Advanced Computer Networks — Cloud Computing (I)

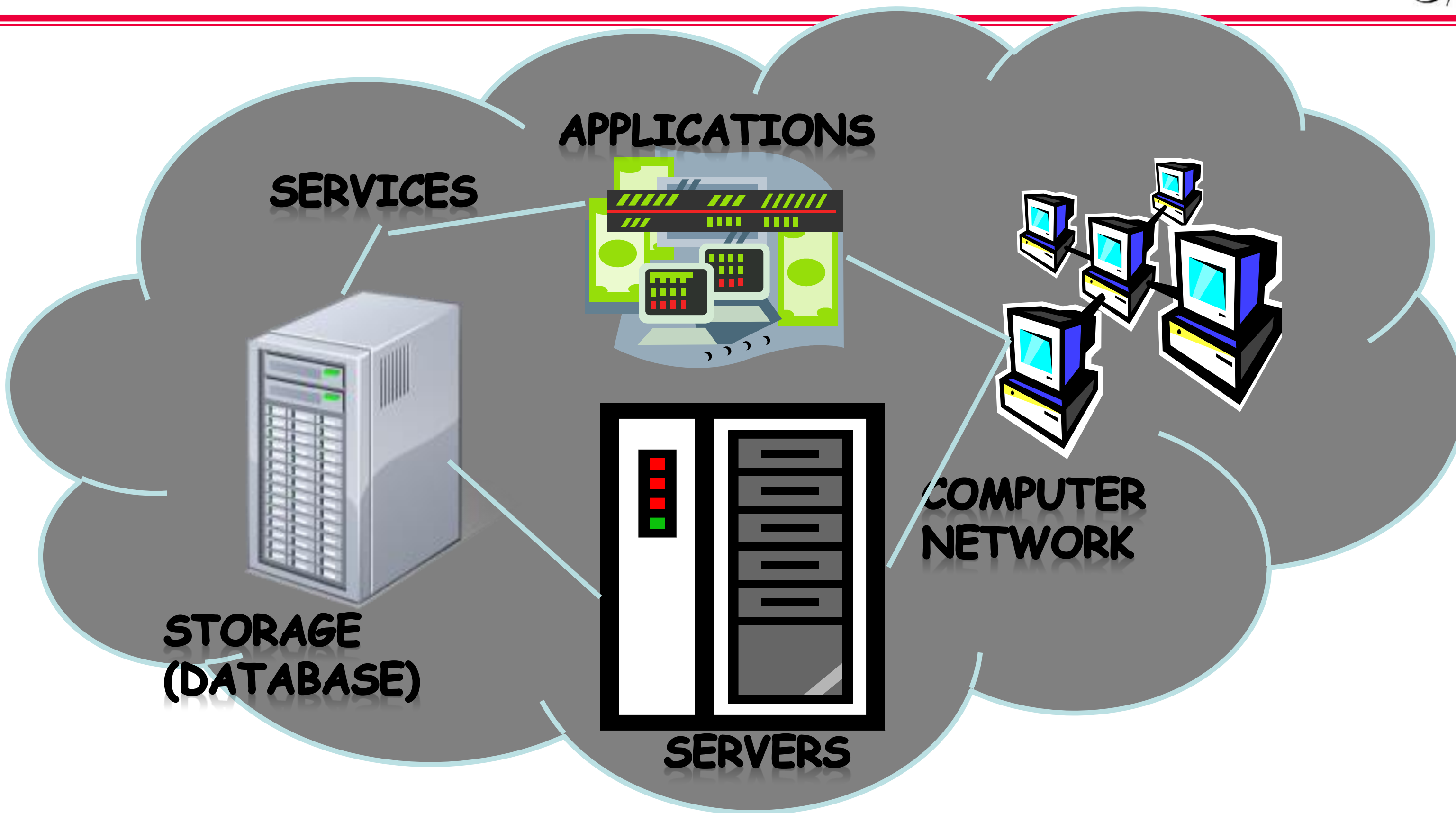
June 13, 2013

Prof. Xiaoming Fu, MSc. Yuan Zhang

Acknowledgement: Revised based on Anthony D. Joseph's slides

GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

What is Cloud Computing



- Shared pool of configurable computing resources
- On-demand network access
- Provisioned by the Service Provider

Cloud Computing Characteristics

Common Characteristics:

- Massive Scale
- Resilient Computing
- Homogeneity
- Geographic Distribution
- Virtualization
- Service Orientation
- Low Cost Software
- Advanced Security

Essential Characteristics:

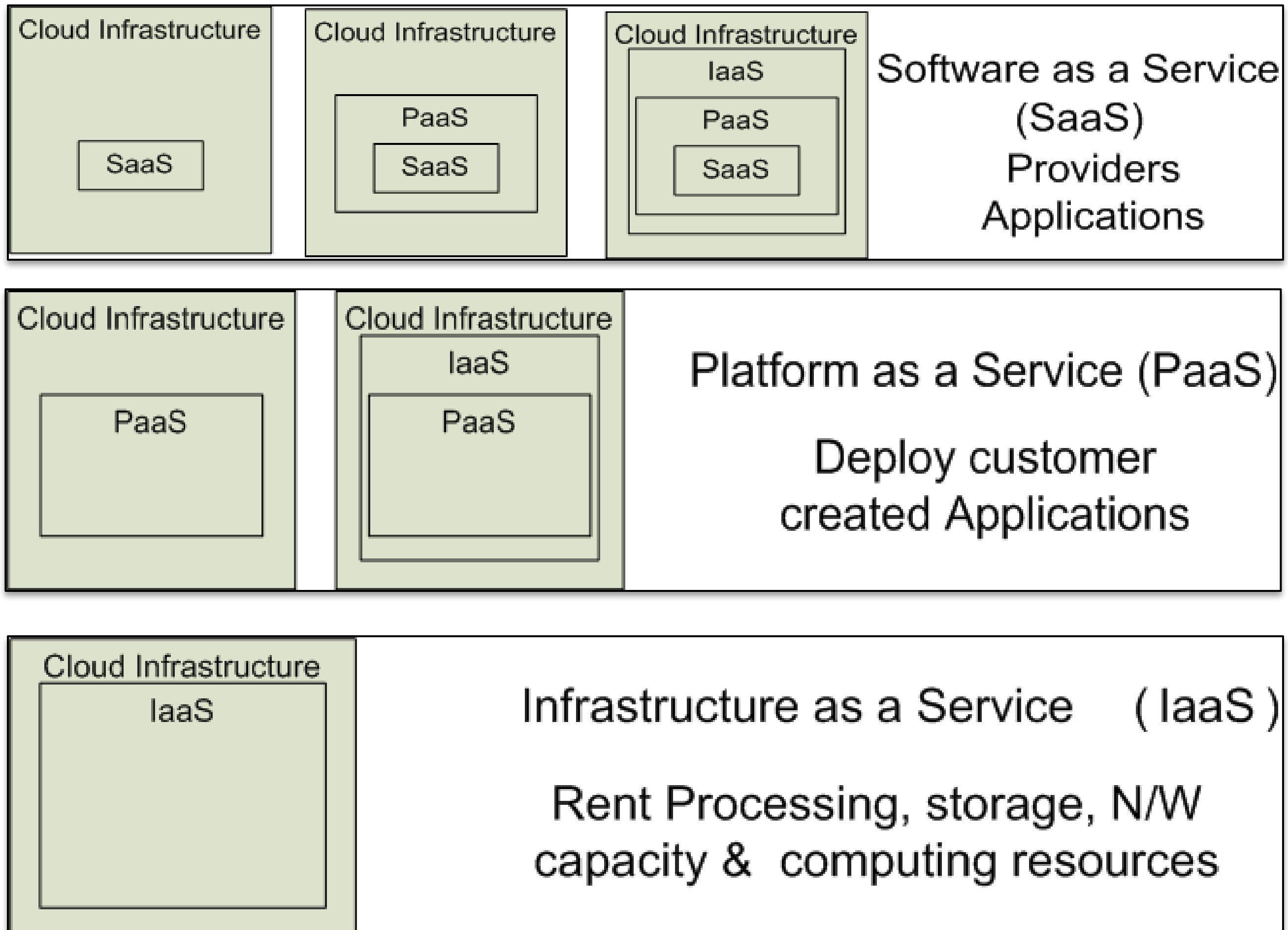
- On Demand Self-Service
- Broad Network Access
- Rapid Elasticity
- Resource Pooling
- Measured Service

Cloud Service Models

Software as a Service (SaaS) | **Platform as a Service (PaaS)** | **Infrastructure as a Service (IaaS)**

