

Distributed Systems and Cloud Computing

Lecture 1

Introduction

Lecture Plan:

- **What is Cloud**
 - Cloud Computing
 - Cloud Computing Services
 - Why Cloud?
 - Drawbacks
- **Distributed Systems**
 - Hadoop
 - Map/Reduce

A Cloud is ...

- **Datacenter hardware and software** that the vendors use to offer the computing resources and services



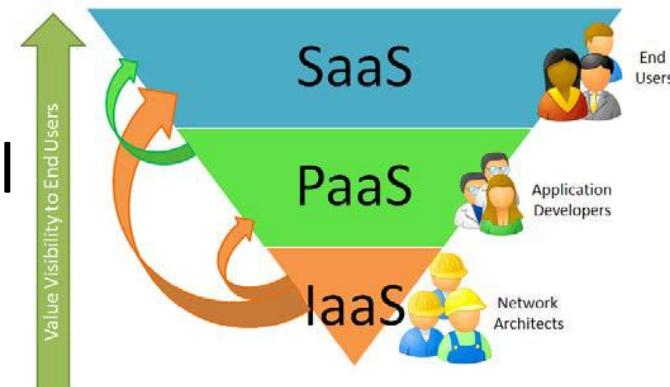
Cloud Computing

- Represents both the cloud & the provided services
- Why call it “cloud computing”?
 - Wikipedia: "the term derives from the fact that most technology diagrams depict the Internet or IP availability by using a drawing of a cloud."

Cloud Computing Services

Three basic services:

- Software as a Service (SAAS) model
 - Apps through browser
- Platform as a Service (PAAS) model
 - Delivery of a computing platform for custom software development as a service
- Infrastructure as a Service (IAAS) model
 - Deliver of computer infrastructure as a service
- XAAS, the list continues to grow...



On-Premises

Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking

Infrastructure as a Service

Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking

Platform as a Service

Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking

Software as a Service

Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking

You Manage

Other Manages

SaaS

- Started around 1999
- Application is licensed to a customer as a service on demand
- Software Delivery Model:
 - Hosted on the vendor's web servers
 - Downloaded at the consumer's device and disabled when on-demand contract is over

PaaS

- **Delivery of an integrated computing platform (to build/test/deploy custom apps) & solution stack as a service.**
- **Deploy your applications & don't worry about buying & managing the underlying hardware and software layers**

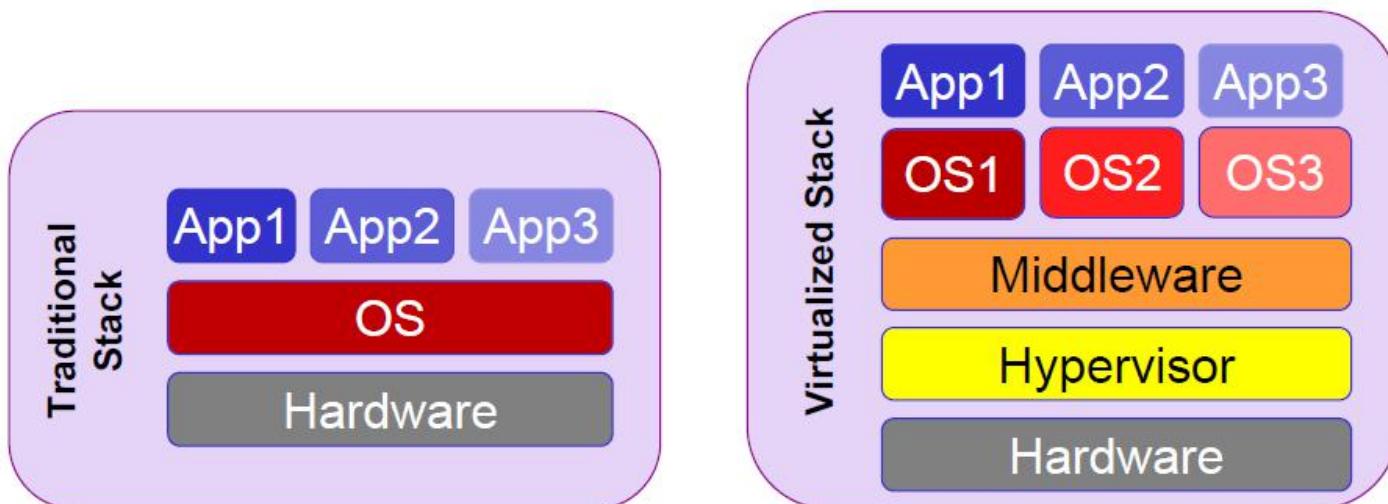
IaaS

- **Delivery of computer infrastructure (typically platform virtualization environment) as a service**
- **Buy resources**
 - Servers
 - Software
 - Data center space
 - Network equipment as fully outsourced services
- **Example:**



IaaS

- Virtualization Technology is a major enabler of IaaS
 - It's a path to share IT resource pools: Web servers, storage, data, network, software and databases.
 - Higher utilization rates



Enabling Technologies

- Virtualization
- Distributed Storage
- Distributed Computing
- Utility Computing
- Network Bandwidth & Latency
- Fault-Tolerant Systems.

Why Cloud Computing?

- Large-Scale Data-Intensive Applications
- Flexibility
- Scalability
- Customized to your current needs:
 - Hardware
 - Software
- Effect:
 - Reduce Cost
 - Reduce Maintenance
 - High Utilization
 - High Availability
 - Reduced Carbon Footprint.

Why Cloud Computing?

- Flexibility
 - Software: Any software platform
 - Access: access resources from any machine connected to the Internet
 - Deploy infrastructure from anywhere at anytime
 - Software controls infrastructure

Why Cloud Computing?

- Scalability
 - Instant
 - Control via software
 - Add/cancel/rebuild resources instantly
 - Start small, then scale your resources up/down as you need
 - illusion of infinite resources available on demand

Why Cloud Computing?

- Customization
 - Everything in your wish list
 - Software platforms
 - Storage
 - Network bandwidth
 - Speed

Why Cloud Computing?

- Cost
 - Pay-as-you-go model
 - Small/medium size companies can tap the infrastructure of corporate giants
 - Time to service/market
 - No upfront cost

Why Cloud Computing?

- Maintenance
 - Reduce the size of a client's IT department
 - Is the responsibility of the cloud vendor
 - This Includes:
 - Software updates
 - Security patches
 - Monitoring system's health
 - System backup
 - ...etc

Why Cloud Computing?

- Availability
 - Having access to software, platform, infrastructure from anywhere at any time
 - All you need is a device connected to the internet
- Reliability

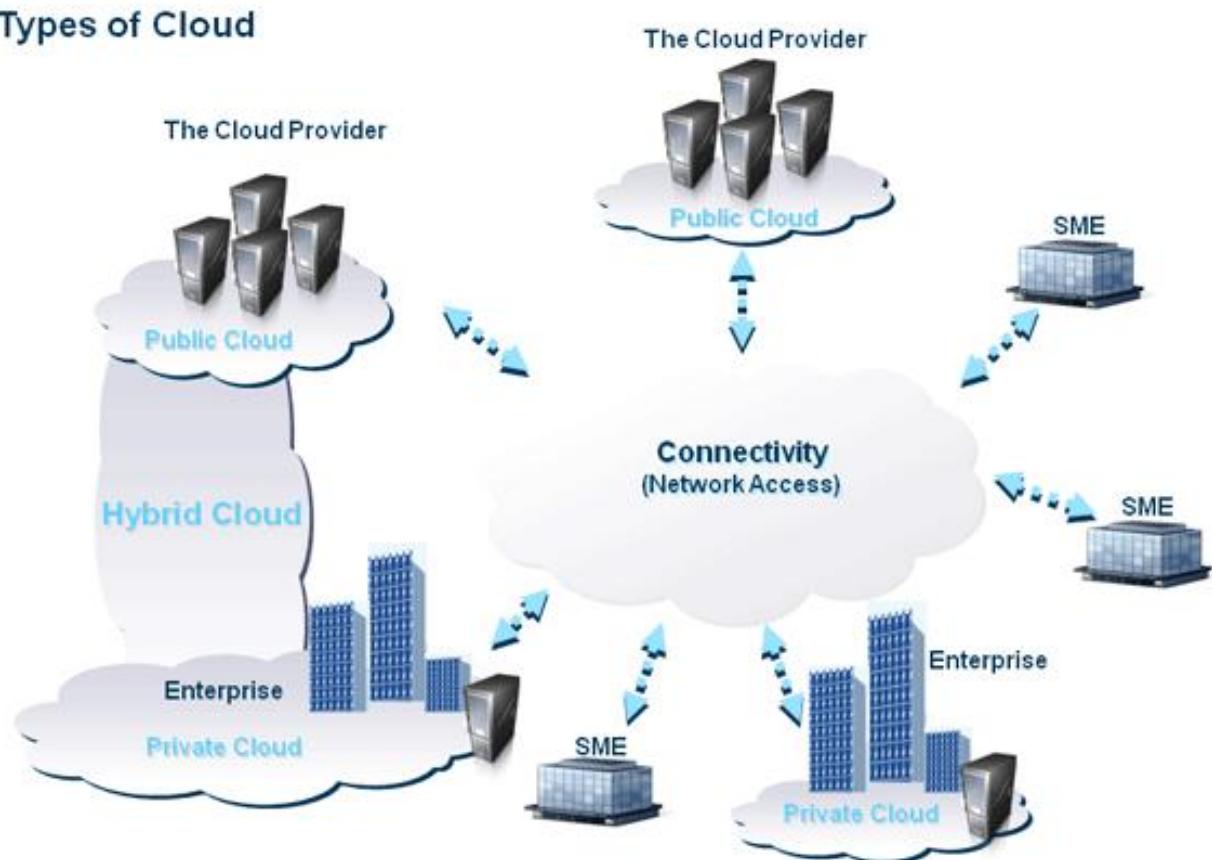
The system's fault tolerance is managed by the cloud providers and users no longer need to worry about it.

Drawbacks

- Security
- Privacy
- Vendor lock-in
- Network-dependent
- Migration.

Types of Clouds

- Public
- Private
- Hybrid
- Community.



DISTRIBUTED SYSTEMS (HADOOP)

What is Hadoop?

- Apache top level project, open-source implementation of frameworks for reliable, scalable, distributed computing and data storage.
- It is a flexible and highly-available architecture for large scale computation and data processing on a network of commodity hardware.



Hadoop's Developers

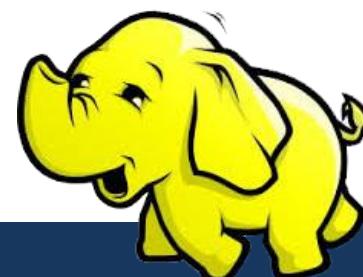


2005: Doug Cutting and Michael J. Cafarella developed Hadoop to support distribution for the [Nutch](#) search engine project.

The project was funded by Yahoo.



2006: Yahoo gave the project to Apache Software Foundation.



Google Origins

2003

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
Google



2004

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

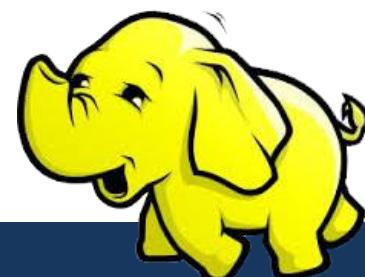
Google, Inc.



2006

Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber
(fay,jeff,sanjay,wilson,debo,wallach,mike,tushar,afikes,robert)@google.com
Google, Inc.



Abstract
Bigtable is a distributed storage system for managing structured data in a highly dynamic environment, supporting petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to entire terabytes for satellite imagery) and latency requirements

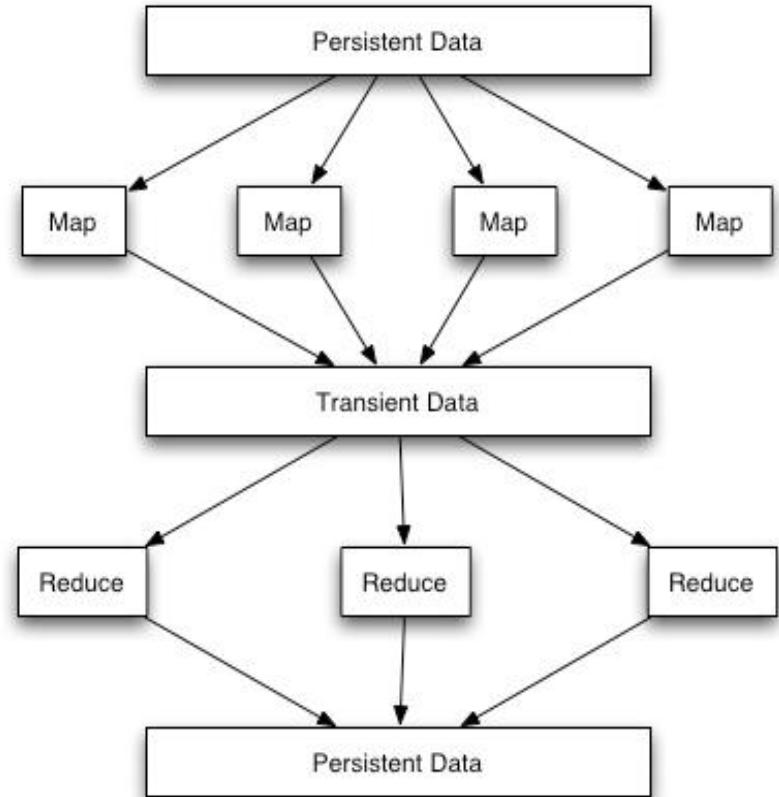
achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead it provides clients with a simple data model that supports dynamic control over data layout and format, allowing clients to reason about the locality properties of data represented in the underlying storage. Data is denoted using row and column names that can be arbitrary strings. Bigtable also treats data as uninterpreted strings.

What is MapReduce

- MapReduce: Abstraction that simplifies writing applications that access massively distributed data

MapReduce Programming Model

- Functional programming that is easily parallelizable
- Split into two phases:
 - Map – Perform custom function on all items in an array
 - Reduce – Collate map results using custom function
- Scales well – computation separated from processing dataflow
- Illustrative example:
 - Map that squares the value of numbers in an array
 $\{1, 2, 3, 4\} \rightarrow \{1, 4, 9, 16\}$
 - Reduce that sums the squares : 30



Additional Materials

- Coulouris, G. F., Dollimore, J., & Kindberg, T. Distributed systems: concepts and design. 5th edition, Pearson Education.
- Fokkink, W. Distributed Algorithms: An Intuitive Approach. MIT Press.
- Erl, T., Puttini, R., & Mahmood, Z. Cloud Computing: Concepts, Technology, & Architecture. Pearson Education.

Lecture 2

Lecture Plan

- Service Oriented Architecture
- RESTful service
 - Rest design pattern
 - Terminology
 - HTTP mapping
- Microservice
 - Monoliths
 - Microservice

Service Oriented Architecture (SOA)

- A means of developing distributed systems where the components are stand-alone services
- Services may execute on different computers from different service providers
- Standard protocols have been developed to support service communication and information exchange

Service-Oriented Architecture (SOA)

Enterprise Architecture Integration(EAI)

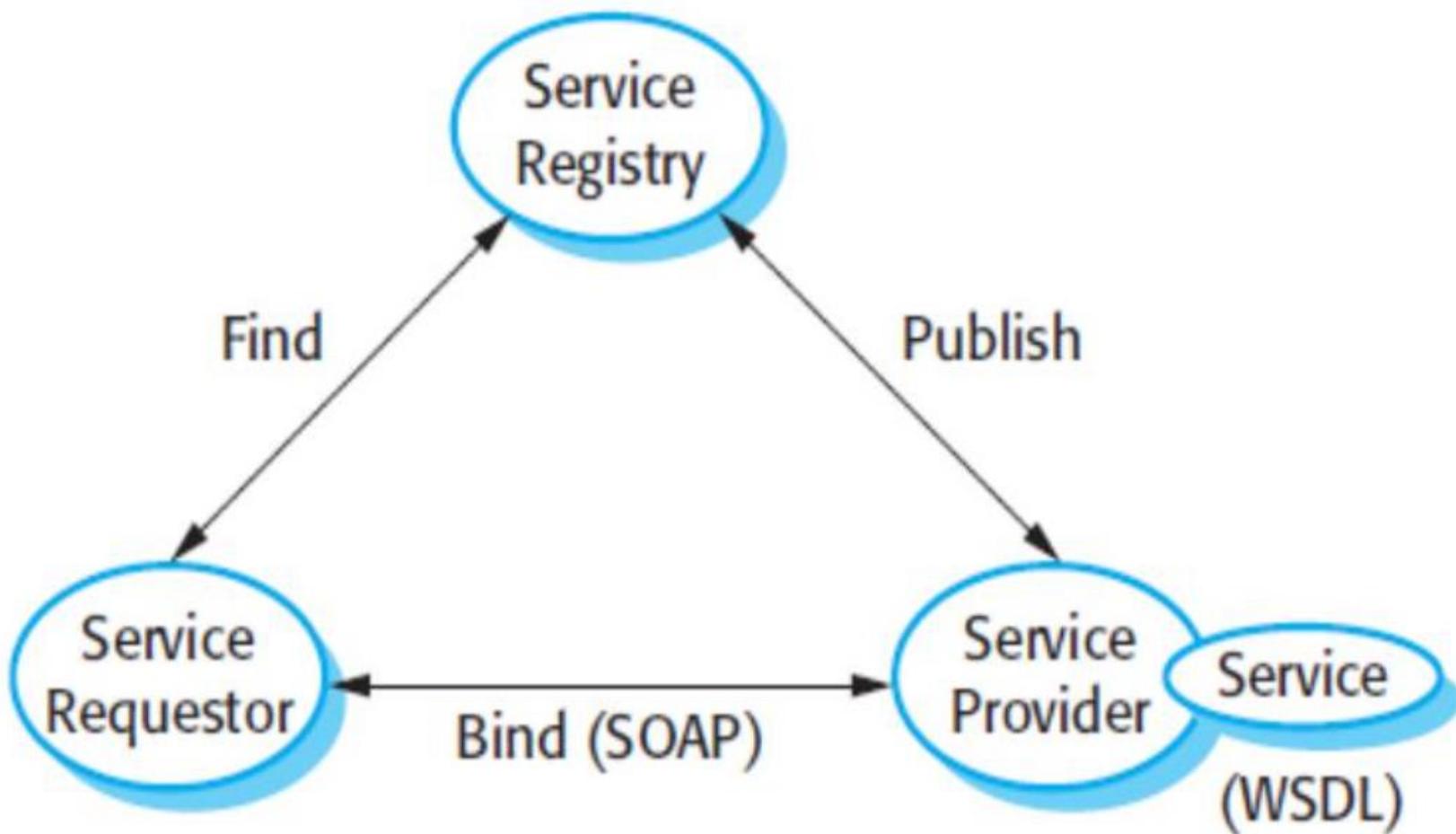
Enterprise Service Bus (EBS)

Containerization
(Google Kubernetes, Docker Swarm, Apache Mesos)

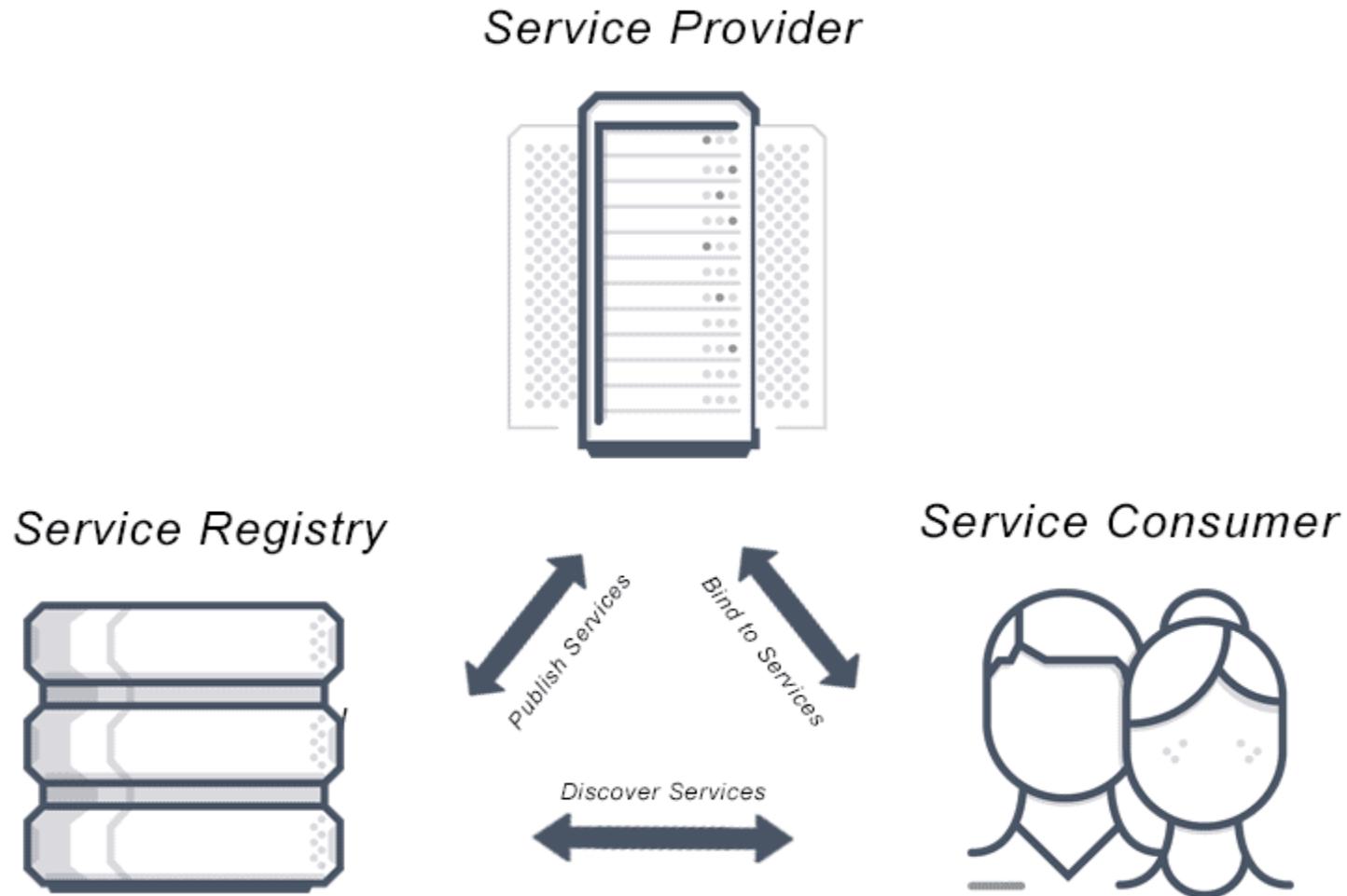
Web Services
(W3C: WSDL/UDDI/SOAP, OASIS: WS-*, RESTful)

Microservices

Service-Oriented Architecture (SOA)

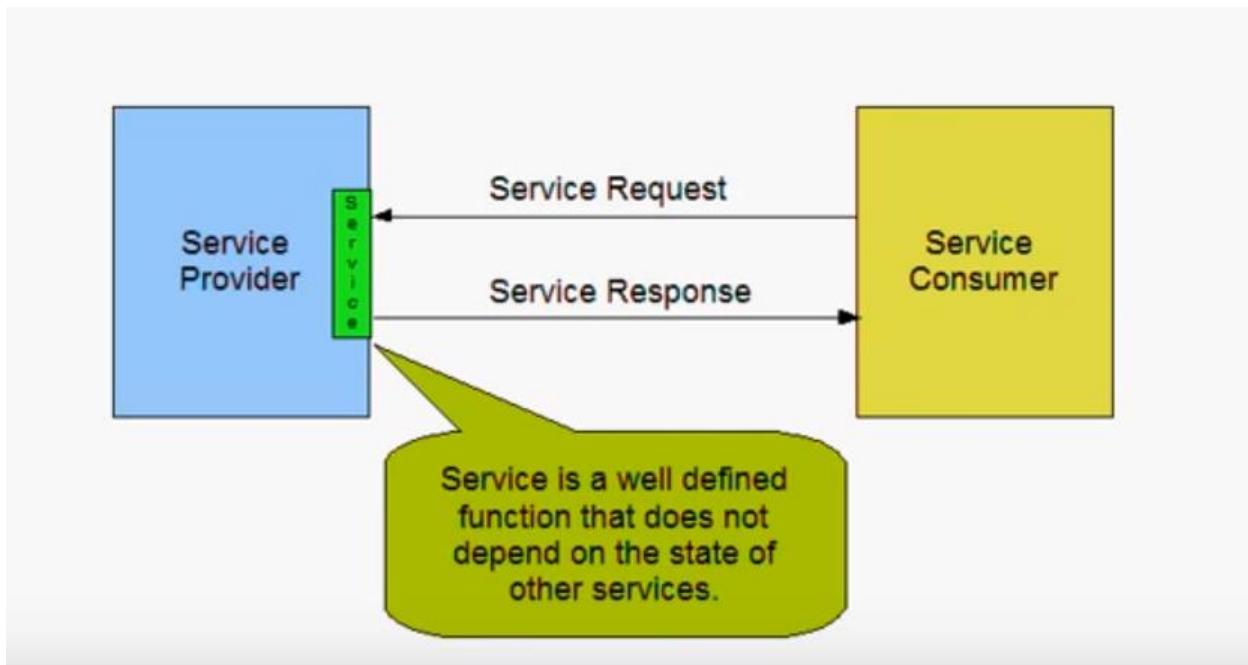


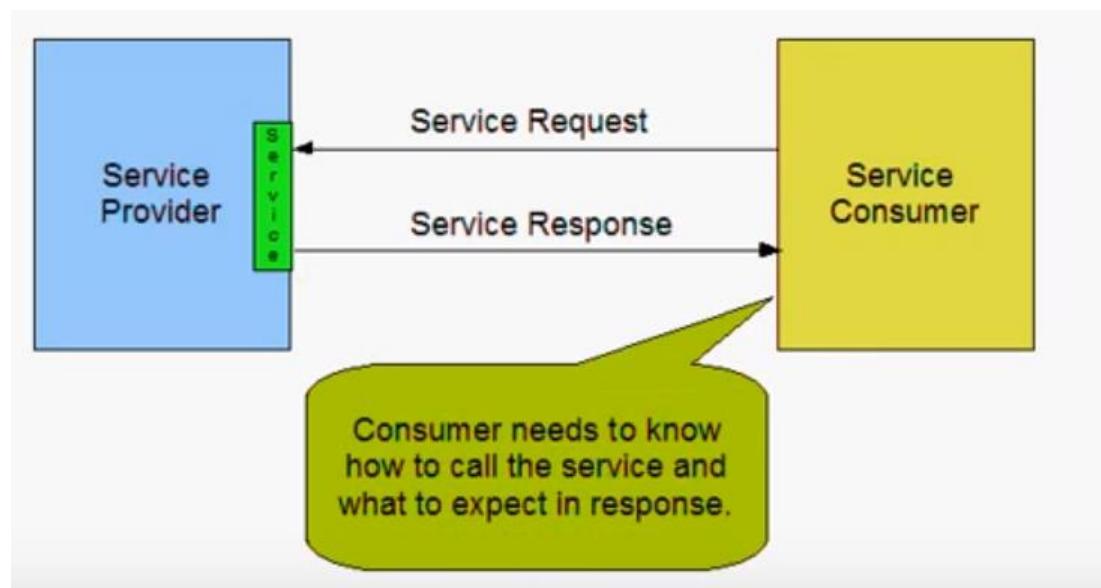
The Service Oriented Architecture Triangle

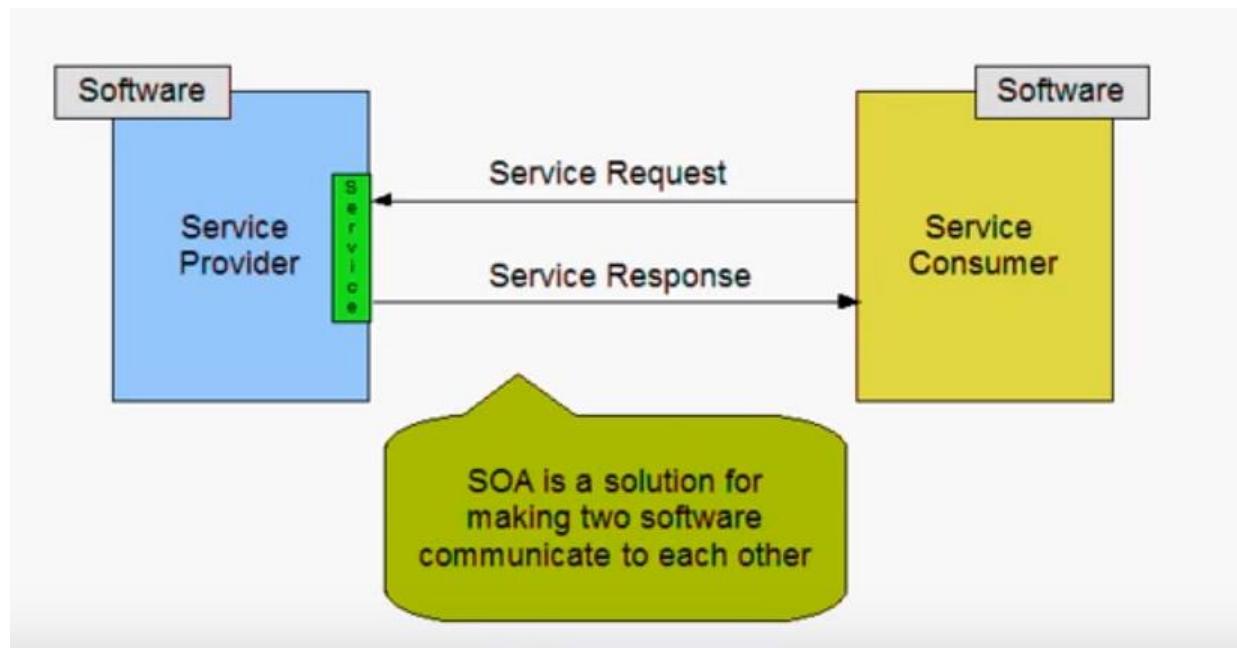


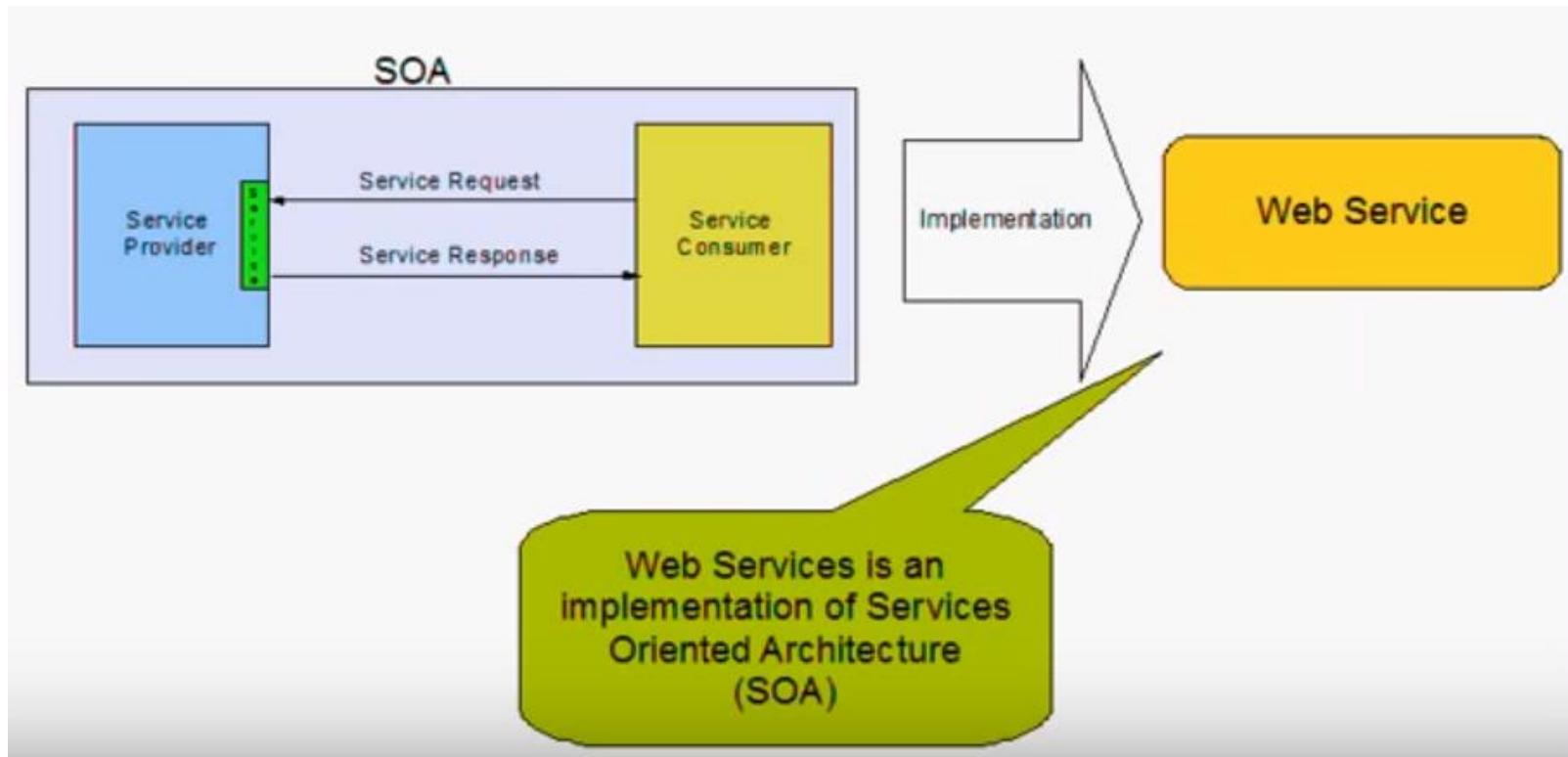
Key Standards

- Simple Object Access Protocol (SOAP)
 - A message exchange standard that supports service communication
 - Defines a uniform way of passing XML-encoded data
 - Defines a way to bind HTTP as the underlying communication protocol
- Web Service Definition Language (WSDL)
 - This standard allows a service interface and its bindings to be defined
- Web Services - Business Process Execution Language (WS-BPEL)
 - This standard allows a service interface and its bindings to be defined
- Universal Description, Discovery, and Integration (UDDI)
 - Defines the components of a service specification that may be used to discover the existence of a service









REPRESENTATIONAL STATE TRANSFER (REST)

Two Fundamental Aspects of the REST Design Pattern

- **Resources:** Every distinguishable entity is a resource. A resource may be a Web site, an HTML page, an XML document, a Web service, a physical device, *etc.*
- **URLs Identify Resources:** Every resource is uniquely identified by a URL. This is Tim Berners-Lee Web Design, Axiom 0. [\(link\)](#)

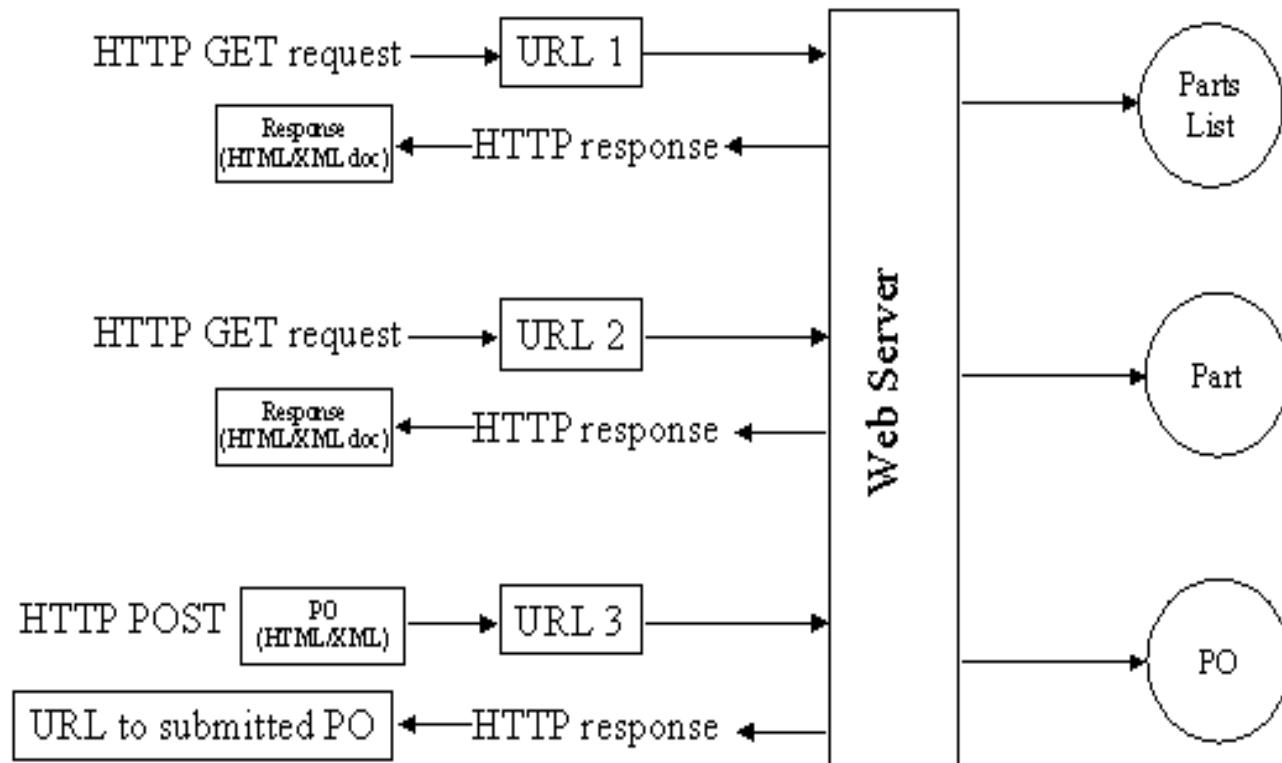
Terminology in REST based network

- **Client-Server:** a pull-based interaction style(Client request data from servers as and when needed).
- **Stateless:** each request from client to server must contain all the information necessary to understand the request, and cannot take advantage of any stored context on the server.
- **Cache:** to improve network efficiency, responses must be capable of being labeled as cacheable or non-cacheable.
- **Uniform Interface:** all resources are accessed with a generic interface (e.g., HTTP GET, POST, PUT, DELETE).
- **Named resources:** the system is comprised of resources which are named using a URL.
- **Interconnected resource representations:** the representations of the resources are interconnected using URLs, thereby enabling a client to progress from one state to another.

Mapped version of CRUD into HTTP methods in RESTful Web Services

Operation	HTTP method
Create	POST
Read	GET
Update	PUT or POST
Delete	DELETE

REST way of Implementing the web services



Service – Get parts list

The web service makes available a URL to a parts list resource

Client uses : <http://www.parts-depot.com/parts>

Document Client receives :

```
<?xml version="1.0"?>
<p:Parts xmlns:p="http://www.parts-depot.com"
           xmlns:xlink="http://www.w3.org/1999/xlink">
    <Part id="00345" xlink:href="http://www.parts-depot.com/parts/00345"/>
    <Part id="00346" xlink:href="http://www.parts-depot.com/parts/00346"/>
    <Part id="00347" xlink:href="http://www.parts-depot.com/parts/00347"/>
    <Part id="00348" xlink:href="http://www.parts-depot.com/parts/00348"/>
</p:Parts>
```

Service – Get detailed part data

The web service makes available a URL to each part resource.

