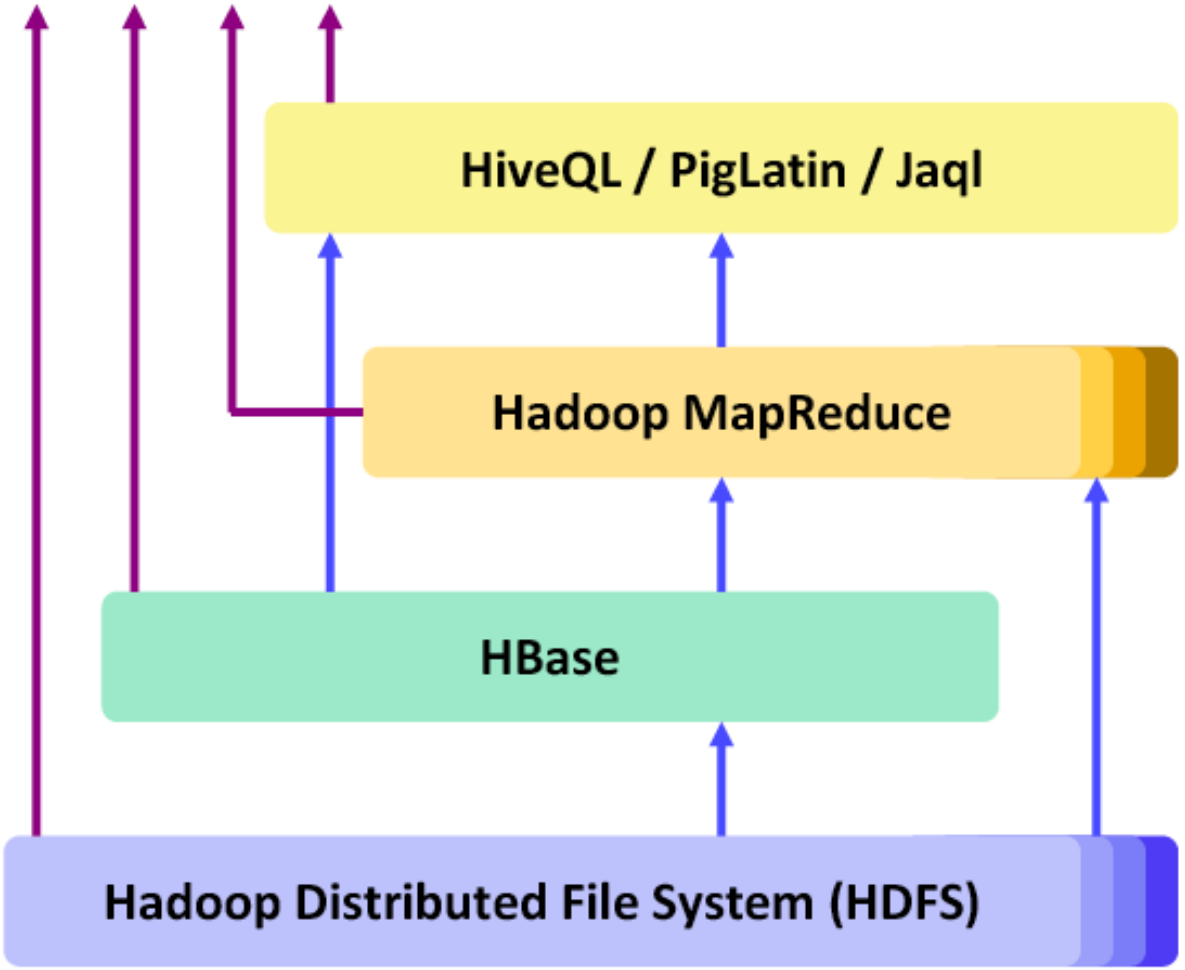
Google Applications using BigTable (cont'd)

Project name	Table size (TB)	Compression ratio	# Cells (billions)	# Column Families	# Locality Groups	% in memory	Latency-sensitive?
Crawl	800	11%	1000	16	8	0%	No
Crawl	50	33%	200	2	2	0%	No
Google Analytics	20	29%	10	1	1	0%	Yes
Google Analytics	200	14%	80	1	1	0%	Yes
Google Base	2	31%	10	29	3	15%	Yes
Google Earth	0.5	64%	8	7	2	33%	Yes
Google Earth	70	–	9	8	3	0%	No
Orkut	9	–	0.9	8	5	1%	Yes
Personalized Search	4	47%	6	93	11	5%	Yes

Table 2: Characteristics of a few tables in production use. *Table size* (measured before compression) and *# Cells* indicate approximate sizes. *Compression ratio* is not given for tables that have compression disabled.