SalesForce CRM

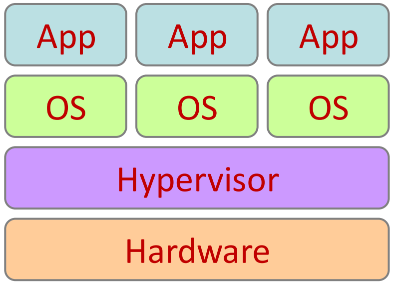
LotusLive



Virtualization

http://www.vmware.com/virtualization/

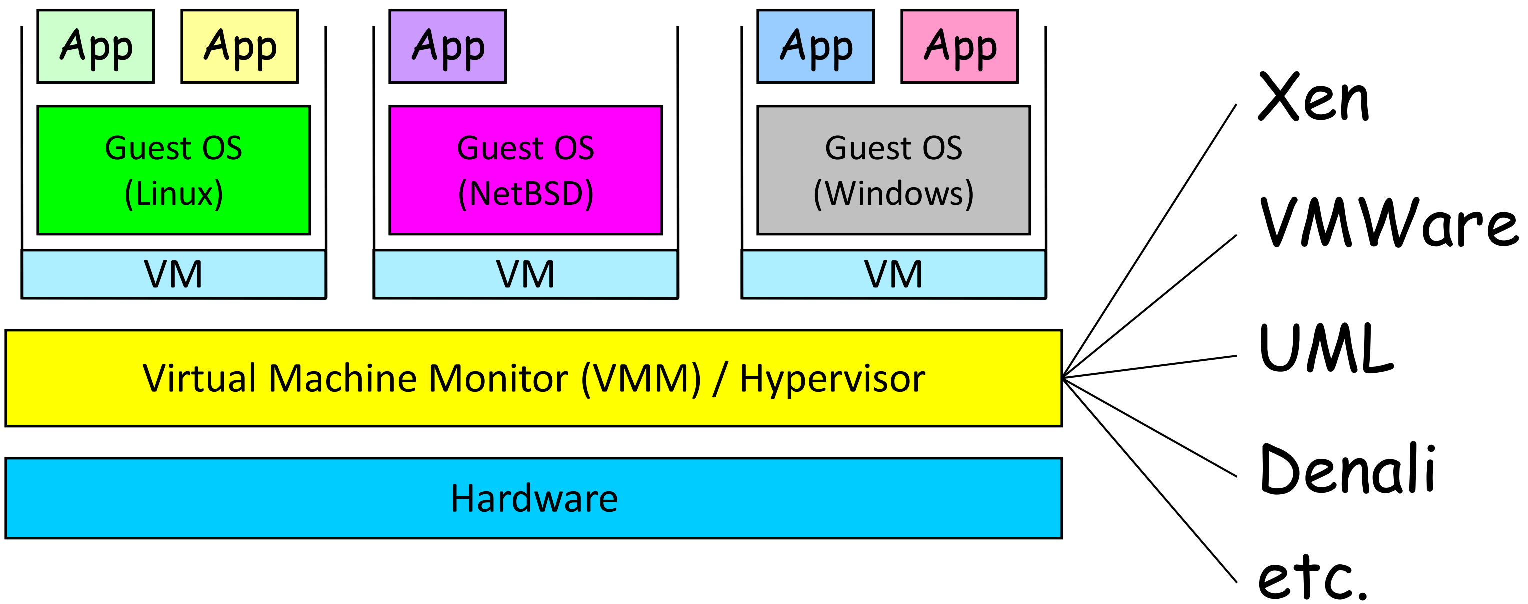
- Virtual workspaces:
  - An abstraction of an execution environment that can be made dynamically available to authorized clients by using well-defined protocols,
  - Resource quota (e.g. CPU, memory share),
  - Software configuration (e.g. OS, provided services).
- Implement on Virtual Machines (VMs):
  - Abstraction of a physical host machine,
  - Hypervisor intercepts and emulates instructions from VMs, and allows management of VMs,
  - VMWare, Xen, etc.
- Provide infrastructure API:
  - Plug-ins to hardware/support infrastructures



Virtualized Stack

Virtual Machines

- VM technology allows multiple virtual machines to run on a single physical machine.



Performance: Para-virtualization (e.g. Xen) is very close to raw physical performance!

* Para-virtualization means a guest OS is recompiled prior to installation inside a VM

Modes of Clouds

Public Cloud

- Computing infrastructure is hosted by cloud vendor at the vendors premises.
- and can be shared by various organizations.
- E.g. : Amazon, Google, Microsoft, Sales force

Private Cloud

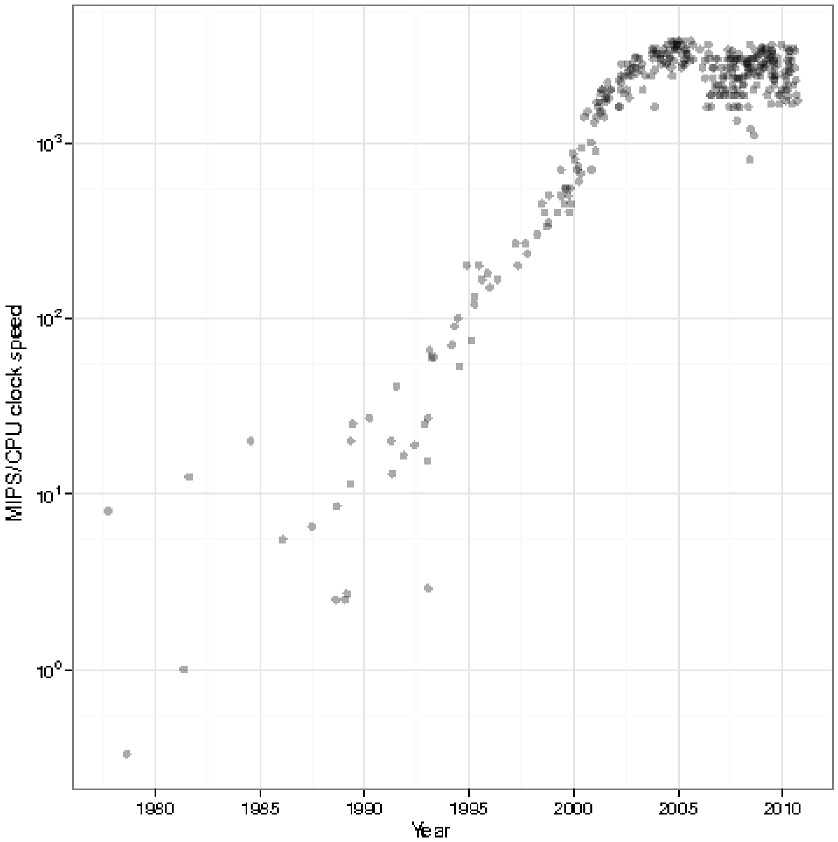
- The computing infrastructure is dedicated to a particular organization and not shared with other organizations.
- more expensive and more secure when compare to public cloud.
- E.g. : HP data center, IBM, Sun, Oracle, 3tera

Hybrid Cloud

- Organizations may host critical applications on private clouds.
- where as relatively less security concerns on public cloud.
- usage of both public and private together is called hybrid cloud.

Background of Cloud Computing

- 1990: Heyday of parallel computing, multi-processors
  - 52% growth in performance per year!
- 2002: The thermal wall
  - Speed (frequency) peaks, but transistors keep shrinking
- The Multicore revolution
  - 15-20 years later than predicted, we have hit the performance wall