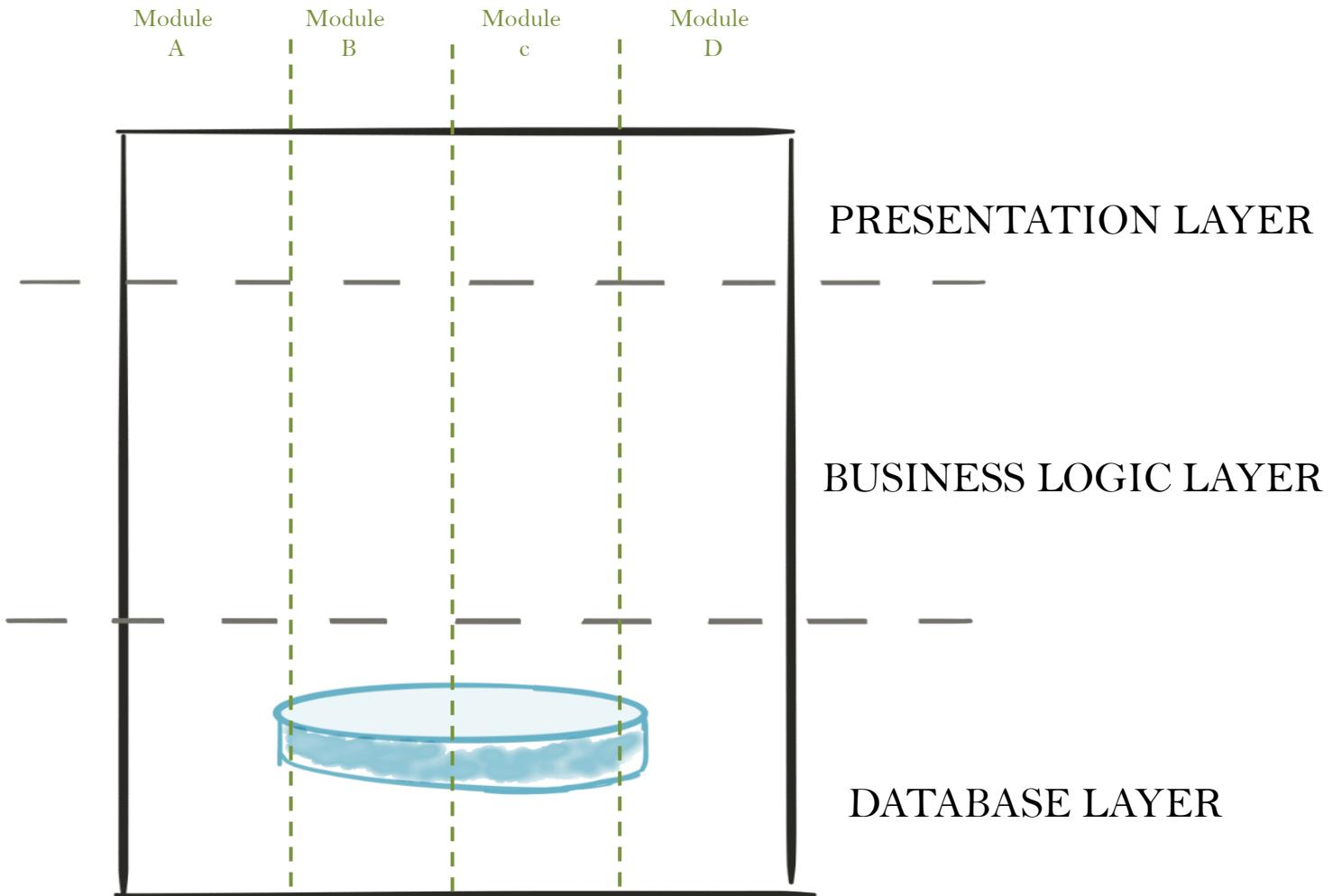
Client uses : <http://www.parts-depot.com/parts/00345>

Document Client receives :

```
<?xml version="1.0"?>
<p:Part xmlns:p="http://www.parts-depot.com" xmlns:xlink="http://www.w3.org/1999/xlink">
    <Part-ID>00345</Part-ID>
    <Name>Widget-A</Name>
    <Description>This part is used within the frap assembly</Description>
    <Specification xlink:href="http://www.parts-depot.com/parts/00345/specification"/> <UnitCost
        currency="USD">0.10</UnitCost>
    <Quantity>10</Quantity>
</p:Part>
```

MICROSERVICES

A monolith



VS

Monolith

Microservices

VS



Monoliths VS Microservices

- Advantages of microservices over monoliths include
 - Support for scaling
 - Scale vertically rather than horizontally
 - Support for change
 - Support hot deployment of updates
 - Support for reuse
 - Use same web service in multiple apps
 - Swap out internally developed web service for externally developed web service
 - Support for separate team development
 - Pick boundaries that match team responsibilities
 - Support for failure

Support for change: hot swapping

- In a large organization (e.g., Facebook, Amazon, AirBnb), will constantly have new features being finished and rolled out to production
- Traditional model: releases
 - Finish next version of software, test, release as a unit once every year or two
- Web enables frequent updates
 - Could update every night or even every hour
- But.... if updating every hour, really do not want website to be down

Hot Swapping

Hot swap X | Microphone Search

All Images Videos News More Tools

About 405,000,000 results (0.58 seconds)

Dictionary

Definitions from [Oxford Languages](#) · [Learn more](#)

Search for a word Search icon

 **hot-swap**

verb INFORMAL

fit or replace (a computer part) with the power still connected.
"the hard disk can be hot-swapped"

[Feedback](#)

Translations and more definitions dropdown arrow

Hot Swapping



Support for reuse

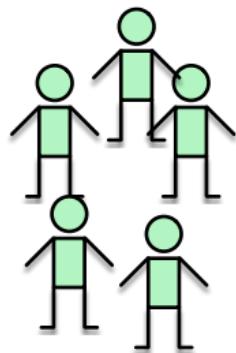
- In a large organization (e.g., Facebook, Amazon, AirBnb), may have many internal products that all depend on a similar core service (e.g., user account storage, serving static assets)
- Would like to
 - be able to build functionality once, reuse in many place
 - swap out an old implementation for a new implementation with a new technology or implementation
 - swap out an internal service for a similar external service

Organization in a monolith



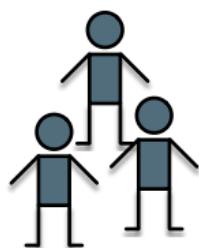
Frontend

Orders, shipping, catalog



Backend

Orders, shipping, catalog

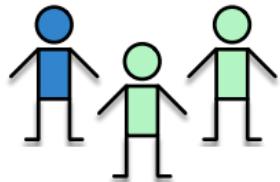


Database

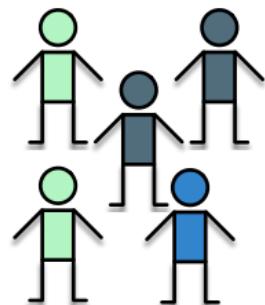
Orders, shipping, catalog

Classic teams:
1 team per “tier”

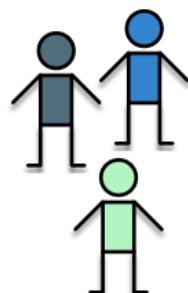
Organization in microservices



Orders



Shipping



Catalog

Example: Amazon

Teams can focus on one business task
And be responsible directly to users

“Full Stack”

“2 pizza teams”

Design for Failure

- Each of the many microservices might fail
 - Services might have bugs
 - Services might be slow to respond
 - Entire servers might go down
 - If you have 60,000 hard disks, 3 fail a day
 - The more microservices there are, the higher the likelihood at least one is currently failing
- Key: design every service assuming that at some point, everything it depends on might disappear - must fail “gracefully”

Support for failure

- Design for idempotency
 - Should be able to retry requests without introducing bad data
- Design for data locality
 - Transactions across microservices are hard to manage
- Design for eventual consistency

Design for idempotency

- Want to design APIs so that executing an action multiple times leads to same resulting state
- Prefer state changes on existing entity rather than creating new entities

Design for data locality

- If datastore server fails or is slow, do not want entire site to go down.
- Decentralizes implementation decisions.
- Allows each service to manage data in the way that makes the most sense for that service
- Also performance benefit: caching data locally in microservices enables faster response
- Rule: Services exchange data **ONLY** through their exposed APIs - **NO** shared databases

Consistency

- One of the rules was “no shared database”
- But surely some state will be shared
- Updates are sent via HTTP request
- No guarantee that those updates occur immediately
- Instead, guarantee that they occur eventually
 - What if a request results in change to resource in one service, but other service has not yet processed corresponding request?
 - May end up with different states in different resources.
 - Logic needs to be written to correctly handle such situations.

Additional Materials

- Coulouris, G. F., Dollimore, J., & Kindberg, T. Distributed systems: concepts and design. 5th edition, Pearson Education. Chapter 9, 10
- <https://blogs.oracle.com/developers/getting-started-with-microservices-part-three>

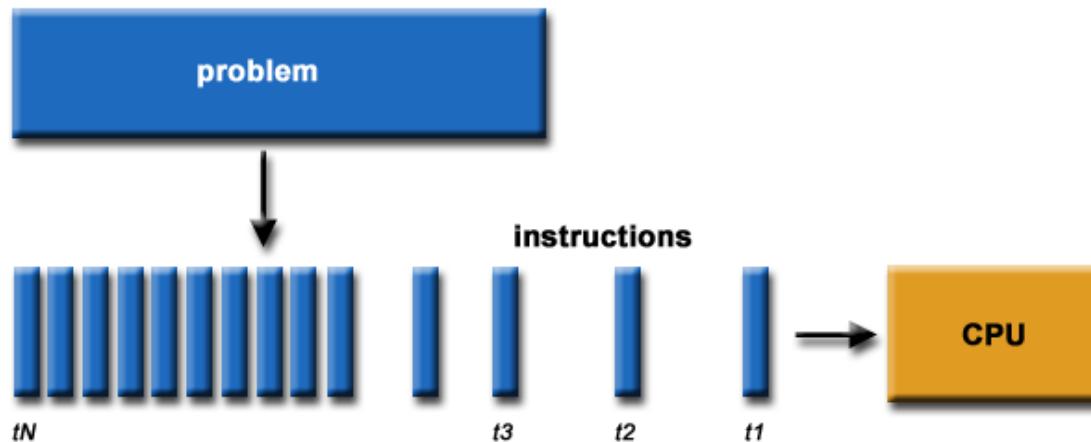
Lecture 3

Lecture plan

- Parallel Computing
- Concepts and terminology
- Parallel Computer Memory Architecture
- Parallel programming models
- Designing Parallel Programs

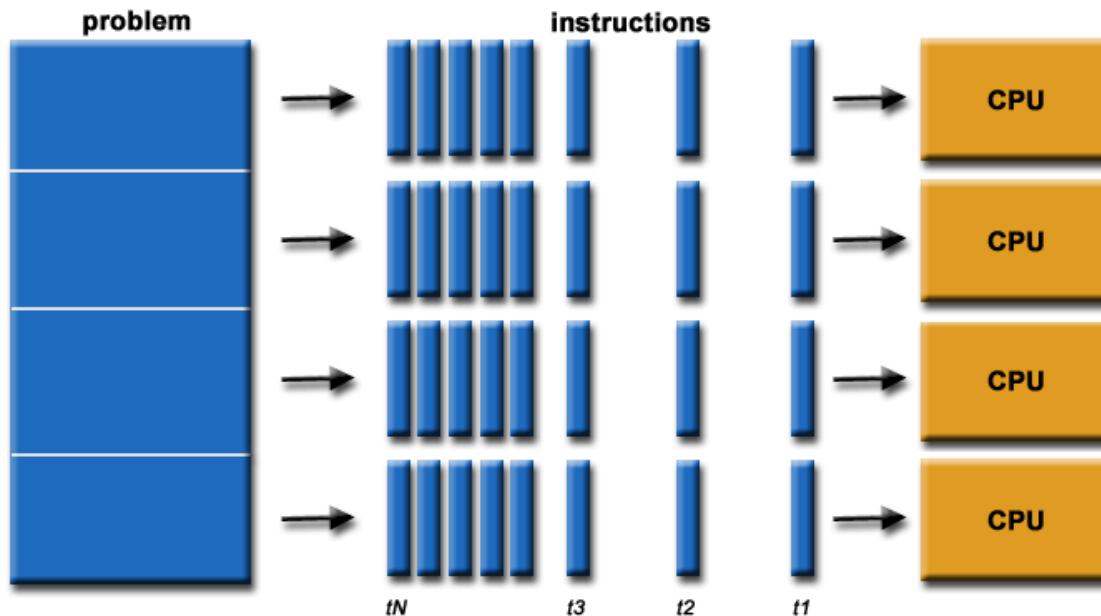
What is Parallel Computing?

- Traditionally, software has been written for *serial* computation:
 - To be run on a single computer having a single Central Processing Unit (CPU);
 - A problem is broken into a discrete series of instructions.
 - Instructions are executed one after another.
 - Only one instruction may execute at any moment in time.



What is Parallel Computing?

- In the simplest sense, *parallel computing* is the simultaneous use of multiple compute resources to solve a computational problem.
 - To be run using multiple CPUs
 - A problem is broken into discrete parts that can be solved in parallel
 - Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs



Parallel Computing: Resources

- The compute resources can include:
 - A single computer with multiple processors;
 - A single computer with (multiple) processor(s) and some specialized computer resources (Graphical Processing Unit ([GPU](#)), Field-Programmable Gate Array ([FPGA](#)))
 - An arbitrary number of computers connected by a network;
 - A combination of these.

Parallel Computing: The computational problem

- The computational problem usually demonstrates characteristics such as the ability to be:
 - Broken apart into discrete pieces of work that can be solved simultaneously;
 - Execute multiple program instructions at any moment in time;
 - Solved in less time with multiple compute resources than with a single compute resource.

Parallel Computing: what for?

- Parallel computing is an evolution of serial computing that attempts to emulate what has always been the state of affairs in the natural world: many complex, interrelated events happening at the same time, yet within a sequence.
- Some examples:
 - Weather and ocean patterns
 - Tectonic plate drift
 - Rush hour traffic
 - Automobile assembly line
 - Daily operations within a business
 - Building a shopping mall

Parallel Computing: what for?

- Traditionally considered to be "the high end of computing"
- Motivated by numerical simulations of complex systems such as:
 - weather and climate
 - chemical and nuclear reactions
 - biological, human genome
 - electronic circuits
 - manufacturing processes, etc.

Parallel Computing: what for?

- Today, commercial applications require the processing of large amounts of data in sophisticated ways. Example applications include:
 - parallel databases, data mining
 - oil exploration
 - web search engines, web based business services
 - computer-aided diagnosis in medicine
 - management of national and multi-national corporations
 - advanced graphics and virtual reality, particularly in the entertainment industry
 - networked video and multi-media technologies
 - collaborative work environments
- Ultimately, parallel computing is an attempt to maximize the infinite but seemingly scarce commodity called time.

Why Parallel Computing?

- Good question. Parallel computing is complex on every aspect!
- The primary reasons for using parallel computing:
 - Save time: wall-clock time
 - Solve larger problems
 - Provide concurrency (do multiple things at the same time)

Limitations of Serial Computing

- Limits to serial computing
- Transmission speeds
- Limits to miniaturization
- Economic limitations

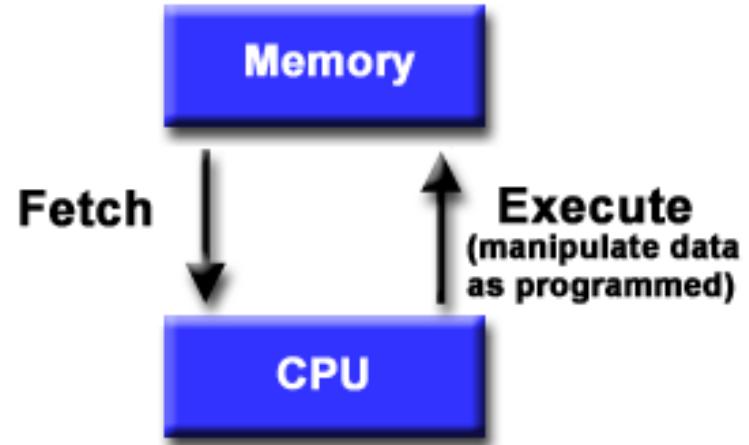
Concepts and Terminology

Von Neumann Architecture

- A common machine model known as the von Neumann computer
- Uses the stored-program concept. The CPU executes a stored program that specifies a sequence of read and write operations on the memory.

Basic Design

- Basic design
 - Memory is used to store both program and data instructions
 - Program instructions are coded data which tell the computer to do something
 - Data is simply information to be used by the program



- A central processing unit (CPU) gets instructions and/or data from memory, decodes the instructions and then *sequentially* performs them.

Flynn's Classical Taxonomy

- There are different ways to classify parallel computers
- One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of ***Instruction*** and ***Data***. Each of these dimensions can have only one of two possible states: ***Single*** or ***Multiple***.

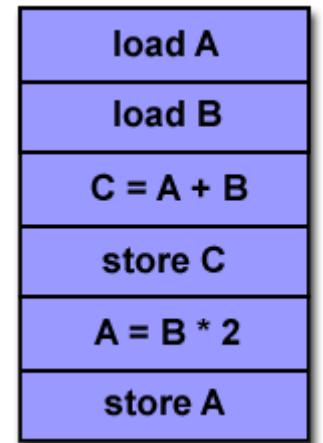
Flynn Matrix

- The matrix below defines the 4 possible classifications according to Flynn

S I S D	S I M D
Single Instruction, Single Data	Single Instruction, Multiple Data
M I S D	M I M D
Multiple Instruction, Single Data	Multiple Instruction, Multiple Data

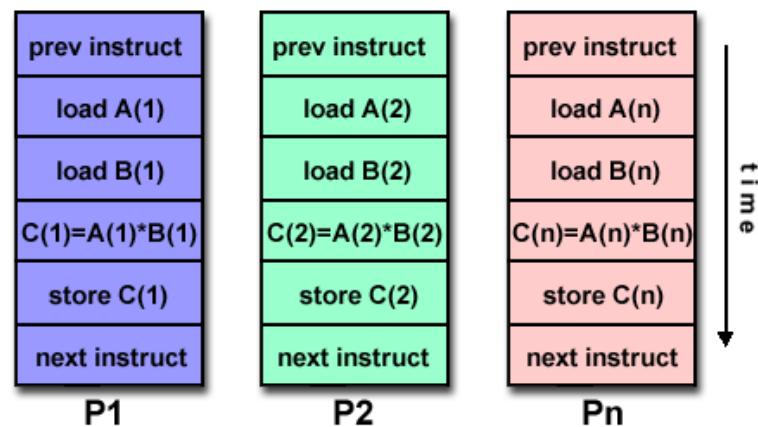
Single Instruction, Single Data (SISD)

- A serial (non-parallel) computer
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
- Single data: only one data stream is being used as input during any one clock cycle
- Deterministic execution
- The oldest and until recently, the most prevalent form of computer
- Examples: most PCs, single CPU workstations and mainframes



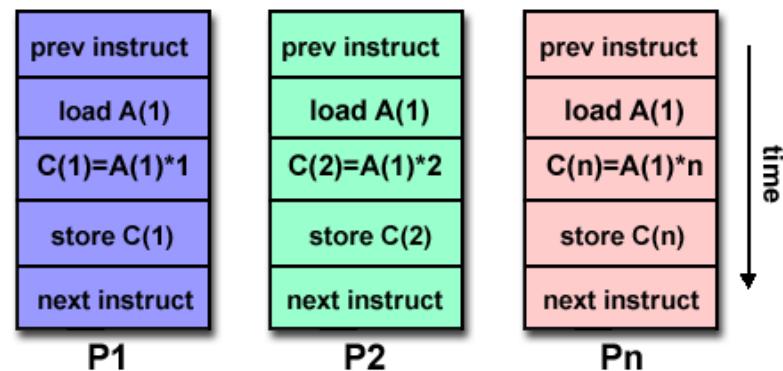
Single Instruction, Multiple Data (SIMD)

- A type of parallel computer
- Single instruction: All processing units execute the same instruction at any given clock cycle
- Multiple data: Each processing unit can operate on a different data element
- This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.
- Best suited for specialized problems characterized by a high degree of regularity, such as image processing.
- Synchronous and deterministic execution



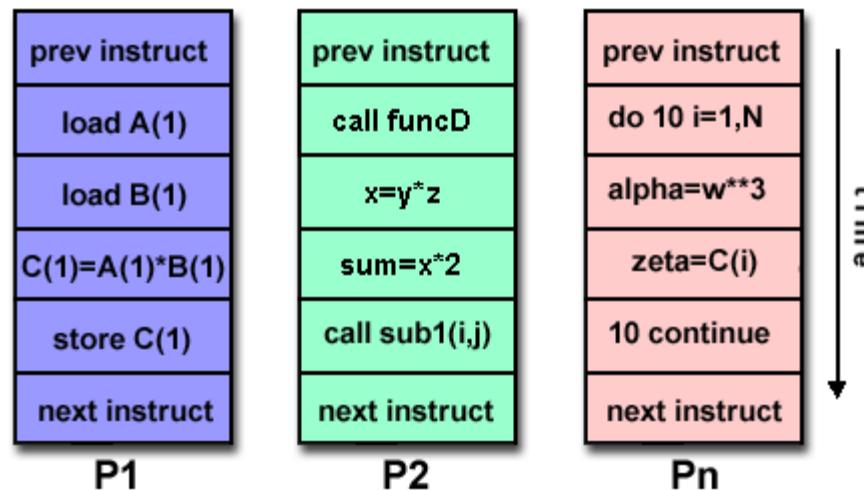
Multiple Instruction, Single Data (MISD)

- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams.
- Some conceivable uses might be:
 - multiple frequency filters operating on a single signal stream
 - multiple cryptography algorithms attempting to crack a single coded message.



Multiple Instruction, Multiple Data (MIMD)

- Most common type of parallel computer. Most modern computers fall into this category.
- Multiple Instruction: every processor may be executing a different instruction stream
- Multiple Data: every processor may be working with a different data stream
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Examples: most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs.



Some General Parallel Terminology

- **Task**
 - A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor.
- **Parallel Task**
 - A task that can be executed by multiple processors safely (yields correct results)
- **Serial Execution**
 - Execution of a program sequentially, one statement at a time. In the simplest sense, this is what happens on a one processor machine. However, virtually all parallel tasks will have sections of a parallel program that must be executed serially.

- **Parallel Execution**
 - Execution of a program by more than one task, with each task being able to execute the same or different statement at the same moment in time.
- **Shared Memory**
 - From a strictly hardware point of view, describes a computer architecture where all processors have direct (usually bus based) access to common physical memory. In a programming sense, it describes a model where parallel tasks all have the same "picture" of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists.
- **Distributed Memory**
 - In hardware, refers to network based memory access for physical memory that is not common. As a programming model, tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing.

- **Communications**
 - Parallel tasks typically need to exchange data. There are several ways this can be accomplished, such as through a shared memory bus or over a network, however the actual event of data exchange is commonly referred to as communications regardless of the method employed.
- **Synchronization**
 - The coordination of parallel tasks in real time, very often associated with communications. Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point.
 - Synchronization usually involves waiting by at least one task, and can therefore cause a parallel application's wall clock execution time to increase.

- **Granularity**
 - In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.
 - **Coarse:** relatively large amounts of computational work are done between communication events
 - **Fine:** relatively small amounts of computational work are done between communication events
- **Observed Speedup**
 - Observed speedup of a code which has been parallelized, defined as:
$$\frac{\text{wall-clock time of serial execution}}{\text{wall-clock time of parallel execution}}$$
 - One of the simplest and most widely used indicators for a parallel program's performance.

- **Parallel Overhead**

- The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as:
 - Task start-up time
 - Synchronizations
 - Data communications
 - Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.
 - Task termination time

- **Massively Parallel**

- Refers to the hardware that comprises a given parallel system - having many processors. The meaning of many keeps increasing, but currently BG/L pushes this number to 6 digits.

- **Scalability**

- Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors. Factors that contribute to scalability include:
 - Hardware - particularly memory-cpu bandwidths and network communications
 - Application algorithm
 - Parallel overhead related
 - Characteristics of your specific application and coding

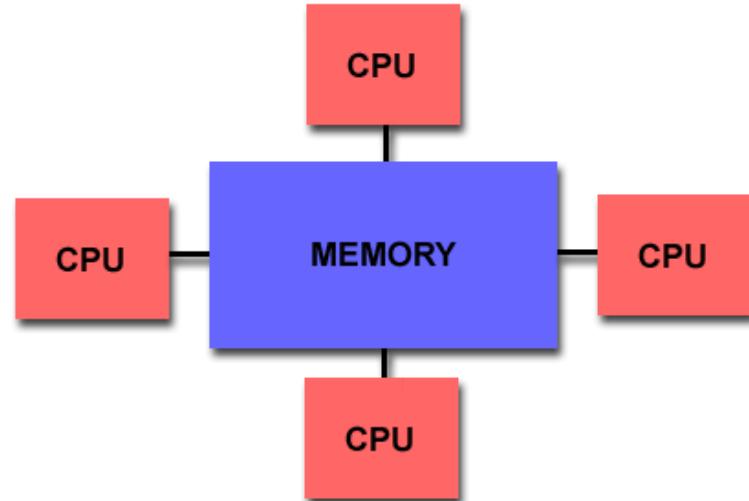
Parallel Computer Memory Architectures

Memory architectures

- Shared Memory
- Distributed Memory
- Hybrid Distributed-Shared Memory

Shared Memory

- Shared memory parallel computers generally have in common the ability for all processors to access all memory as global address space.
- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Shared memory machines can be divided into two main classes based upon memory access times: **UMA** and **NUMA**.



Shared Memory : UMA vs. NUMA

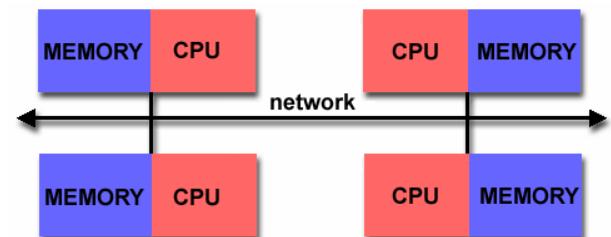
- Uniform Memory Access (UMA):
 - Most commonly represented today by Symmetric Multiprocessor (SMP)
 - Identical processors
 - Equal access and access times to memory
 - Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.
- Non-Uniform Memory Access (NUMA):
 - Often made by physically linking two or more SMPs
 - One SMP can directly access memory of another SMP
 - Not all processors have equal access time to all memories
 - Memory access across link is slower
 - If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA

Shared Memory: Pro and Con

- Advantages
 - Global address space provides a user-friendly programming perspective to memory
 - Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs
- Disadvantages:
 - Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increase traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
 - Programmer responsibility for synchronization constructs that insure "correct" access of global memory.
 - Expense: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

Distributed Memory

- Distributed memory systems require a communication network to connect inter-processor memory.
- Processors have their own local memory. Memory addresses in one processor do not map to another processor – no concept of global address space across all processors.
- Each processor operates independently. Changes it makes to its local memory have no effect on the memory of other processors.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.
- The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet.

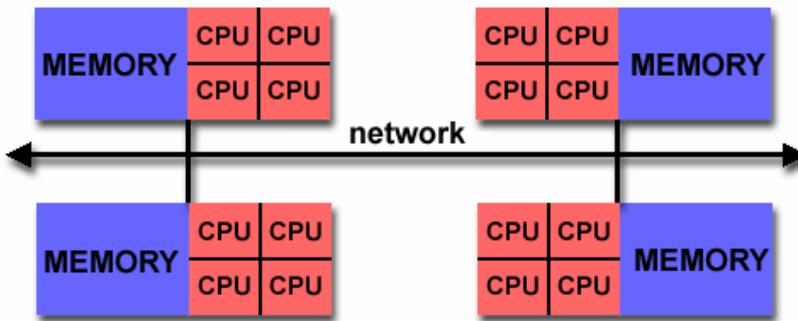


Distributed Memory: Pro and Con

- Advantages
 - Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
 - Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
 - Cost effectiveness: can use commodity, off-the-shelf processors and networking.
- Disadvantages
 - The programmer is responsible for many of the details associated with data communication between processors.
 - It may be difficult to map existing data structures, based on global memory, to this memory organization.
 - Non-uniform memory access (NUMA) times

Hybrid Distributed-Shared Memory

- The largest and fastest computers in the world today employ both shared and distributed memory architectures.

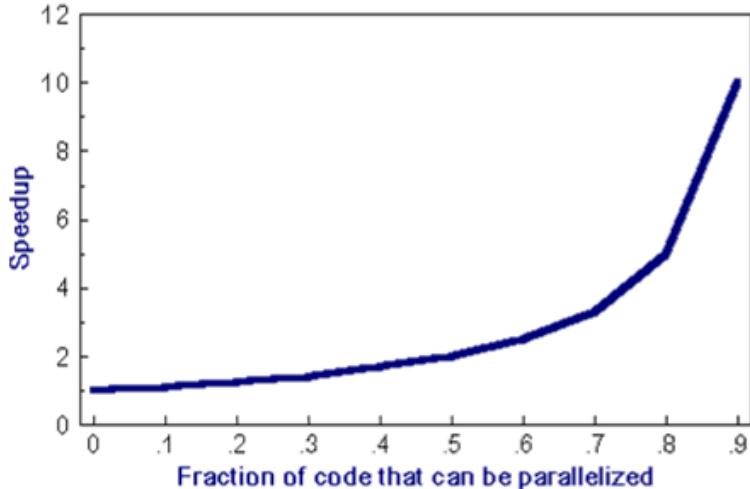


- The shared memory component is usually a cache coherent SMP machine. Processors on a given SMP can address that machine's memory as global.
- The distributed memory component is the networking of multiple SMPs. SMPs know only about their own memory - not the memory on another SMP. Therefore, network communications are required to move data from one SMP to another.

Amdahl's Law

Amdahl's Law states that potential program speedup is defined by the fraction of code (P) that can be parallelized:

$$\text{speedup} = \frac{1}{1 - P}$$



- If none of the code can be parallelized, $P = 0$ and the speedup = 1 (no speedup). If all of the code is parallelized, $P = 1$ and the speedup is infinite (in theory).
- If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.

Amdahl's Law

- Introducing the number of processors performing the parallel fraction of work, the relationship can be modelled by
- where P = parallel fraction, N = number of processors and S = serial fraction

$$\text{speedup} = \frac{1}{\frac{P + S}{N}}$$

Amdahl's Law

- It soon becomes obvious that there are limits to the scalability of parallelism. For example, at $P = .50$, $.90$ and $.99$ (50%, 90% and 99% of the code is parallelizable)

N	speedup		
	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02

Parallel Programming Models

Parallel Programming Models

- Shared Memory Model
- Threads Model
- Message Passing Model
- Data Parallel Model
- Other Models

Shared Memory Model

- In the shared-memory programming model, tasks share a common address space, which they read and write asynchronously.
- Various mechanisms such as locks / semaphores may be used to control access to the shared memory.

Threads Model

- In the threads model of parallel programming, a single process can have multiple, concurrent execution paths.
- Threads are commonly associated with shared memory architectures and operating systems.

Message Passing Model

- The message passing model demonstrates the following characteristics:
 - A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine as well across an arbitrary number of machines.
 - Tasks exchange data through communications by sending and receiving messages.
 - Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

Data Parallel Model

- The data parallel model demonstrates the following characteristics:
 - Most of the parallel work focuses on performing operations on a data set. The data set is typically organized into a common structure, such as an array or cube.
 - A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure.
 - Tasks perform the same operation on their partition of work, for example, "add 4 to every array element".
- On shared memory architectures, all tasks may have access to the data structure through global memory. On distributed memory architectures the data structure is split up and resides as "chunks" in the local memory of each task.

Designing Parallel Programs

Things to consider...

- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

- The first step in developing parallel software is to understand the problem that you wish to solve in parallel. If you are starting with a serial program, this necessitates understanding the existing code also.
- Before spending time in an attempt to develop a parallel solution for a problem, determine whether or not the problem is one that can actually be parallelized.

Example of Parallelizable Problem

Calculate the potential energy for each of several thousand independent conformations of a molecule. When done, find the minimum energy conformation.

- This problem is able to be solved in parallel. Each of the molecular conformations is independently determinable. The calculation of the minimum energy conformation is also a parallelizable problem.

Example of a Non-parallelizable Problem

Calculation of the Fibonacci series (1,1,2,3,5,8,13,21,...) by use of the formula:

$$F(k + 2) = F(k + 1) + F(k)$$

- This is a non-parallelizable problem because the calculation of the Fibonacci sequence as shown would entail dependent calculations rather than independent ones. The calculation of the $k + 2$ value uses those of both $k + 1$ and k . These three terms cannot be calculated independently and therefore, not in parallel.

Identify the program's *hotspots*

- Know where most of the real work is being done. The majority of scientific and technical programs usually accomplish most of their work in a few places.
- Profilers and performance analysis tools can help here
- Focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.

Identify *bottlenecks* in the program

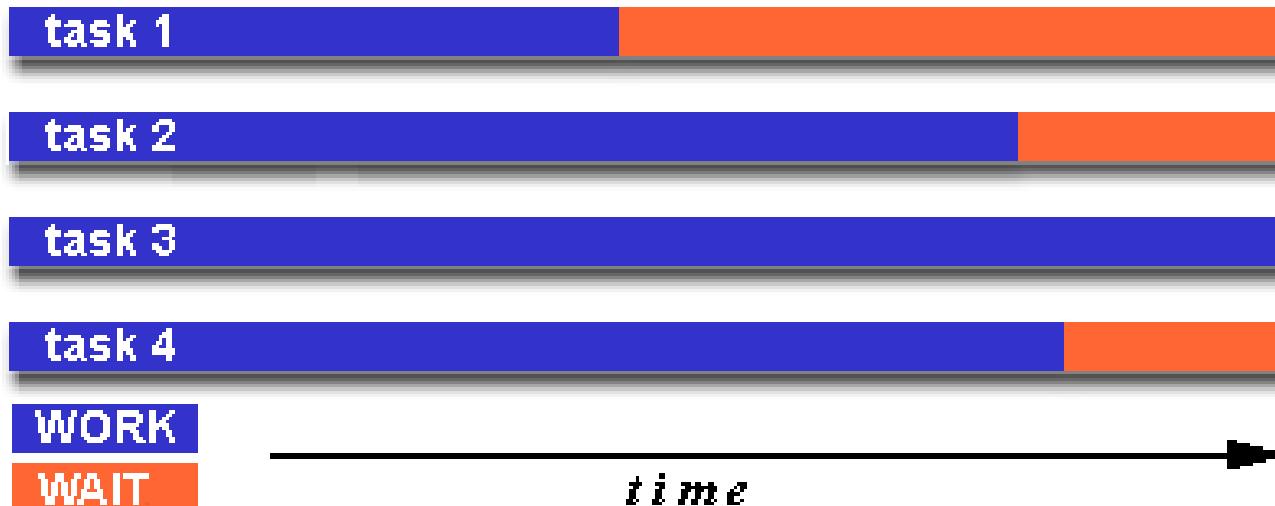
- Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? For example, I/O is usually something that slows a program down.
- May be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas

Other considerations

- Identify inhibitors to parallelism. One common class of inhibitor is *data dependence*, as demonstrated by the Fibonacci sequence above.
- Investigate other algorithms if possible. This may be the single most important consideration when designing a parallel application.

Load Balancing

- Load balancing refers to the practice of distributing work among tasks so that ***all*** tasks are kept busy ***all*** of the time. It can be considered a minimization of task idle time.
- Load balancing is important to parallel programs for performance reasons. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance.



How to Achieve Load Balance?

- **Equally partition the work each task receives**
 - For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks.
 - For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the tasks.
 - If a heterogeneous mix of machines with varying performance characteristics are being used, be sure to use some type of performance analysis tool to detect any load imbalances. Adjust work accordingly.

How to Achieve Load Balance?

- **Use dynamic work assignment**
 - Certain classes of problems result in load imbalances even if data is evenly distributed among tasks
 - When the amount of work each task will perform is intentionally variable, or is unable to be predicted, it may be helpful to use a ***scheduler - task pool*** approach. As each task finishes its work, it queues to get a new piece of work.
 - It may become necessary to design an algorithm which detects and handles load imbalances as they occur dynamically within the code.

Complexity

- In general, parallel applications are much more complex than corresponding serial applications. Not only do you have multiple instruction streams executing at the same time, but you also have data flowing between them.
- The costs of complexity are measured in programmer time in virtually every aspect of the software development cycle:
 - Design
 - Coding
 - Debugging
 - Tuning
 - Maintenance
- Adhering to "good" software development practices is essential when working with parallel applications - especially if somebody besides you will have to work with the software.

Resource Requirements

- The primary intent of parallel programming is to decrease execution wall clock time, however in order to accomplish this, more CPU time is required. For example, a parallel code that runs in 1 hour on 8 processors actually uses 8 hours of CPU time.
- The amount of memory required can be greater for parallel codes than serial codes, due to the need to replicate data and for overheads associated with parallel support libraries and subsystems.
- For short running parallel programs, there can be a decrease in performance compared to a similar serial implementation. The overhead costs associated with setting up the parallel computing can comprise a significant portion of the total execution time for short runs.

Scalability

- The ability of a parallel program's performance to scale is a result of a number of interrelated factors. Simply adding more machines is rarely the answer.
- The algorithm may have inherent limits to scalability. At some point, adding more resources causes performance to decrease. Most parallel solutions demonstrate this characteristic at some point.
- Hardware factors play a significant role in scalability.
 - Memory-CPU bus bandwidth on an SMP machine
 - Communications network bandwidth
 - Amount of memory available on any given machine or set of machines
 - Processor clock speed
- Parallel support libraries and subsystems software can limit scalability independent of your application.

Performance Analysis and Tuning

- As with debugging, monitoring and analyzing parallel program execution is significantly more of a challenge than for serial programs.

Additional Materials

- Borut Robič, Patricio Bulić, and Roman Trubec, T. Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms. Springer. Chapter 1, 2
- Google

Lecture 4

Transactions

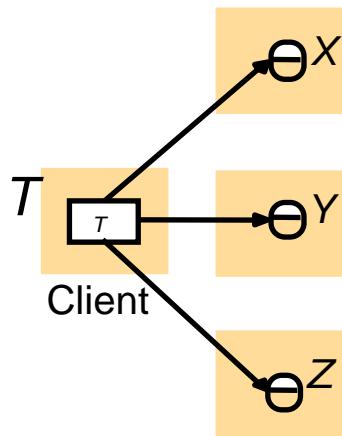
- A **distributed transaction** is a set of operations on data that is performed across two or more data repositories (especially databases). It is typically coordinated across separate nodes connected by a network but may also span multiple databases on a single server.

Topics in Distributed Transactions

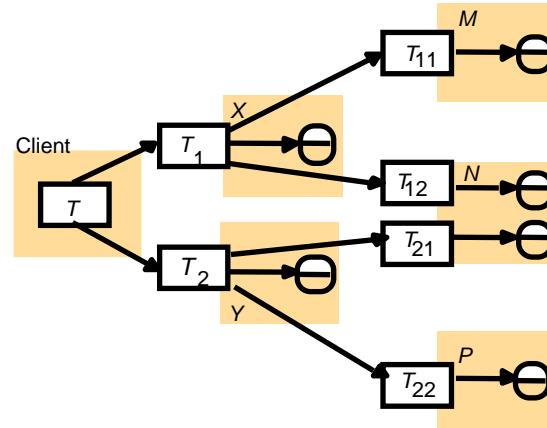
- **Atomicity:** property requires that either all the servers involved in the same transaction commit the transaction or all of them abort. Agreement among servers are necessary.
- **Transaction recovery:** is to ensure that all objects are recoverable. The values of the objects reflect all changes made by committed transactions and none of those made by aborted ones.

Distributed transactions

(a) Flat transaction



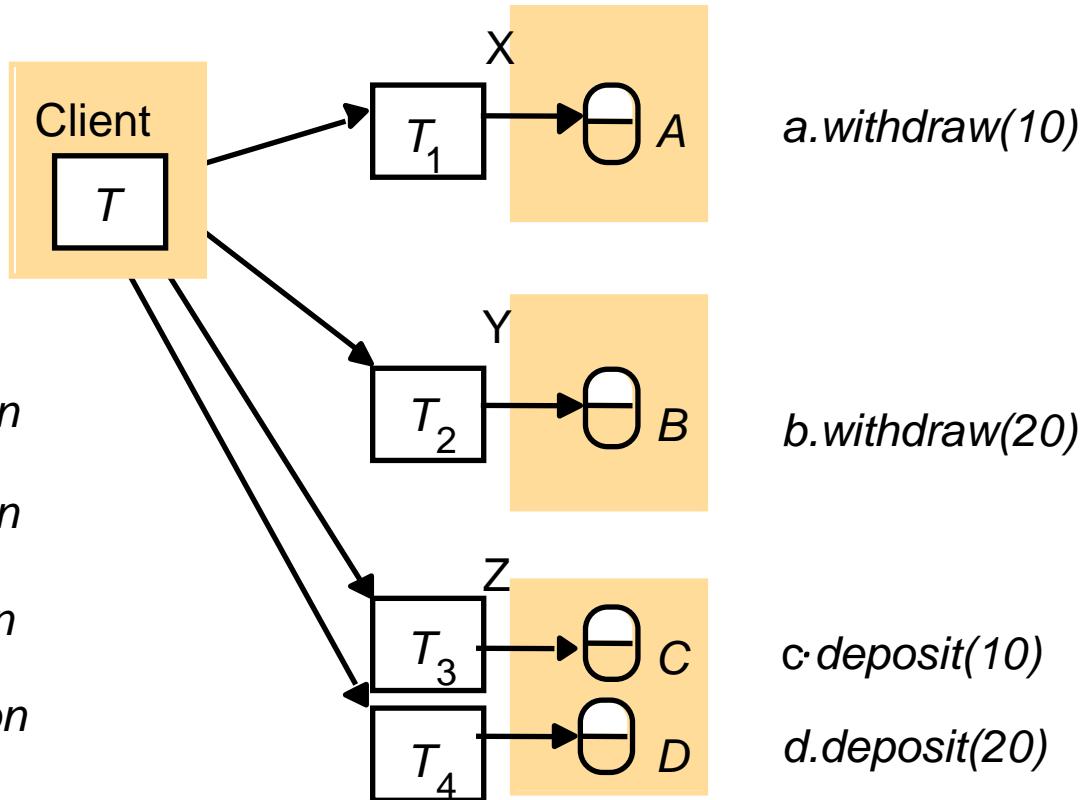
(b) Nested transactions



- **Flat transaction** send out requests to different servers and each request is completed before client goes to the next one.
- **Nested transaction** allows sub-transactions at the same level to execute concurrently.

```

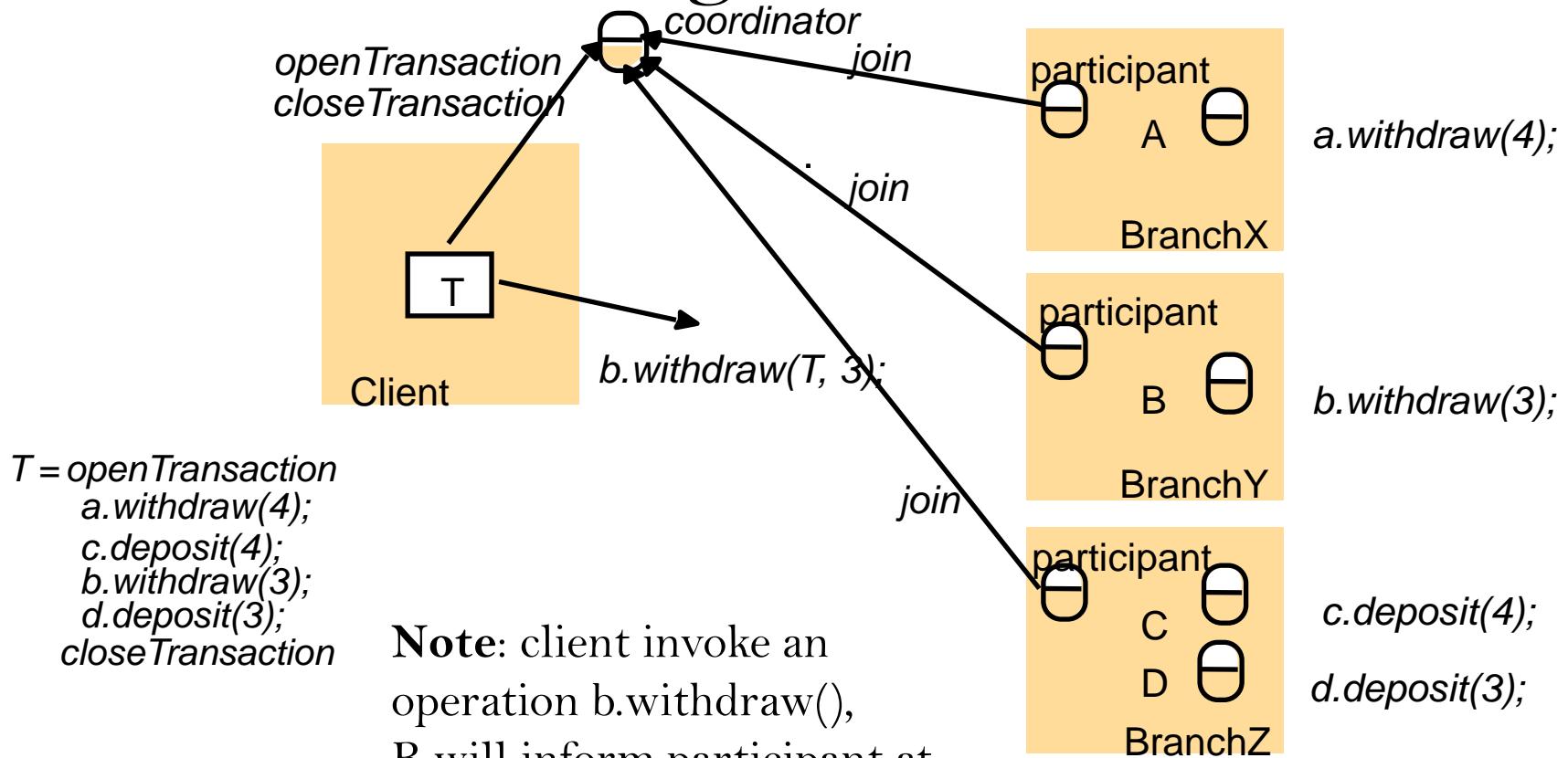
T = openTransaction
openSubTransaction
  a.withdraw(10);
openSubTransaction
  b.withdraw(20);
openSubTransaction
  c.deposit(10);
openSubTransaction
  d.deposit(20);
closeTransaction
  
```



Coordinator of a distributed transaction

- A client starts a transaction by sending an *openTransaction()* request to a **coordinator**. The coordinator returns the TID to the client. The TID must be unique (serverIP and number unique to that server)
- Coordinator is responsible for committing or aborting it. Each other server in a transaction is a participant. **Participants** are responsible for cooperating with the coordinator in carrying out the commit protocol and keep track of all recoverable objects managed by it.
- Each coordinator has a set of references to the participants. Each participant records a reference to the coordinator.