From [F. Chang et al.: Bigtable: A Distributed Storage System for Structured Data, OSDI 2006]

Architecture Sample 2: The Hadoop-way (e.g. Yahoo)



Hadoop 1.0 vs. Hadoop 2.0 Ecosystem

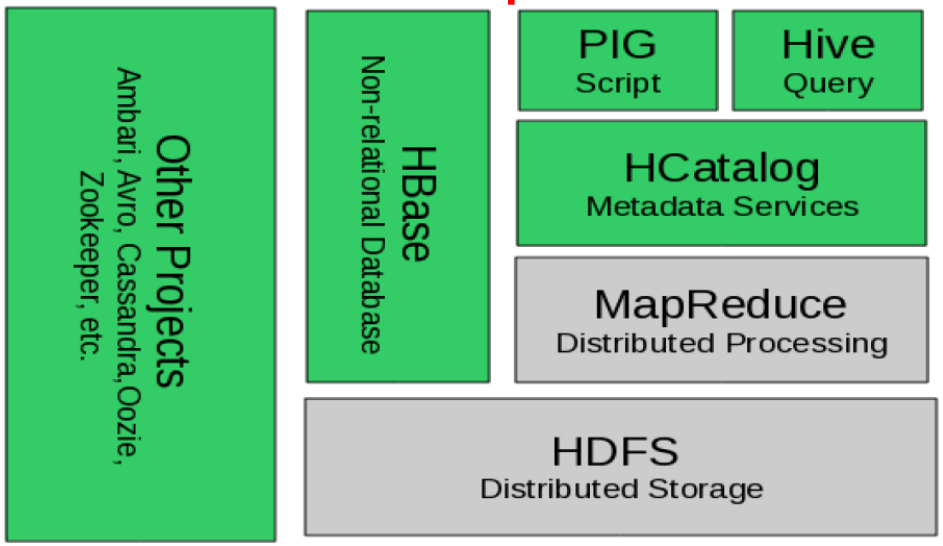


Figure 2.1 The Hadoop 1.0 ecosystem, MapReduce and HDFS are the core components, while other are built around the core.

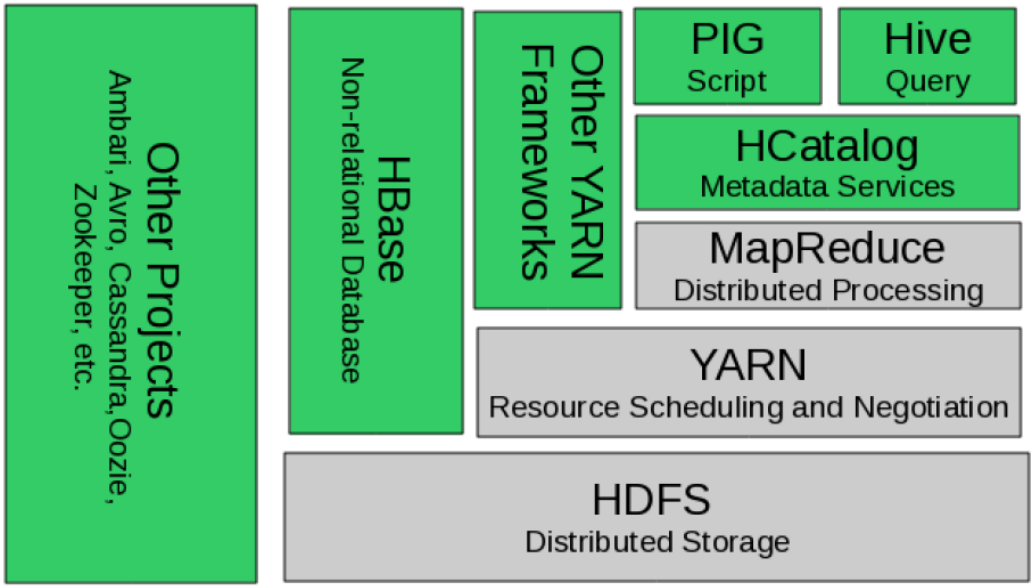


Figure 2.2 YARN adds a more general interface to run non-MapReduce jobs within the Hadoop framework