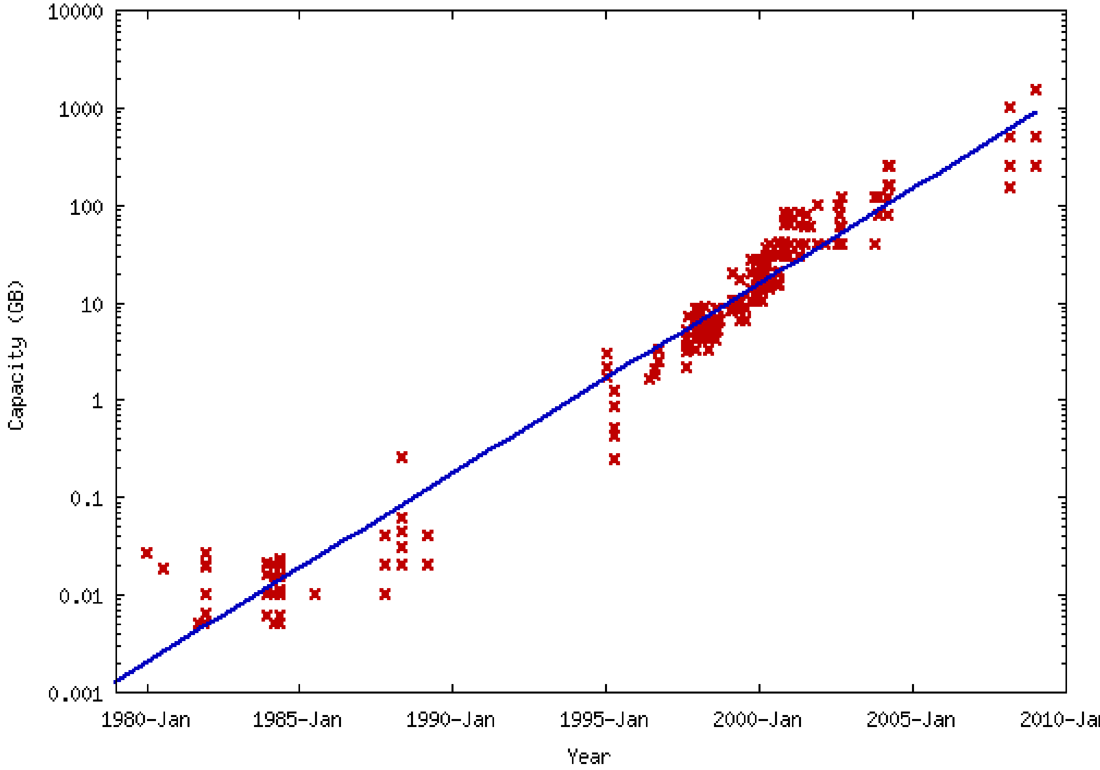
At the same time…

- Amount of stored data is exploding…

Data Deluge

- Billions of users connected through the net
  - WWW, FB, twitter, cell phones, …
  - 80% of the data on FB was produced last year

- Storage getting cheaper
  - Store more data!



Solving the Impedance Mismatch

- Computers not getting faster, and we are drowning in data
  - How to resolve the dilemma?
- Solution adopted by web-scale companies
  - Go massively *distributed* and *parallel*

Distributed Systems

- Distributed Systems/Computing
  - *Loosely coupled* set of computers, communicating through message passing, solving a common goal
- Distributed computing is *challenging*
  - Dealing with *partial failures*
  - Dealing with *asynchrony*
- Distributed Computing versus Parallel Computing?
  - distributed computing=parallel computing + partial failures

Dealing with Distribution

- We have seen several of the tools that help with distributed programming
  - Message Passing Interface (MPI)
  - Distributed Shared Memory (DSM)
  - Remote Procedure Calls (RPC)

- But, distributed programming is still very hard
  - Programming for scale, fault-tolerance, consistency, …

Distributed Systems

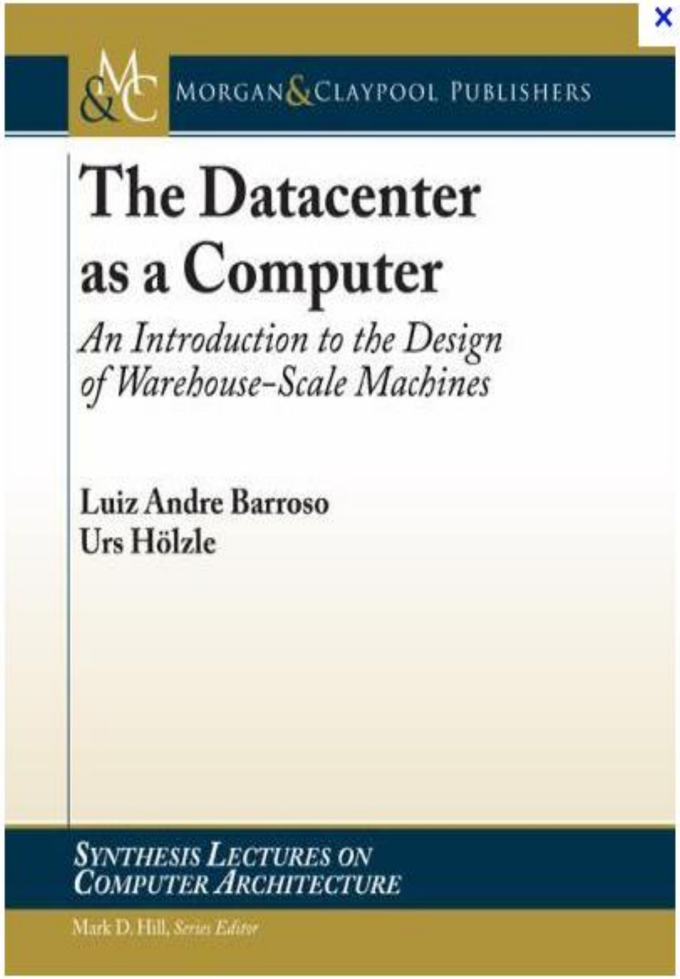
- Distributed Systems/Computing
  - *Loosely coupled* set of computers, communicating through message passing, solving a common goal
- Distributed computing is *challenging*
  - Dealing with *partial failures* (examples?)
  - Dealing with *asynchrony* (examples?)
- Distributed Computing versus Parallel Computing?
  - distributed computing=parallel computing + partial failures

Dealing with Distributed Programming

- We have seen several of the tools that help with distributed programming
  - Message Passing Interface (MPI)
  - Distributed Shared Memory (DSM)
  - Remote Procedure Calls (RPC)

- But, distributed programming is still very hard
  - Programming for scale, fault-tolerance, consistency, …

The Datacenter is the new Computer



- *"Program"* == Web search, email, map/GIS, …
- *"Computer"* == 10,000's computers, storage, network
- Warehouse-sized facilities and workloads
- *Built from less reliable components than traditional datacenters*

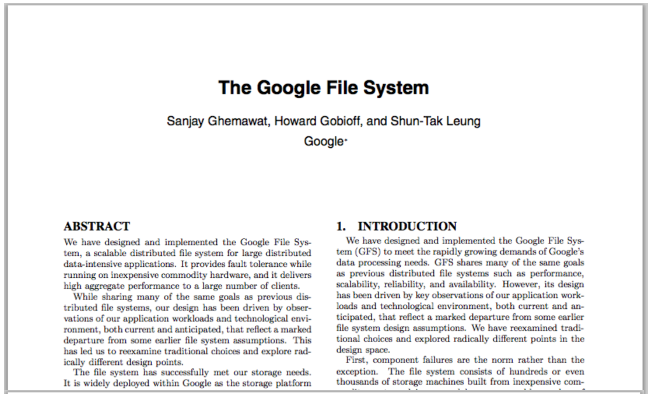
Datacenter/Cloud Operating System

- Data sharing
  - Google File System, key/value stores
- Programming Abstractions
  - Google MapReduce, PIG, Hive, Spark
- Multiplexing of resources
  - Apache projects: Mesos, YARN (MapReduce v2), ZooKeeper, BookKeeper, …

Google Cloud Infrastructure

- Google File System (GFS), 2003
  - Distributed File System for entire cluster
  - Single namespace

- Google MapReduce (MR), 2004
  - Runs queries/jobs on data
  - Manages work distribution & fault-tolerance
  - Colocated with file system



The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

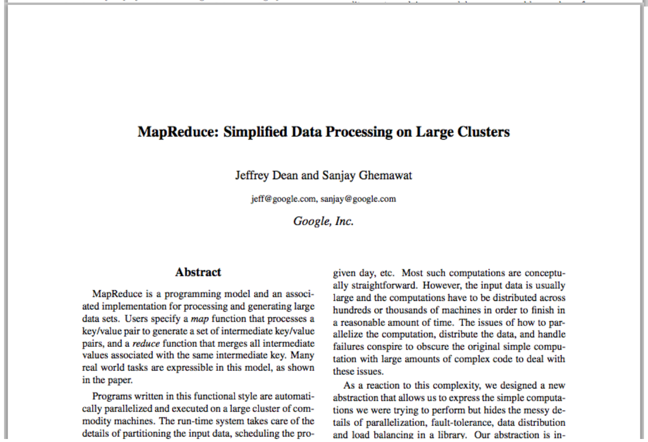
While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive com-



MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the pro-

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is in-

- Apache open source versions Hadoop DFS and Hadoop MR

GFS/Hadoop DFS Insights

- *Petabyte* storage
  - Files split into large blocks (128 MB) and replicated across several nodes
  - Big blocks allow high throughput sequential reads/writes
- Data *striped* on hundreds/thousands of servers
  - Scan 100 TB on 1 node @ 50 MB/s = 24 days
  - Scan on 1000-node cluster = 35 minutes

GFS/HDFS Insights (2)

- *Failures* will be the norm
  - Mean time between failures for 1 node = 3 years
  - Mean time between failures for 1000 nodes = 1 day
- Use *commodity* hardware
  - Failures are the norm anyway, buy cheaper hardware
- No complicated consistency models
  - Single writer, append-only data

MapReduce Model

- Data type: key-value **records**
- **Map** function:

$$(K_{in}, V_{in}) \rightarrow list(K_{inter}, V_{inter})$$

- Group all identical K_{inter} values and pass to reducer
- **Reduce** function:

$$(K_{inter}, list(V_{inter})) \rightarrow list(K_{out}, V_{out})$$

Example: Word Count

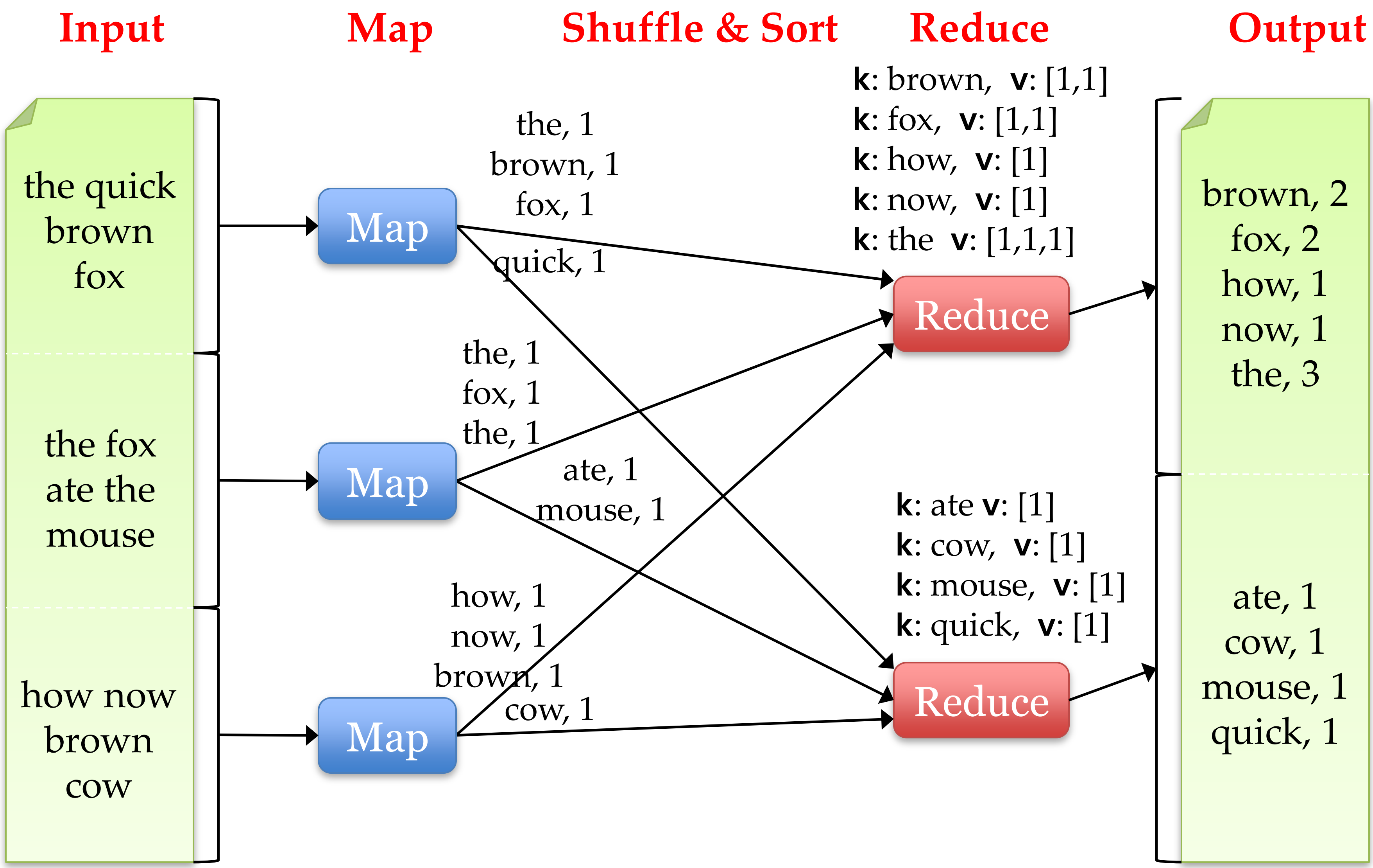
Input: key is filename, value is a line in input file

```
def mapper(file, line):
    foreach word in line.split():
        output(word, 1)
```

Intermediate: key is a word, value is 1

```
def reducer(key, values):
    output(key, sum(values))
```

Word Count Execution



MapReduce Insights

- Restricted key-value model
  - Same **fine-grained operation** (Map & Reduce) repeated on big data
  - Operations must be **deterministic**
  - Operations must be **idempotent/no side effects**
    - **Idempotent:** means an operation can be applied multiple times without changing the result beyond the initial application
  - Only communication is through the shuffle
  - Operation (Map & Reduce) output saved (on disk)

What is MapReduce Used For?

- At **Google**:
  - Index building for Google Search
  - Article clustering for Google News
  - Statistical machine translation

- At **Yahoo!**:
  - Index building for Yahoo! Search
  - Spam detection for Yahoo! Mail

- At **Facebook**:
  - Data mining
  - Ad optimization
  - Spam detection

MapReduce Pros

- Distribution is completely **transparent**
  - Not a single line of distributed programming (ease, correctness)
- Automatic **fault-tolerance**
  - Determinism enables running failed tasks somewhere else again
  - Saved intermediate data enables just re-running failed reducers
- Automatic **scaling**
  - As operations as side-effect free, they can be distributed to any number of machines dynamically
- Automatic **load-balancing**
  - Move tasks and speculatively execute duplicate copies of slow tasks (*stragglers)*

MapReduce Cons

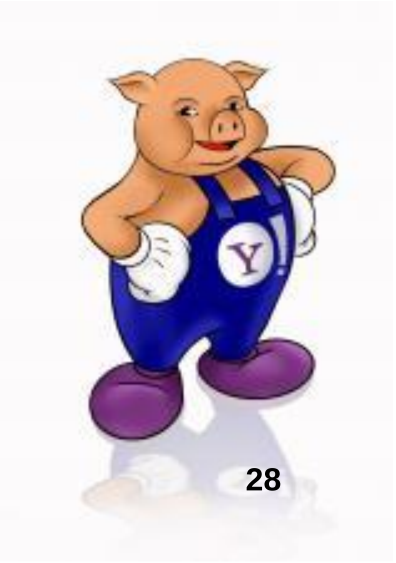
- Restricted programming model
  - Not always natural to express problems in this model
  - Low-level coding necessary
  - Little support for iterative jobs (lots of disk access)
  - High-latency (batch processing)

- Addressed by follow-up research
  - **Pig** and **Hive** for high-level coding
  - **Spark** for iterative and low-latency jobs

Pig

- High-level language:
  - Expresses sequences of MapReduce jobs
  - Provides relational (SQL) operators (JOIN, GROUP BY, etc)
  - Easy to plug in Java functions

- Started at Yahoo! Research
  - Runs about 50% of Yahoo!’s jobs



Hive

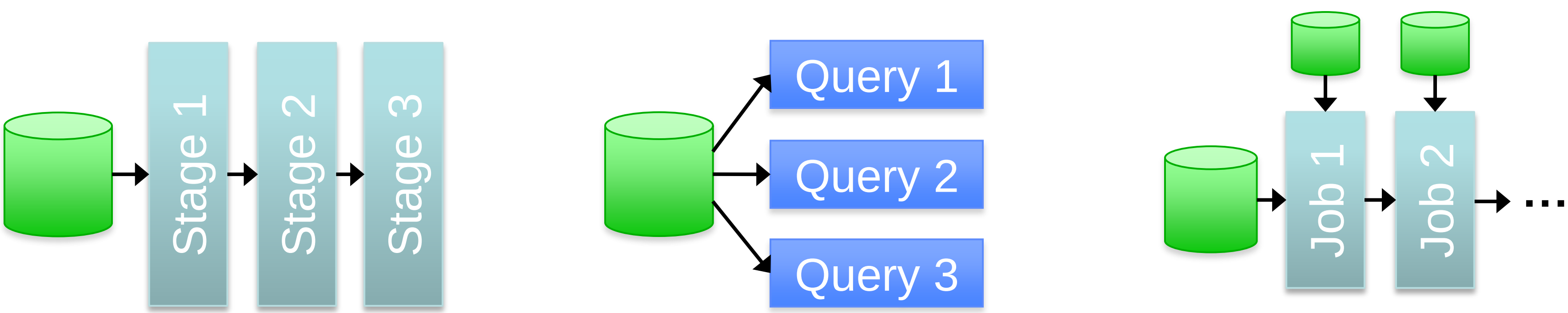
- Relational database built on Hadoop
  - Maintains table schemas
  - SQL-like query language (which can also call Hadoop Streaming scripts)
  - Supports table partitioning, complex data types, sampling, some query optimization