Distributed banking transaction



Note: client invoke an operation $b.\text{withdraw}()$,
B will inform participant at BranchY to join coordinator.
The coordinator is in one of the servers, e.g. BranchX

One-phase atomic commit protocol

- A transaction comes to an end when the client requests that a transaction be committed or aborted.
- Simple way is coordinator to communicate the commit or abort request to all the participants in the transaction and to keep on repeating the request until all of them have acknowledged that they had carried it out.
- Inadequate because when the client requests a commit, it does not allow a server to make a unilateral decision to abort a transaction. E.g., deadlock avoidance may force a transaction to abort at a server when locking is used. So, any server may fail, or abort and client is not aware.

Two-phase commit protocol

- Allow any participant to abort its part of a transaction. Due to atomicity, the whole transaction must also be aborted.
- **In the first phase**, each participant votes for the transaction to be committed or aborted. Once voted to commit, not allowed to abort it. So before votes to commit, it must ensure that it will eventually be able to carry out its part, even if it fails and is replaced.
- A participant is said to be in a **prepared** state if it will eventually be able to commit it. So each participant needs to save the altered objects in the permanent storage device together with its status-prepared.

Two-phase commit protocol

- **In the second phase,** every participant in the transaction carries out the joint decision. If any one participant votes to abort, the decision must be to abort. If all the participants vote to commit, then the decision is to commit the transaction.
- The problem is to ensure that all of the participants vote and that they all reach the same decision. It is an example of consensus. It is simple if no error occurs. However, it should work when servers fail, message lost or servers are temporarily unable to communicate with one another.

Two-phase commit protocol

- If the client requests abort, or if the transaction is aborted by one of the participants, the coordinator informs the participants immediately.
- It is when the client asks the coordinator to commit the transaction that two-phase commit protocol comes into use.
- In the first phase, the coordinator asks all the participants if they are prepared to commit; and in the second, it tells them to commit or abort the transaction.

Two-phase commit protocol

Phase 1 (voting phase):

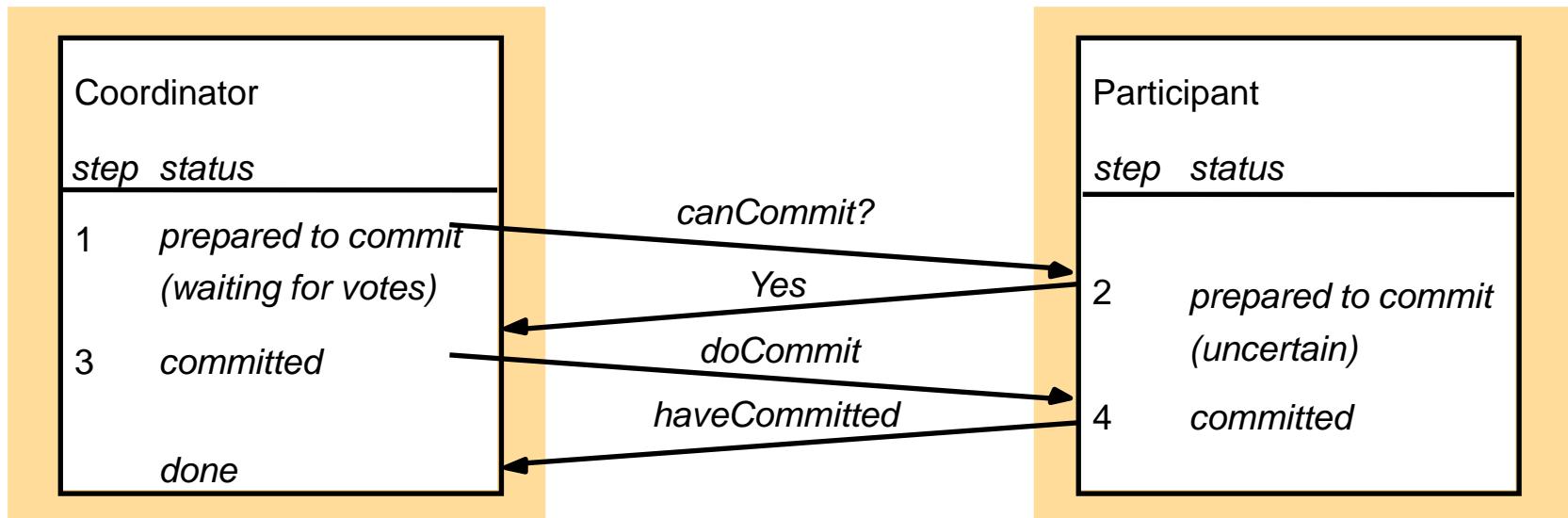
1. The coordinator sends a *canCommit?* request to each of the participants in the transaction.
2. When a participant receives a *canCommit?* request it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If the vote is *No* the participant aborts immediately.

Two-phase commit protocol

Phase 2 (completion according to outcome of vote):

3. The coordinator collects the votes (including its own).
 - (a) If there are no failures and all the votes are *Yes* the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.
 - (b) Otherwise the coordinator decides to abort the transaction and sends *doAbort* requests to all participants that voted *Yes*.
4. Participants that voted *Yes* are waiting for a *doCommit* or *doAbort* request from the coordinator. When a participant receives one of these messages it acts accordingly and in the case of commit, makes a *haveCommitted* call as confirmation to the coordinator.

Communication in Two-phase commit protocol



Two-phase commit protocol

- Consider when a participant has voted Yes and is waiting for the coordinator to report on the outcome of the vote by telling it to commit or abort.
- Such a participant is uncertain and cannot proceed any further. The objects used by its transaction cannot be released for use by other transactions.
- Participant makes a `getDecision` request to the coordinator to determine the outcome. If the coordinator has failed, the participant will not get the decision until the coordinator is replaced resulting in extensive delay for participant in uncertain state.
- Timeout are used since exchange of information can fail when one of the servers crashes, or when messages are lost So process will not block forever.

Performance of two-phase commit protocol

- Provided that all servers and communication channels do not fail, with N participants
- N number of *canCommit?* Messages and replies
- Followed by N *doCommit* messages
- The cost in messages is proportional to $3N$
- The cost in time is three rounds of message.
- The cost of *haveCommitted* messages are not counted, which can function correctly without them- their role is to enable server to delete stale coordinator information.

Failure of Coordinator

- When a participant has voted *Yes* and is waiting for the coordinator to report on the outcome of the vote, such participant is in uncertain stage. If the coordinator has failed, the participant will not be able to get the decision until the coordinator is replaced, which can result in extensive delays for participants in the uncertain state.
- One alternative strategy is allow the participants to obtain a decision from other participants instead of contacting coordinator. However, if all participants are in the uncertain state, they will not get a decision.

Concurrency Control in Distributed Transactions

- Concurrency control for distributed transactions: each server applies local concurrency control to its own objects, which ensure transactions serializability locally.
- However, the members of a collection of servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner. Thus, **global serializability** is required.

Locks

- Lock manager at each server decide whether to grant a lock or make the requesting transaction wait.
- However, it cannot release any locks until it knows that the transaction has been committed or aborted at all the servers involved in the transaction.
- A lock managers in different servers set their locks independently of one another. It is possible that different servers may impose different orderings on transactions.

Timestamp ordering concurrency control

- In a single server transaction, the coordinator issues a unique timestamp to each transaction when it starts. Serial equivalence is enforced by committing the versions of objects in the order of the timestamps of transactions that accessed them.
- In distributed transactions, we require that each coordinator issue globally unique time stamps. The coordinators must agree as to the ordering of their timestamps. \langle local timestamp, server-id \rangle , the agreed ordering of pairs of timestamps is based on a comparison in which the server-id is less significant.
- The timestamp is passed to each server whose objects perform an operation in the transaction.

Timestamp ordering concurrency control

- To achieve the same ordering at all the servers, The servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner. E.g. If T commits after U at server X, T must commits after U at server Y.
- Conflicts are resolved as each operation is performed. If the resolution of a conflict requires a transaction to be aborted, the coordinator will be informed and it will abort the transaction at all the participants.

Locking

T	U
Write(A) at X locks A	Write(B) at Y locks B
Read(B) at Y waits for U	Read(A) at X waits for T

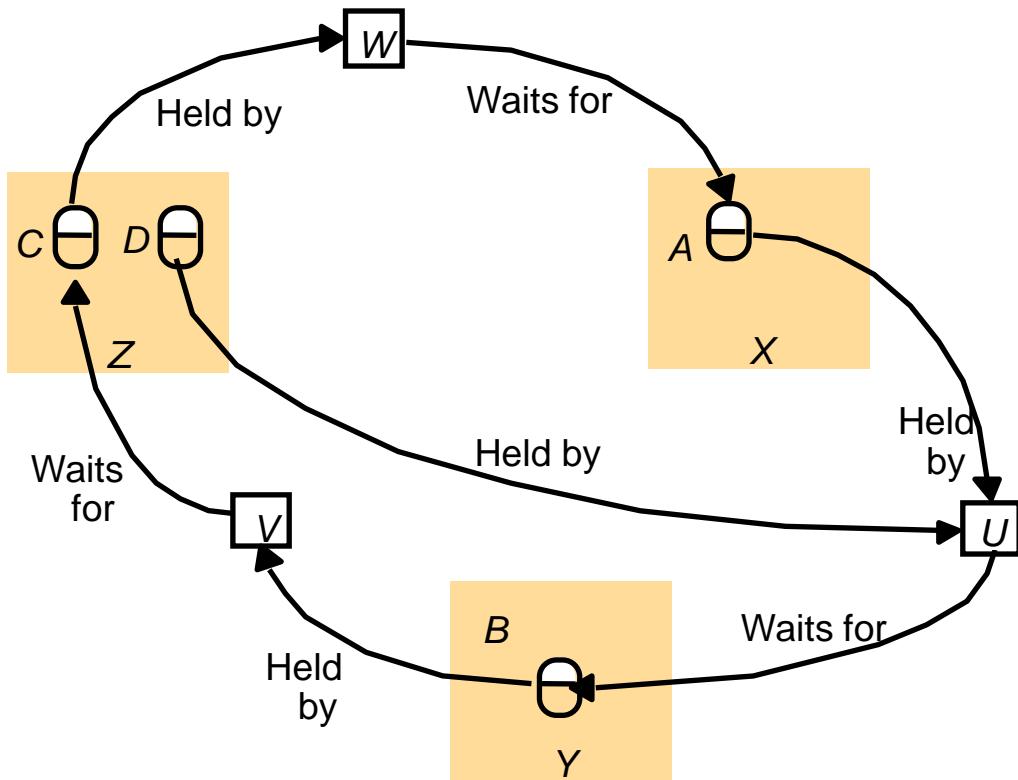
T before U in one server X and U before T in server Y. These different ordering can lead to cyclic dependencies between transactions and a distributed deadlock situation arises.

Distributed Deadlock

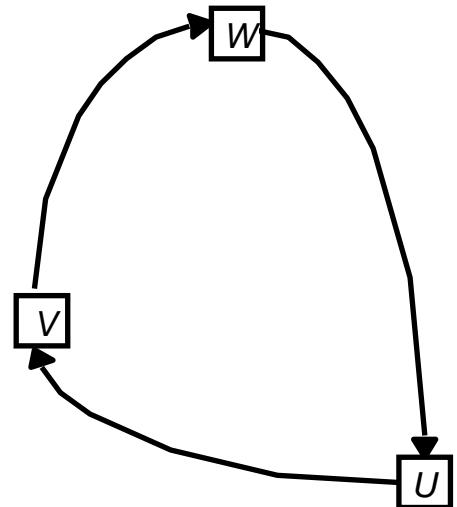
- Deadlocks can arise within a single server when locking is used for concurrency control. Servers must either prevent or detect and resolve deadlocks.
- Using timeout to resolve deadlock is a clumsy approach. Why? Another way is to detect deadlock by detecting cycles in a wait for graph.

Distributed Deadlock

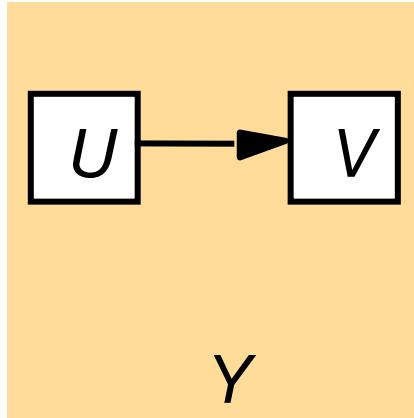
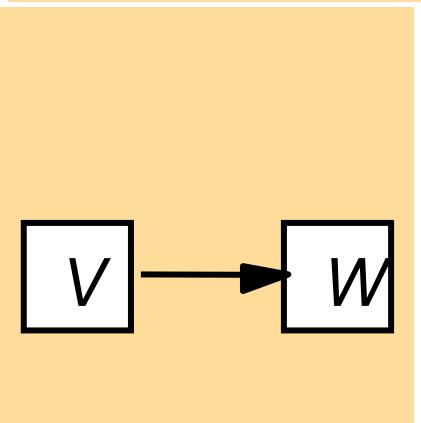
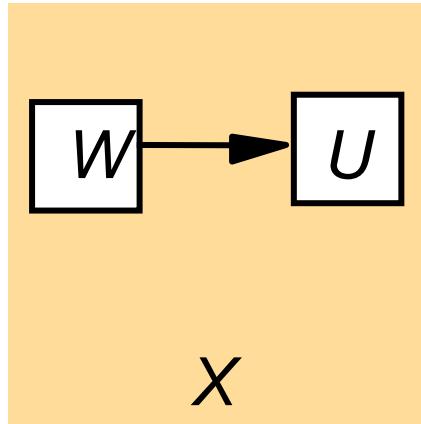
(a)



(b)



local wait-for graph



Local and global wait-for graphs

Global wait for graph is held in part by each of the several servers involved. Communication between these servers is required to find cycles in the graph.

Simple solution: one server takes on the role of global deadlock detector. From time to time, each server sends the latest copy of its local wait-for graph.

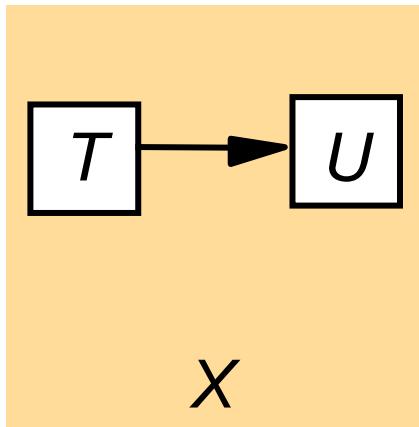
Disadvantages: poor availability, lack of fault tolerance and no ability to scale. The cost of frequent transmission of local wait-for graph is high.

Z

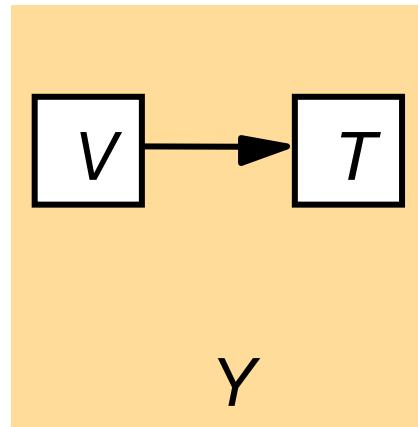
Phantom deadlock

- A deadlock that is detected but is not really a deadlock is called a phantom deadlock.
- As the procedure of sending local wait-for graph to one place will take some time, there is a chance that one of the transactions that holds a lock will meanwhile have released it, in which case the deadlock will no longer exist.

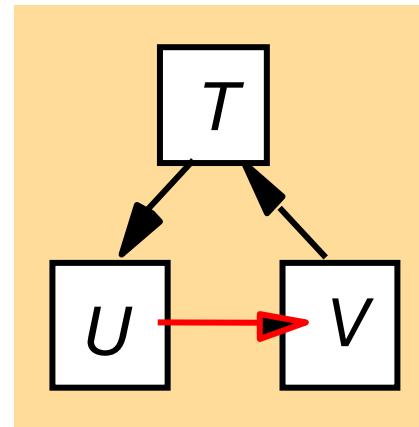
local wait-for graph



local wait-for graph



global deadlock detector



suppose U releases object at X and request object held by V . $U \rightarrow V$

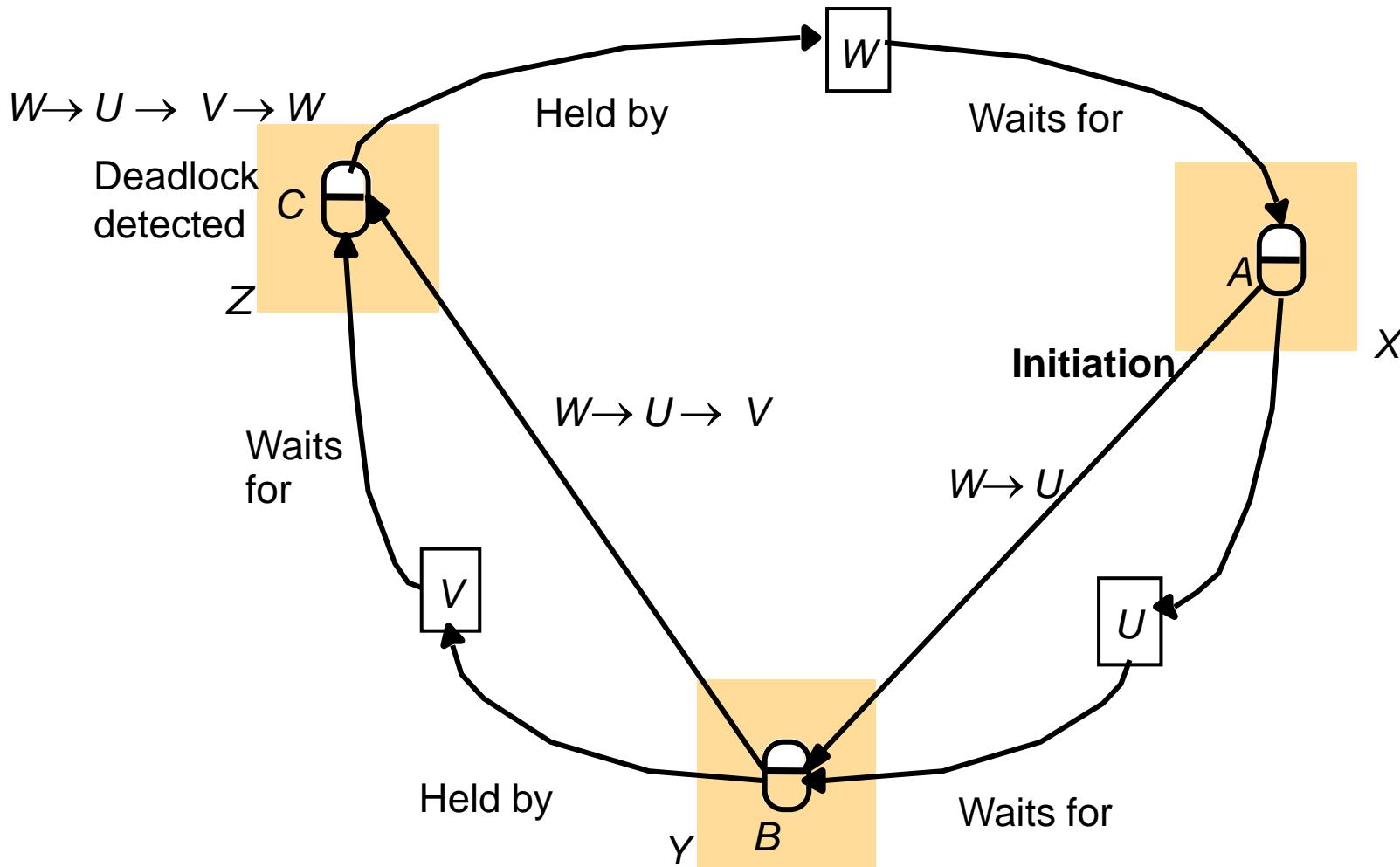
Then the global detector will see deadlock. However, the edge from T to U no longer exist.

However, if two-phase locking is used, transactions can not release locks and then obtain more locks, and phantom deadlock cycles cannot occur in the way suggested above.

Edge Chasing / Path Pushing

- Distributed approach for deadlock detection. No global wait-for graph is constructed, but each of the servers has knowledge about some of its edges. The servers attempt to find cycles by forwarding messages called probes, which follow the edges of the graph throughout the distributed system.
- A probe message consists of transaction wait-for relationships representing a path in the global wait-for graph.

Probes transmitted to detect deadlock



Transaction recovery

- Atomic property of transactions can be described in two aspects:
 - Durability: objects are saved in permanent storage and will be available indefinitely thereafter. Acknowledgement of a client's commit request implies that all the effects of the transaction have been recorded in permanent storage as well as in the server's volatile object.
 - Failure atomicity: the effects of transactions are atomic even when the server crashes.
 - Both can be realized by recovery manager.

Recovery manager

- Tasks of a recovery manager:
 - Save objects in permanent storage (in a recovery file) for committed transactions;
 - To restore the server's objects after a crash;
 - To reorganize the recovery file to improve the performance of recovery;
 - To reclaim storage space in the recovery file.

Additional Materials

- Coulouris, G. F., Dollimore, J., & Kindberg, T. Distributed systems: concepts and design. 5th edition, Pearson Education. Chapter 16, 17
- T. Özsü, P. Valduriez Principles of Distributed Database Systems. 3rd edition. Chapter 10, 11

Additional materials

- <https://hazelcast.com/glossary/distributed-transaction/>
- <https://www.geeksforgeeks.org/two-phase-commit-protocol-distributed-transaction-management/>
- <https://www.geeksforgeeks.org/flat-nested-distributed-transactions/>
- https://en.wikipedia.org/wiki/Microsoft_Distributed_Transaction_Coordinator
- https://www.tutorialspoint.com/distributed_dbms/distributed_dbms_commit_protocols.htm
- <https://medium.com/@macworks/distributed-commit-66c866975fb6>

Lecture 5

CAP Theorem and NoSQL Databases

CAP Theorem (Brewer's Theorem)

CAP theorem states that there are three basic requirements which exist in a special relation when designing applications for a distributed architecture.

Consistency

All reads receive the most recent write or an error

Availability

All reads contain data, but it might not be the most recent

Partition Tolerance

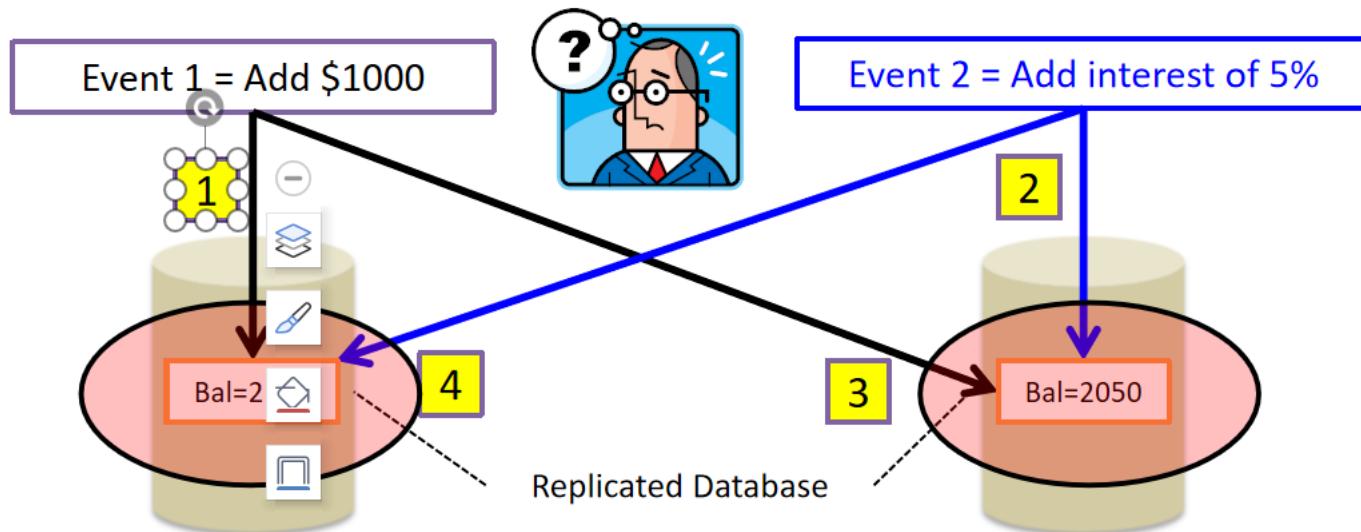
The system continues to operate despite network failures

The CAP Theorem



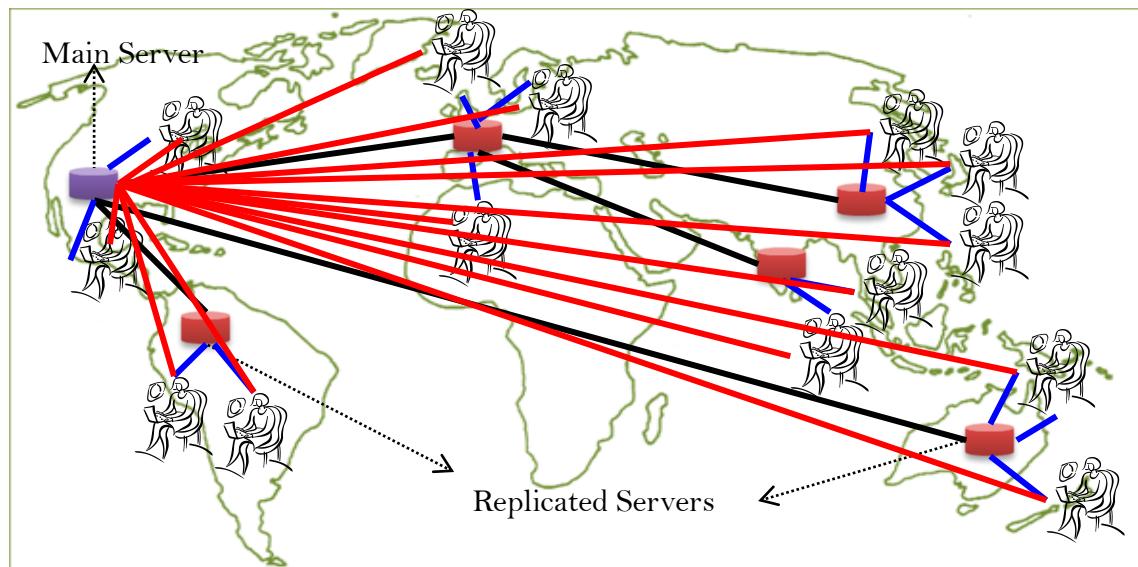
CAP Theorem (Brewer's Theorem)

Consistency - This means that the data in the database remains consistent after the execution of an operation. For example, after an update operation all clients see the same data.



CAP Theorem (Brewer's Theorem)

Availability - This means that the system is always on (service guarantee availability), no downtime.



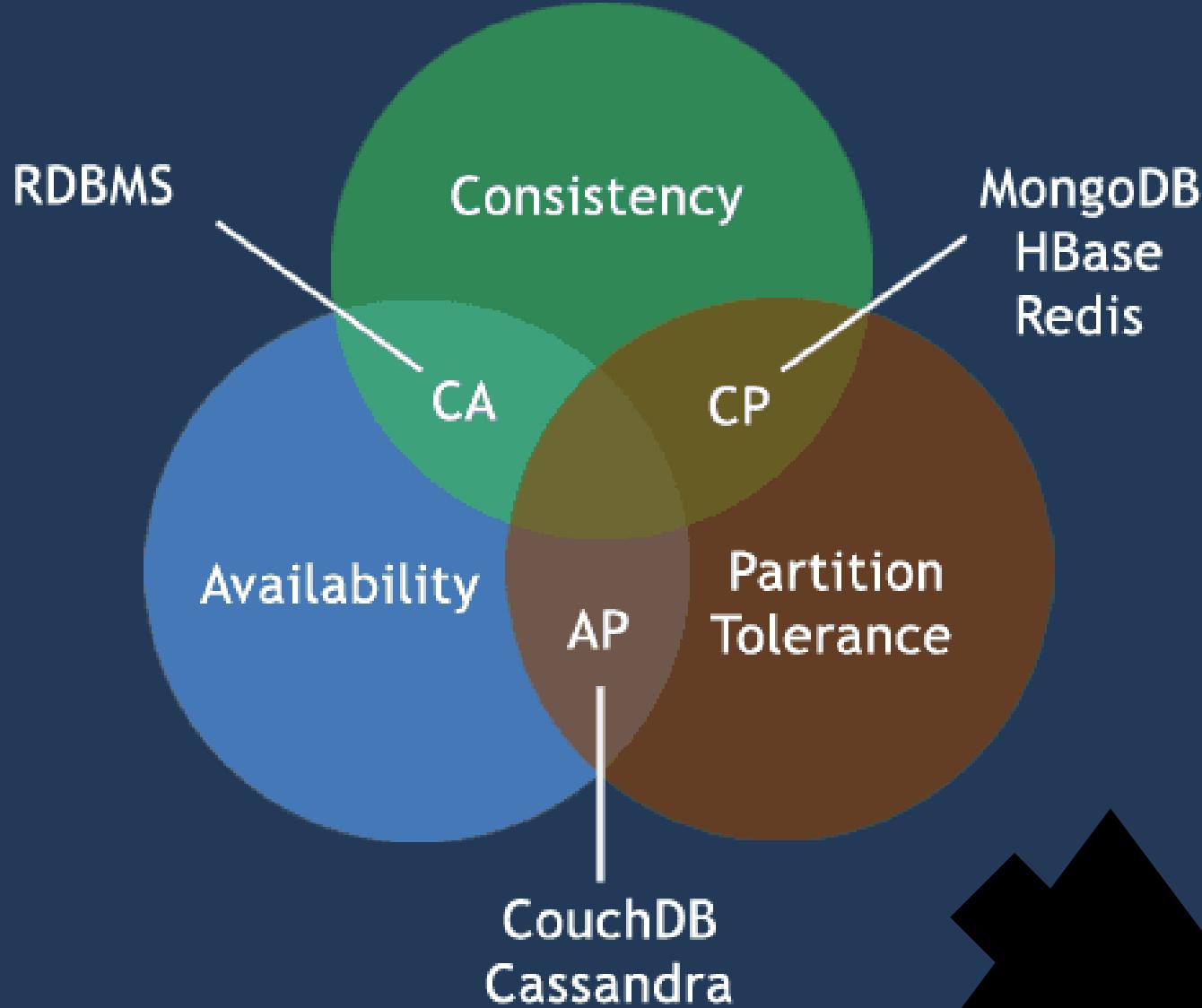
CAP Theorem (Brewer's Theorem)

Partition Tolerance - This means that the system continues to function even if the communication among the servers is unreliable, i.e., the servers may be partitioned into multiple groups that cannot communicate with one another.

CAP Theorem (Brewer's Theorem)

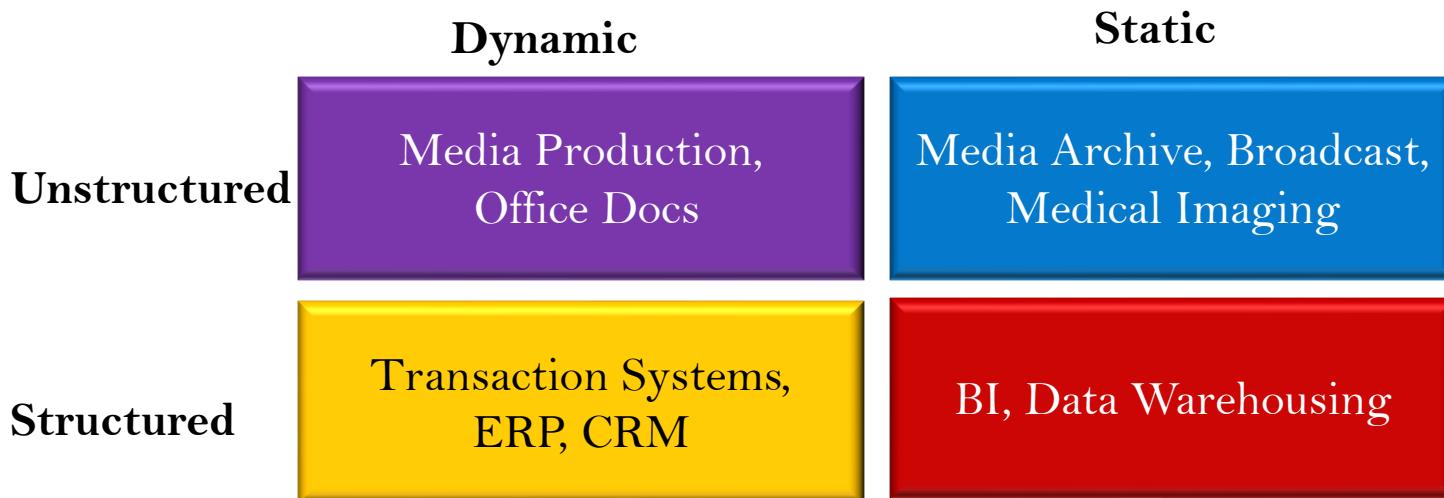
- In theory it is impossible to fulfill all 3 requirements. CAP provides the basic requirements for a distributed system to follow 2 of the 3 requirements.
 - CA - Single site cluster, therefore all nodes are always in contact. When a partition occurs, the system blocks.
 - CP -Some data may not be accessible, but the rest is still consistent/accurate.
 - AP - System is still available under partitioning, but some of the data returned may be inaccurate.

CAP Theorem



Why Classifying Data?

- Segmenting data into one of the following 4 quadrants can help in designing and developing a pertaining storage solution



- Relational databases are usually used for structured data
- File systems or *NoSQL databases* can be used for (static), unstructured data

NoSQL

- 1998 first used for a relational database that omitted the use of SQL (by Carlo Strozzi)
- 2009 used for conferences of advocates of non-relational databases
 - Eric Evans
 - Blogger, developer at Rackspace

NoSQL movement – the whole point of seeking alternatives is that you need to solve a problem that relational databases are a bad fit for

NoSQL

- Not – no to SQL
 - Another option, not the only one
- Not – not only SQL
 - Oracle DB or PostgreSQL would fit the definition
- Next Generation Databases mostly addressing some of the points: being non-relational, distributed, open-source and horizontally scalable.
- The original intention has been modern web-scale databases. Often more characteristics apply as: schema-free, easy replication support, simple API, eventually consistent (BASE, not ACID), a huge data amount, and more (ACID – **atomicity, consistency, isolation, durability**)

Motivation for NoSQL Databases

- Huge amounts of data are now handled in real-time
- Both data and use cases are getting more and more dynamic
- Social networks (relying on graph data) have gained impressive momentum
 - Special type of NoSQL databases: graph databases
- Full-texts have always been treated shabbily by RDBMS

NoSQL Databases

Five Advantages

1. Elastic scaling

- i. “Classical” database administrators scale up – buy bigger servers as database load increases
- ii. Scaling out – distributing the database across multiple hosts as load increases

2. Big Data

- i. Volumes of data that are being stored have increased massively
- ii. Opens new dimensions that cannot be handled with RDBMS

NoSQL Databases

Five Advantages

3. Goodbye DBAs (see you later?)

- i. Automatic repair, distribution, tuning, ...
- ii. vs. expensive, highly trained DBAs of RDBMS

4. Economics

- i. Based on commodity servers
- ii. less costs per transaction/second

5. Flexible Data Models

- i. Non-existing/relaxed data schema
- ii. structural changes cause no overhead

NoSQL Databases

Five Challenges

- 1. Maturity**
 - i. Still in pre-production phase
 - ii. Key features yet to be implemented
- 2. Support**
 - i. Mostly open source, result from start-ups
 - i. Enables fast development
 - ii. Limited resources or credibility
- 3. Administration**
 - i. Require lot of skill to install and effort to maintain

NoSQL Databases

Five Challenges

4. Analytics and Business Intelligence

- i. Focused on web apps scenarios
 - i. Modern Web 2.0 applications
 - ii. Insert-read-update-delete
- ii. Limited ad-hoc querying
- iii. Even a simple query requires significant programming expertise

5. Expertise

- 4. Few number of NoSQL experts available in the market

Data Assumptions

RDBMS	NoSQL
integrity is mission-critical	OK as long as most data is correct
data format consistent, well-defined	data format unknown or inconsistent
data is of long-term value	data are expected to be replaced
data updates are frequent	write-once, read multiple (no updates, or at least not often)
predictable, linear growth	unpredictable growth (exponential)
non-programmers writing queries	only programmers writing queries
regular backup	replication
access through master server	sharding across multiple nodes

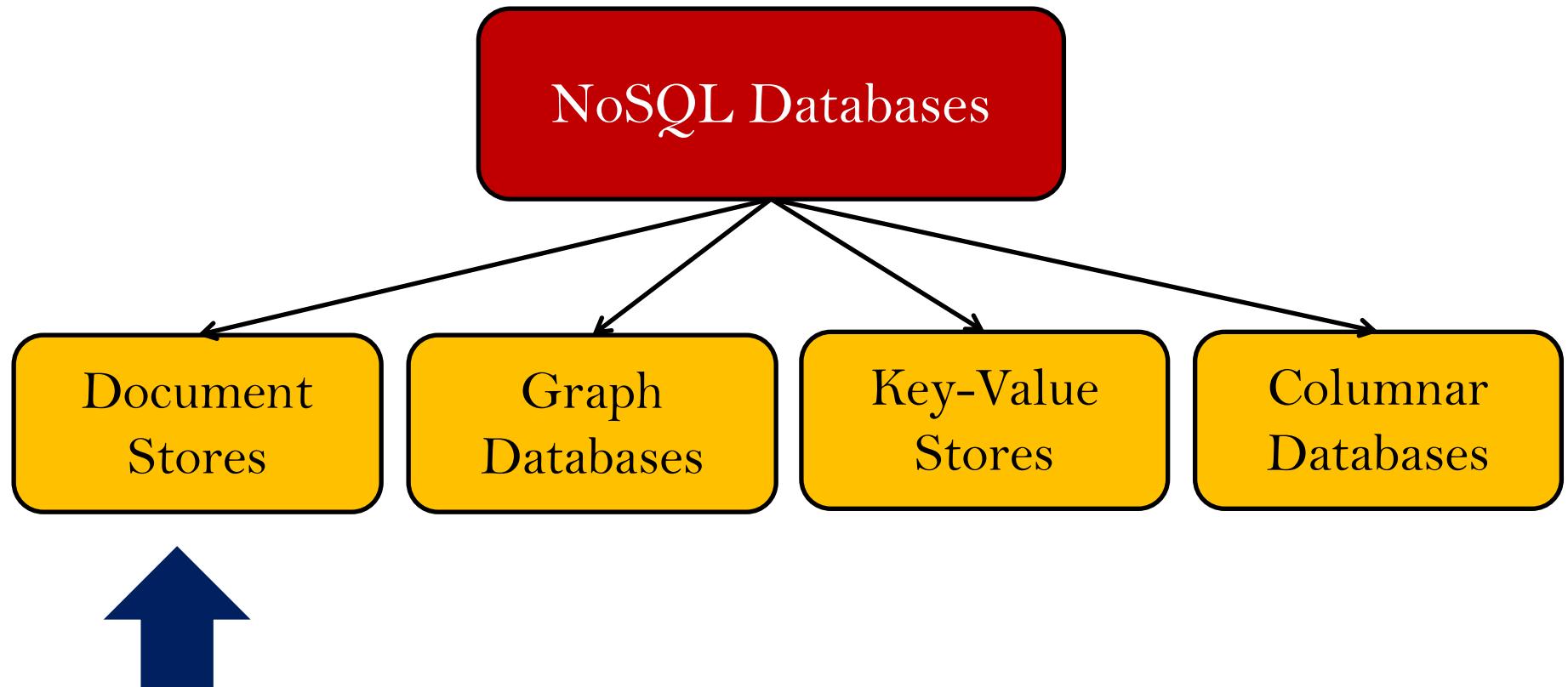
NoSQL Databases

Schemalessness

1. When we want to store data in a RDBMS, we need to define a schema
2. Advocates of schemalessness rejoice in freedom and flexibility
 - i. Allows to easily change your data storage as we learn more about the project
 - ii. Easier to deal with non-uniform data
3. Fact: there is usually an implicit schema present
 - i. The program working with the data must know its structure

Types of NoSQL Databases

- Here is a limited taxonomy of NoSQL databases:



Document Stores

- Documents are stored in some standard format or encoding (e.g., XML, JSON, PDF or Office Documents)
 - These are typically referred to as Binary Large Objects (BLOBs)
- Documents can be indexed
 - This allows document stores to outperform traditional file systems
- E.g., MongoDB and CouchDB (both can be queried using MapReduce)

Document Databases

Basic Characteristics

- Documents are the main concept
 - Stored and retrieved
 - XML, JSON, ...
- Documents are
 - Self-describing
 - Hierarchical tree data structures
 - Can consist of maps, collections (lists, sets, ...), scalar values, nested documents, ...
- Documents in a collection are expected to be similar
 - Their schema can differ
- Document databases store documents in the value part of the key-value store
 - Key-value stores where the value is examinable

Document Databases

Representatives



Lotus Notes
Storage Facility

Document Databases

Suitable Use Cases

Event Logging

- Many different applications want to log events
 - Type of data being captured keeps changing
- Events can be sharded (i.e. divided) by the name of the application or type event

Content Management Systems, Blogging Platforms

- Managing user comments, user registrations, profiles, web-facing documents, ...

Web Analytics or Real-Time Analytics

- Parts of the document can be updated
- New metrics can be easily added without schema changes
 - E.g. adding a member of a list, set,...