HBase

- Can be considered as the Open-source version of BigTable
- Semi-structured data storage
- Developed initially by Powerset (an NLP company)
- Now part of Apache's (open-source) Hadoop project
  - Like BigTable, HBase tables can serve as Data input/output store for MapReduce jobs run in Hadoop
  - Based on HDFS
- Access via Java API, REST and others
- Used by, e.g. Facebook's Messaging Platform

Apache Cassandra

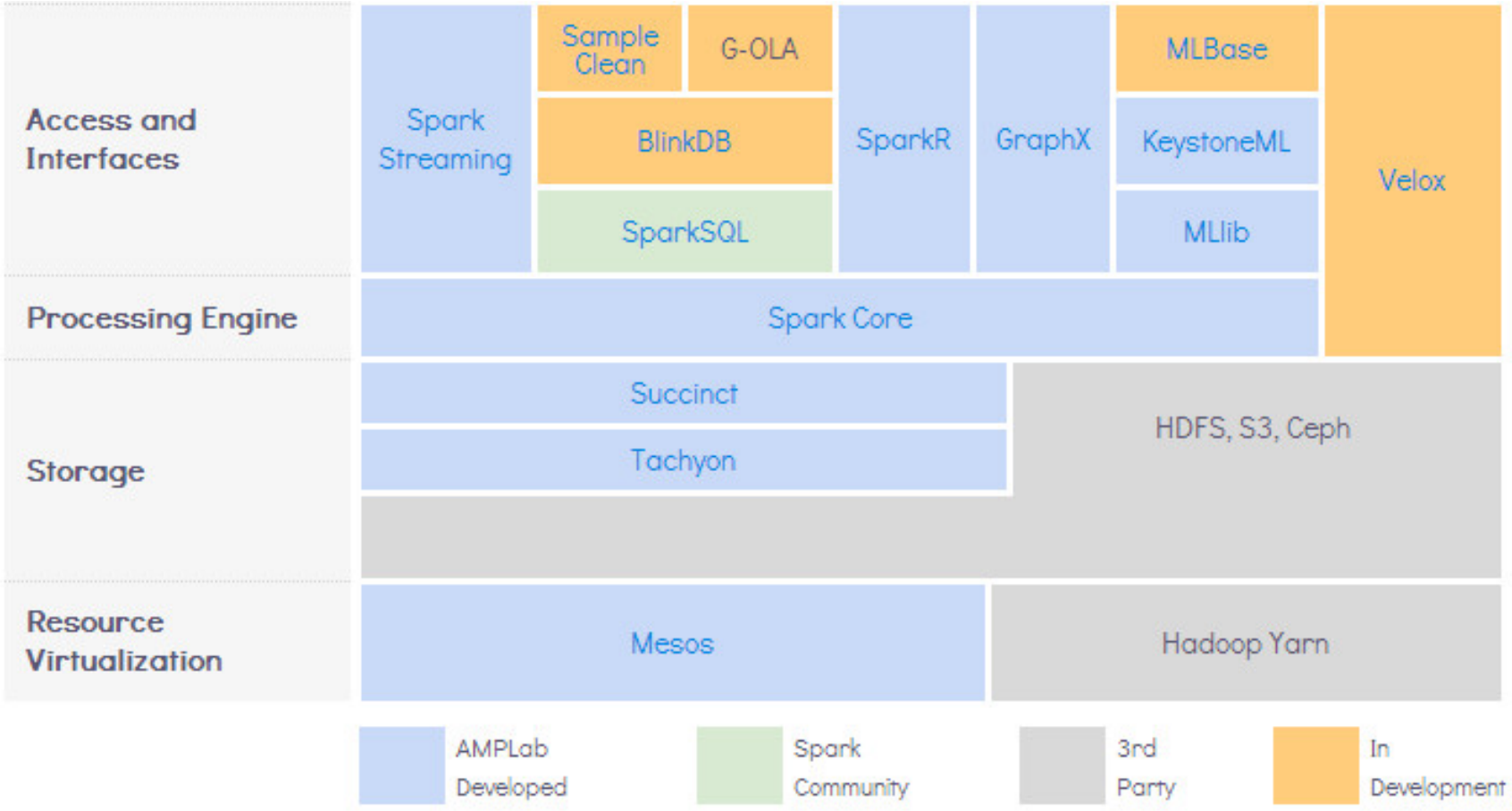
- BigTable data model running on an Amazon Dynamo-like (P2P) infrastructure
- Developed initially by Facebook
- Now part of Apache Software Foundation (Open-source)
- Differences w.r.t. Hbase
  - Standalone system
  - Not based on HDFS
  - Storage approach similar to Distributed Hash Table (DHT)
  - Tunable consistency levels