- Developed at Facebook
  - Used for many Facebook jobs

Spark Motivation

Complex jobs, interactive queries and online processing all need one thing that MapReduce lacks:

Efficient primitives for **data sharing**



Iterative job **Interactive mining** **Stream processing**

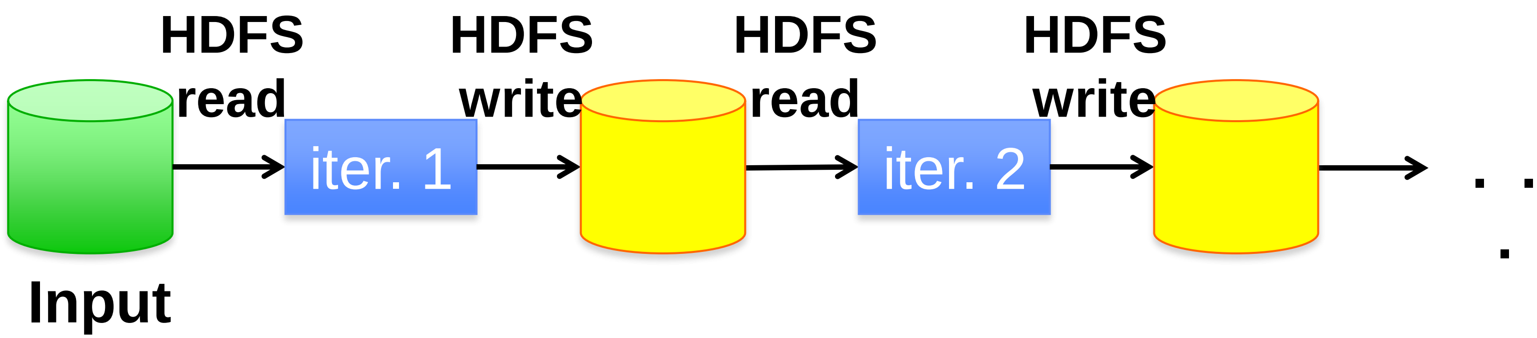
Spark Motivation

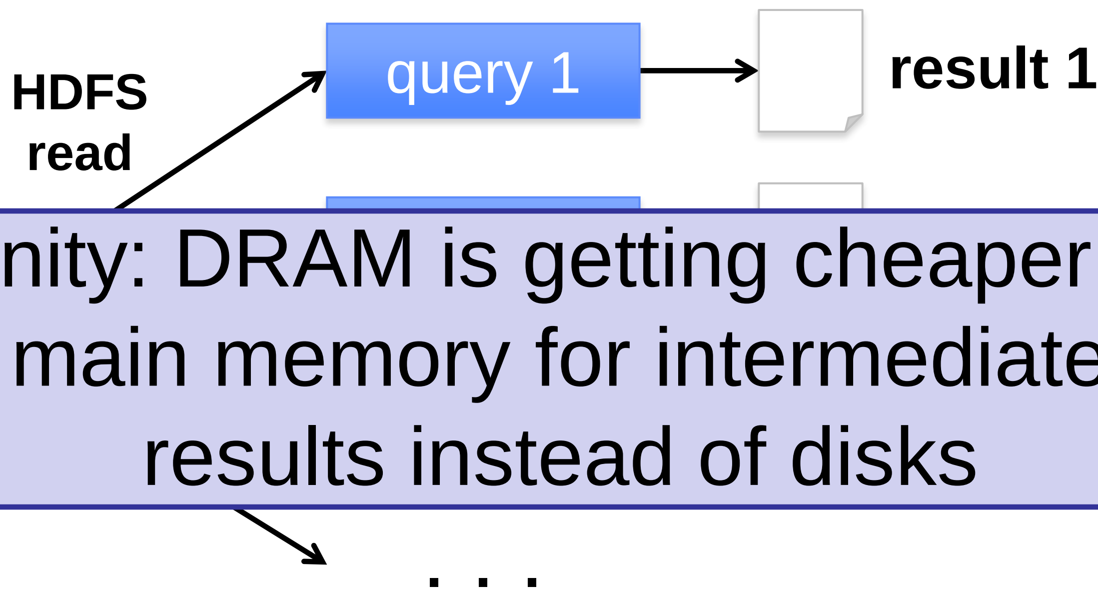
Complex jobs, interactive queries and online processing all need one thing that MapReduce lacks:

Efficient primitives for **data sharing**

Problem: in MapReduce, the only way to share data across jobs is using stable storage (e.g. file system) ➔ slow!

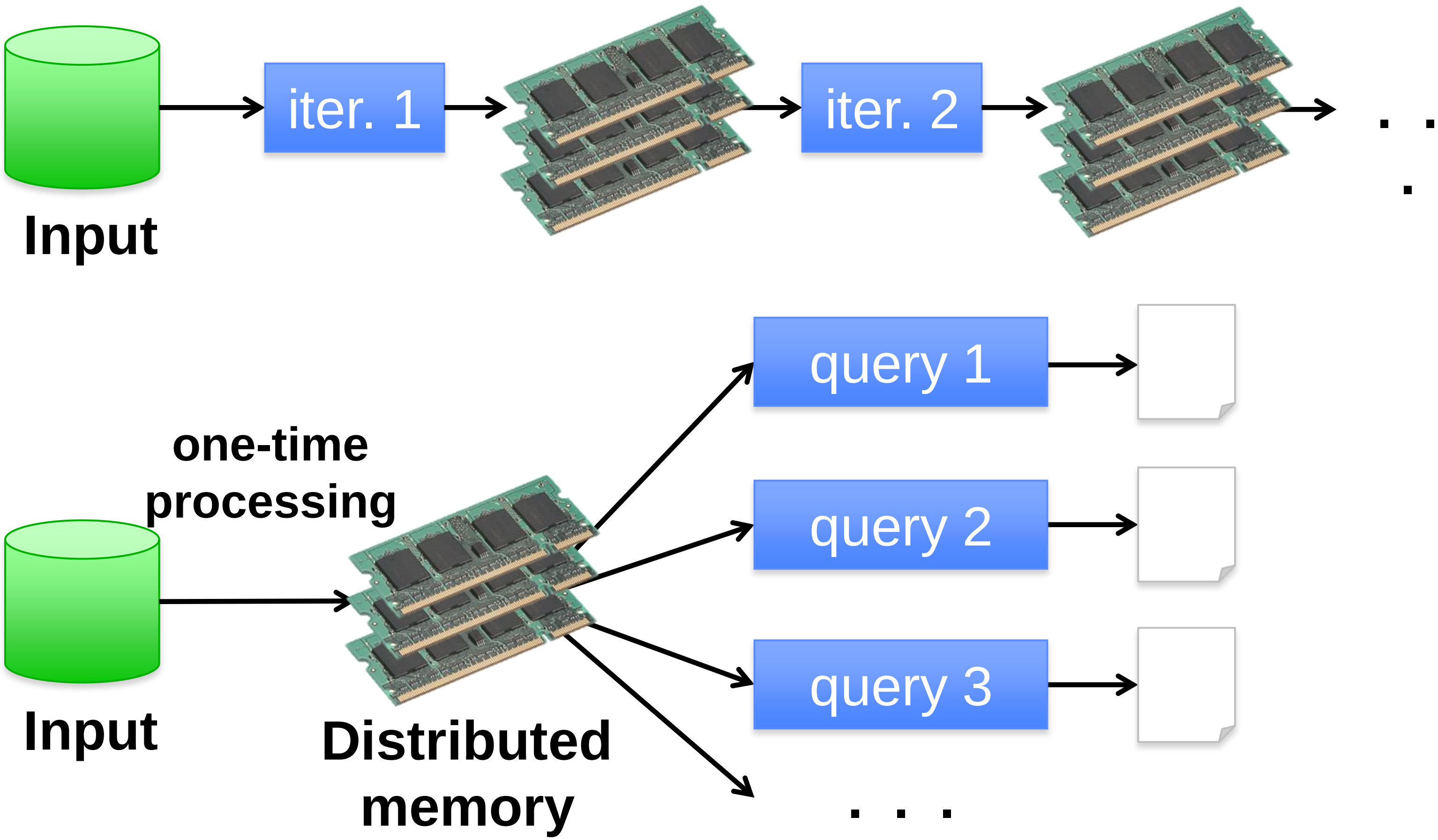
Examples





Opportunity: DRAM is getting cheaper ➔ use main memory for intermediate results instead of disks