E-Commerce Applications

- Flexible schema for products and orders
- Evolving data models without expensive data migration

Document Databases

When Not to Use

Complex Transactions Spanning Different Operations

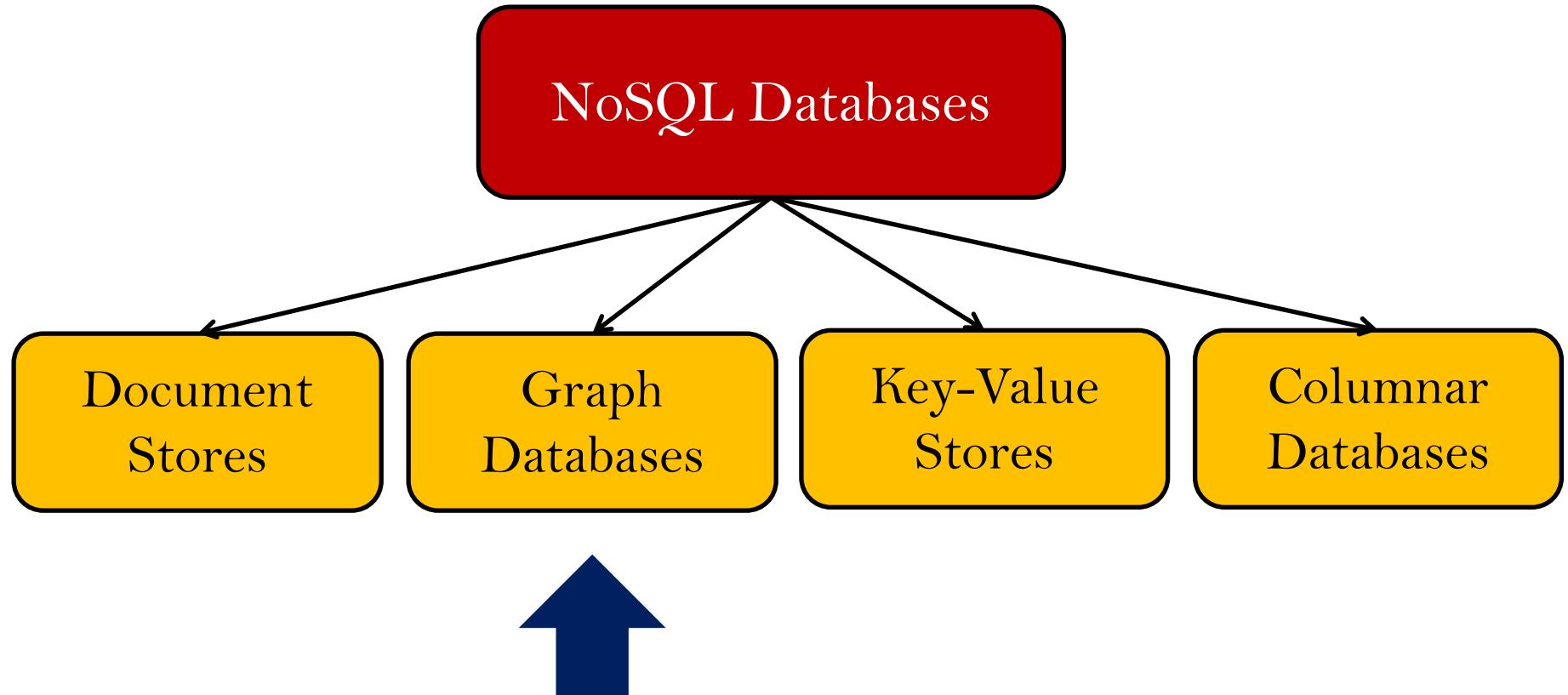
- Atomic cross-document operations
 - Some document databases do support (e.g., RavenDB)

Queries against Varying Aggregate Structure

- Design of aggregate is constantly changing → we need to save the aggregates at the lowest level of granularity
 - i.e. to normalize the data

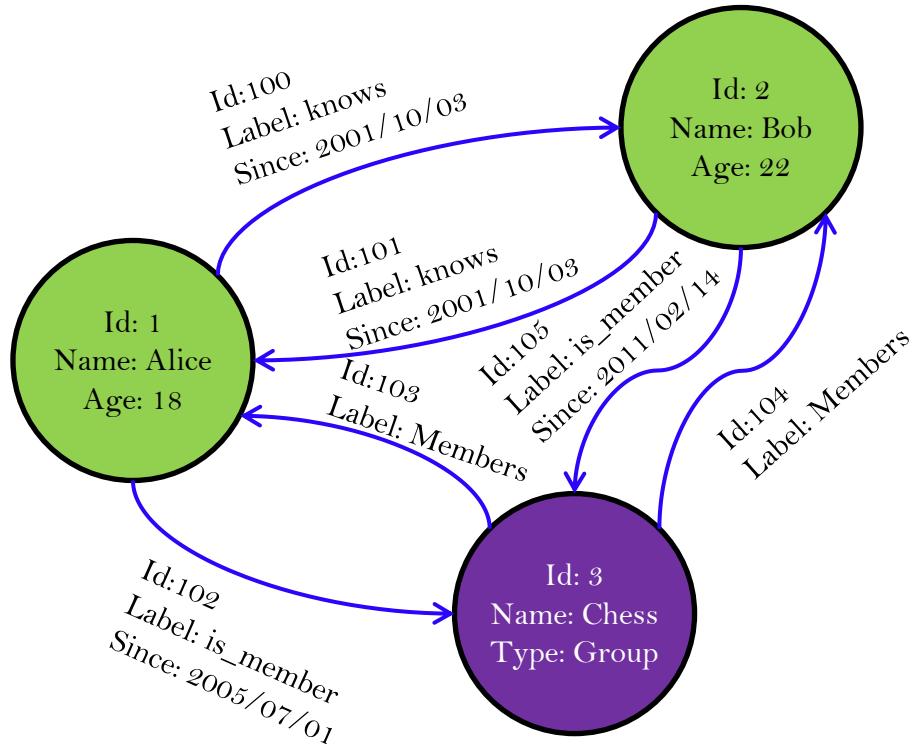
Types of NoSQL Databases

- Here is a limited taxonomy of NoSQL databases:



Graph Databases

- Data are represented as vertices and edges



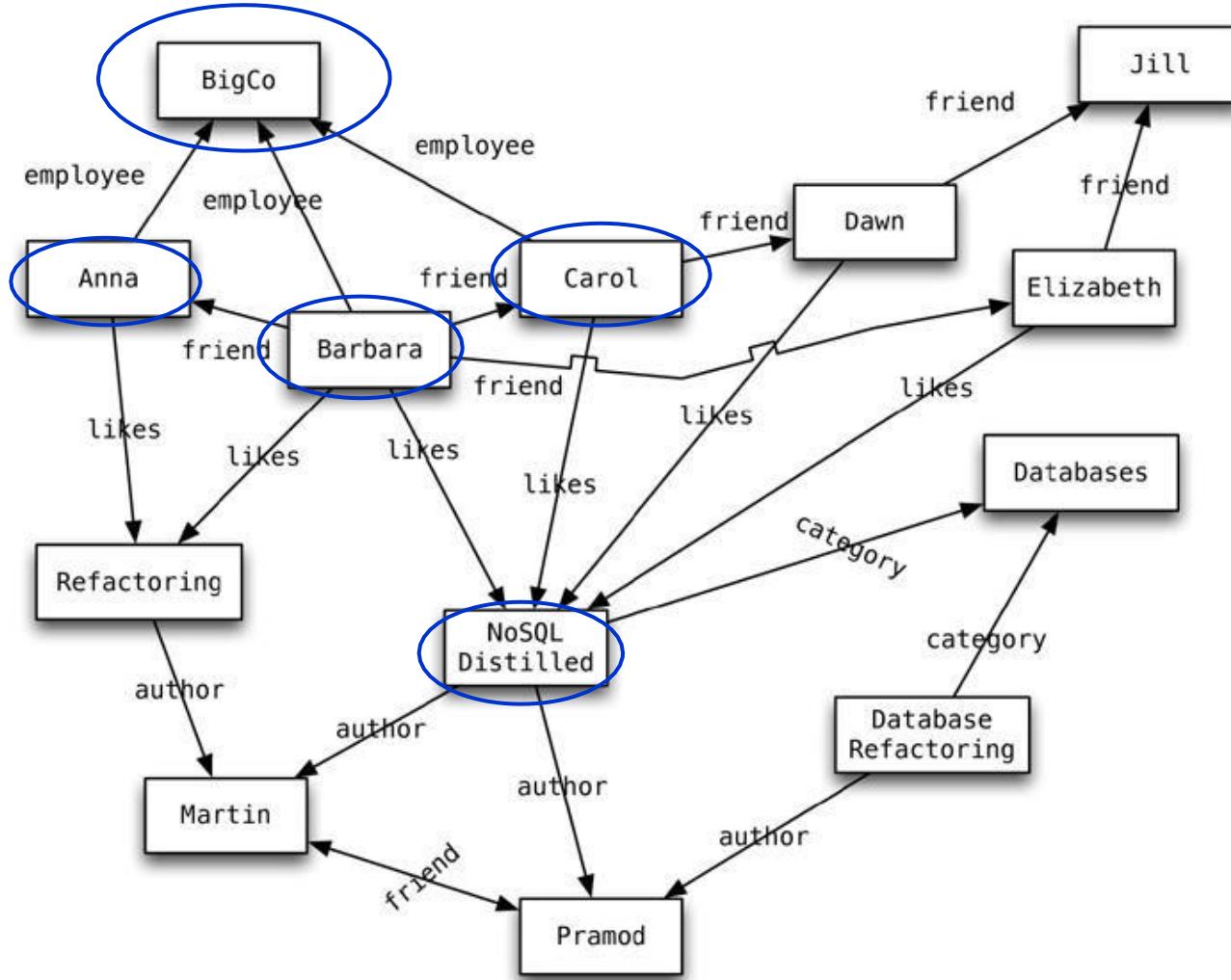
- Graph databases are powerful for graph-like queries (e.g., find the shortest path between two elements)
- E.g., Neo4j and VertexDB

Graph Databases

Basic Characteristics

- To store entities and relationships between these entities
 - Node is an instance of an object
 - Nodes have properties
 - e.g., name
 - Edges have directional significance
 - Edges have types
 - e.g., likes, friend, ...
- Nodes are organized by relationships
 - Allow to find interesting patterns
 - e.g., “Get all nodes employed by Big Co that like NoSQL Distilled”

Example:



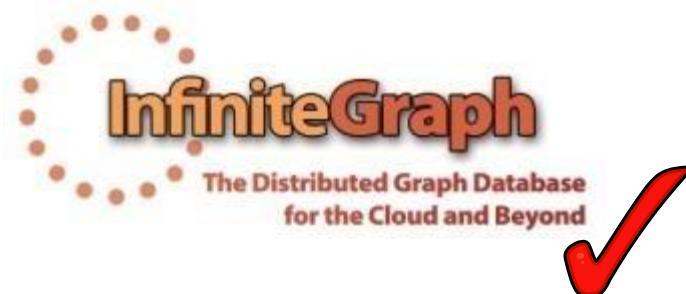
Graph Databases

RDBMS vs. Graph Databases

- When we store a graph-like structure a in RDBMS, it is for single type of relationship
 - “Who is my manager”
- Adding another relationship usually means a lot of schema changes
- In RDBMS we model the graph beforehand based on the Traversal we want
 - If the Traversal changes, the data will have to change
 - In graph databases the relationship is not calculated at query time but persisted

Graph Databases

Representatives



FlockDB

Graph Databases

Suitable Use Cases

Connected Data

- Social networks
- Any link-rich domain is well suited for graph databases

Routing, Dispatch, and Location-Based Services

- Node = location or address that has a delivery
- Graph = nodes where a delivery has to be made
- Relationships = distance

Recommendation Engines

- “your friends also bought this product”
- “when invoicing this item, these other items are usually invoiced”

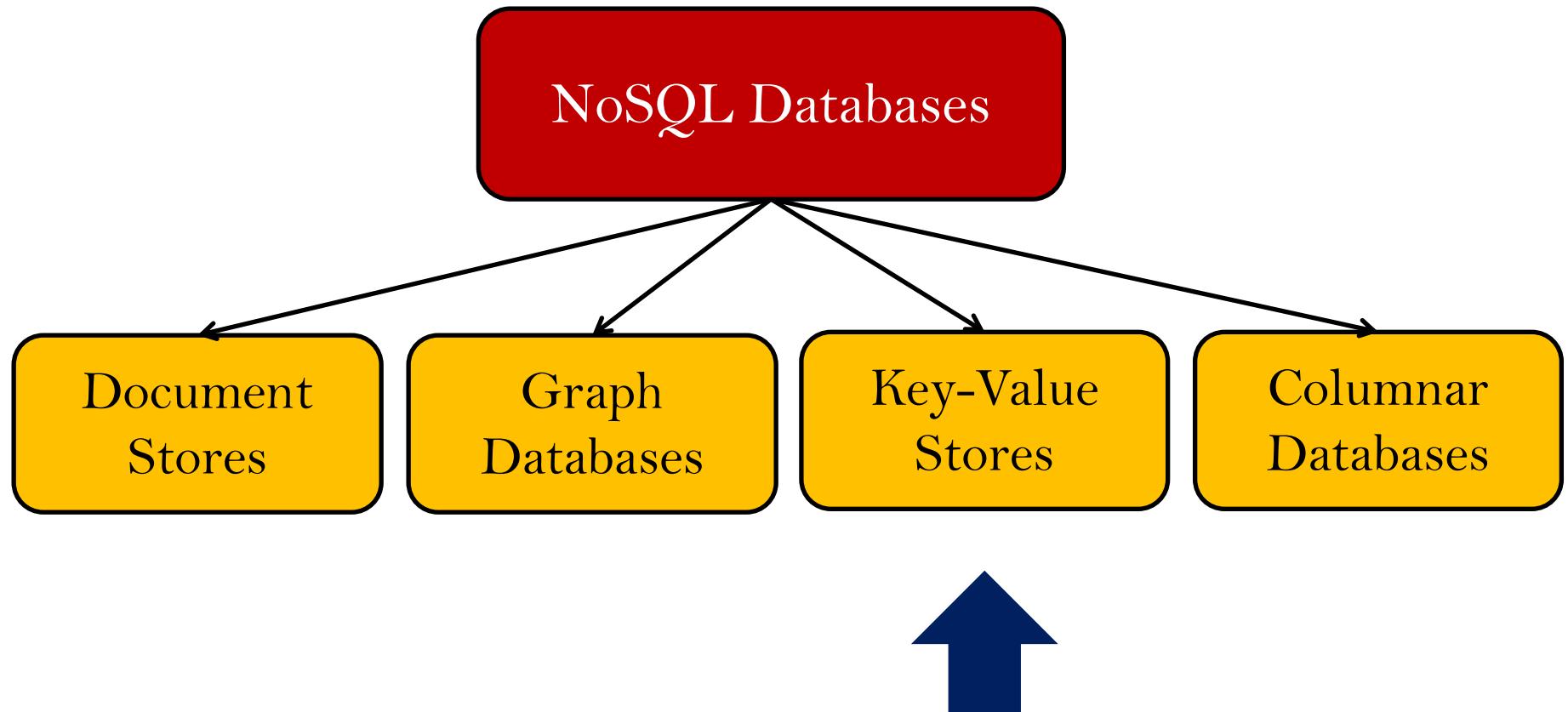
Graph Databases

When Not to Use

- When we want to update all or a subset of entities
 - Changing a property on all the nodes is not a straightforward operation
 - e.g., analytics solution where all entities may need to be updated with a changed property
- Some graph databases may be unable to handle lots of data
 - Distribution of a graph is difficult or impossible

Types of NoSQL Databases

- Here is a limited taxonomy of NoSQL databases:



Key-Value Stores

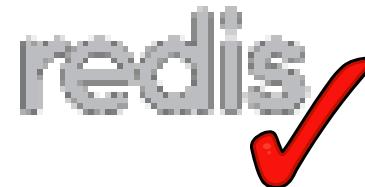
- Keys are mapped to (possibly) more complex value (e.g., lists)
- Keys can be stored in a hash table and can be distributed easily
- Such stores typically support regular CRUD (create, read, update, and delete) operations
 - That is, no joins and aggregate functions
- E.g., Amazon DynamoDB and Apache Cassandra

Key-value store

Basic characteristics

- The simplest NoSQL data stores
- A simple hash table (map), primarily used when all access to the database is via primary key
- A table in RDBMS with two columns, such as ID and NAME
 - ID column being the key
 - NAME column storing the value
 - A BLOB that the data store just stores
- Basic operations:
 - Get the value for the key
 - Put a value for a key
 - Delete a key from the data store
- Simple: great performance, easily scaled
- Simple: not for complex queries, aggregation needs

Key-value store Representatives



MemcachedDB



ORACLE®
BERKELEY DB

Hamster DB
embedded database



not
open-source

open-source
version



**Project
Voldemort**

Key-value store

Suitable Use Cases

Storing Session Information

- Every web session is assigned a unique session_id value
- Everything about the session can be stored by a single PUT request or retrieved using a single GET
- Fast, everything is stored in a single object

User Profiles, Preferences

- Every user has a unique user_id, user_name + preferences such as language, colour, time zone, which products the user has access to,
...
- As in the previous case:
 - Fast, single object, single GET/PUT

Shopping Cart Data

- Similar to the previous cases

Key-value store

When Not to Use

Relationships among Data

- Relationships between different sets of data
- Some key-value stores provide link-walking features
 - Not usual

Multioperation Transactions

- Saving multiple keys
 - Failure to save any one of them → revert or roll back the rest of the operations

Query by Data

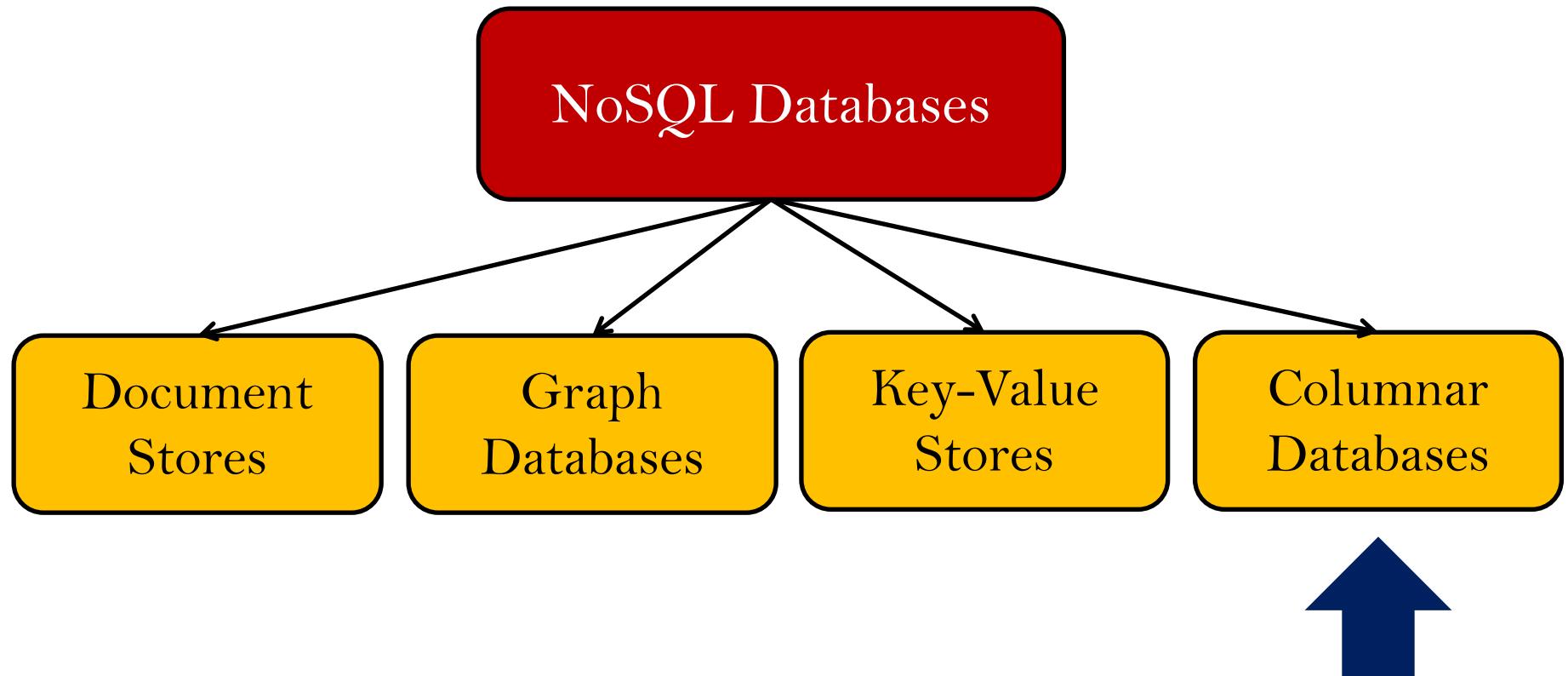
- Search the keys based on something found in the value part

Operations by Sets

- Operations are limited to one key at a time
- No way to operate upon multiple keys at the same time

Types of NoSQL Databases

- Here is a limited taxonomy of NoSQL databases:



Columnar Databases

- Columnar databases are a hybrid of RDBMSs and Key-Value stores
 - Values are stored in groups of zero or more columns, but in Column-Order (as opposed to Row-Order)

Record 1			
Alice	3	25	Bob
4	19	Carol	0
45			

Row-Order

Column A			
Alice	Bob	Carol	
3	4	0	25
19	45		

Columnar

Column A = Group A			
Alice	Bob	Carol	
3	25	4	19
0	45		

Column Family {B, C}

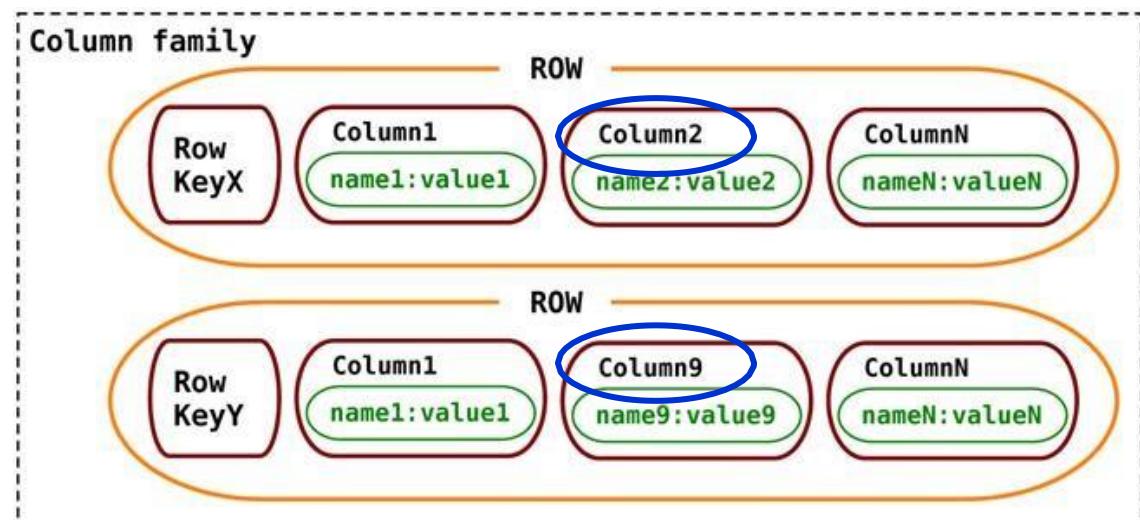
Columnar with Locality Groups

- Values are queried by matching keys
- E.g., HBase and Vertica

Column-Family Stores

Basic Characteristics

- Also “columnar” or “column-oriented”
- Column families = rows that have many columns associated with a row key
- Column families are groups of related data that is often accessed together
 - e.g., for a customer we access all profile information at the same time, but not orders



Column-Family Stores

Representatives

Google's
BigTable



Cassandra



HYPERTABLE

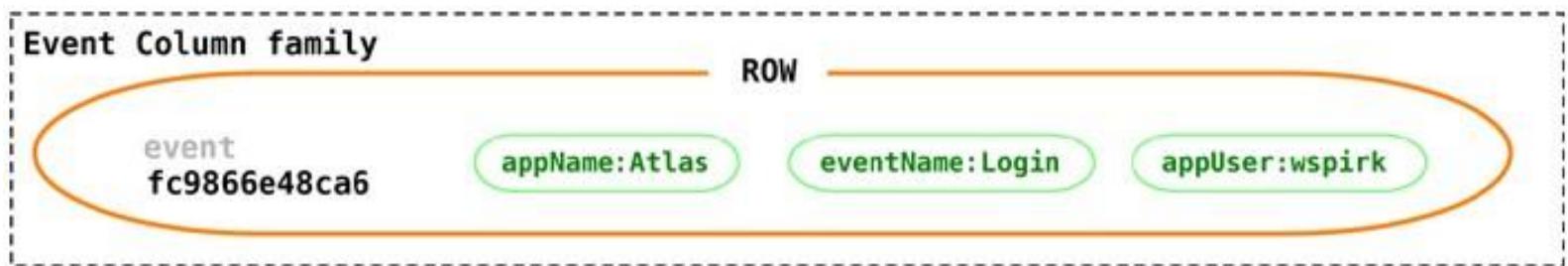


SimpleDB

amazon.com

Column-Family Stores

Suitable Use Cases



Event Logging

- Ability to store any data structures → good choice to store event information

Content Management Systems, Blogging Platforms

- We can store blog entries with tags, categories, links, and trackbacks in different columns
- Comments can be either stored in the same row or moved to a different keyspace
- Blog users and the actual blogs can be put into different column families

Column-Family Stores

When Not to Use

Systems that Require ACID Transactions

- Column-family stores are not just a special kind of RDBMSs with variable set of columns!

Aggregation of the Data Using Queries

- (such as SUM or AVG)
- Have to be done on the client side

For Early Prototypes

- We are not sure how the query patterns may change
- As the query patterns change, we have to change the column family design

References

- <http://nosql-database.org/>
- Pramod J. Sadalage – Martin Fowler: **NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence**
- Eric Redmond – Jim R. Wilson: **Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement**
- Sherif Sakr – Eric Pardede: **Graph Data Management: Techniques and Applications**
- Shashank Tiwari: **Professional NoSQL**

Lecture 6

Introduction to HDFS

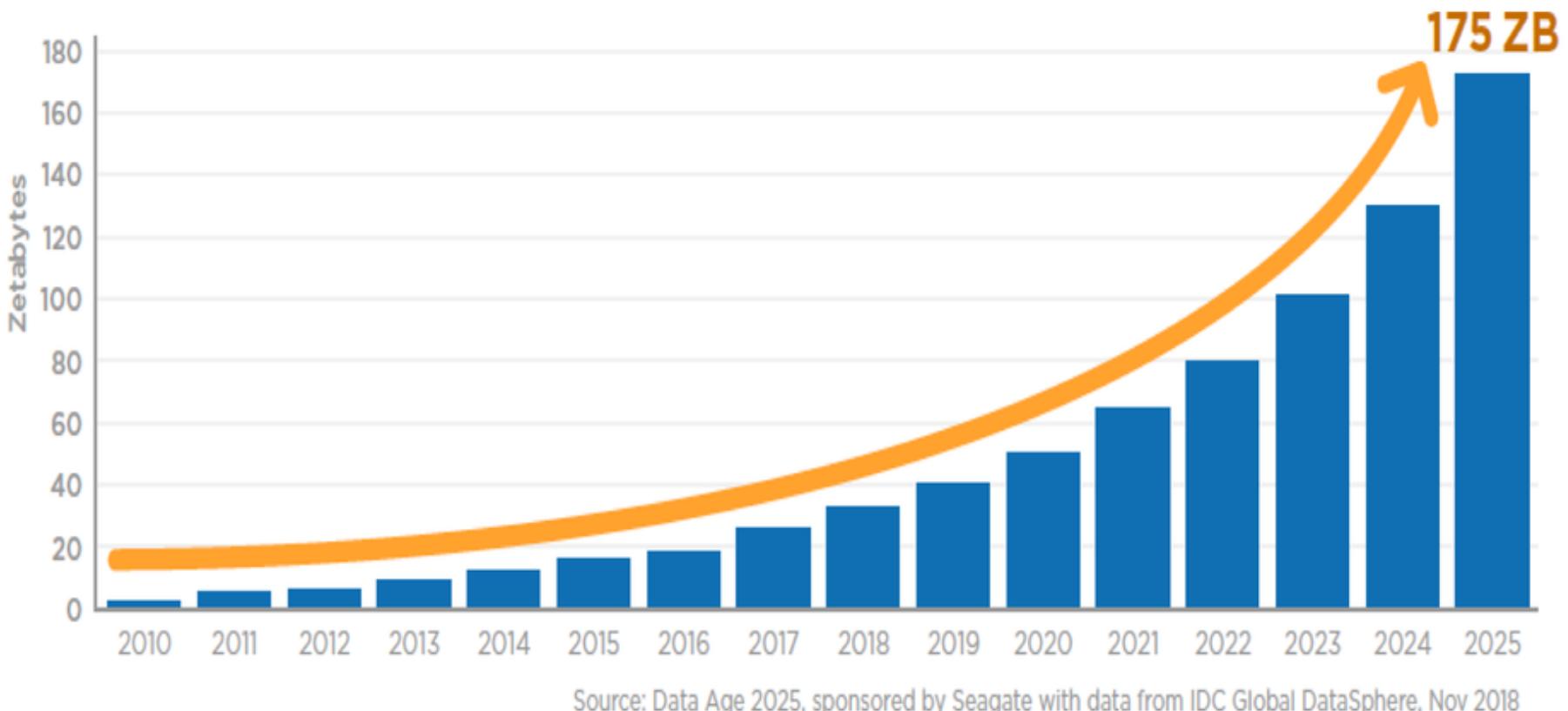
Agenda

- Brief history of data management and Hadoop
- Introduction to HDFS and Hadoop
- Understanding Hadoop concepts and terms
- HDFS Architecture
- Read and Write operations on Hadoop
- Hadoop High Availability

Learning Outcomes

- You can explain what vertical and horizontal scaling is
- You can list server roles in HDFS
- You can explain how topology affects replica placement
- You can explain what chunk / block size is used for
- You can explain in detail how HDFS client reads and writes data

Why Distributed Systems?



<https://medium.com/analytics-vidhya/the-5-vs-of-big-data-2758bfcc51d>

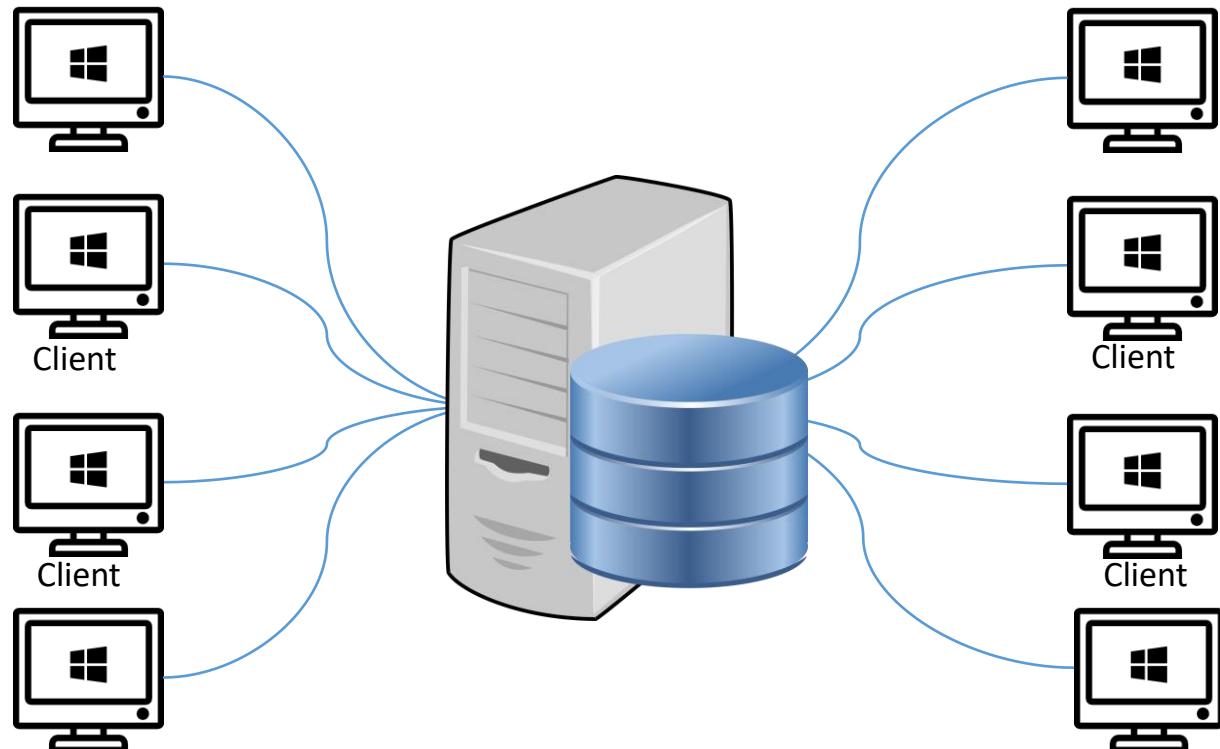
Why Distributed Systems

- Storage for Large Datasets
 - Limitations of conventional RDBMS
 - The cost of data storage is high
 - Handling data in different formats
 - RDBMS can manipulate only structured data
 - Data getting generated with high speed
 - Real-time processing of data

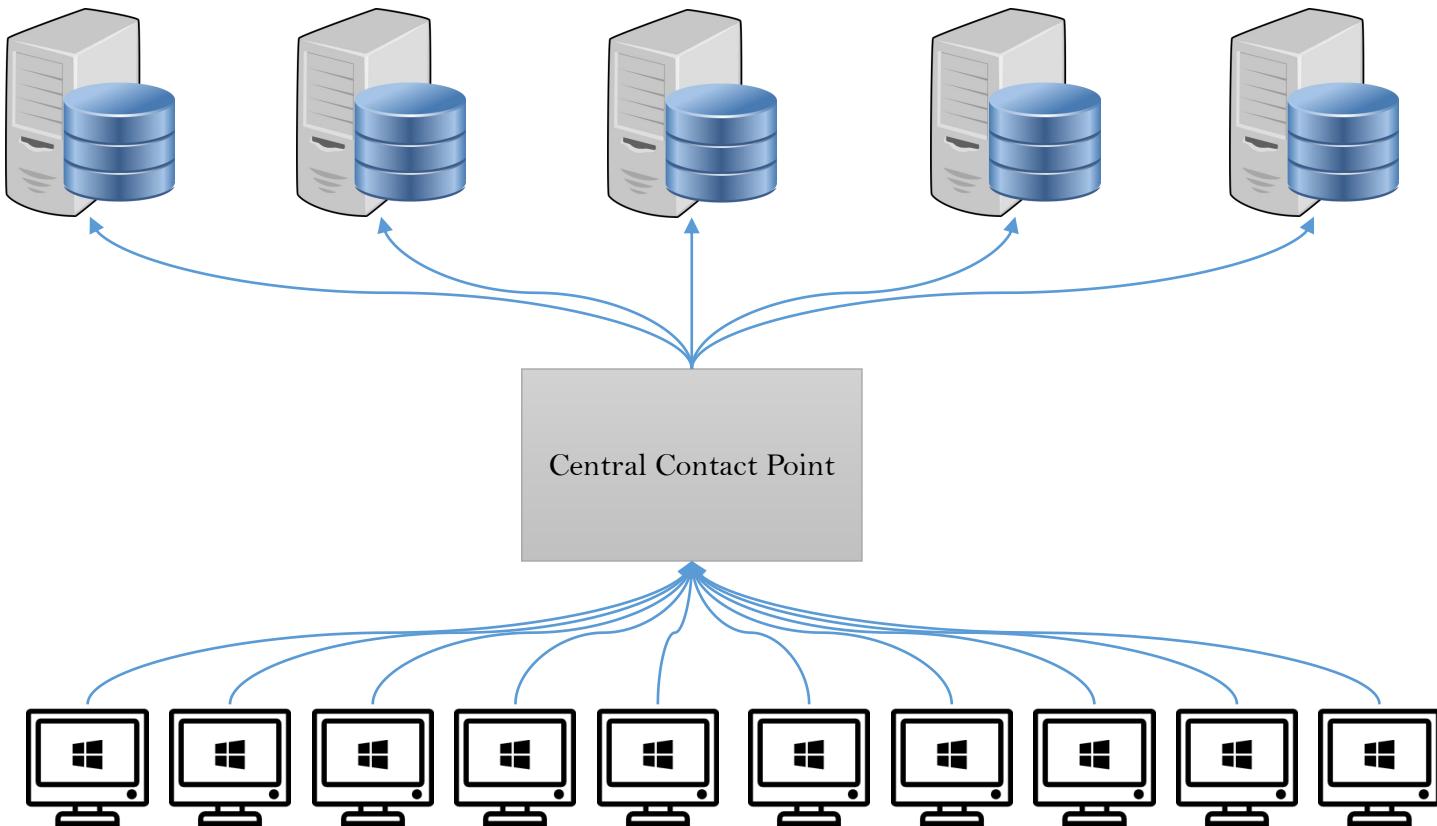
Hadoop vs Relational Database Management Systems

	Traditional RDBMS	Hadoop
Data size	Gigabytes	Petabytes
Access	Interactive and batch	Batch
Updates	Read and write many times	Write once, read many times
Transactions	ACID	None (BASE)
Structure	Schema-on-write	Schema-on-read
Integrity	High	Low
Scaling	Nonlinear	Linear

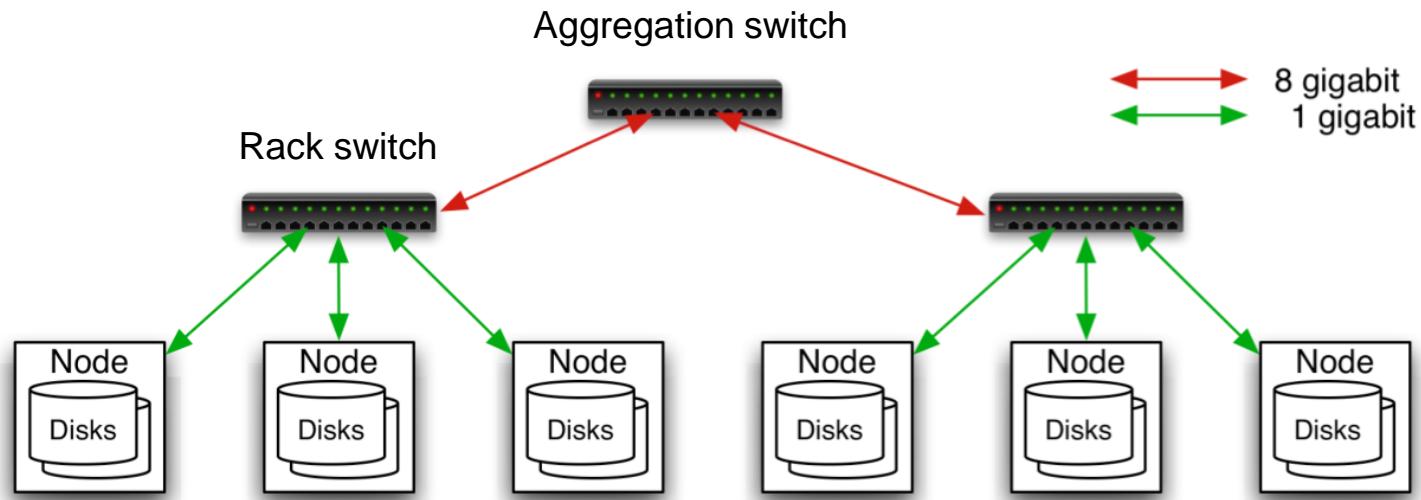
Vertical Scaling



Horizontal Scaling



Commodity Hardware

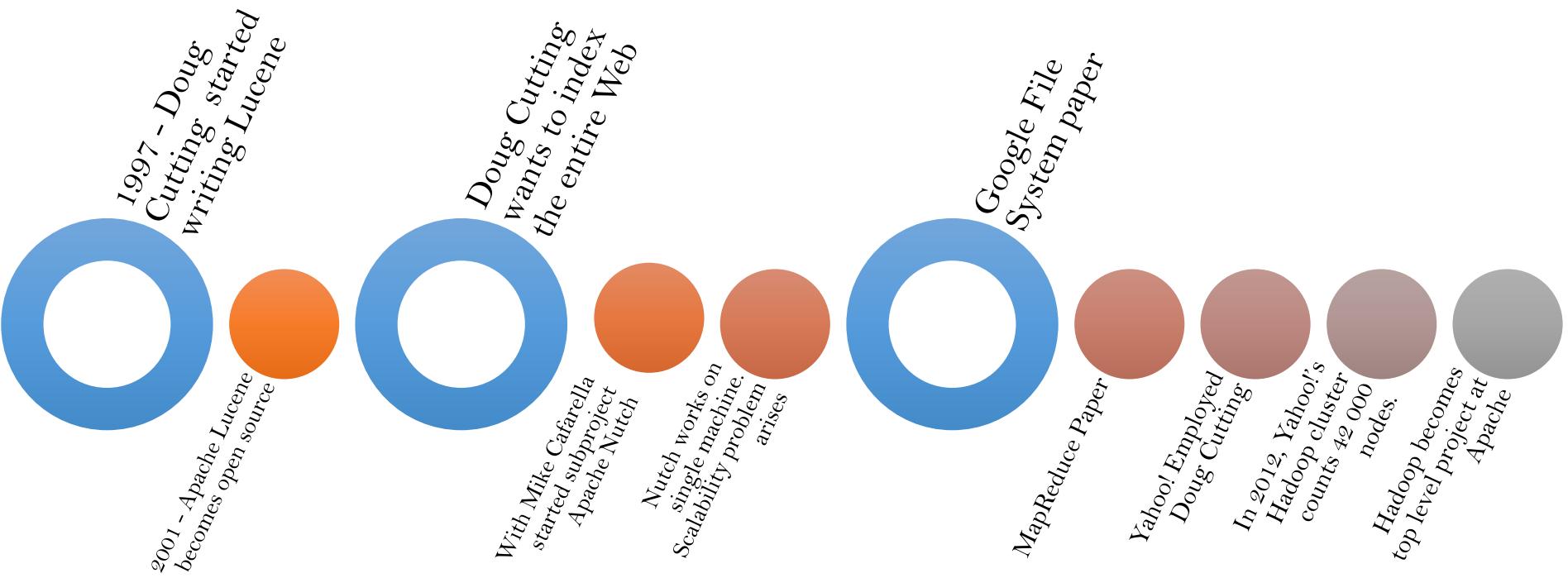


- Typically, in 2 level architecture
 - Nodes are commodity PCs
 - 30-40 nodes/rack
 - Uplink from rack is 3-4 gigabit
 - Rack-internal is 1 gigabit

Challenges

- Concurrency
- Partial failure
- Performance

History of Hadoop development



Google File System Key Messages

- Components failures are a norm rather than the exception: the quantity and quality of the components virtually guarantee that some are not functional at any given time.
- Most files are huge by traditional standards.
- Write – once – read –many. Most files are mutated by appending new data rather than overwriting existing data. Once written, the files are only read, and often only sequentially.

The Design of DFS

- Designed for:
 - Very large files
 - Streaming data access
 - Commodity hardware
- Not a good fit for:
 - Low latency data access
 - Lots of small files
 - Multiple writers, arbitrary file modifications

- **Hadoop Distributed File System:** A reliable, high bandwidth, low-cost data storage cluster that facilitates the management of related files across machines.
- **MapReduce engine:** A high performance parallel/distributed data processing implementation of the MapReduce algorithm.