Apache Hive

- Data warehouse infrastructure built on top of Hadoop
- Initially developed by Facebook
- Now part of Apache Software Foundation (Open-source)
- Use to analyze large datasets stored in
  - HDFS
  - Amazon S3
- Support SQL-like query language called HiveSQL

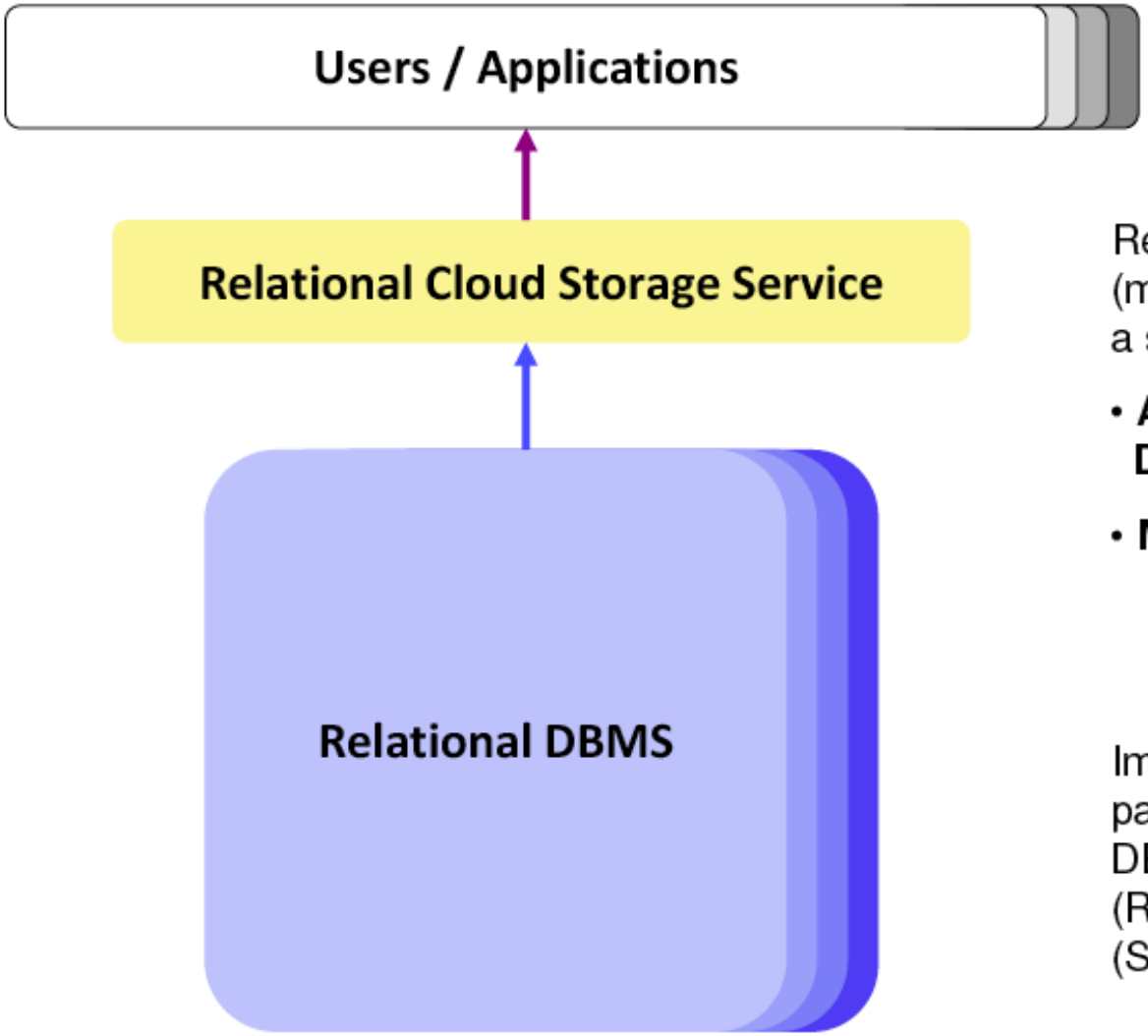
Beyond Hadoop/MapReduce: Another Main-stream Big Data Processing Framework