Goal: In-Memory Data Sharing



10-100 × faster than network and disk

Datacenter Scheduling Problem

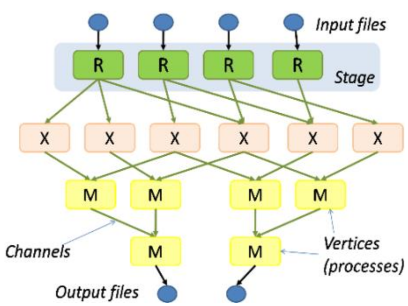
- Rapid innovation in datacenter computing frameworks
- **No single framework optimal for all applications**
- Want to run multiple frameworks in a single datacenter
  - …to maximize utilization
  - …to share data between frameworks

hadoop MapReduce

Google Pregel

Pig

RAILS



Dryad

CIEL

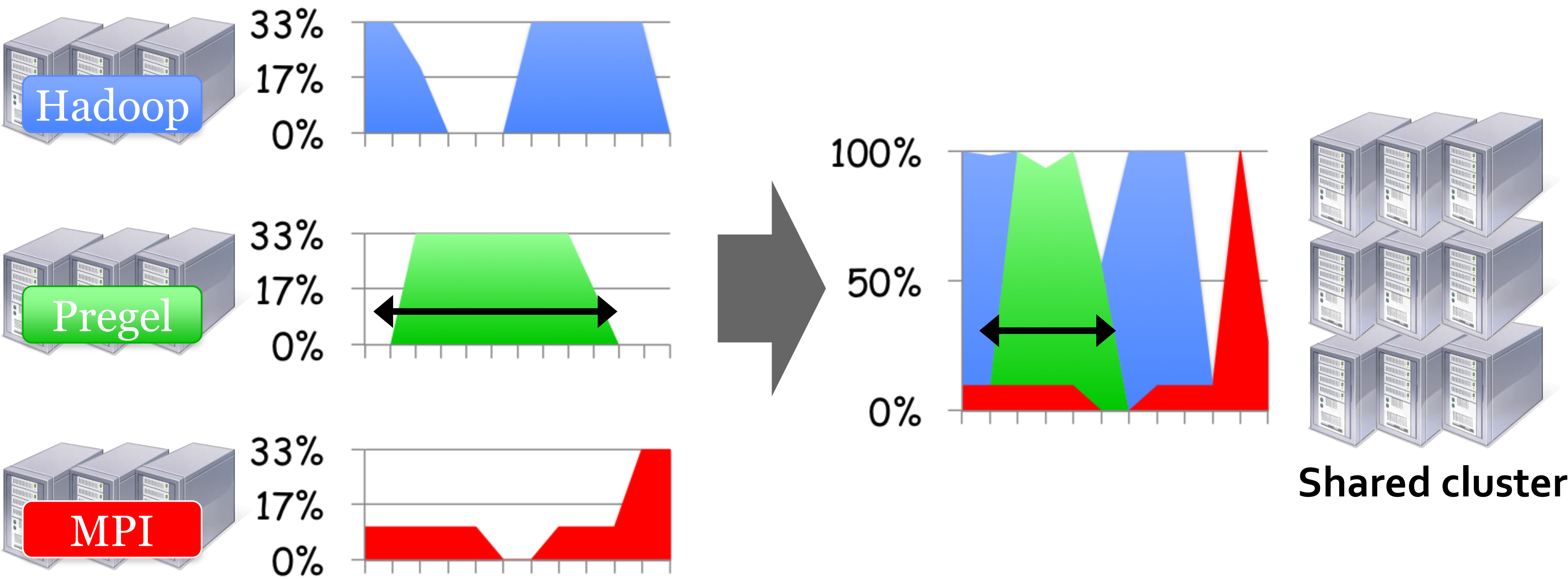
S4 distributed stream computing platform

Google Percolator

MPI-2

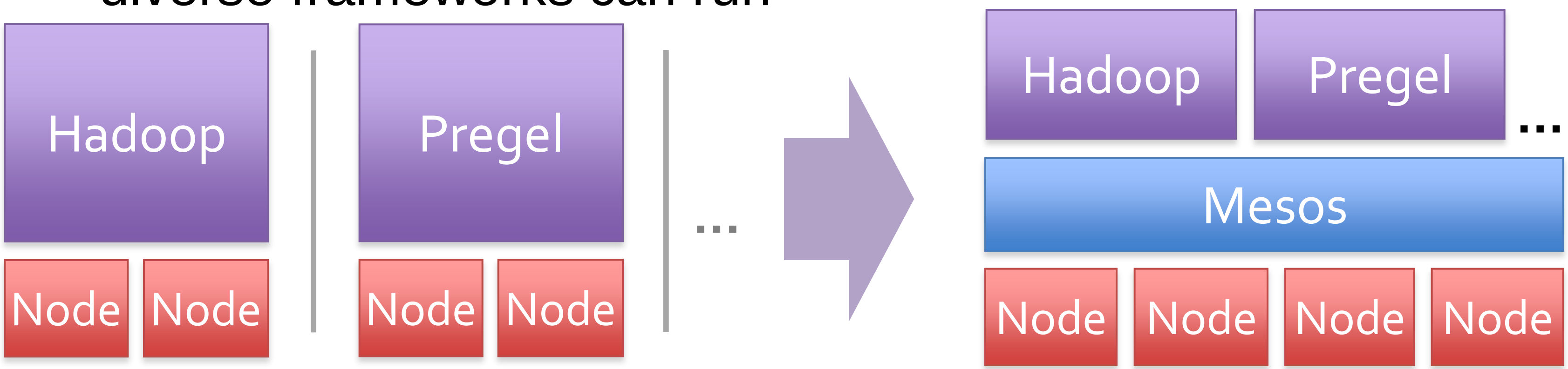
Where We Want to Go

Today: static partitioning **Dynamic sharing**



Solution: Apache Mesos

- Mesos is a common resource sharing layer over which diverse frameworks can run



- Run multiple instances of the *same* framework
  - Isolate production and experimental jobs
  - Run multiple versions of the framework concurrently

- Build *specialized frameworks* targeting particular problem domains
  - Better performance than general-purpose abstractions

Mesos Goals

- **High utilization** of resources
- **Support diverse frameworks** (current & future)
- **Scalability** to 10,000's of nodes
- **Reliability** in face of failures