HDFS Concepts: Blocks

- HDFS Block 128Mb by default
- Unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage.
 - 1) File can be larger than any single disk in the network → blocks of a file can be stored in any of the disks in the cluster.
 - 2) Making the unit of abstraction block simplifies the storage subsystem (easy to calculate how many can be stored on a given disk) and minimize the cost of seek
 - 3) Blocks fit well with replication for providing fault tolerance and high availability

DATA.TXT 1100Mb

128Mb

128Mb

128Mb

128Mb

128Mb

128Mb

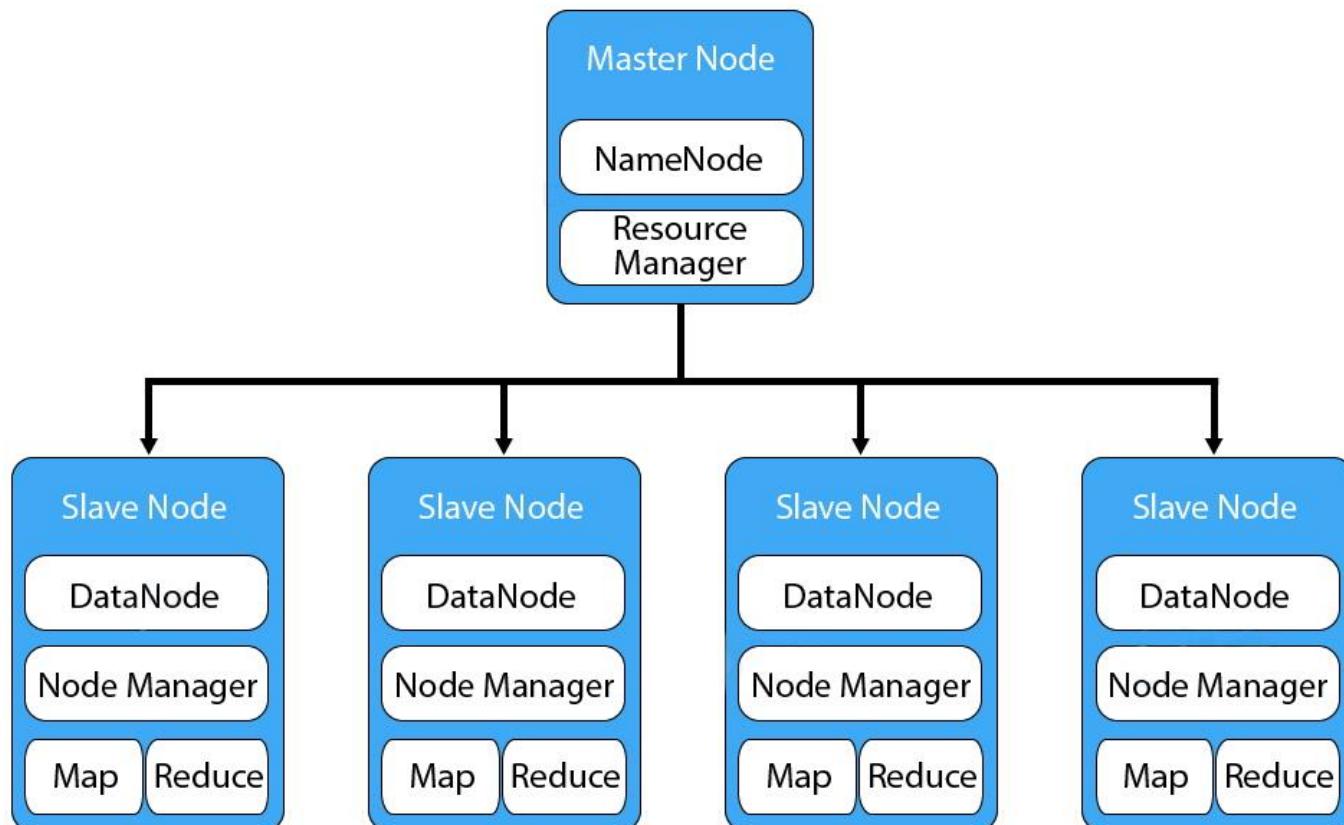
128Mb

128Mb

76

HDFS Concepts: Namenodes and Datanodes

- An HDFS cluster has two types of nodes operating in a master-worker pattern: a namenode (the master) and a number of datanodes (workers)



Namenode (Master Node)

- Manages the filesystem namespace
- Maintains the filesystem tree and the metadata for all the files and directories in the tree
- This information is stored persistently on the local disk in the form of two files: the namespace image and the edit log.
- Namenode knows the datanodes on which all the blocks for a given file are located; (but it does not store block locations persistently, because this information is reconstructed from datanodes when the system starts)
- Client access the filesystem by communicating with the namenode and datanodes
- Namenode periodically communicates with each data node in HeartBeat messages to give it instructions and collect its state

Namenodes

- Single point of failure
- Important to make it resilient to failure:
 - 1) back up the files that make up the persistent state of the filesystem metadata
 - 2) to run a secondary namenode. Its main role is to periodically merge the namespace image with the edit log. Usually runs on a separate physical machine because it requires plenty of CPU and as much memory as the namenode to perform the merge.

System is designed to minimize the master's involvement in all operations.

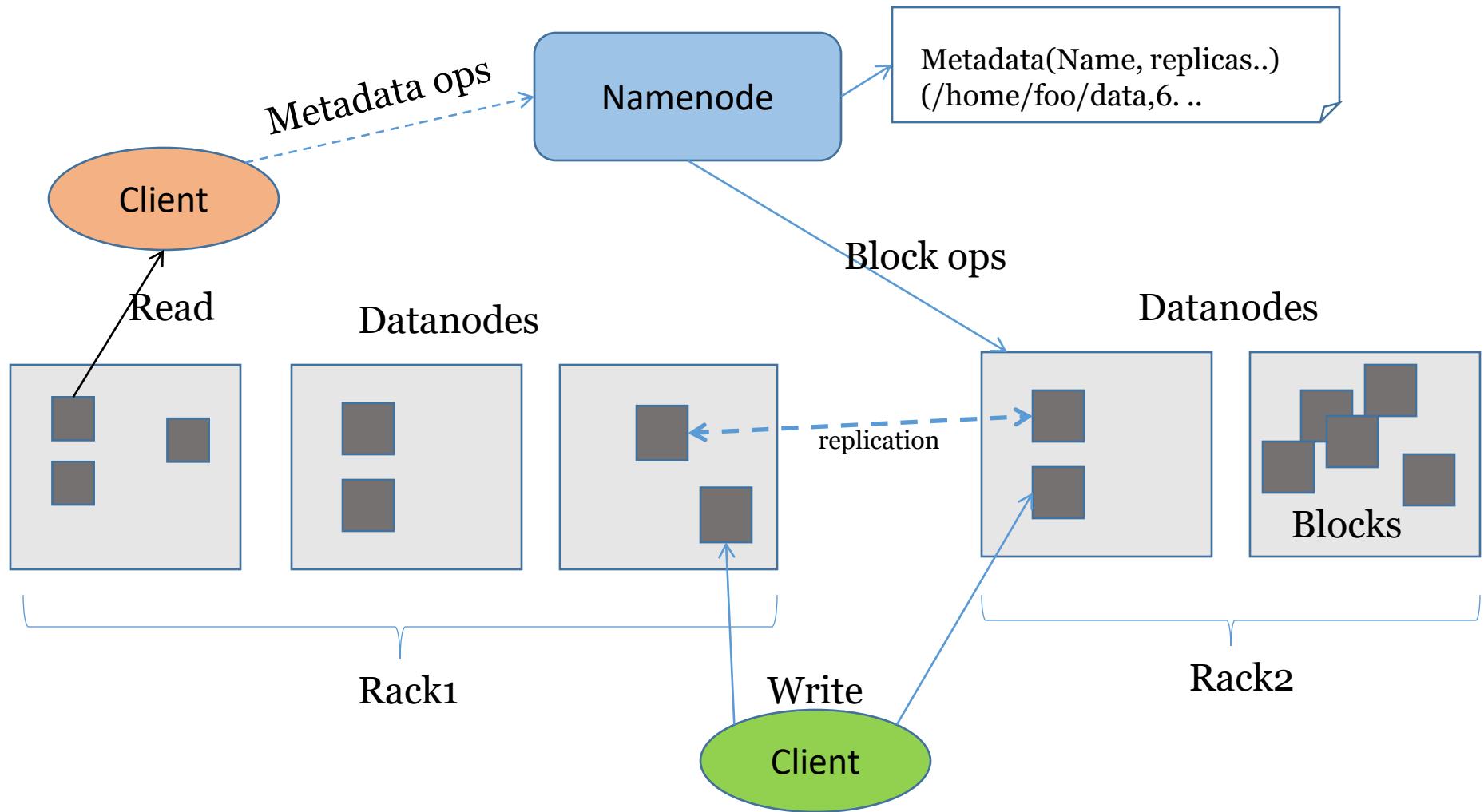
Datanode

- Datanodes are the workhorses of the filesystem.
- They store and retrieve blocks when they are told to
- They report back to the namenode periodically with lists of blocks that they are storing (heartbeat messages are sent every 3 seconds)

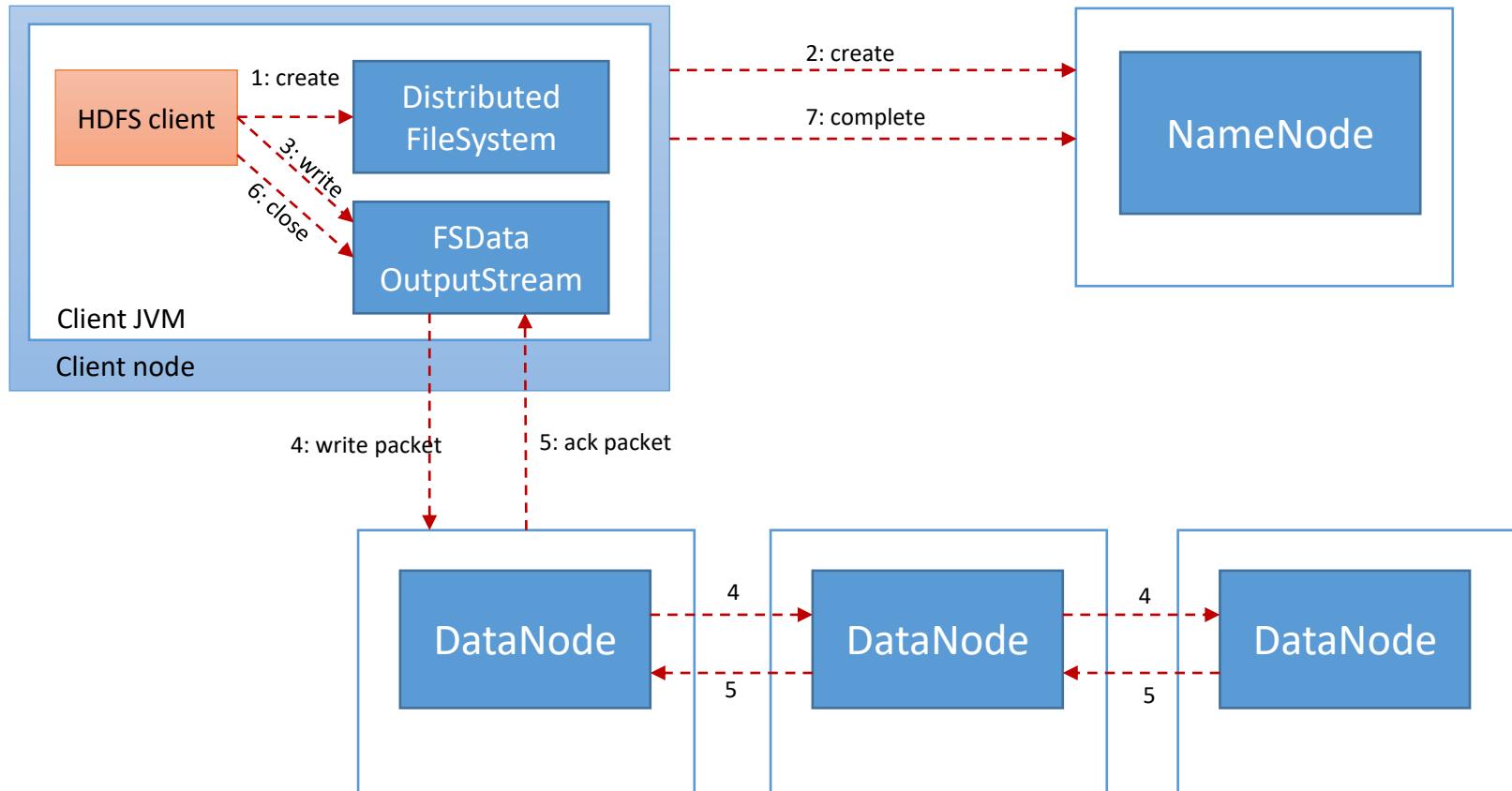
Block Caching

- Normally a datanode reads blocks from disk, but frequently accessed blocks may be cached in the datanodes` memory in a *block cache*.
- Job schedulers (for MapReduce, Spark and other frameworks) can use cached blocks for increased performance
- A small lookup table used in a join is a good candidate for caching, for example.
- Users or applications instruct the namenode which files to cache by adding a cache directive to a cache pool.

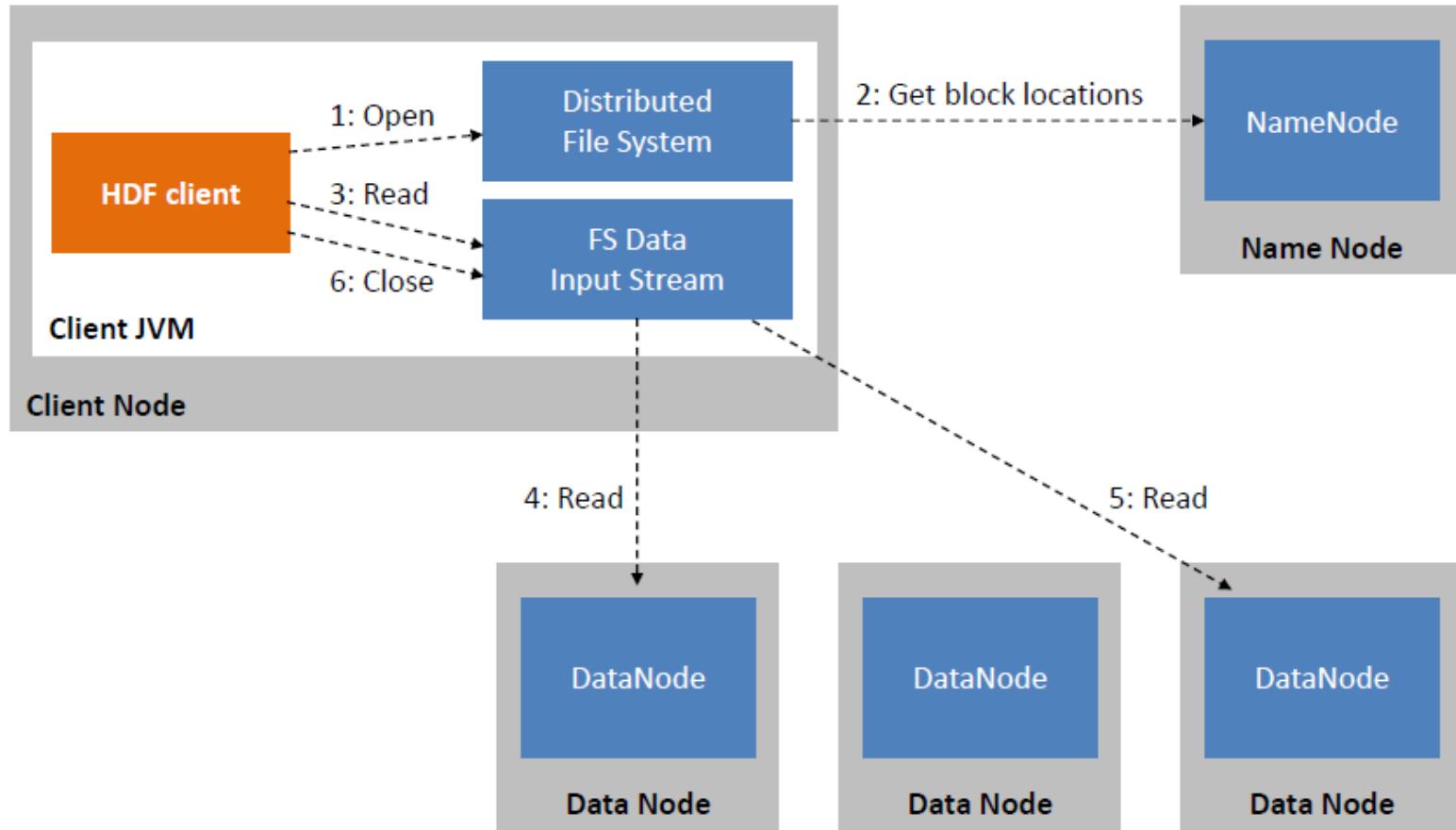
HDFS Architecture



Hadoop write operation

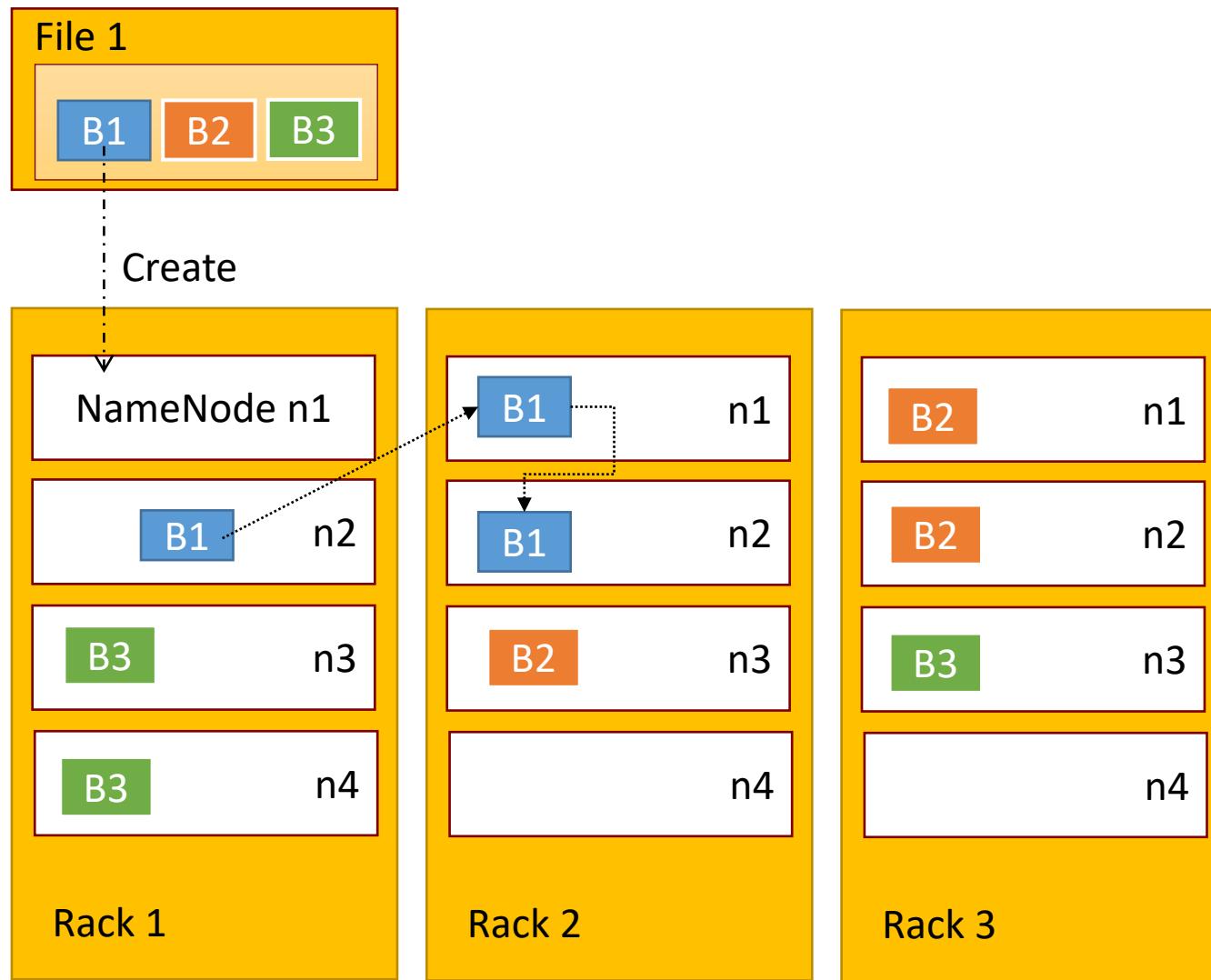


How the files are read in HDFS?

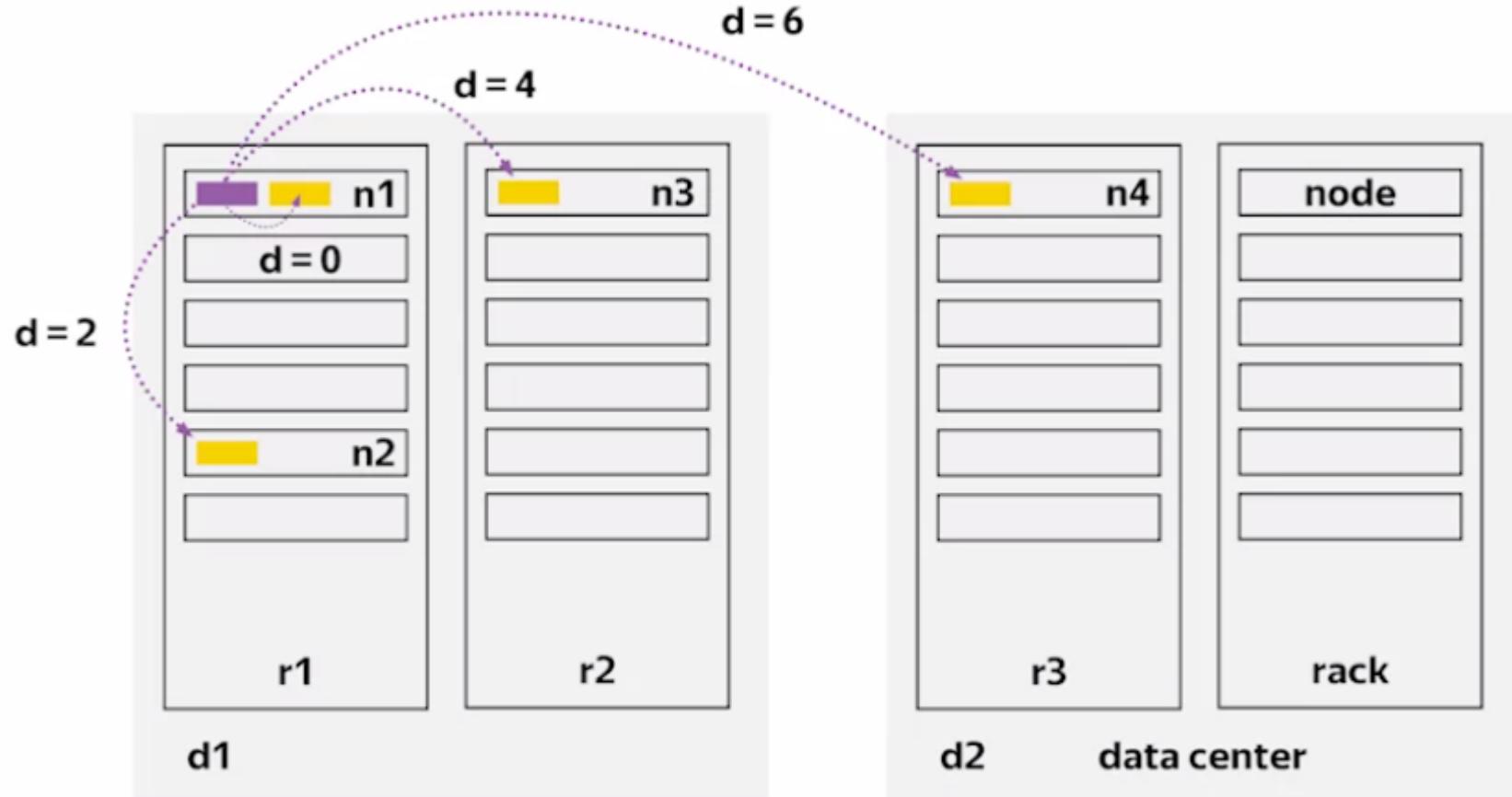


How the replication is done in HDFS?

HDFS Client



Rack awareness



Replica states

Being written (RBW) - corresponds to the replica of the last block of unclosed file. Generally, the replica goes to this state every time when it's just created or reopened to append. Other replicas belonging to the same block can differ from current replica. RBW replica is visible for readers.

Waiting to be recovered (RWR) - represents the state when DataNode restarts or dies. When it happens, all replicas held by this DataNode goes to RWR state. After restarting RWR replica can become outdated. If it happens, it will be discarded. In the other case, it will participate in recovery process.

Under recovery (RUR) - replica is in this state when its recovery is triggered by the lease expiration. This situation is called *lease recovery*.

Temporary - replicas in this state are created only for replication and balancing. It looks almost like RBW replica with the difference that it's not visible for readers.

Finalized - for this state the replica is fully written and closed. Its data and meta data are consistent.

Reading

- Hadoop: The Definitive Guide, Tom White, 4th Edition, O Reilly
- Google File Systems, Sanjay Ghemawat, Howard Gobioff
- <https://blog.cloudera.com/understanding-hdfs-recovery-processes-part-1/>
- <https://gerardnico.com/db/hadoop/hdfs/replication>

Introduction to HBase

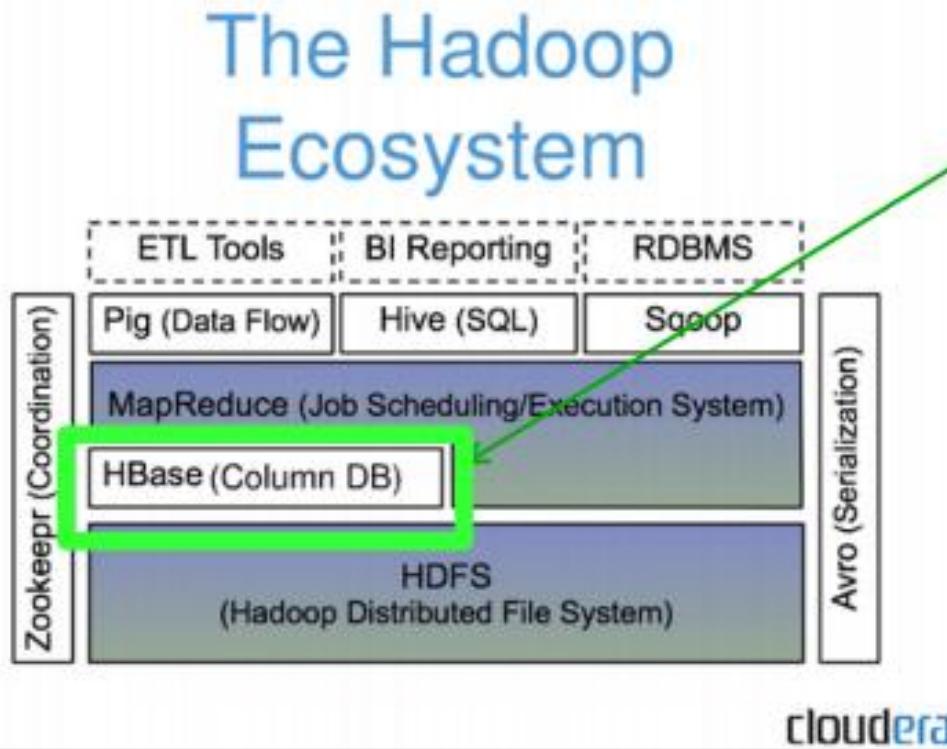
Agenda

- What is HBase
- How does it relate to HDFS
- What is HBase Data Model?
- Underlying HBase Architecture
- Using HBase in Practice

What is Hbase

- HBase is a distributed **column-oriented** data store built on top of HDFS
- Hadoop database, a distributed, scalable (horizontally scalable), big data store
- Modeled after Google's Big Table (designed to provide quick random access to huge amounts of structured data)
- NoSQL data store

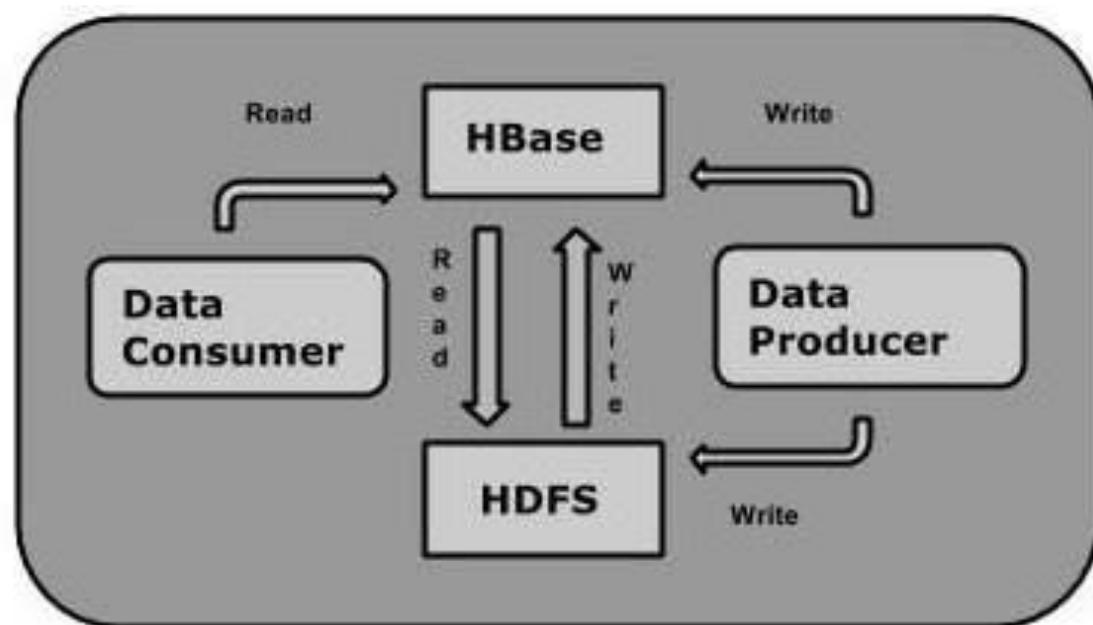
The Hadoop Ecosystem



- HBase is built on top of HDFS
- HBase files are internally stored in HDFS

Storing Data in HBase

- One can store the data in HDFS either directly or through HBase.
- Data consumer reads/accesses the data in HDFS randomly using HBase.
- HBase sits on top of the Hadoop File System and provides read and write access.



Storage Mechanism in HBase

- HBase is a **column-oriented database** and the tables in it are sorted by row.
- The table schema defines only column families, which are the key value pairs.
- A table have multiple column families and each column family can have any number of columns.
- Subsequent column values are stored contiguously on the disk. Each cell value of the table has a timestamp.
- In an HBase:
 - Table is a collection of rows.
 - Row is a collection of column families.
 - Column family is a collection of columns.
 - Column is a collection of key value pairs.

Storage in HBase

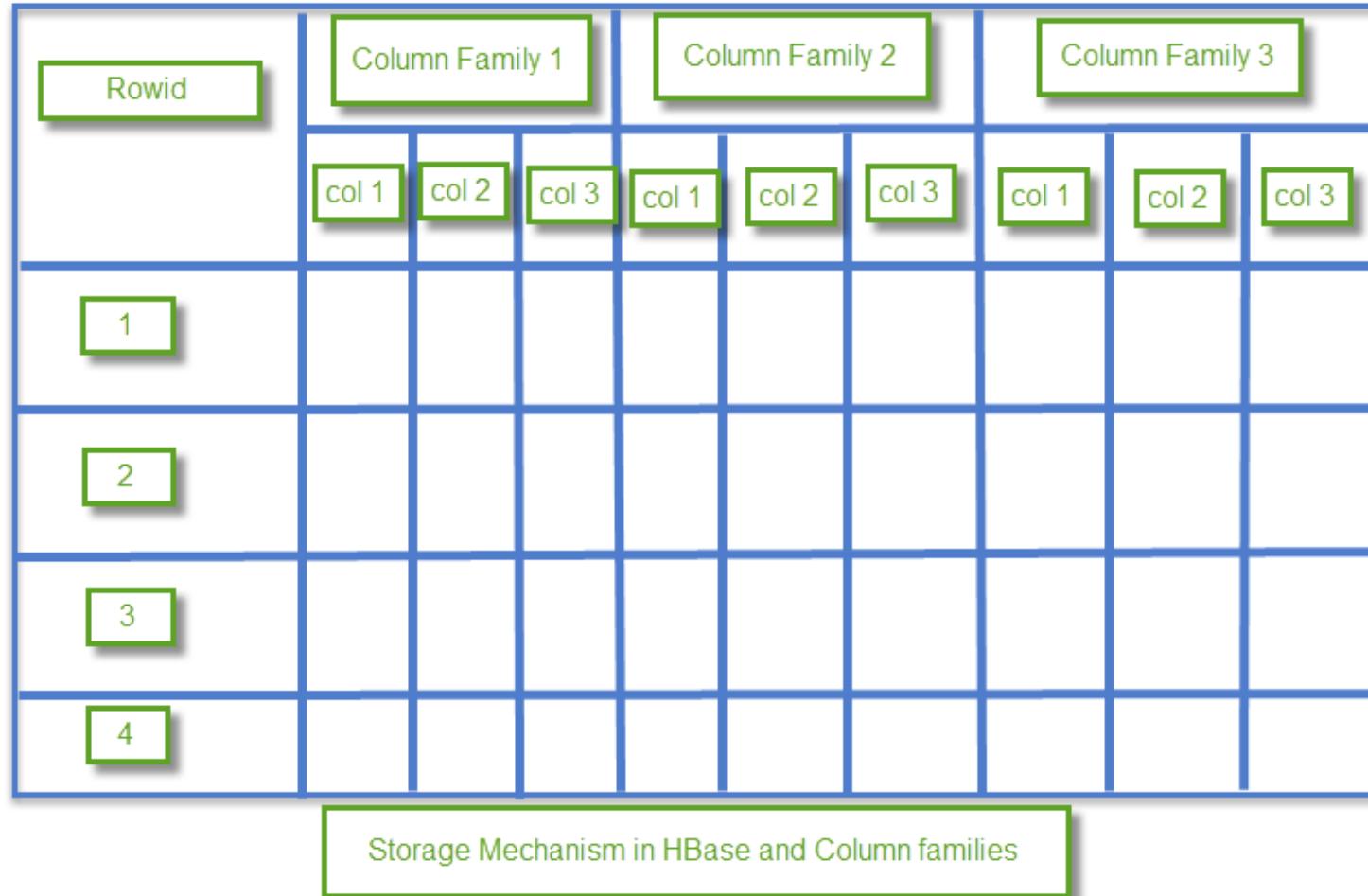


Table: Collection of rows present.

Row: Collection of column families.

Column Family: Collection of columns.

Column: Collection of key-value pairs.

Namespace: Logical grouping of tables.

Cell: A {row, column, version} tuple exactly specifies a cell definition in HBase.

Column Families in a Column-Oriented DB

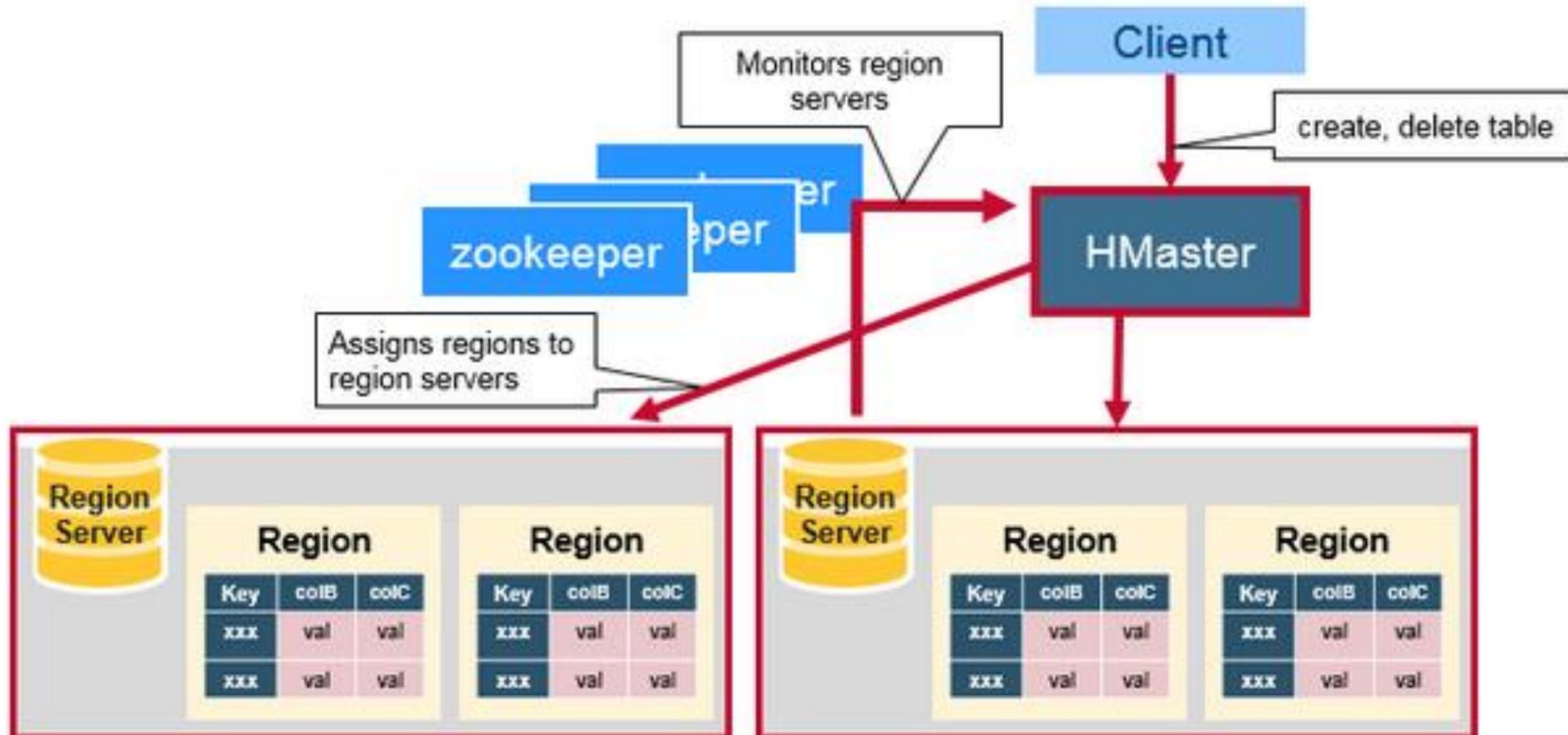
COLUMN FAMILIES

Row key	personal data		professional data	
empid	name	city	designation	salary
1	raju	hyderabad	manager	50,000
2	ravi	chennai	sr.engineer	30,000
3	rajesh	delhi	jr.engineer	25,000

HBase Architecture

- In HBase, tables are split into regions and are served by the region servers.
- Regions are vertically divided by column families into “Stores”. Stores are saved as files in HDFS.
- HBase has three major components:
 - client library
 - master server
 - region servers
- Region servers can be added or removed as per requirement.

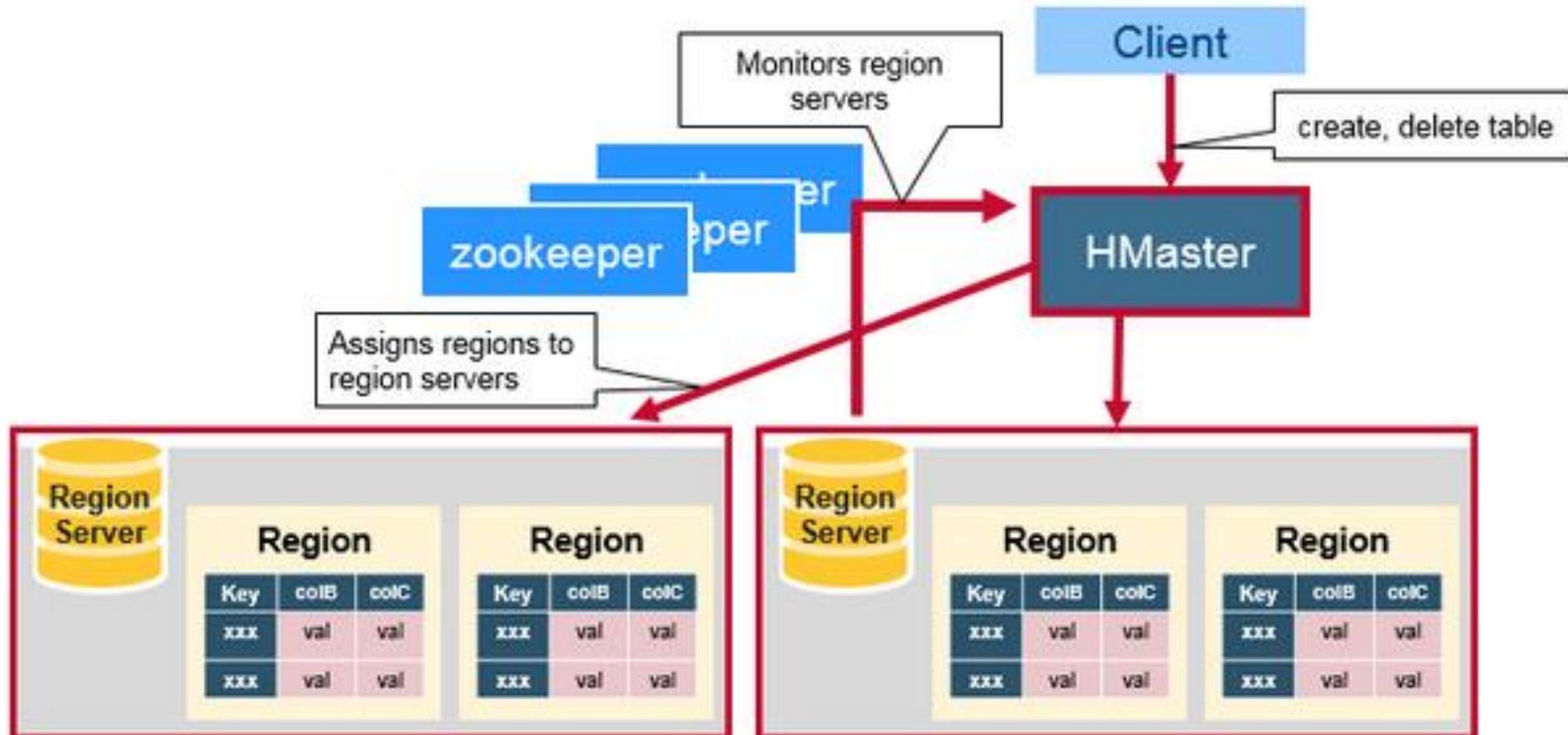
HBase Architecture



Master Server in HBase

- The master server
 - Assigns regions to the region servers and takes the help of Apache Zookeeper for this task.
 - Handles load balancing of the regions across region servers. It unloads the busy servers and shifts the regions to less occupied servers.
 - Maintains the state of the cluster by negotiating the load balancing.
 - Is responsible for schema changes and other metadata operations such as creation of tables and column families.