- Spark & Berkeley Data Analytic Stack (BDAS) by UC Berkeley



Reference: https://amplab.cs.berkeley.edu/software/

Alternative: Relational Database Management System (RDMS) as a Service



Relational Model and SQL (maybe with restrictions) as a service, e.g.

- **Amazon Relational Database Service (RDS)**
- **Microsoft SQL Azure**

Implemented on top of parallel clusters of common DBMS Servers, e.g. MySQL (RDS) or MS SQL Server (SQL Azure)

MapReduce vs. Parallel RDBMS

- MapReduce

+ Very Scalable, Fault-tolerant and Automatic Load-balancing

+ Operates well in Heterogeneous Clusters

- Writing Map/Reduce jobs is more complicated than writing SQL queries

- Performance largely depends on the skill of the programmer

- Parallel RDBMS

+ Usually very good and consistent performance

+ Flexible and proven interface (SQL)

+ SQL-queries are automatically optimized for transaction performance

- Scaling is rather Limited (10's of nodes)
- Does NOT work well in Heterogeneous Clusters
- Not very Fault-Tolerant

MapReduce vs. Parallel RDBMS

	Parallel DBMS	MapReduce
Schema Support	☑	☒
Indexing	☑	☒
Programming Model	Stating what you want (declarative: SQL)	Presenting an algorithm (procedural: C/C++, Java, …)
Optimization	☑	☒
Scaling	1 – 500	10 - 5000
Fault Tolerance	Limited	Good
Execution	Pipelines results between operators	Materializes results between phases

SQL vs. MapReduce

- Selection / projection / aggregation
  - SQL Query:

```
SELECT year, SUM(price)
FROM sales
WHERE area_code = "US"
GROUP BY year
```

  - Map/Reduce job:

```
map(key, tuple) {
  int year = YEAR(tuple.date);
  if (tuple.area_code = "US")
    emit(year, {'year' => year, 'price' => tuple.price });
}

reduce(key, tuples) {
  double sum_price = 0;
  foreach (tuple in tuples) {
    sum_price += tuple.price;
  }
  emit(key, sum_price);
}
```

SQL vs. MapReduce (cont'd)

- Sorting
  - SQL Query:

```
SELECT *
FROM sales
ORDER BY year
```

  - Map/Reduce job:

```
map(key, tuple) {
  emit(YEAR(tuple.date) DIV 10, tuple);
}

reduce(key, tuples) {
  emit(key, sort(tuples));
}
```

NoSQL (Not-only SQL) vs. RDBMS

- RDBMS provides too much:
  - ACID (Atomicity, Consistency, Isolation, Durability) transactions
  - Complex Query Language
  - Lots and lots of knobs to turn
- RDBMS provides too little:
  - Lack of (cost-effective) scalability, availability
  - Not enough schema/data-type flexibility
- NoSQL
  - Lots of optimization and tuning possible for Analytics
  - Flexible programming model
- NoSQL can borrow many good ideas from RDBMS
  - Declarative Language
  - Parallelization and Optimization Techniques
  - Value of Data Consistency

Source: Raghu Ramakrishnan of Microsoft

Recap

- MapReduce – A Computational Model for Big Data Processing
- The MapReduce Runtime, GFS/ HDFS
- Sample Applications for MapReduce
- Typical Architectures for Big Data Processing Systems

Further Reading

- Jeffrey Dean and Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters

  http://research.google.com/archive/mapreduce.html

- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung: The Google File System

  http://research.google.com/archive/gfs.html

- Siba Mohammad, Sebastian Breb, Eike Schallehn, “Cloud Data Management: A Short Overview and Comparison of Current Approaches,” 24th GI-Workshop on Foundations of Databases, May 2012. slides available at:

  http://wwwiti.cs.uni-magdeburg.de/iti_db/lehre/advdb/cloud.pdf