http://incubator.apache.org/mesos/

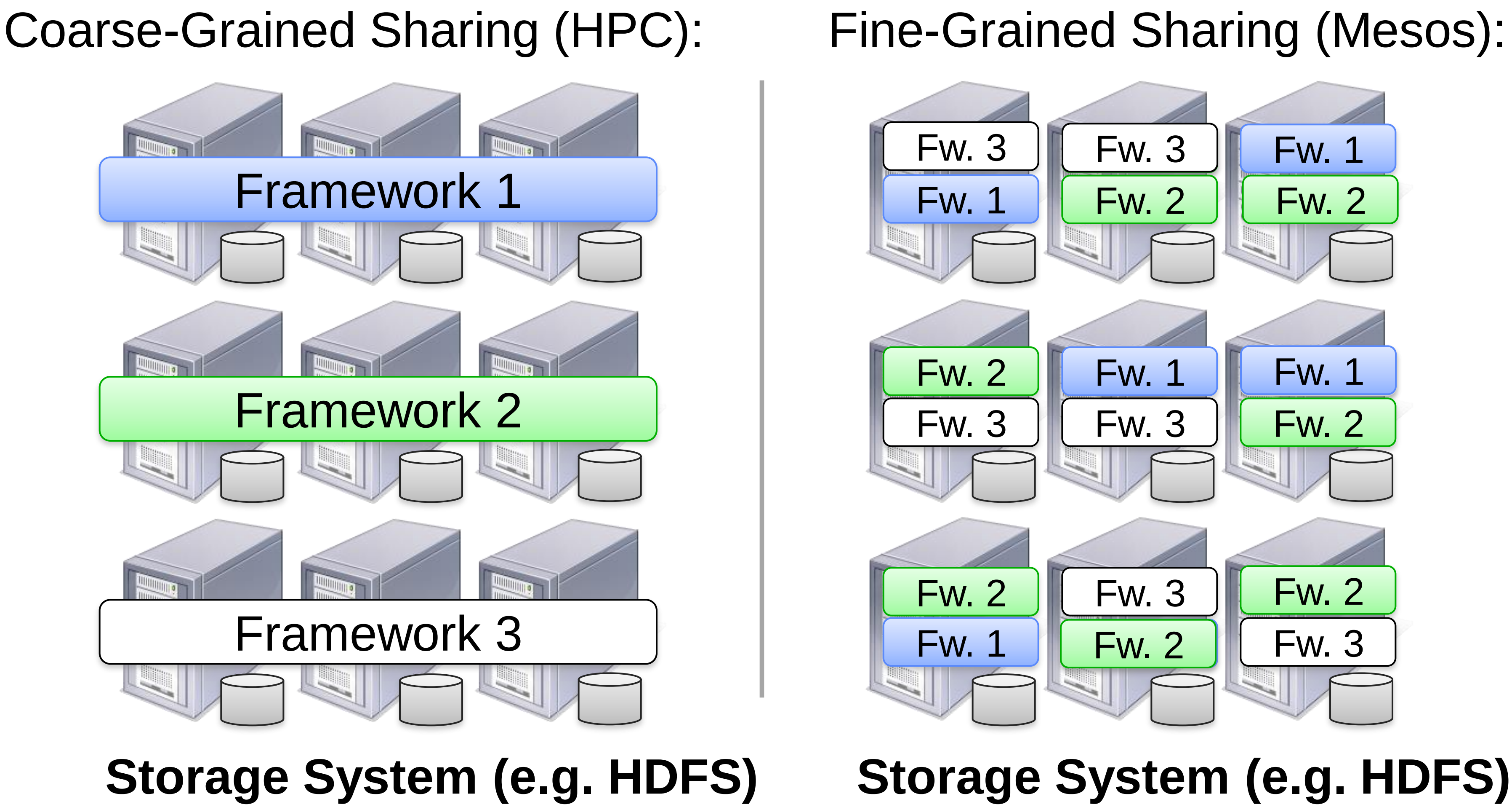
Resulting design: Small microkernel-like core that pushes scheduling logic to frameworks

Mesos 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



+ 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

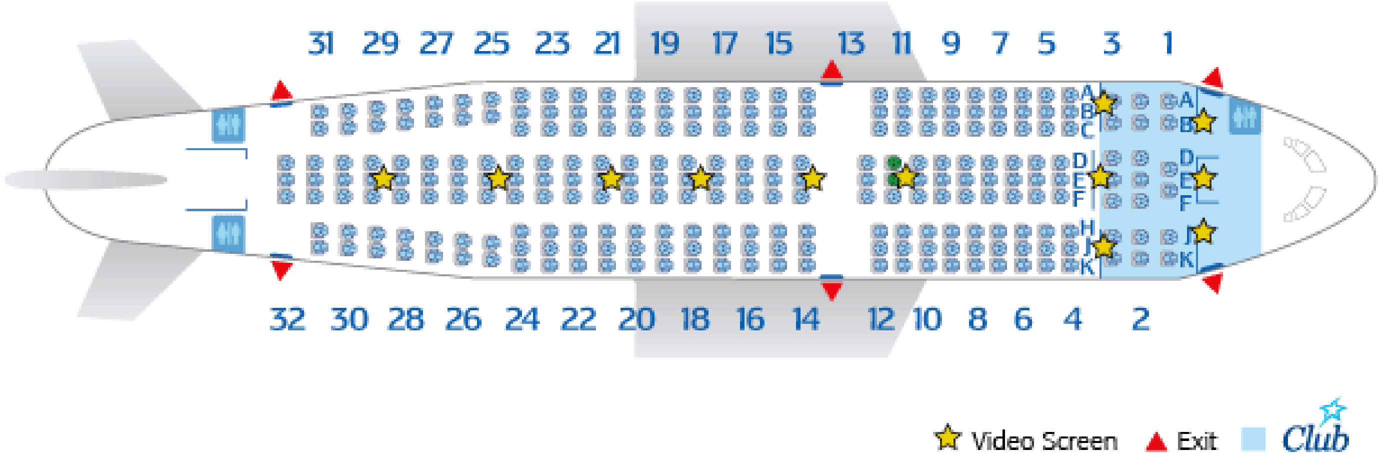
– Future frameworks may have unanticipated needs

Element 2: Resource Offers

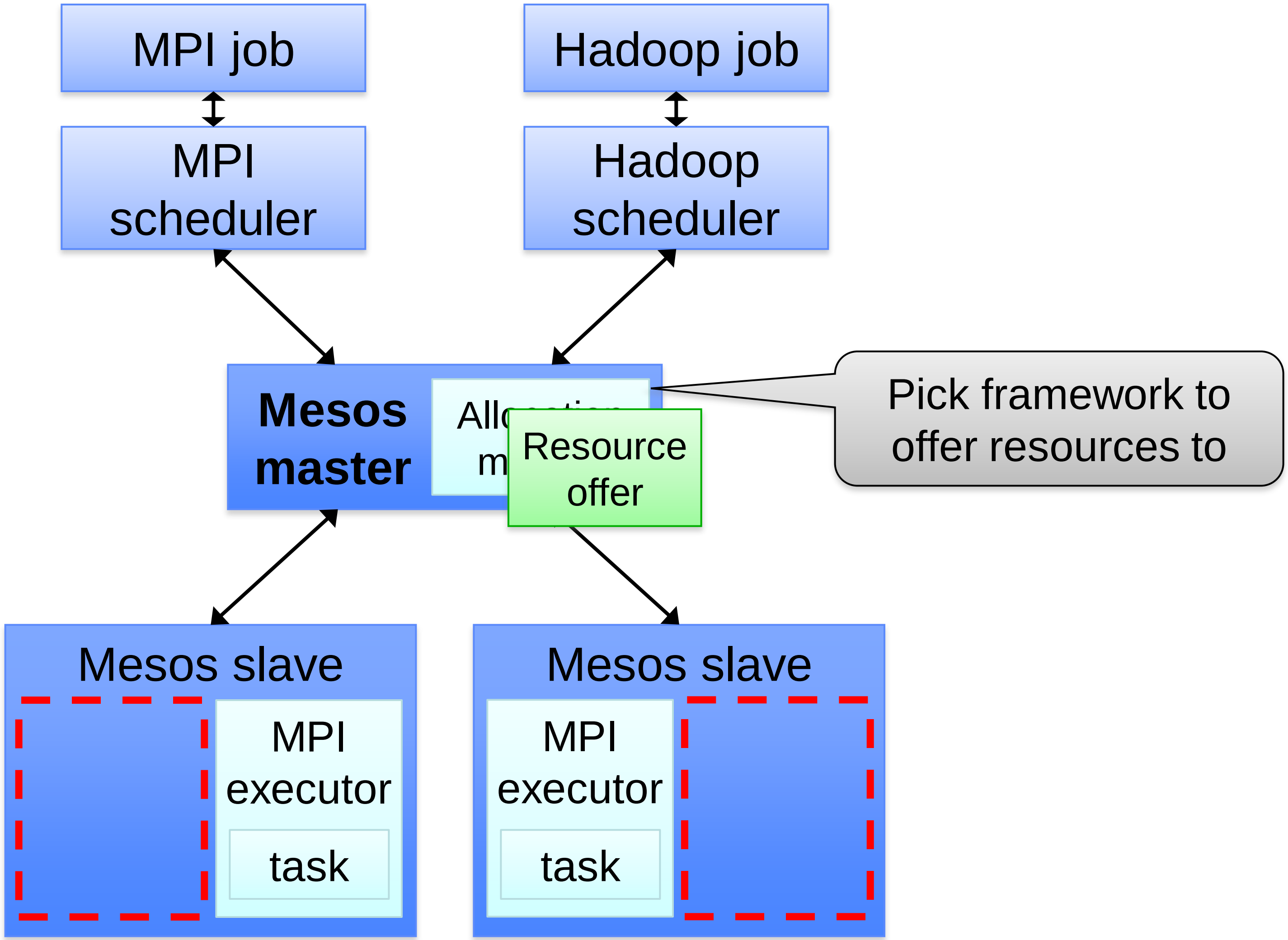
- Mesos: Resource offers
  - Offer available resources to frameworks, let them pick which resources to use and which tasks to launch