HBase Architecture



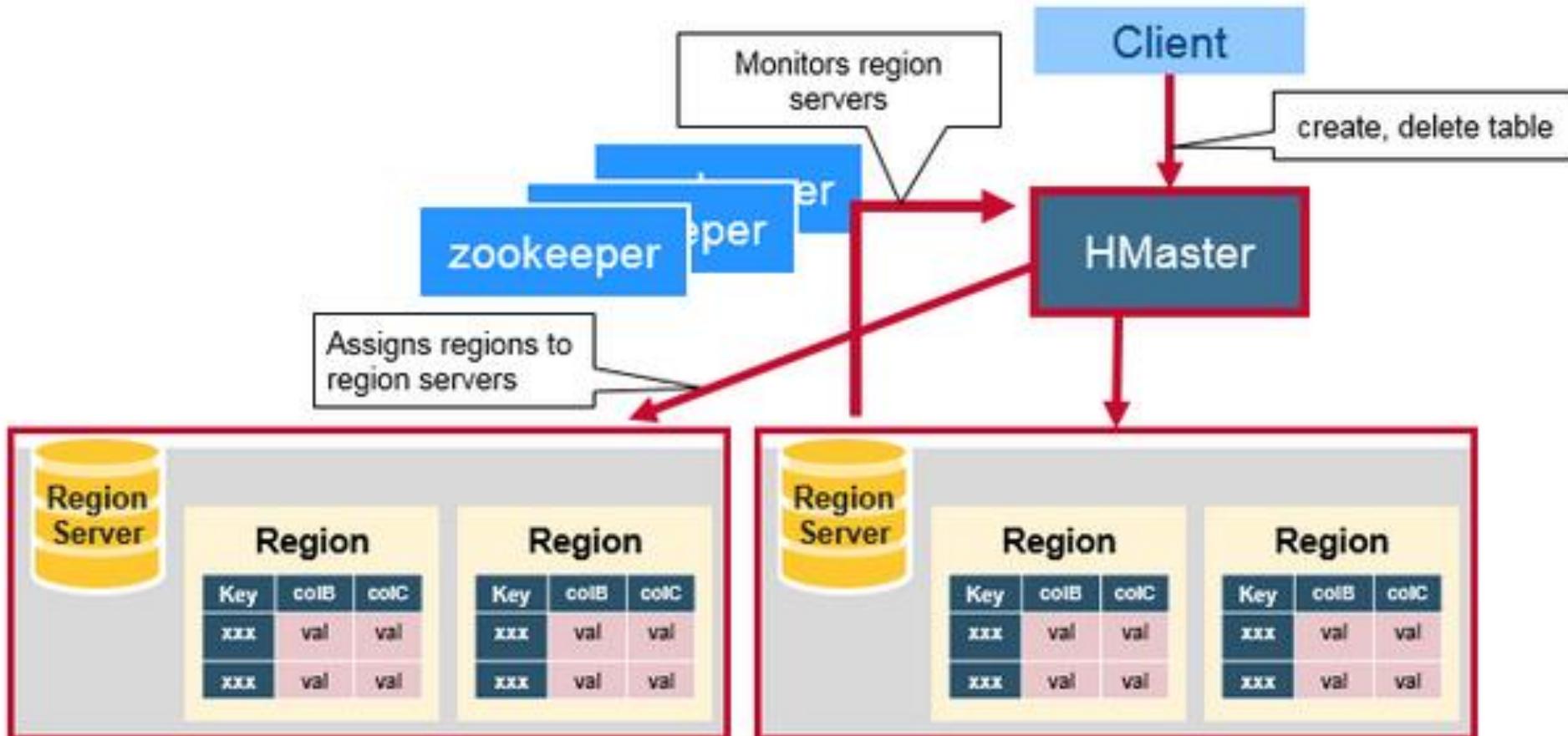
Regions in HBase

- Regions are tables that are split up and spread across the region servers.

Region Server in HBase

- The region servers have regions. Servers
 - communicate with the client and handle data-related operations
 - handle read and write requests for all regions under it
 - decide the size of the region by following the region size thresholds.

HBase Architecture



HBase key messages

- HBase is not a relational database
- HBase is sparse – lots of null empty values,
- HBase is distributed – share-nothing architecture,
- HBase is persistent, multi-dimensional
- Sorted map or Key/value store

Features of HBase

- HBase is linearly scalable.
- It has automatic failure support.
- It provides consistent read and write.
- It integrates with Hadoop.
- It has java API for client.
- It provides data replication across clusters.

HBase and HDFS Comparison

HDFS	HBase
HDFS is a distributed file system suitable for storing large files.	HBase is a database built on top of the HDFS.
HDFS does not support fast individual record lookups.	HBase provides fast lookups for larger tables.
It provides high latency batch processing; no concept of batch processing.	It provides low latency access to single rows from billions of records (Random access).
It provides only sequential access of data.	HBase internally uses Hash tables and provides random access, and it stores the data in indexed HDFS files for faster lookups.
Not good for updates	Support for updates

HBase and RDBMS Comparison

HBase	RDBMS
HBase is schema-less, it doesn't have the concept of fixed columns schema; defines only column families.	An RDBMS is governed by its schema, which describes the whole structure of tables.
It is built for wide tables. HBase is horizontally scalable.	It is thin and built for small tables. Hard to scale.
No transactions are there in HBase.	RDBMS is transactional.
It has de-normalized data.	It will have normalized data.
It is good for semi-structured as well as structured data.	It is good for structured data

Where to Use HBase

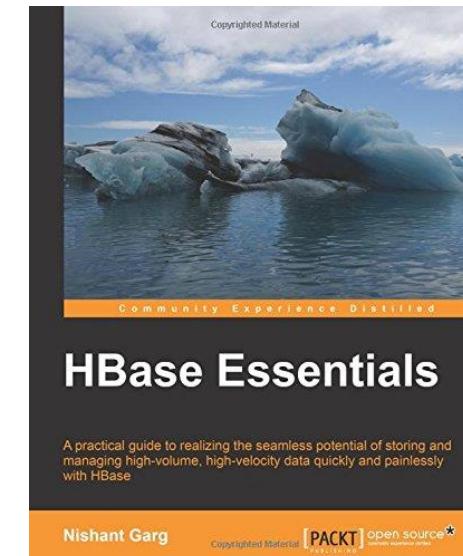
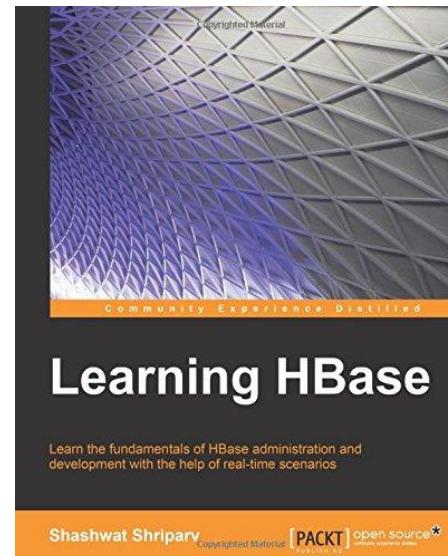
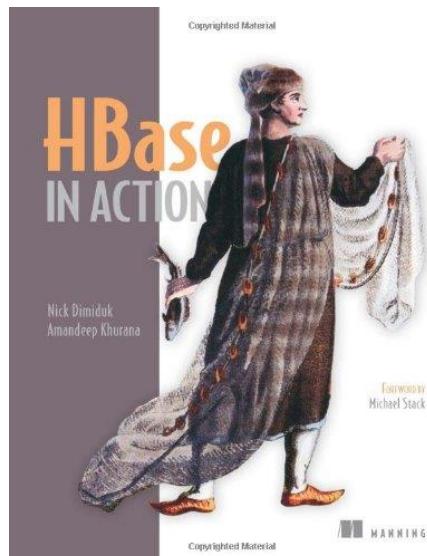
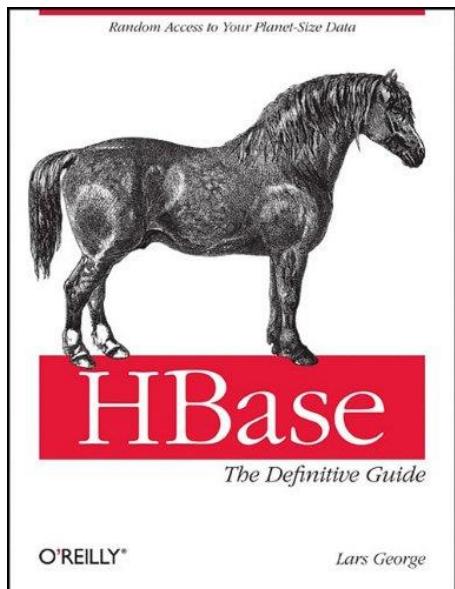
- Apache HBase is used to have random, real-time read/write access to Big Data.
- Need to perform many thousands of operations per second on multiple TB/PB of data
- It hosts very large tables on top of clusters of commodity hardware.
- Apache HBase is a non-relational database modeled after Google's Bigtable. Bigtable acts up on Google File System, likewise Apache HBase works on top of Hadoop and HDFS.

Applications of HBase

- HBase is used whenever we need to provide fast random access to available data.
- Companies such as Facebook, Twitter, Yahoo, and Adobe use HBase internally.
- Access patterns are well-known and simple
 - Telecom Industry (storing billions of call detailed recording, providing real-time access to logs and billing information to customers)
 - Banking industry (to store, process and update vast volumes of data and perform analysis, including fraud detection)

Additional Resources

- <http://hbase.apache.org/>



MapReduce and YARN

Agenda

- MapReduce
- Yarn

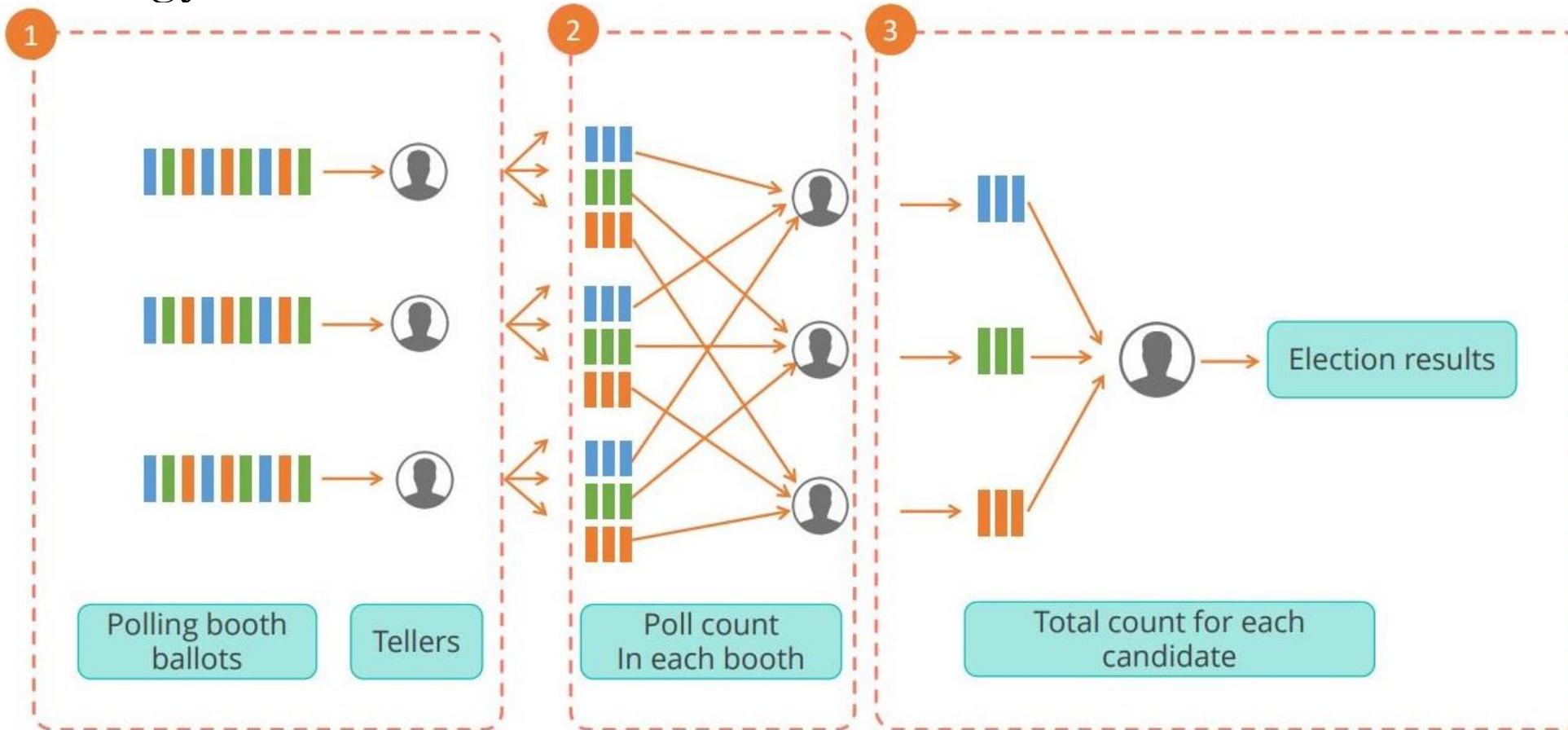
MapReduce

What is MapReduce

- MapReduce is a programming model that processes and analyzes huge data sets logically into separate clusters.
- While Map sorts the data, Reduce segregates it into logical clusters, thus removing ‘bad’ data and retaining the necessary information

MapReduce Analogy

- The MapReduce steps are illustrated using manual vote counting after an election as an analogy:



Map Execution

Map phase

Reads assigned input split from HDFS

Parses input into records as key-value pairs

Applies map function to each record

Informs master node of its completion

Partition phase

Each mapper must determine which reducer will receive each of the outputs

For any key, the destination partition is the same

Number of partition = Number of reducers

Shuffle phase

Fetches input data from all map tasks for the portion corresponding to the reduce task's bucket

Sort phase

Merge sort of all map occurs in a single run

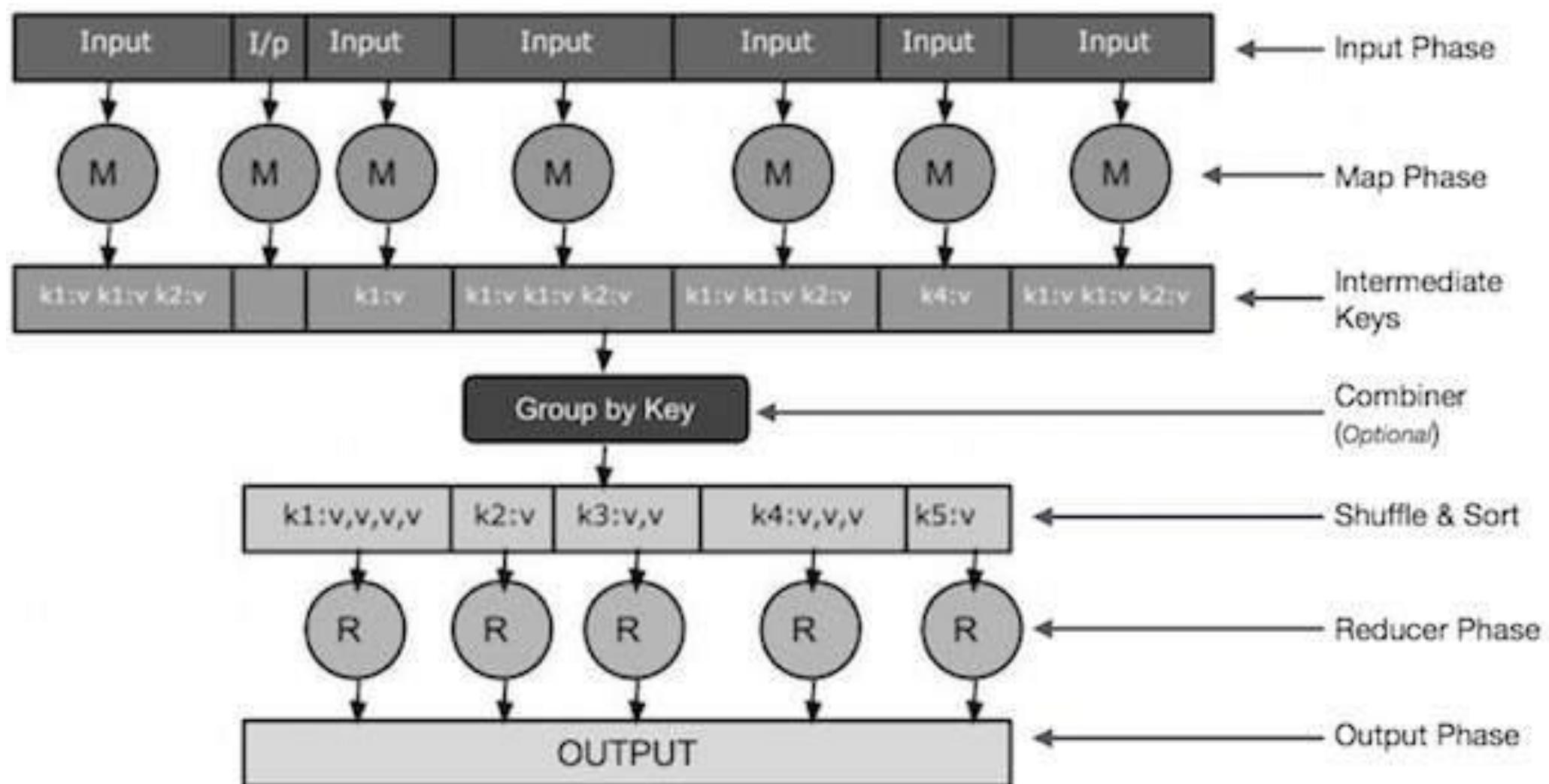
Reduce phase

Applies user-defined reduce function to the merged run

Arguments: key and corresponding list of values

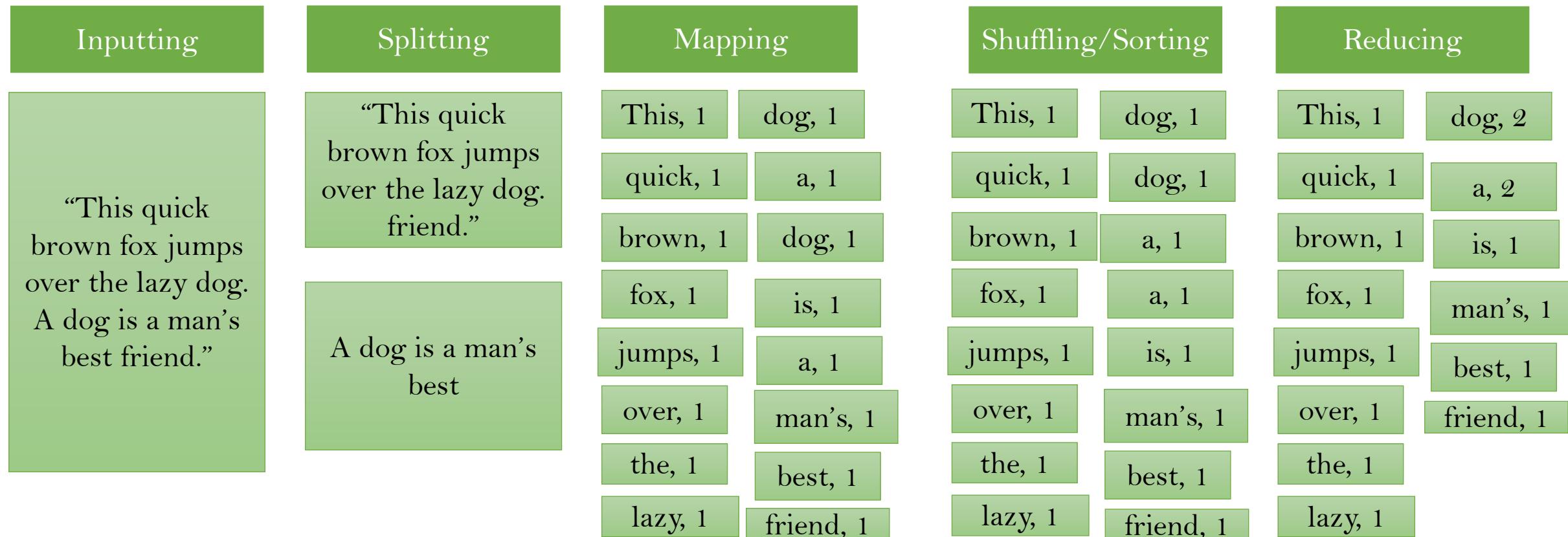
Writes output to a file in HDFS

Map Execution

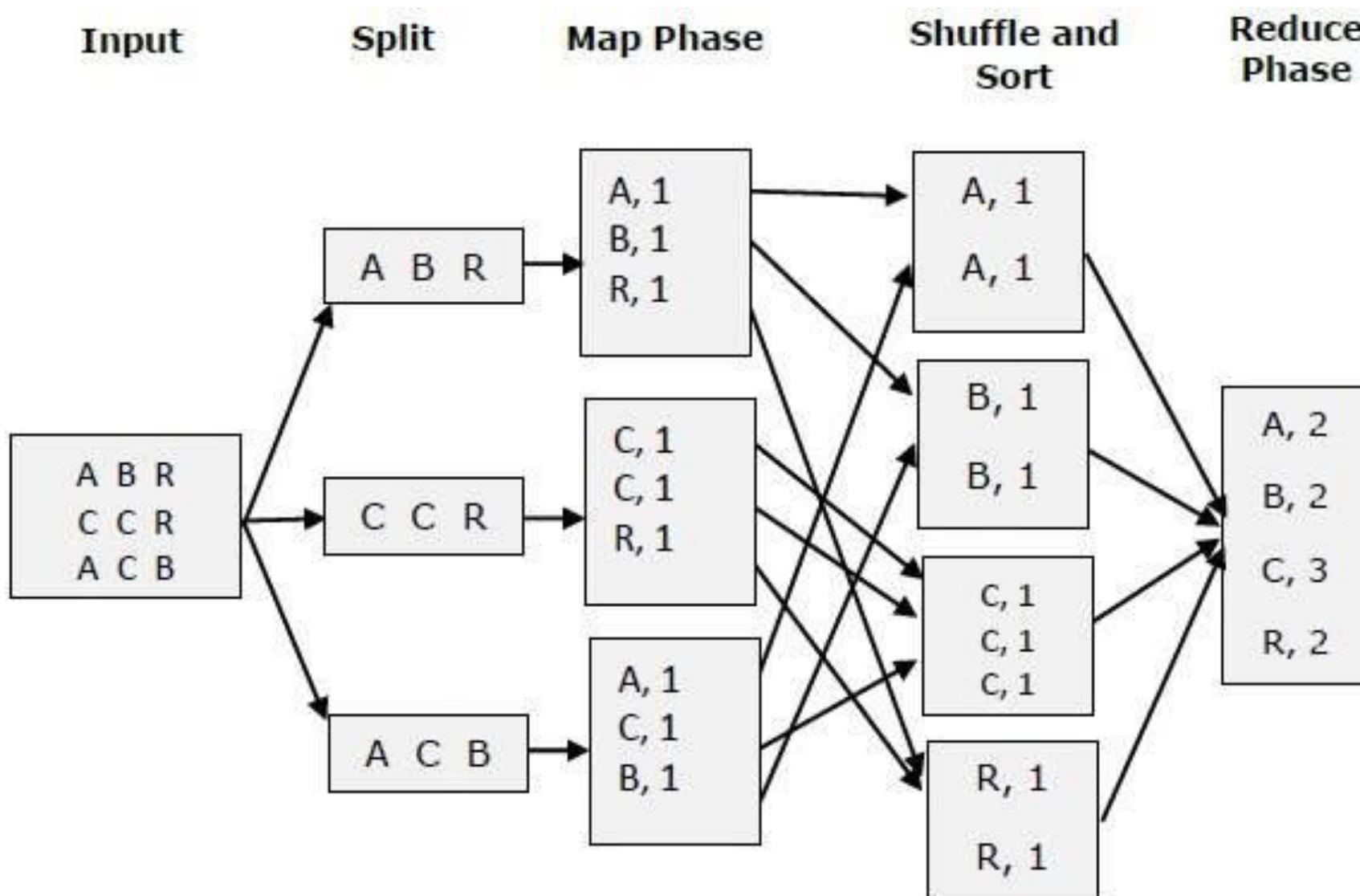


MapReduce example

- Count the number of words: “This quick brown fox jumps over the lazy dog. A dog is a man’s best friend.”

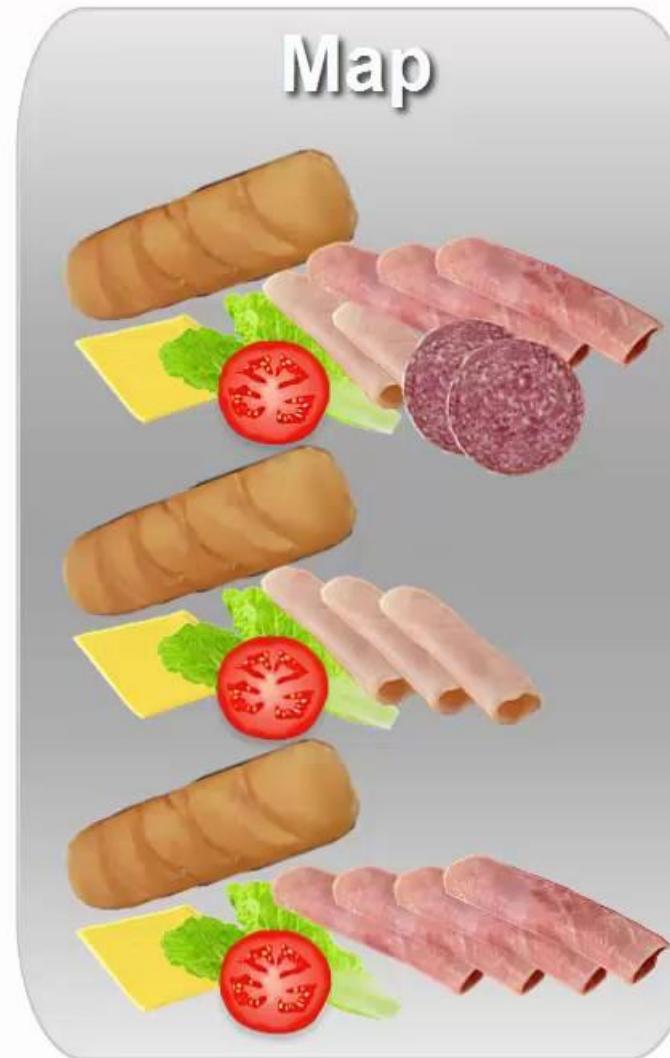


MapReduce example

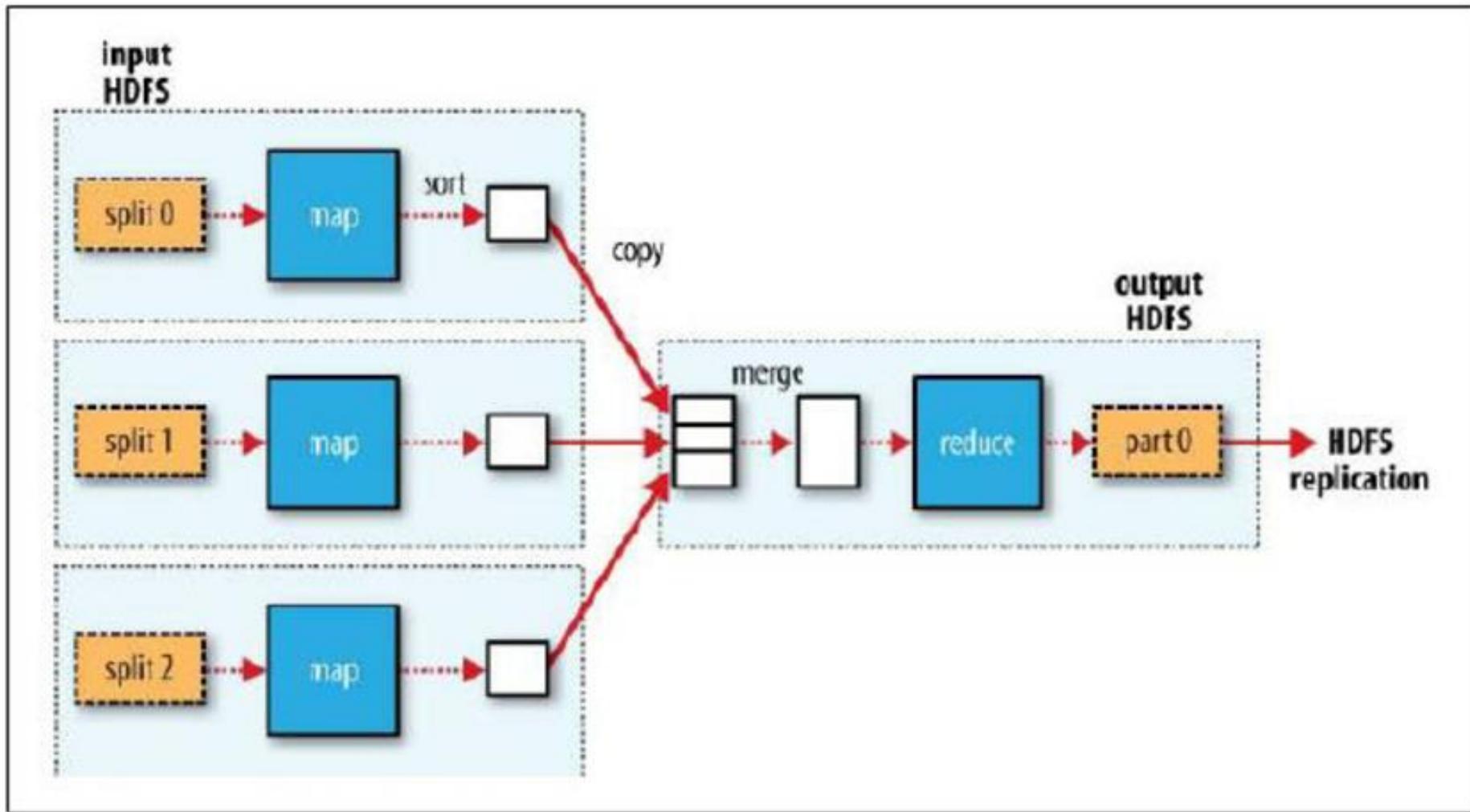




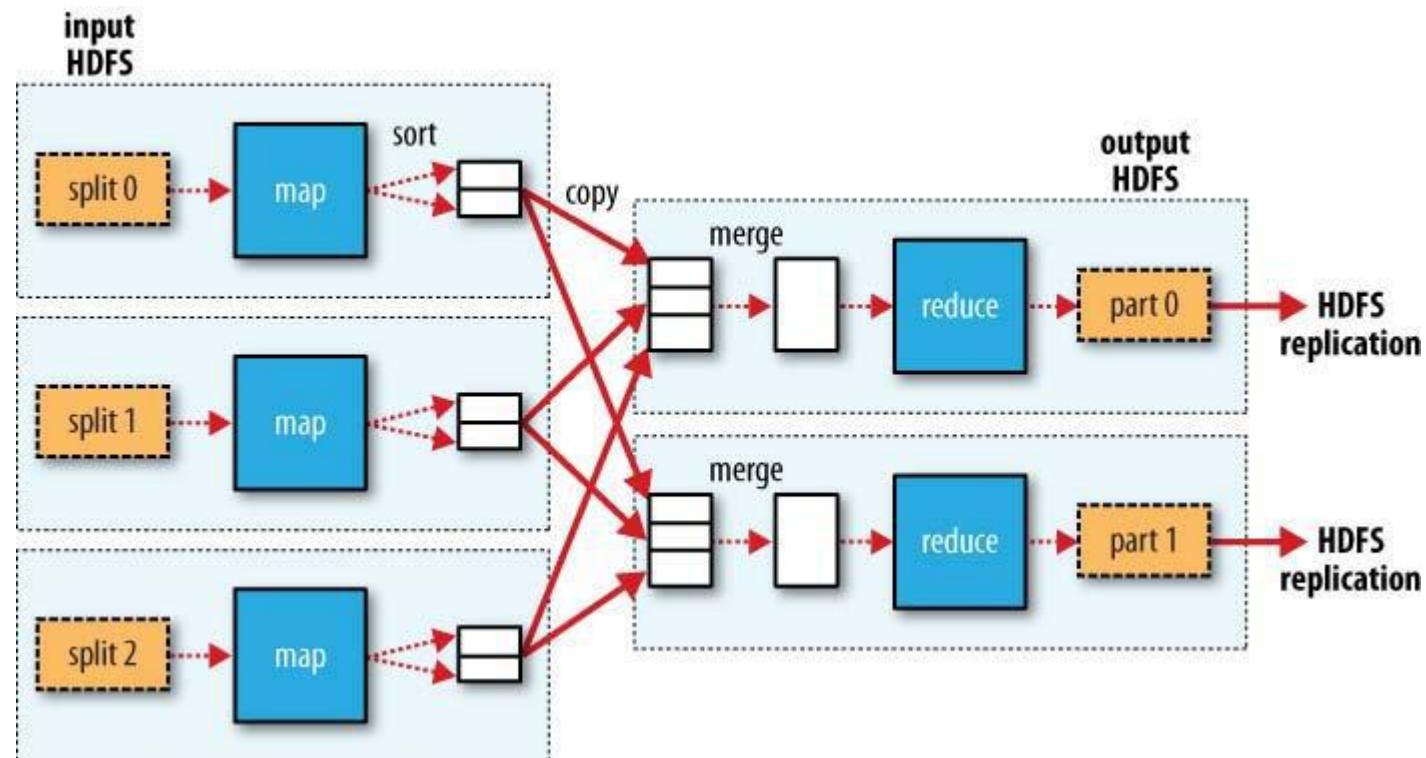
IBM Cloudant®



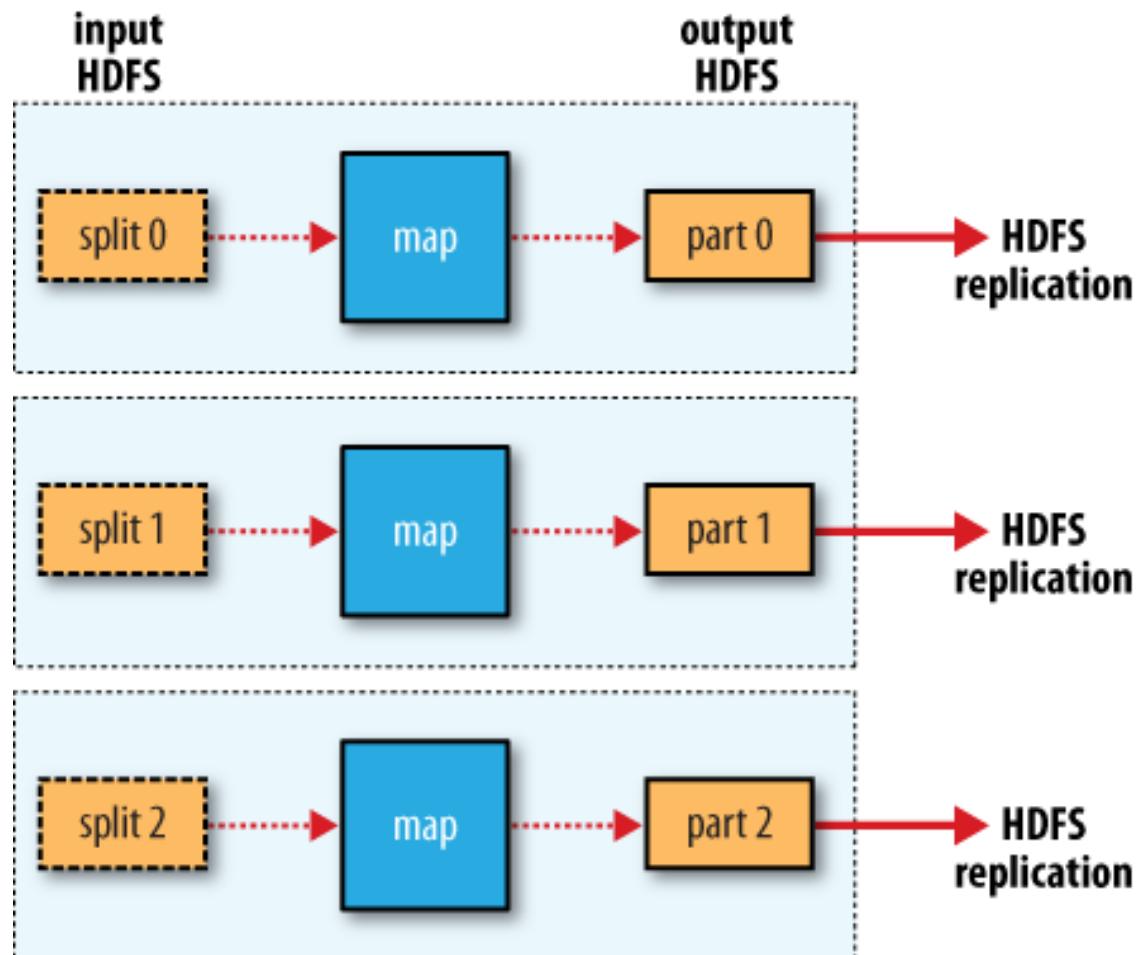
MapReduce with single reduce task



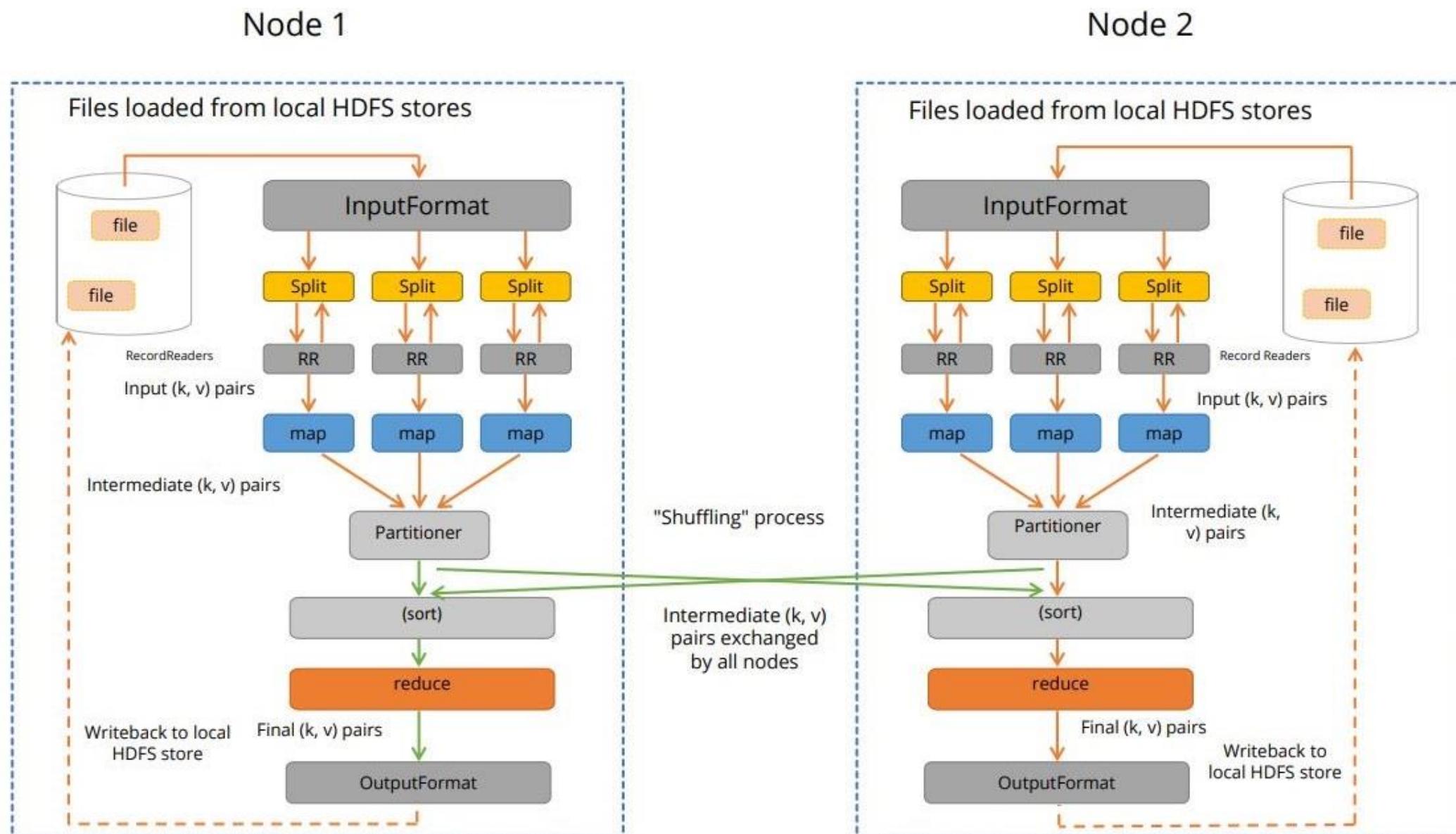
MapReduce with multiple reduce tasks



MapReduce with no reduce task



Map Execution: Distributed Two Node Environment



MapReduce Essentials

- The job input is specified in **key-value** pairs. Each job consists of two stages.
 - 1) A user defined map function is applied to each input record to produce a list of intermediate key-value pairs
 - 2) A user defined reduce function is called once for each distinct key in the map output. Then the list of intermediate values associated with that key is passed.

MapReduce essentials

- The number of reduce tasks can be defined by the users.
- Each reduce task is assigned a set of record groups which are intermediate records corresponding to a group of keys.
- For each group, a user-defined reduce function is applied to the recorded values.
- The reduce tasks read from every map task, and each read returns the record groups for that reduce task.
- Note that the reduce phase cannot start until all mappers have finished processing.

MapReduce Jobs

- A job is a full MapReduce program which typically causes multiple maps and reduces functions to be run in parallel over the life of the program. Many copies of the map and reduce functions are forked for parallel processing across the input dataset.
- A task is a map or reduces function executed on a subset of data. With this understanding of “job”, and “task”, the ApplicationMaster and NodeManager functions become easy to comprehend.

MapReduce Jobs

- Application Master (YARN)
 - The Application Master is responsible for the execution of a single application or MapReduce job.
 - It divides the job requests into tasks and assigns those tasks to Node Managers running on the slave node.
- Node Manager (YARN)
 - The Node Manager has a number of dynamically created resource containers. The size of a container depends on the number of resources it contains, such as memory, CPU, disk, and network IO.
 - It executes map and reduces tasks by launching these containers when instructed by the MapReduce Application Master.

Characteristics of MapReduce

- MapReduce is designed to handle very large scale data in the range of petabytes and exabytes.
- It works well to write once and read many data, also known as WORM data.
- MapReduce allows parallelism without mutexes.
- The Map and Reduce operations are performed by the same processor.
- Operations are provisioned near the data as data locality is preferred.
- Commodity hardware and storage is leveraged in MapReduce.
- The runtime takes care of splitting and moving data for operations.

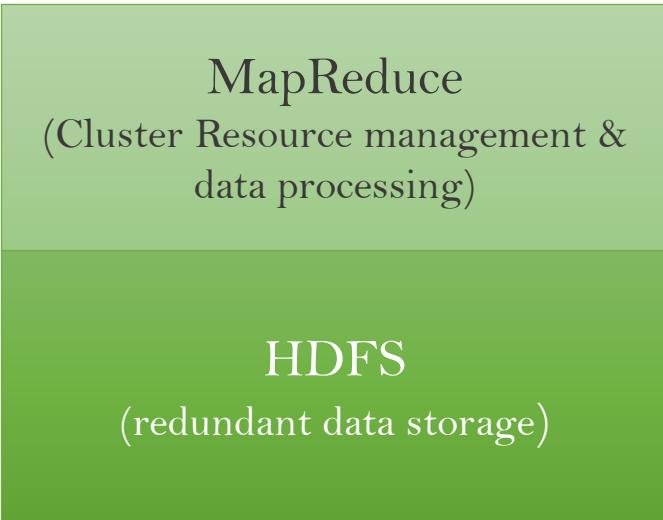
Job Work Interaction

Job Submission	Distribution of input split	Coordination with Node Manager	Resubmission of tasks	Success or failure status
Hadoop MapReduce job is submitted by a client in the form of an input file or a number of input split of files containing data	MapReduce Application Master distributes the input split to separate Node Managers	MapReduce Application Master coordinates with the Node Managers	MapReduce Application resubmits the task(s) to an alternate Node Manager if the Data Node fails	Resource Manager gathers the final output and informs the client of the success or failure status

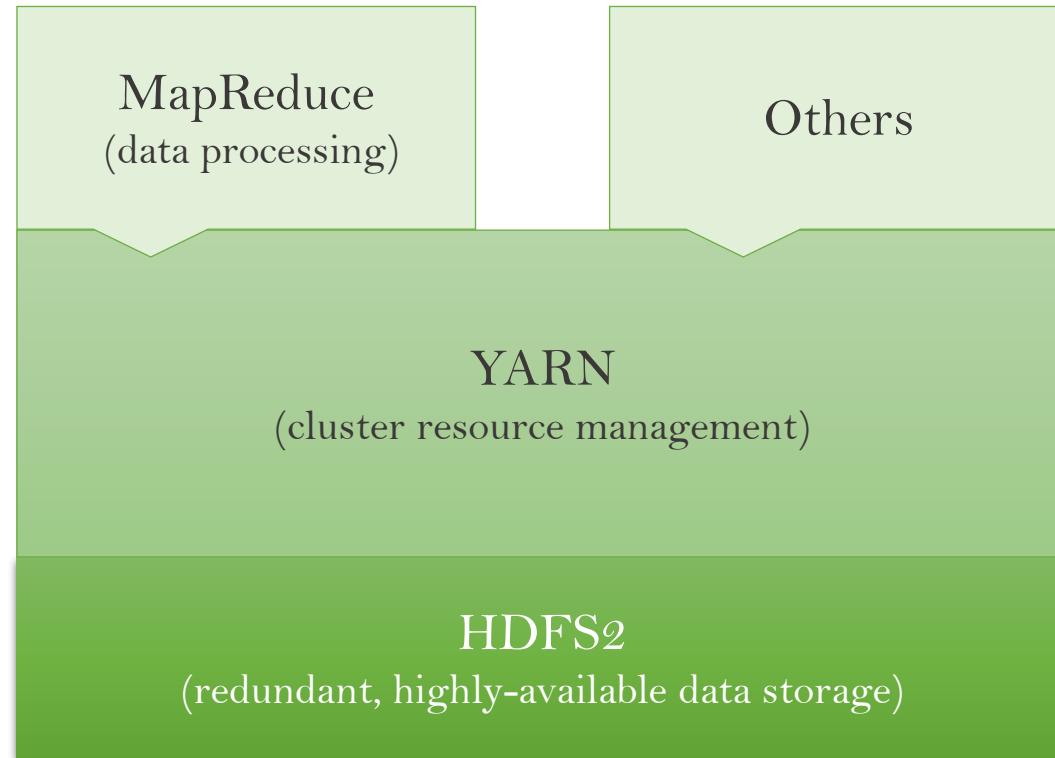
YARN

Hadoop 1.x vs Hadoop 2.x

Hadoop 1.x

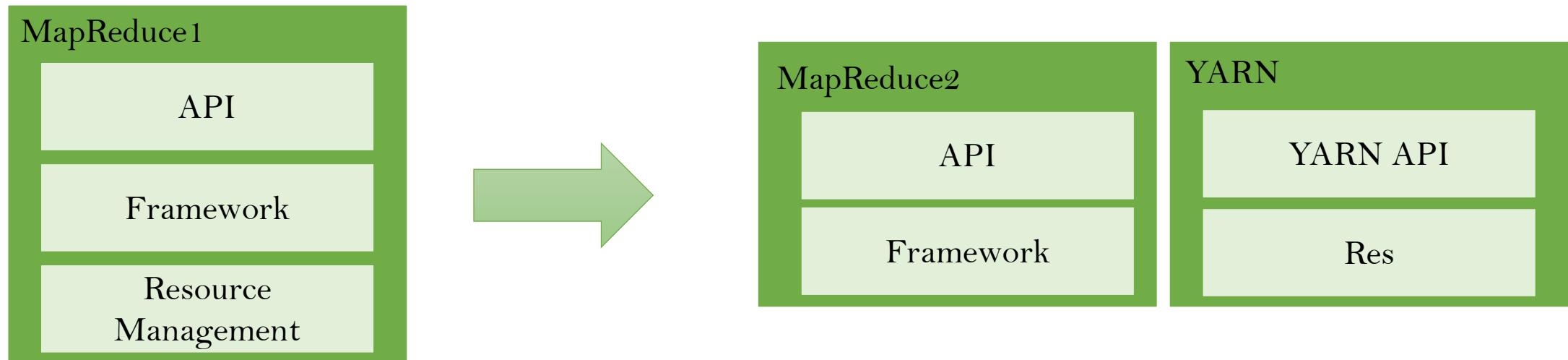


Hadoop 2.x



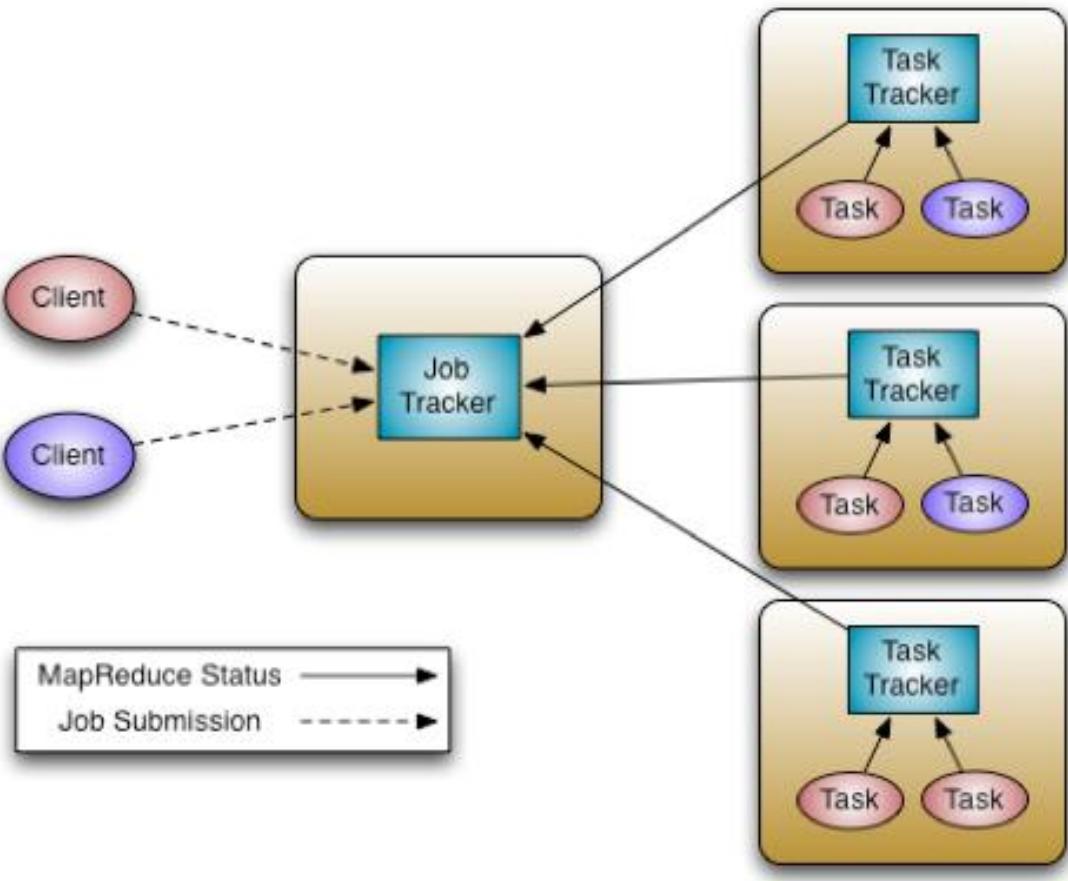
What is YARN

- YARN stands for “Yet Another Resource Negotiator”
- YARN provides its core services via two types of long running daemon: a resource manager (one per cluster) to manage the use of resources and node managers running on all the nodes in the cluster to launch and monitor containers
- Container executes an application specific process with a constrained set of resources (memory, CPU, hd)



Map Reduce 1 Execution Framework

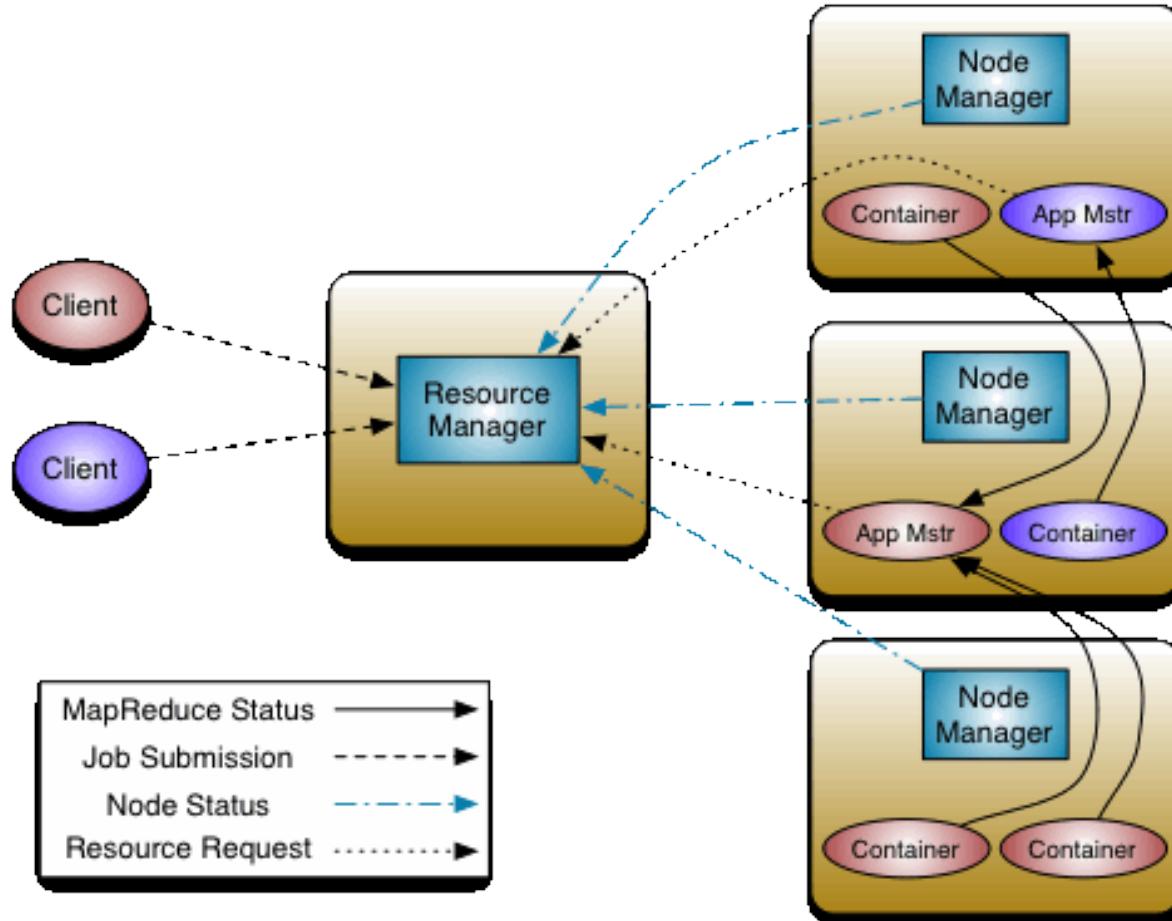
1. Job Tracker is a Master daemon
2. Responsible to assign and track task execution progress
3. Task Trackers are slave daemons
4. They run on systems where data nodes reside
5. Responsible to spawn a child jvm to execute Map, Reduce and intermediate tasks



Motivation for YARN

- Scalability
 - YARN can run on larger cluster than MapReduce 1. MapReduce 1 hits scalability bottlenecks in the region of 4000 nodes and 40000 tasks, stemming from the fact that the job-tracker has to manage both jobs and tasks.
- Utilization
 - In MapReduce 1 each task-tracker is configured with a static allocation of fixed size slots, which are divided into map slots and reduce slots at configuration time. A map slot can only be used to run a map task, reduce slot can only be used to run reduce tasks.
- Multitenancy
 - Other types of distributed applications beyond MapReduce are supported

YARN Architecture



Job Tracker 1.0 responsibility is now split:

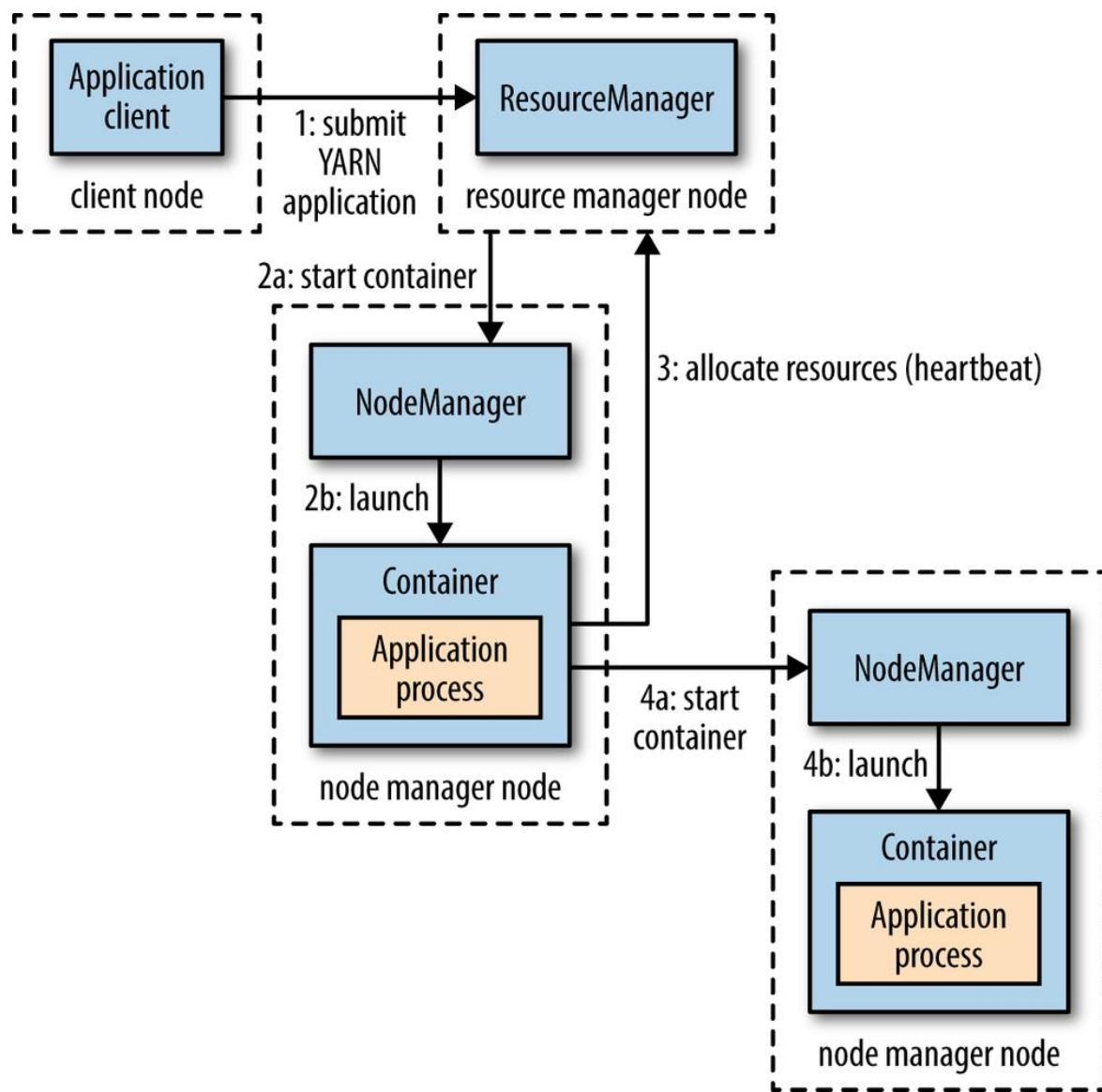
- Resource Manager manages the resources allocation in the cluster
- Application Master manages resources needed for individual applications

Node Manager is a generalized task tracker

A container executes an application specific process

How YARN runs an application

1. Client contacts the resource manager and asks it to run an application master process (1)
2. The resource manager then finds a node manager that can launch the application master in a container (2)
3. Depending on the application, application master can simply run a computation in the container it is running in and return the result to client or it can request more containers from Resource Manager (3) and run a distributed computation (4)

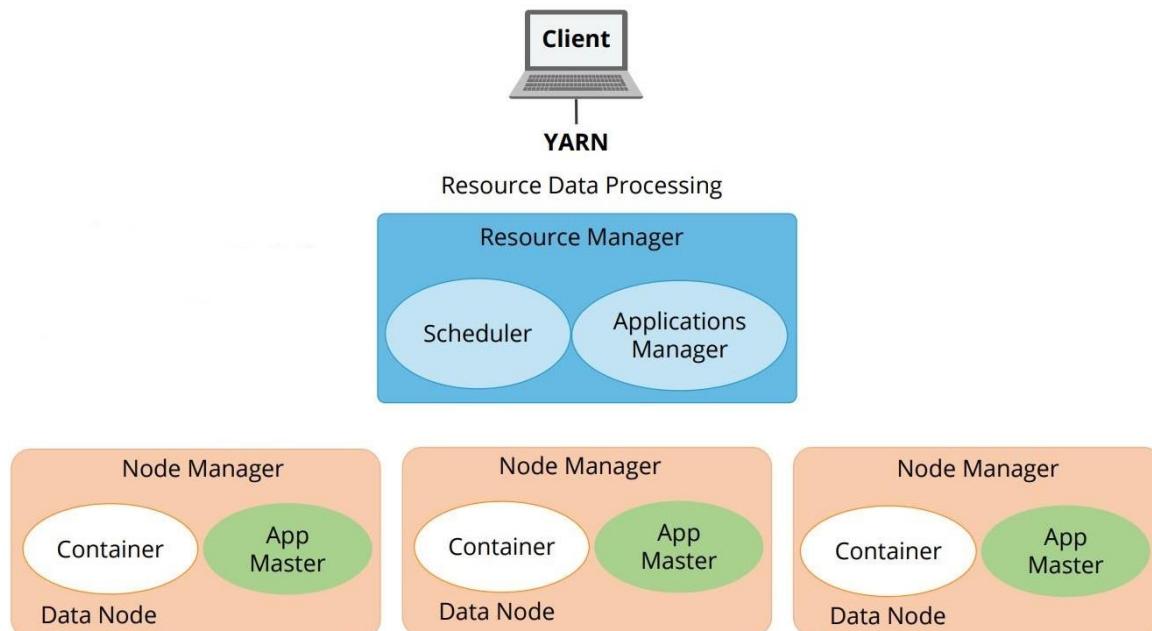


YARN Components

1. Client: To submit Map Reduce jobs
2. Resource Manager: To manage the use of resources across the cluster
3. Container: Name given to a package of resources including RAM, CPU, Network, HDD etc.
4. Node Manager: to oversee the containers running on the cluster nodes
5. Application Master: which negotiates with the Resource Manager for resources and runs the application-specific process (Map or Reduce tasks) in those clusters

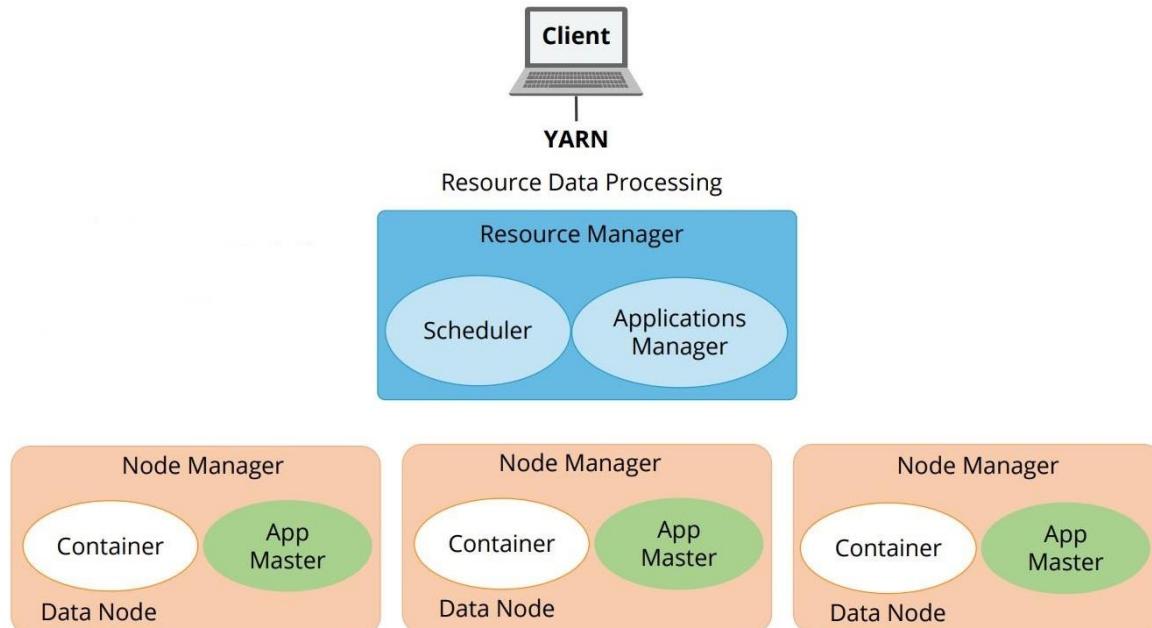
YARN: Resource Manager

- The Resource Manager, which is usually one per cluster, is the master server. Resource Manager knows the location of the Data Node and how many resources they have. This information is referred to as Rack Awareness.
- The Resource Manager runs several services, the most important of which is the Resource Scheduler that decides how to assign the resources.



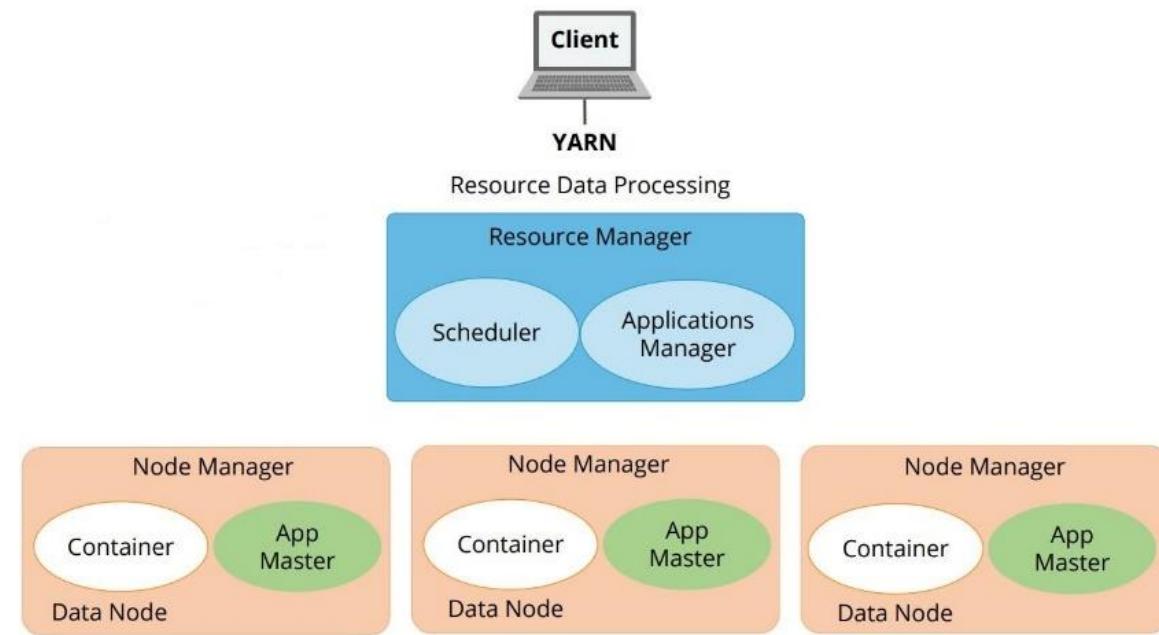
YARN: Application Master

- The Application Master is a framework-specific process that negotiates resources for a single application, that is, a single job or a directed acyclic graph of jobs, which runs in the first container allocated for the purpose.
- Each Application Master requests resources from the Resource Manager and then works with containers provided by Node Managers.



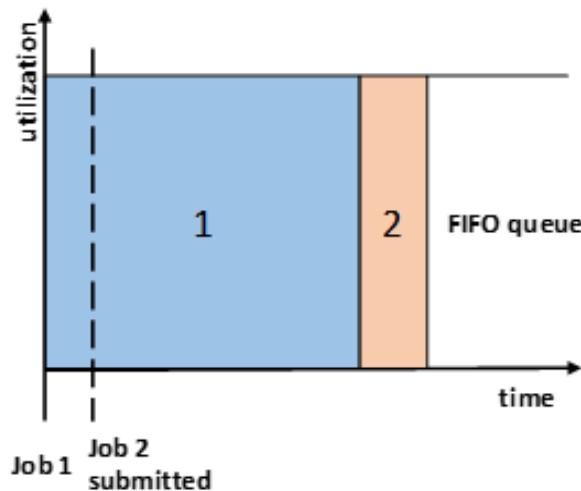
YARN: Node Manager

- The Node Managers can be many in one cluster. They are the slaves of the infrastructure. When it starts, it announces itself to the RM and periodically sends a heartbeat to the RM.
- Each Node Manager offers resources to the cluster. The resource capacity is the amount of memory and the number of v-cores, short for the virtual core. At run-time, the Resource Scheduler decides how to use this capacity.
- A container is a fraction of the Node Manager capacity, and it is used by the client to run a program. Each Node Manager takes instructions from the Resource Manager and reports and handles containers on a single node.



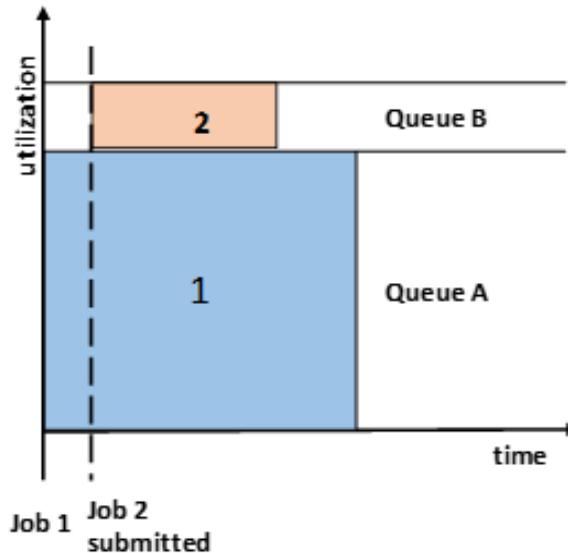
Scheduling in YARN

FIFO: scheduler places applications in a queue and runs them in order of submission the small job is blocked until the large job completes



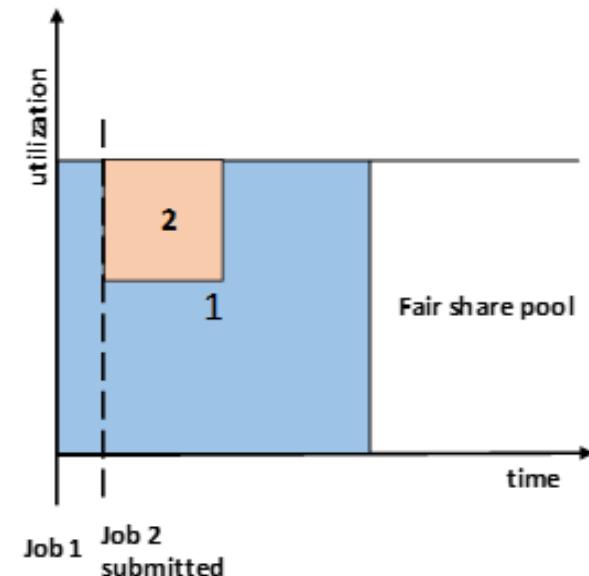
(a) FIFO scheduler

Capacity scheduler: a separate dedicated queue allows the small job to start as soon as it is submitted. This means that the large job finishes later than when using FIFO Scheduler



(b) Capacity Scheduler

Fair Scheduler: dynamically balance resources between all running jobs. Large job starts – it is the only job, so it gets all the resources in the cluster. When the second (small) job starts it is allocated half of the cluster resources so that each job is using its fair share of resources.



(c) Fair Scheduler

Figure 1: YARN Schedulers' cluster utilization vs. time

Reading

- Hadoop: The Definitive Guide, Chapter 2 Map Reduce, by Tom White
- Hadoop: The Definitive Guide, Chapter 4 Yarn, by Tom White
- MapReduce: Simplified Data Processing on Large Clusters, by Jeffrey Dean and Sanjay Ghemawat, Google, Inc.

Lecture 9

Hive

Lecture Plan

- What is HIVE
- Hive Components
- What is Hive Data Model?
- Underlying Hive Architecture
- Using Hive in practice

What is Hive

- Data warehouse infrastructure Build on top of Hadoop for querying and large data sets
- A system for managing and querying unstructured data as if it were structured
 - Stores schema in database
 - Uses Map-Reduce for execution
 - HDFS for storage

Why Hive?

- Hadoop is great!
- MapReduce is very low level
- Lack of expressiveness
- Higher level data processing language are needed

Hive Features

- Designed for On-Line Analytical Processing
- SQL type language for querying
- It is familiar, fast, scalable, and extensible

OLAP

Hive QL or HQL

Can plug in map/reduce scripts
in language of choice

Hive is NOT

- Relational database
- Database for On-Line Transactional Processing
- Language for real-time queries and row-level updates



Online transaction processing, or OLTP, is a class of information systems that facilitate and manage transaction-oriented applications, typically for data entry and retrieval transaction processing (Wikipedia)

History

- Early Hive development Work started at Facebook in 2007
- Hive is and Apache project under Hadoop



<https://hive.apache.org/>

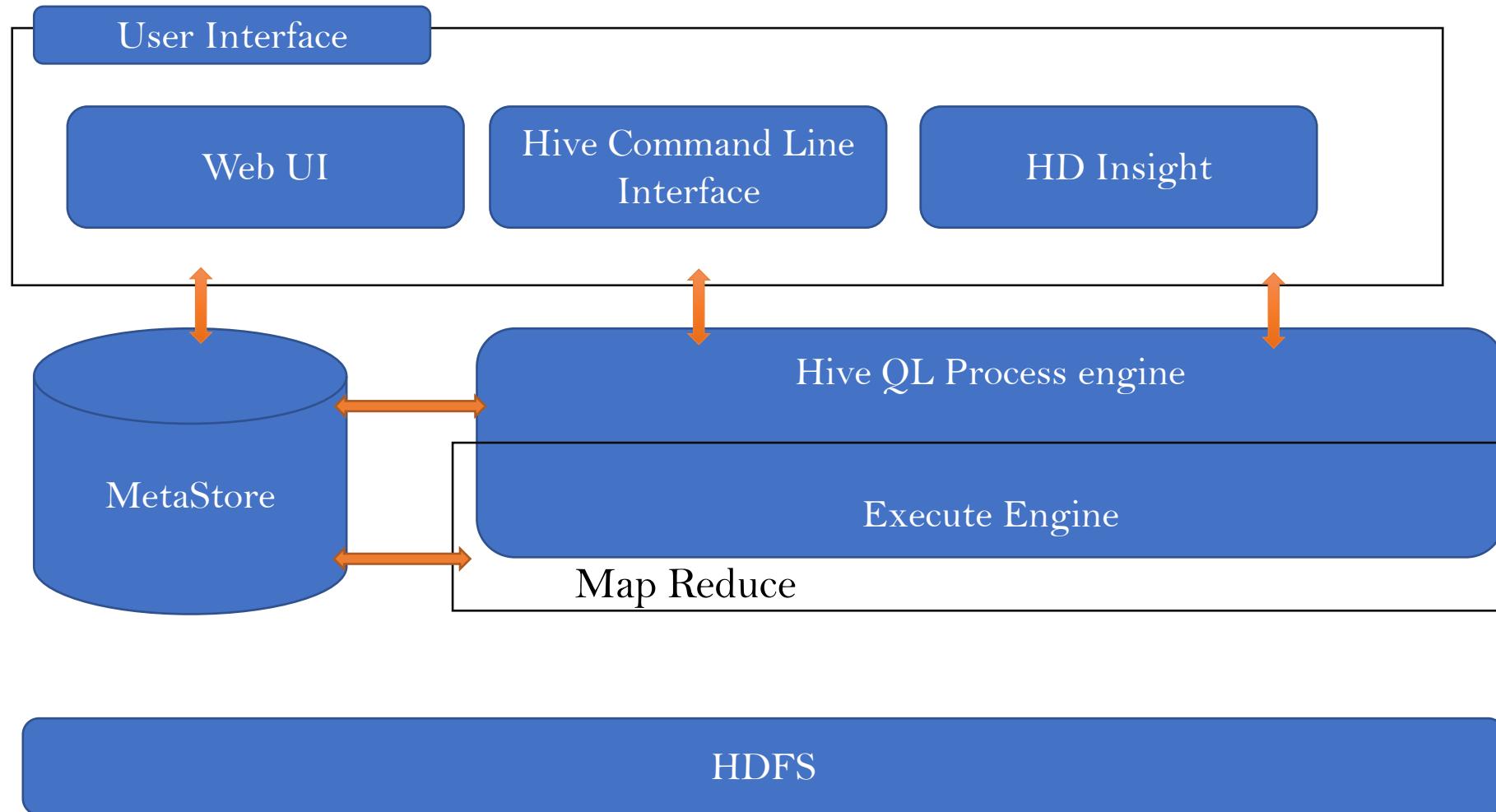
- Data Warehouse infrastructure for Hadoop
- Enables developers to utilize custom mappers and SQL-Like query
- Provides tools to enable ETL on large data
- Language reducers



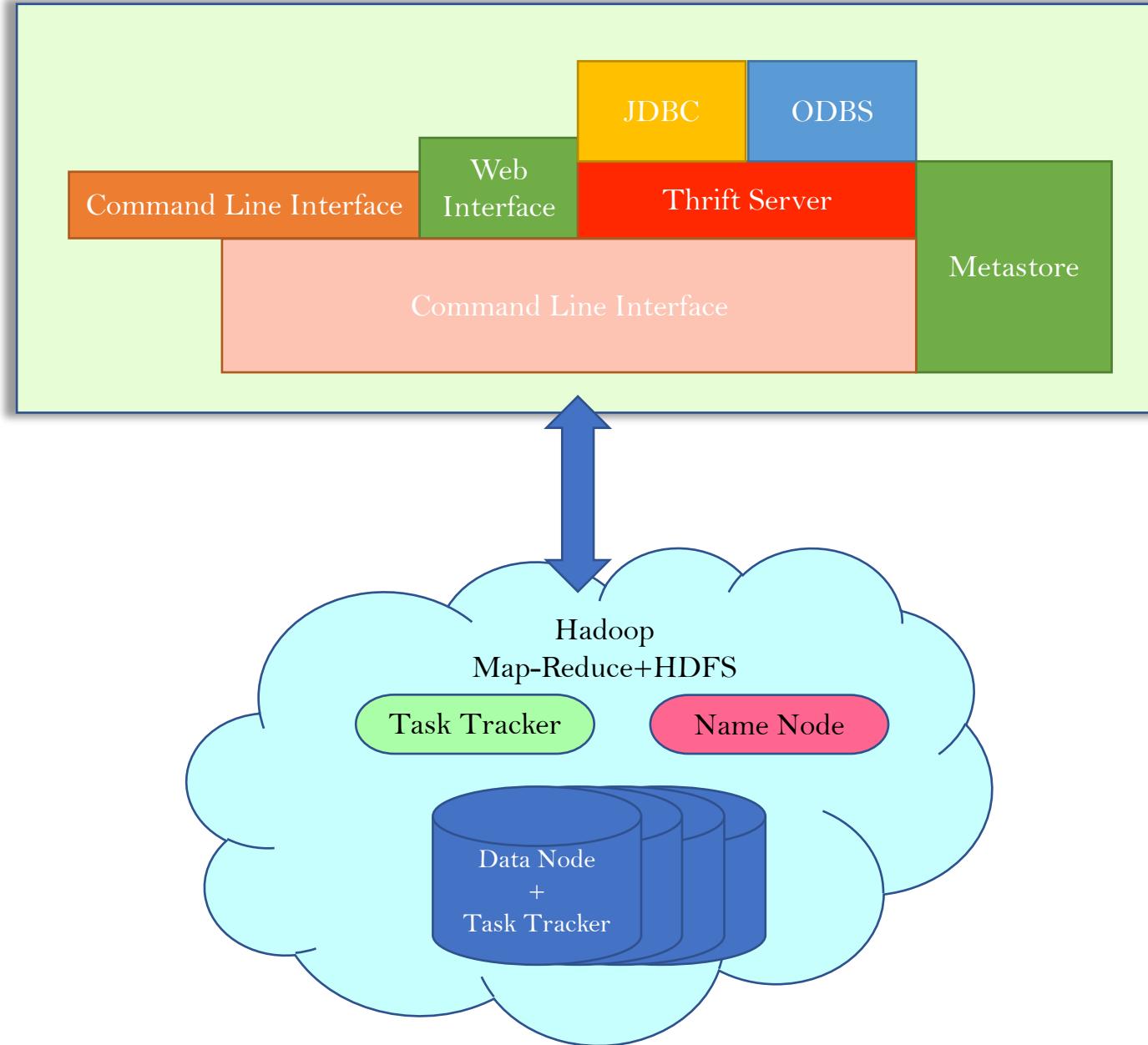
Hive Components

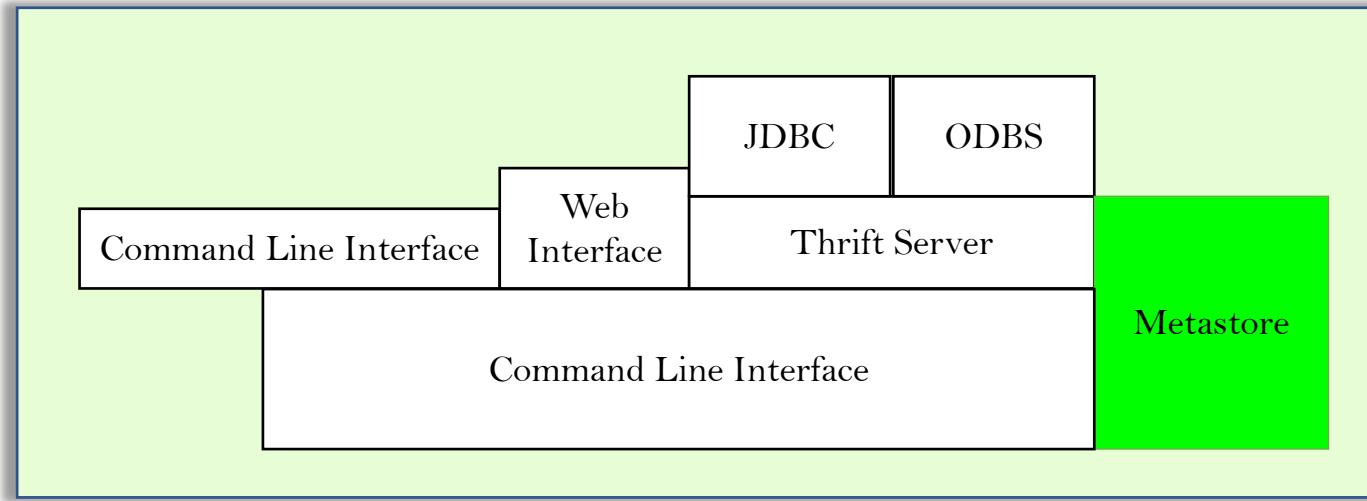
- Shell
- Driver
- MetaStore
- Compiler
- Execution Engine

Hive Architecture



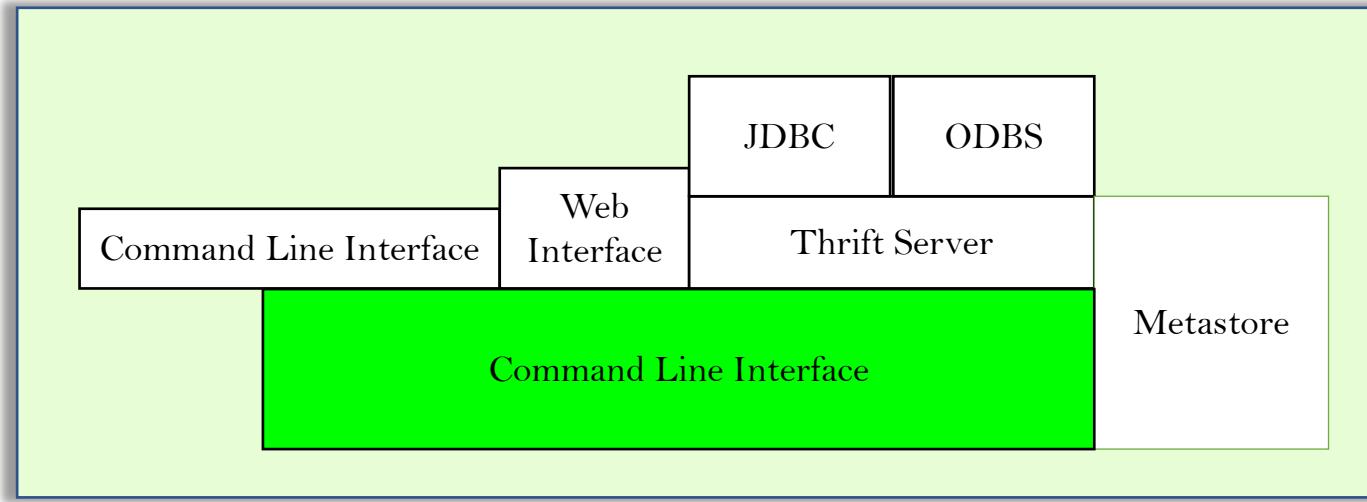
Hive Architecture and Components



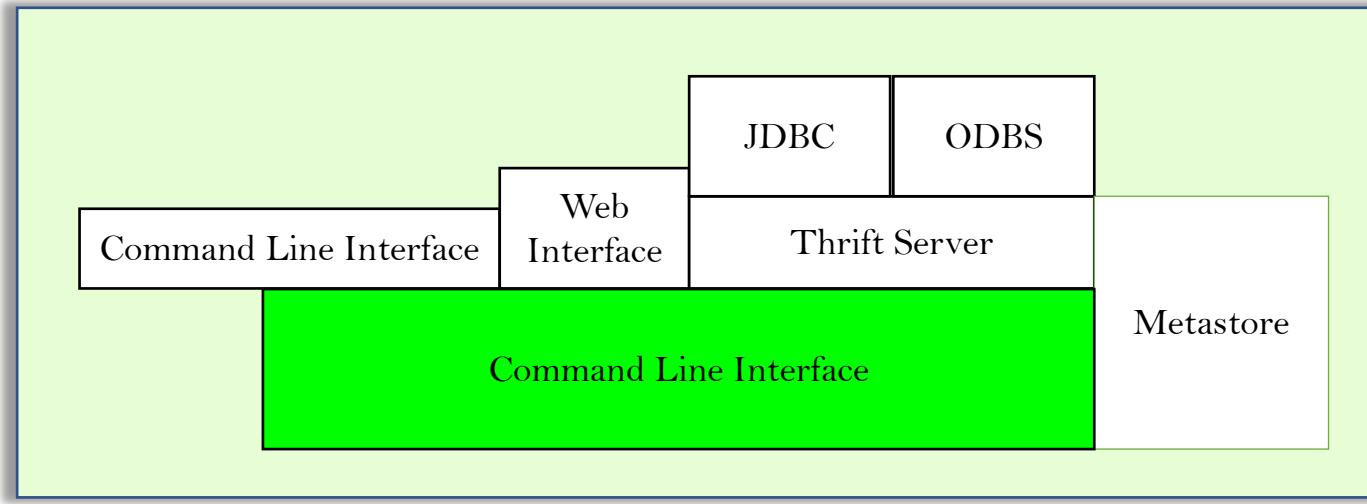


- Metastore

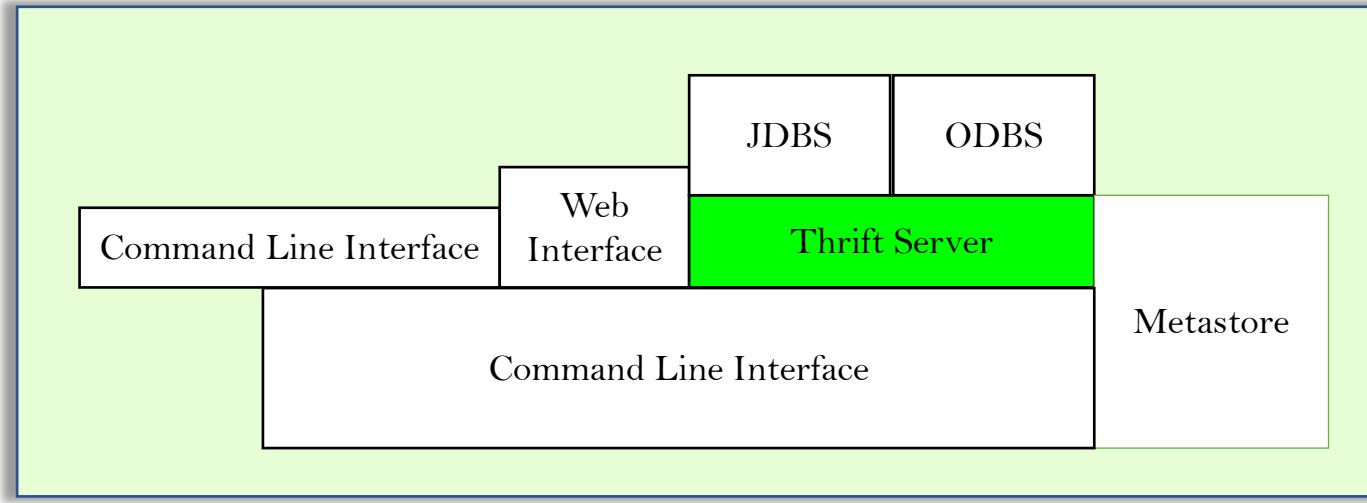
- Store the system catalog and meta data about tables, columns, partitions etc
- Stored on a traditional RDBMS