  \+ Keeps Mesos simple, lets it support future frameworks

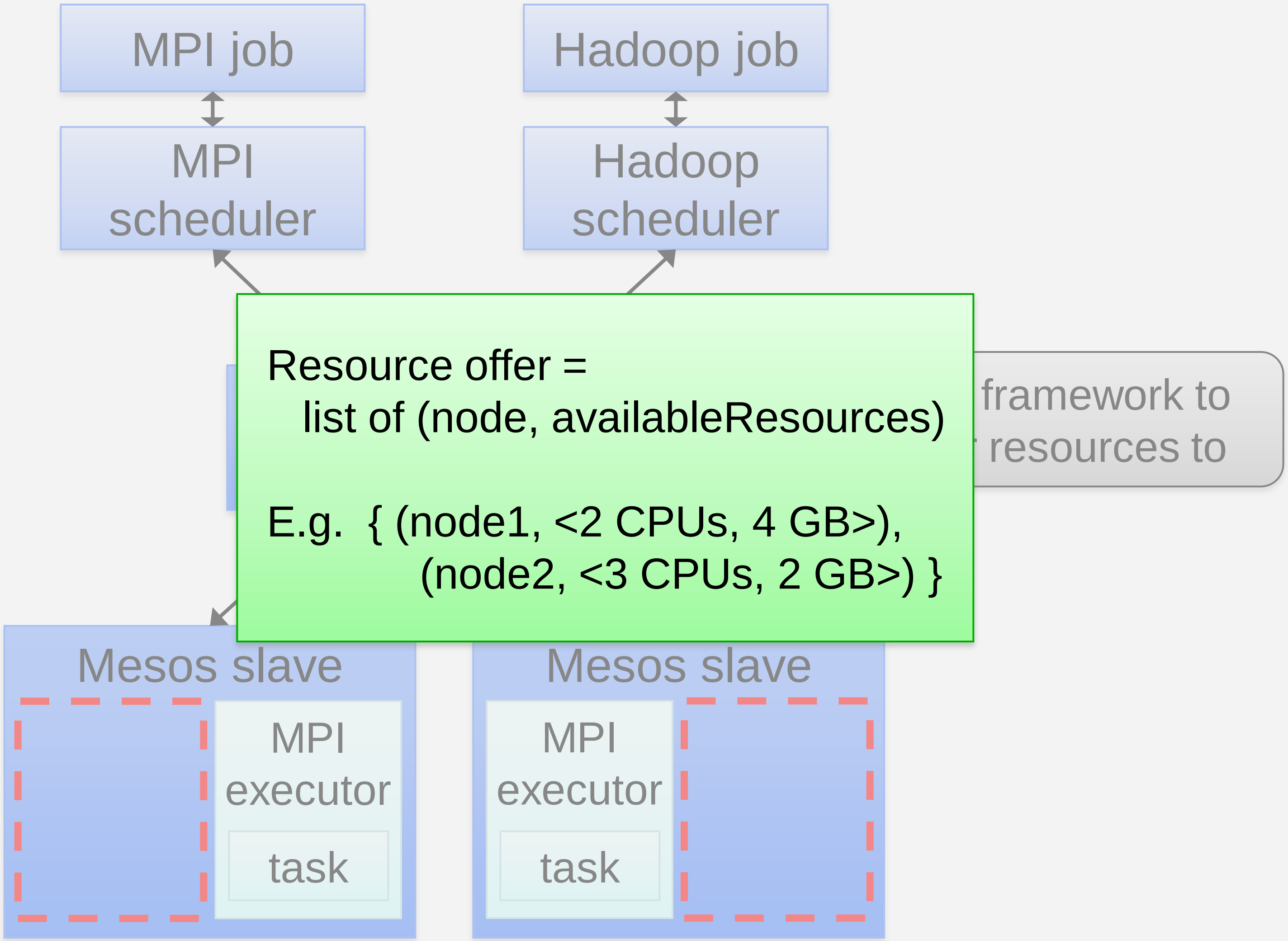
  \- Decentralized decisions might not be optimal



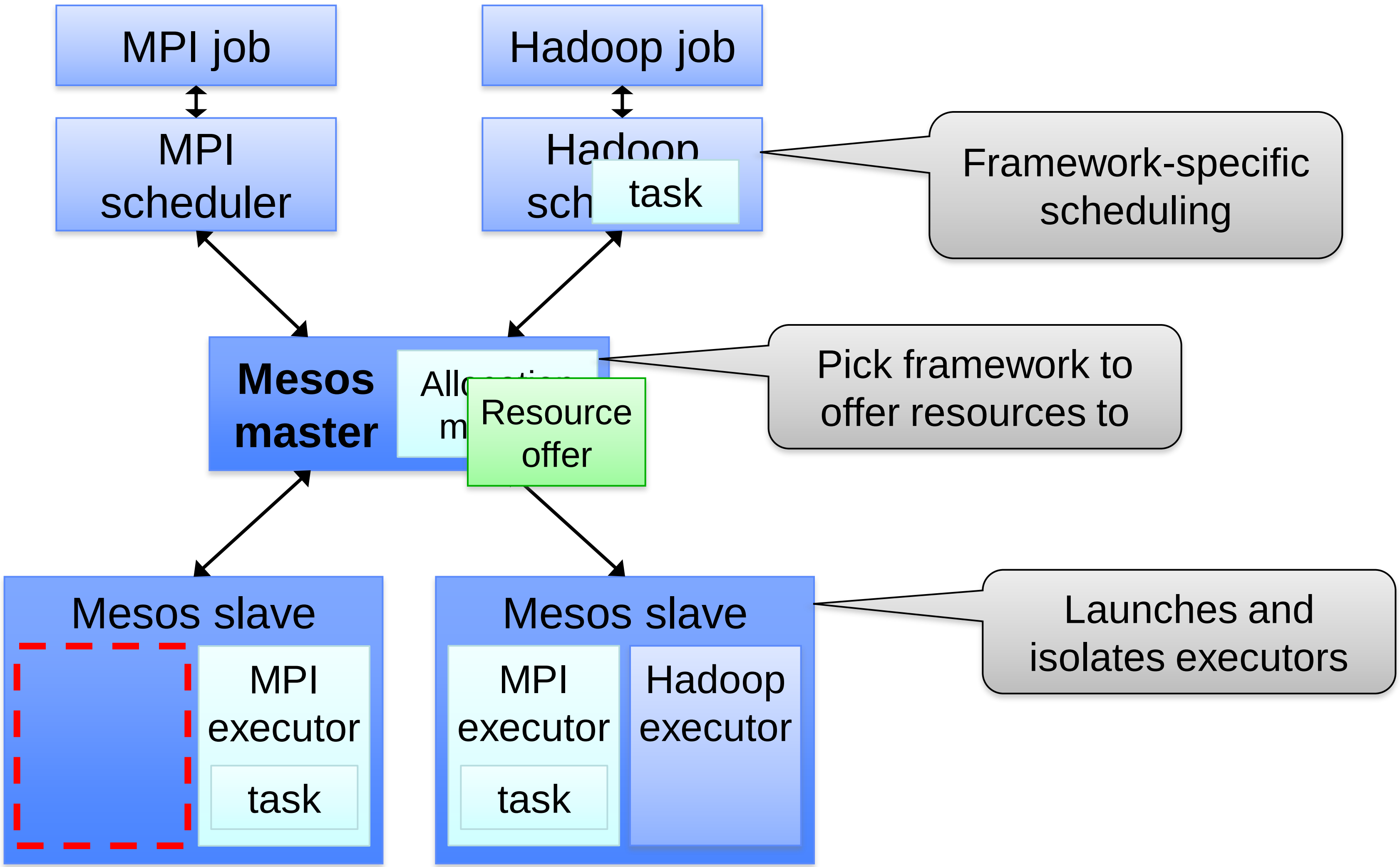
Mesos Architecture



Mesos Architecture



Mesos Architecture



Summary

- Cloud computing/datacenters are the new computer
  – Emerging “Datacenter/Cloud Operating System” appearing

- Pieces of the DC/Cloud OS
  – High-throughput filesystems (GFS/HDFS)
  – Job frameworks (MapReduce, Spark, Pregel)
  – High-level query languages (Pig, Hive)
  – Cluster scheduling (Apache Mesos)