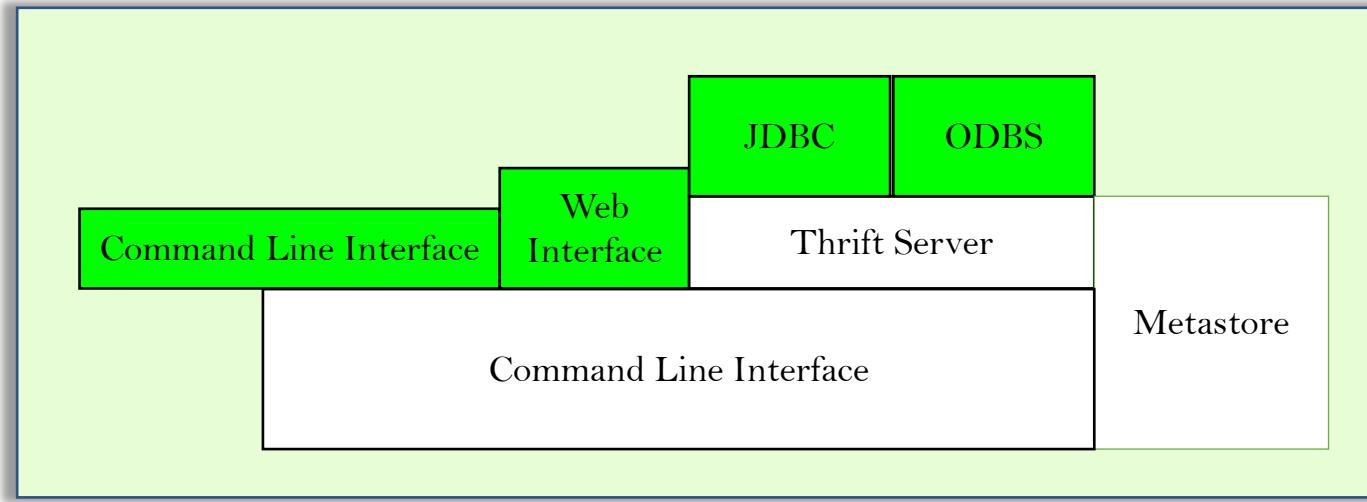
- Driver:
 - Manages the lifecycle of HiveQL statement
 - Maintains a session handle and any session statistics



- Compiler
 - The Component that compiles HiveQL into a directed acyclic graph of map/reduce
- Optimizer
 - Consists of a chain of transformations
 - Performs Column Pruning, Partition Pruning, Repartitioning of Data
- Executor
 - Executes the tasks produced by the compiler in proper dependency order
 - Interface with the underlying Hadoop instance

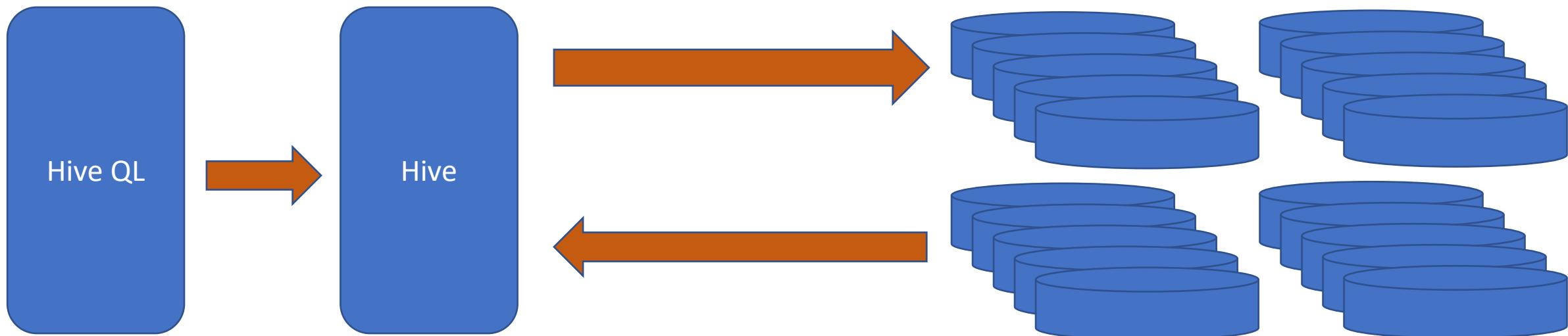


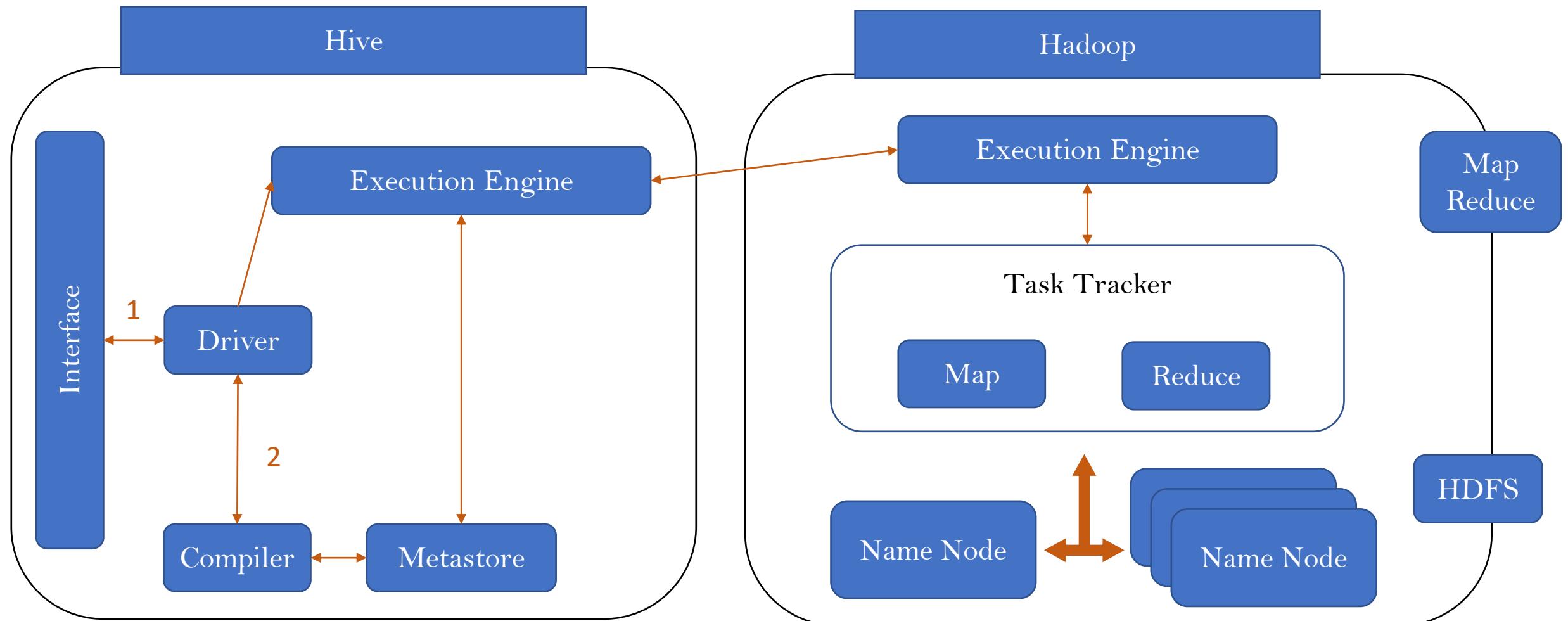
- Hive Server
 - Provides a Thrift interface and a JDBC/ODBC server enables Hive integration with other application

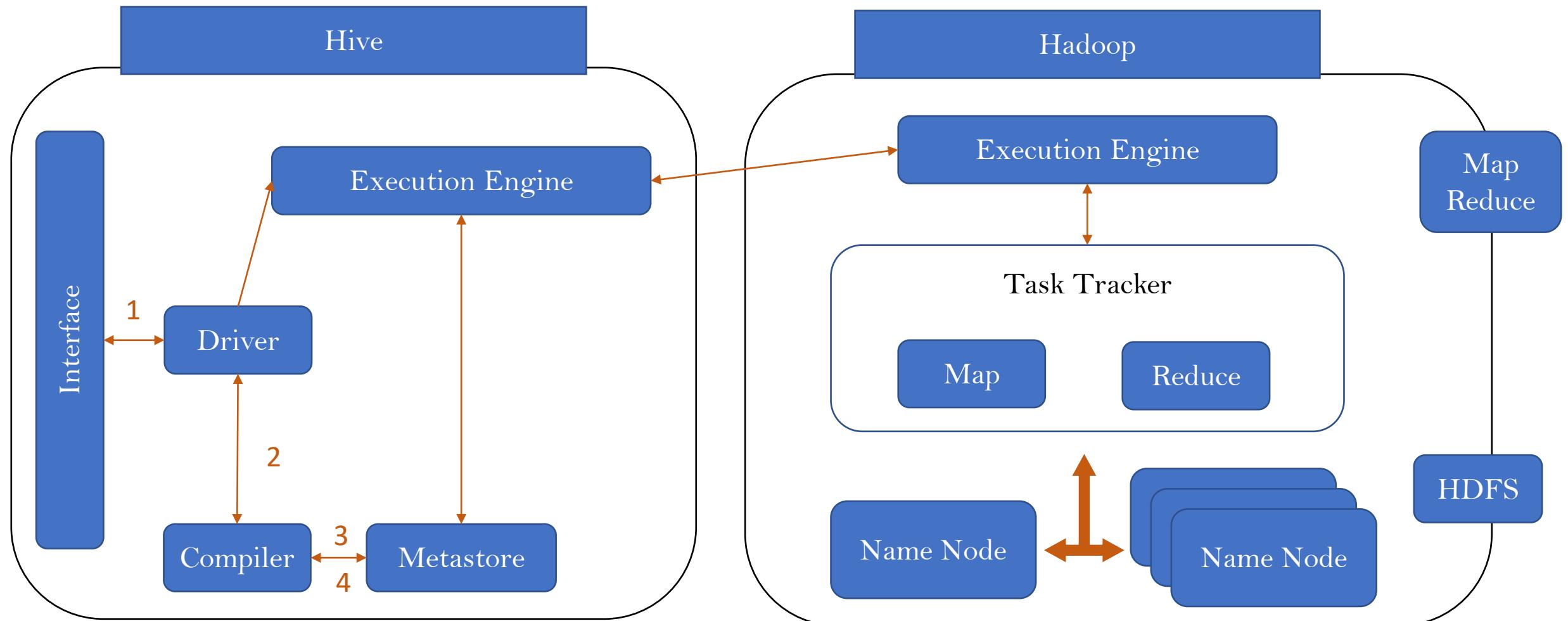


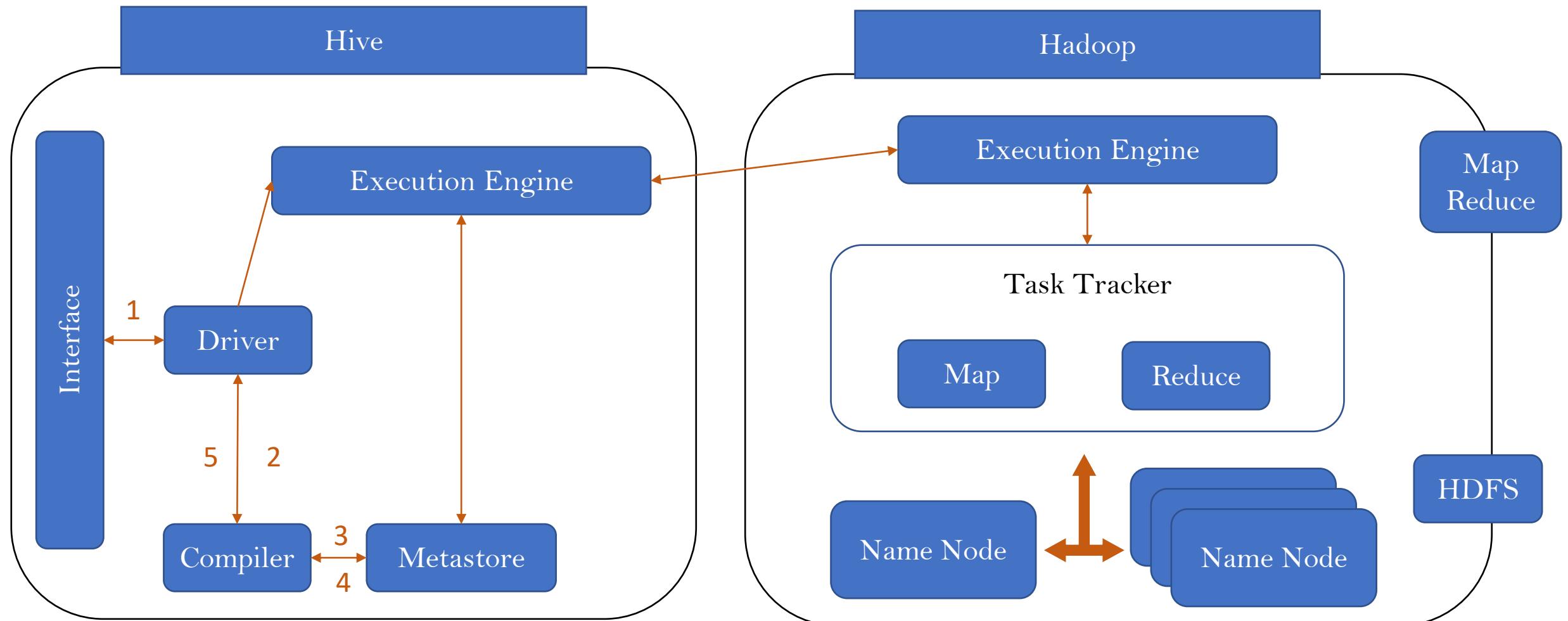
- Client Components
 - Command Line Interface
 - Web User Interface
 - JDBC
 - Java Data-Base Connectivity (driver)
 - ODBC driver
 - Open Data-Base Connectivity (driver)

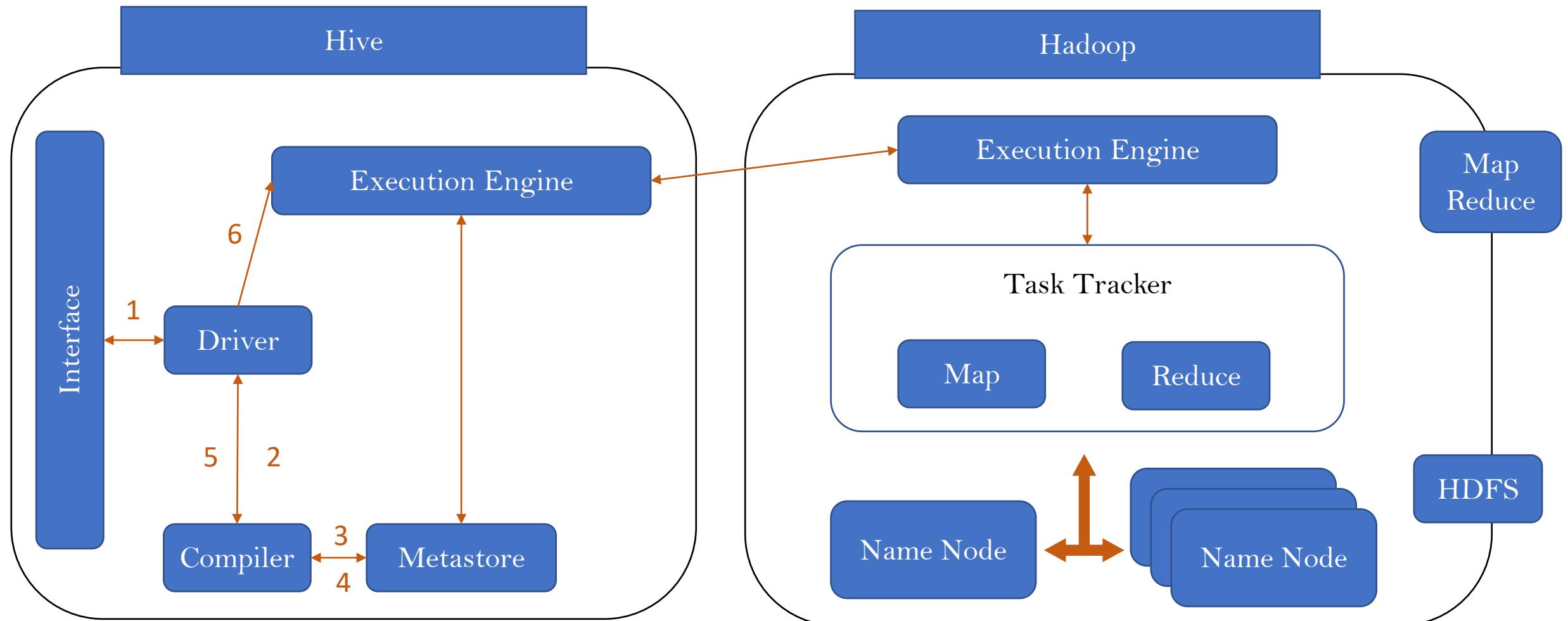
HDFS

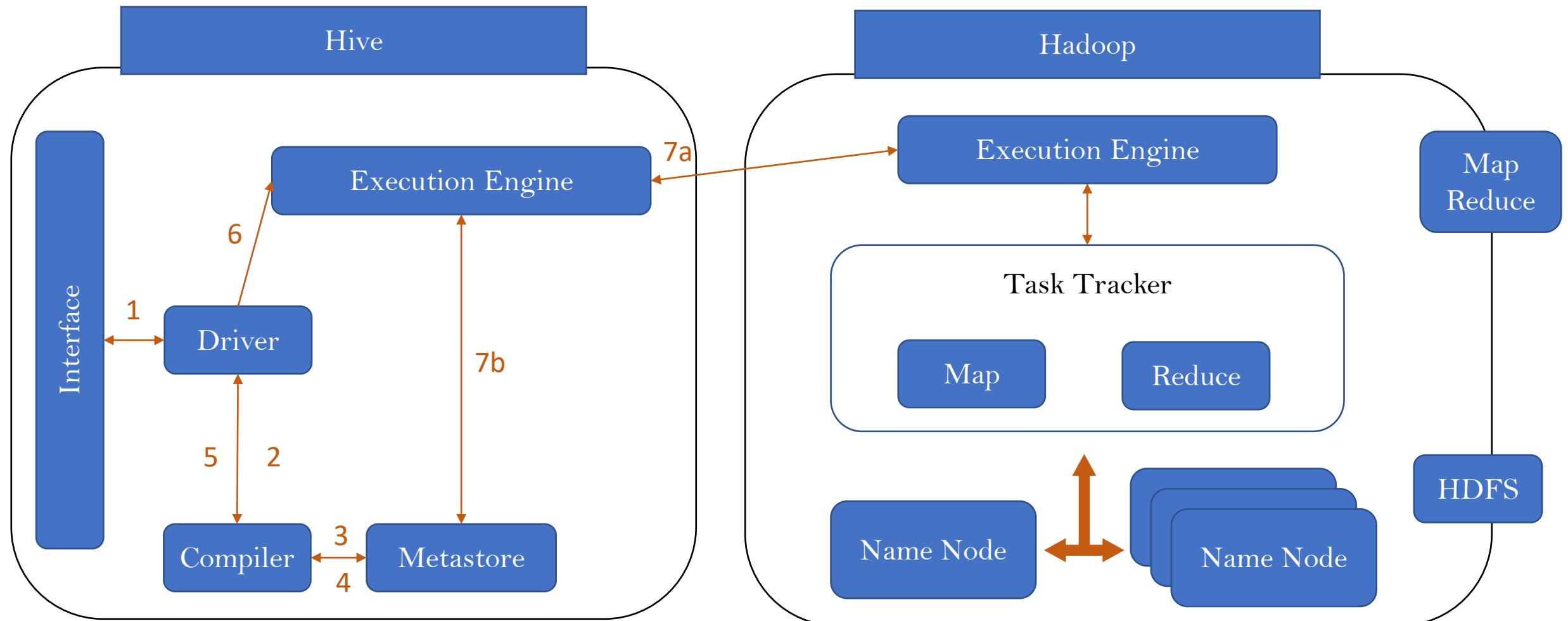


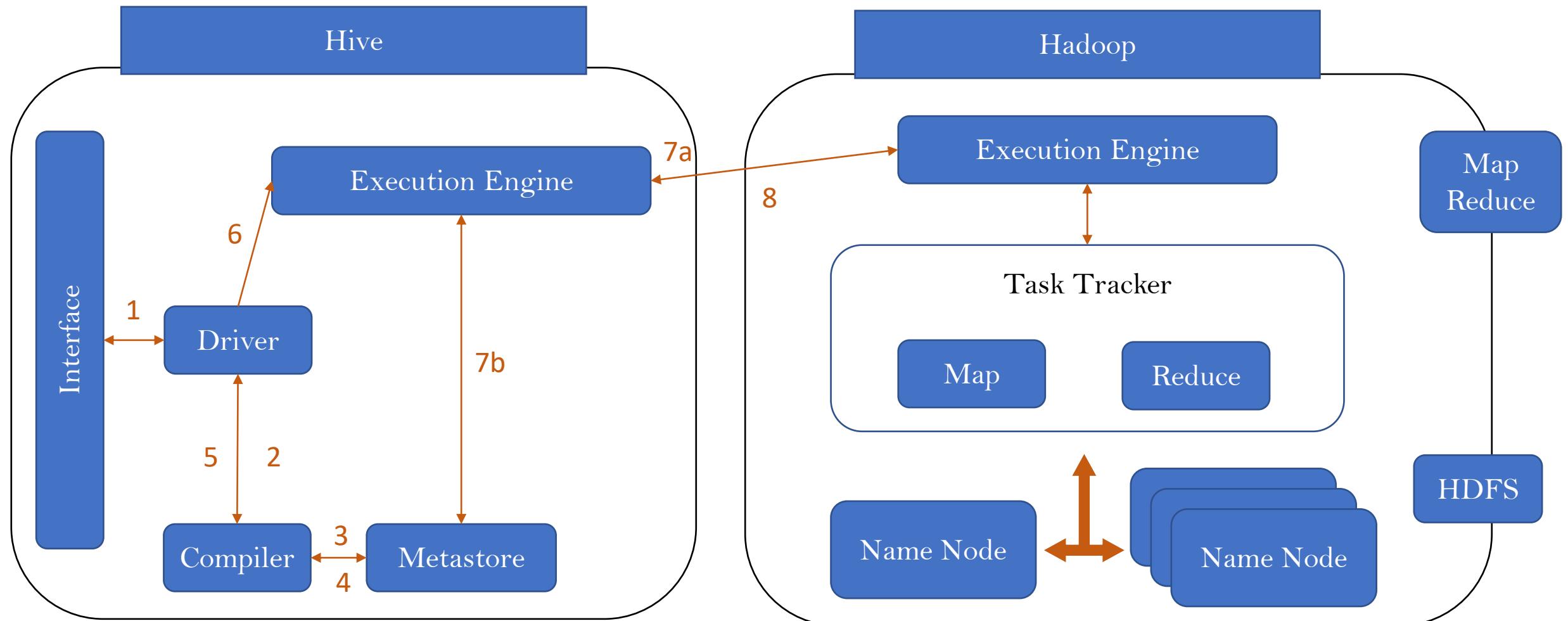


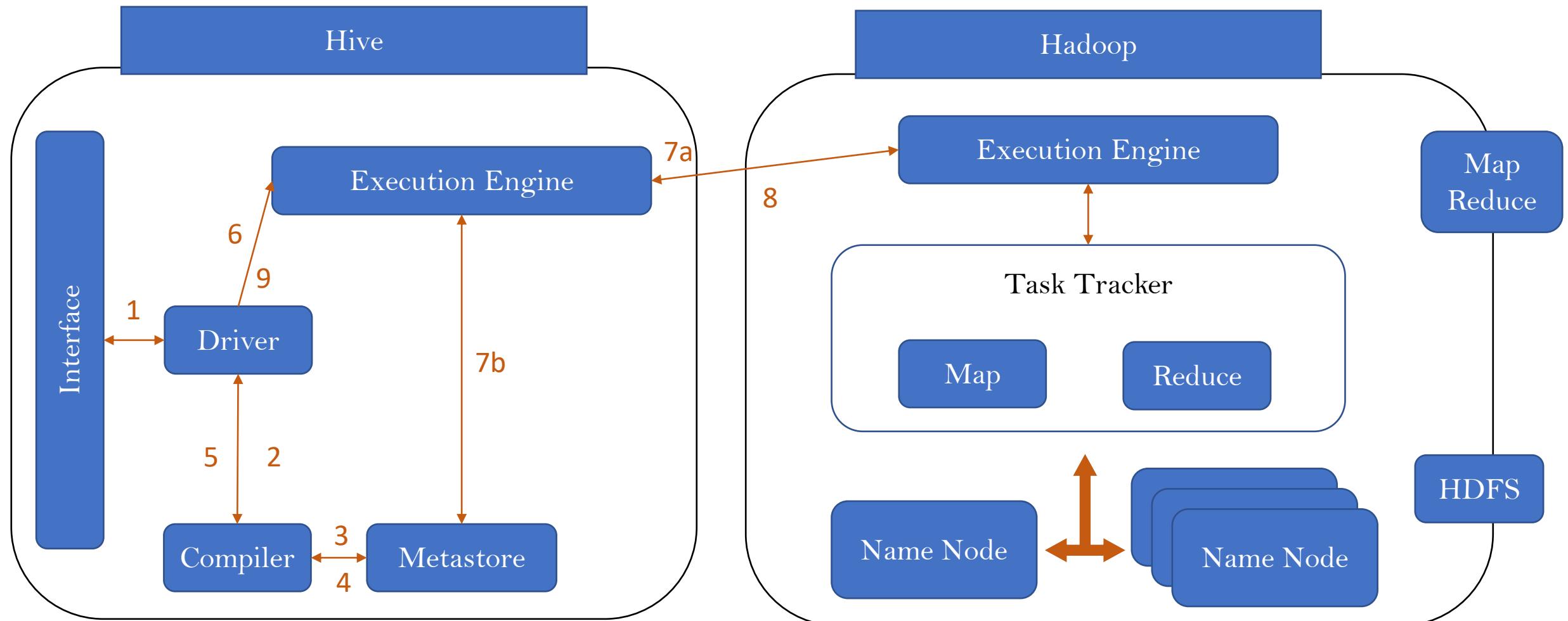


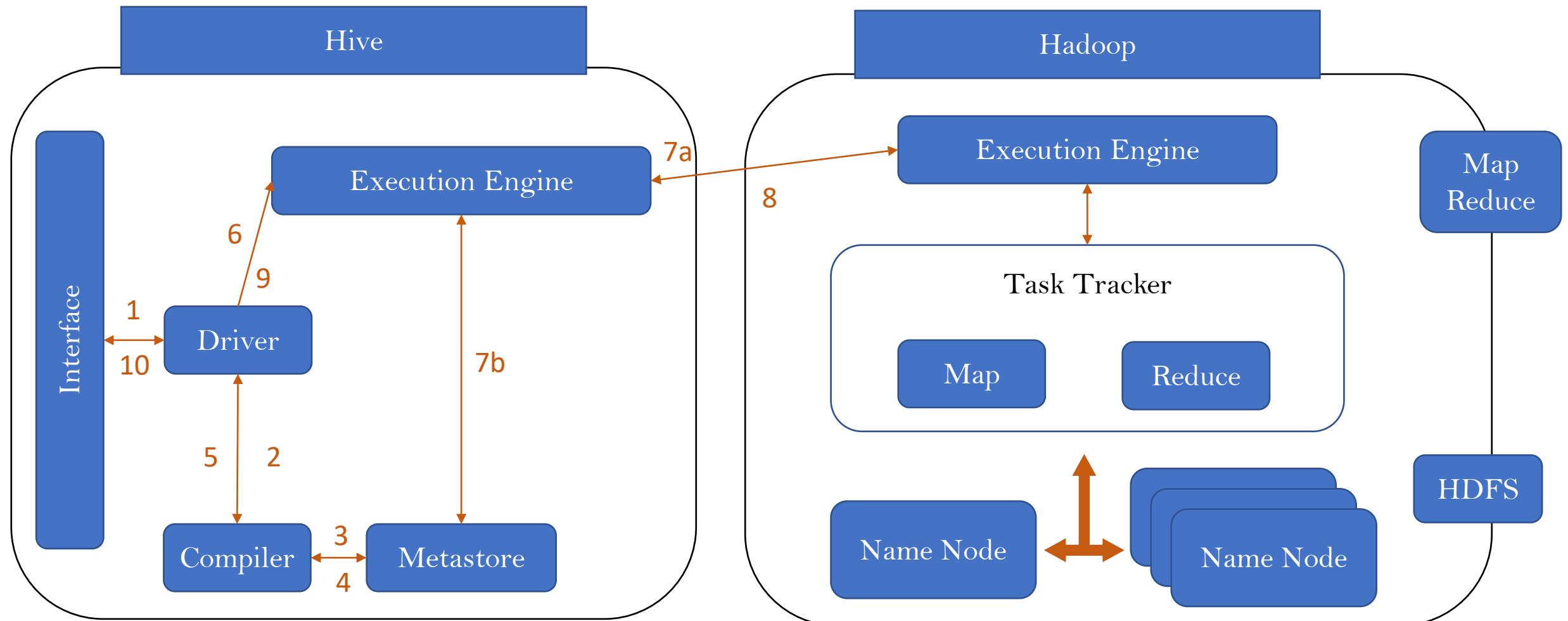












Hive's Data Units

- Database
- Tables
- Partitions
- Buckets (or Cluster)

Very Similar to SQL and Rational DBs

Data Model

- Table: (maps to a HDFS directory)
 - Like tables in relational DBs Each table has corresponding directory in HDFS
- Partition: (maps to sub-directory under the table)
 - Analogous to dense indexes on partition columns
 - Nested sub-directories in HDFS for each combination of partition column values
 - Allows users to efficiently retrieve rows
- Bucket (maps to files under each partition)

Hive Data Structure

- Traditional Database concepts
 - Tables, Rows, Columns, Partitions
- Supports primitive types
 - Integers, Floats, Doubles, Strings
- Additional types and structures
 - Associative arrays
 - map <key-type, value-type>
 - Lists
 - list <element type>
 - Structs
 - Struct <file name: file type...>

Hive Interface

- Command Line interface
- Web interface or Hue
- Java Database connectivity

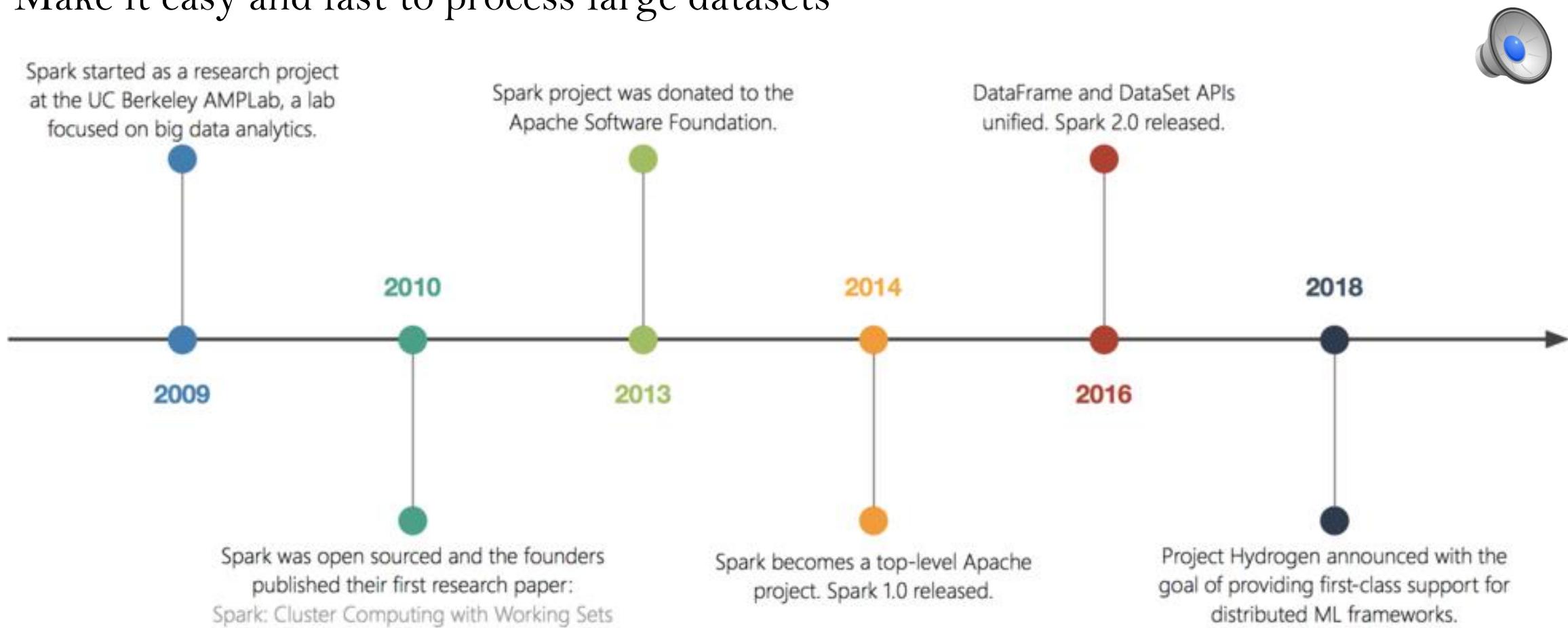
Apache Spark

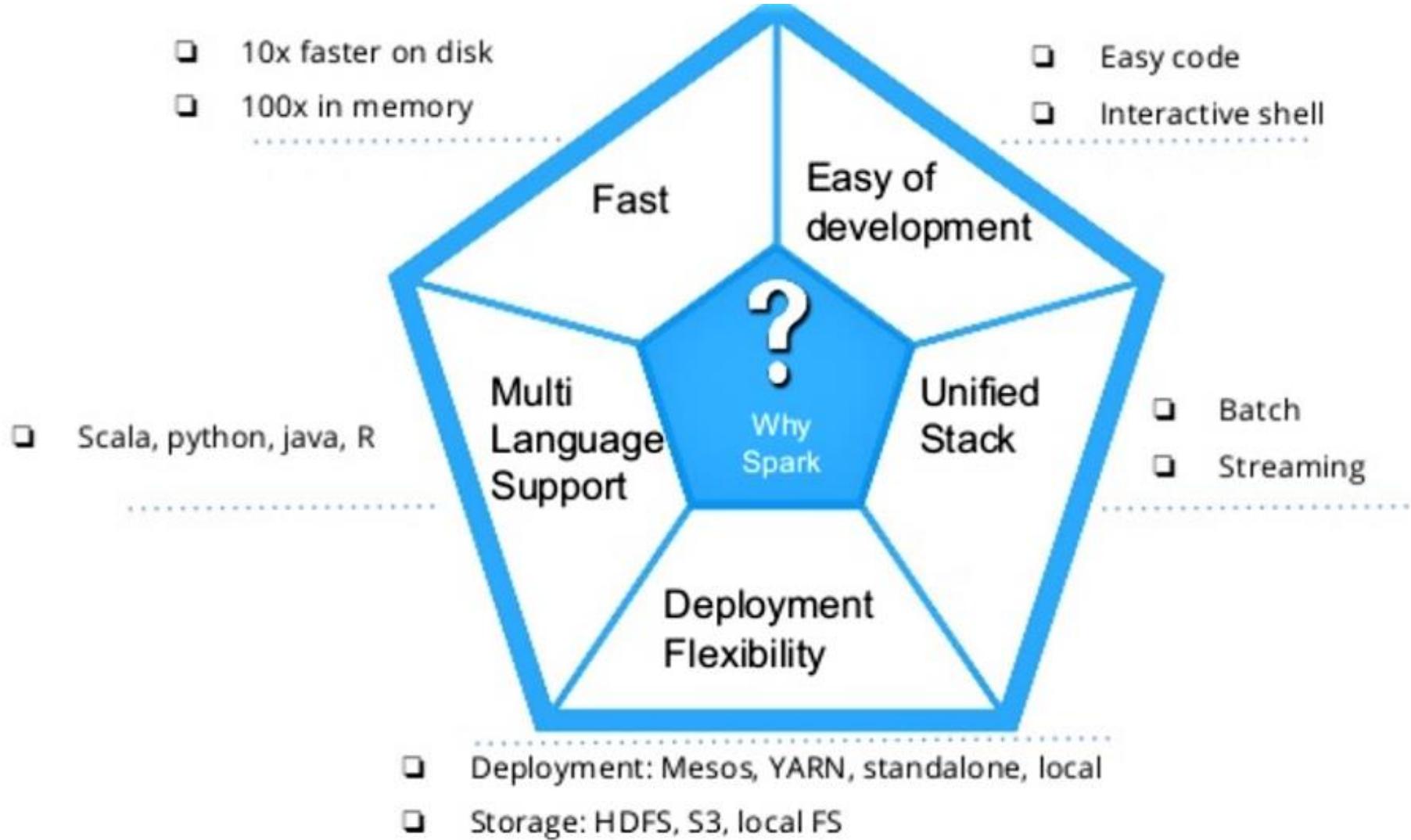
Agenda

- Introduction to Spark
- Apache Spark stack
- Spark Components
- Resilient Distributed Datasets
- RDD Operations

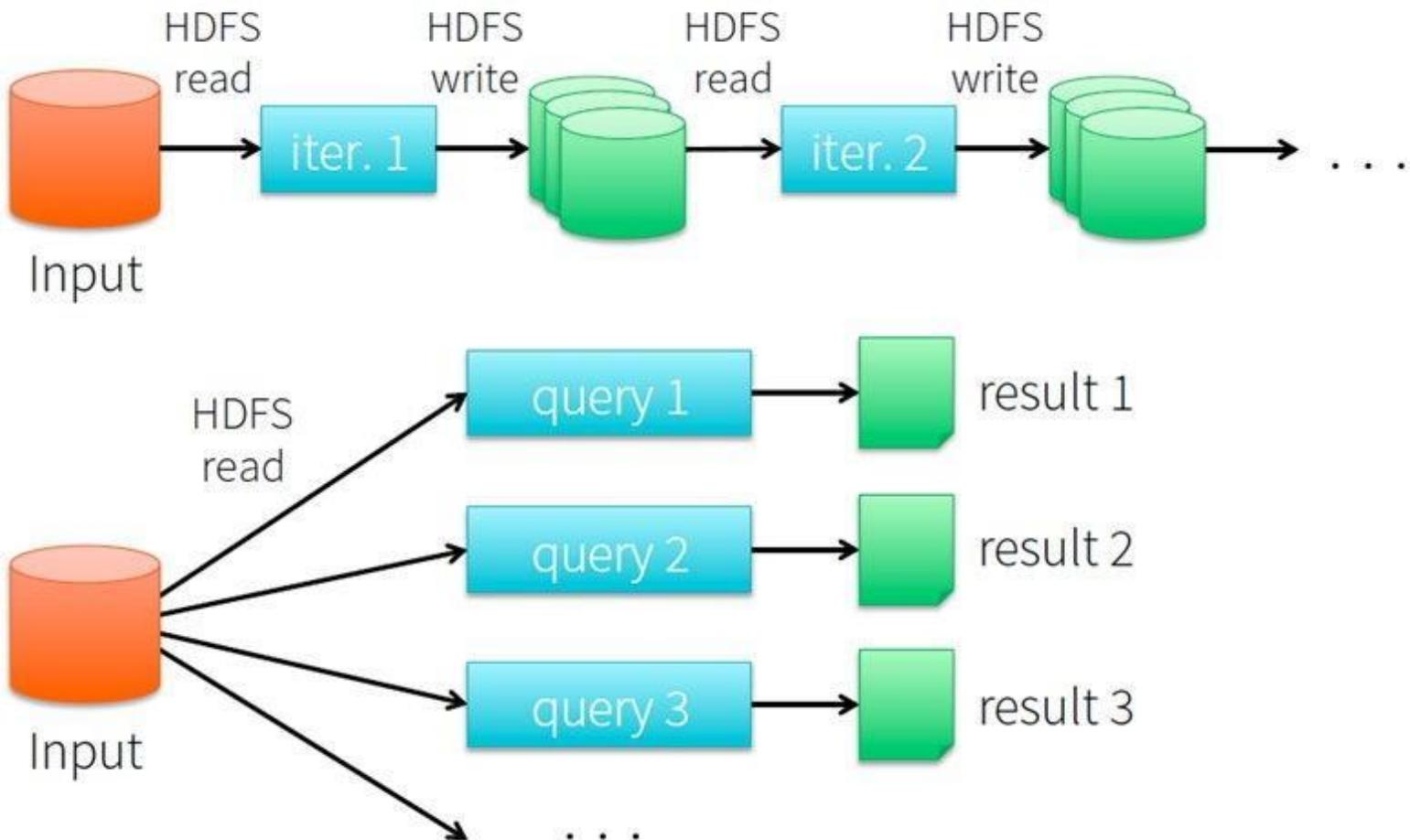
Apache Spark

- Fast and general cluster computing engine that generalizes the MapReduce model
- Make it easy and fast to process large datasets



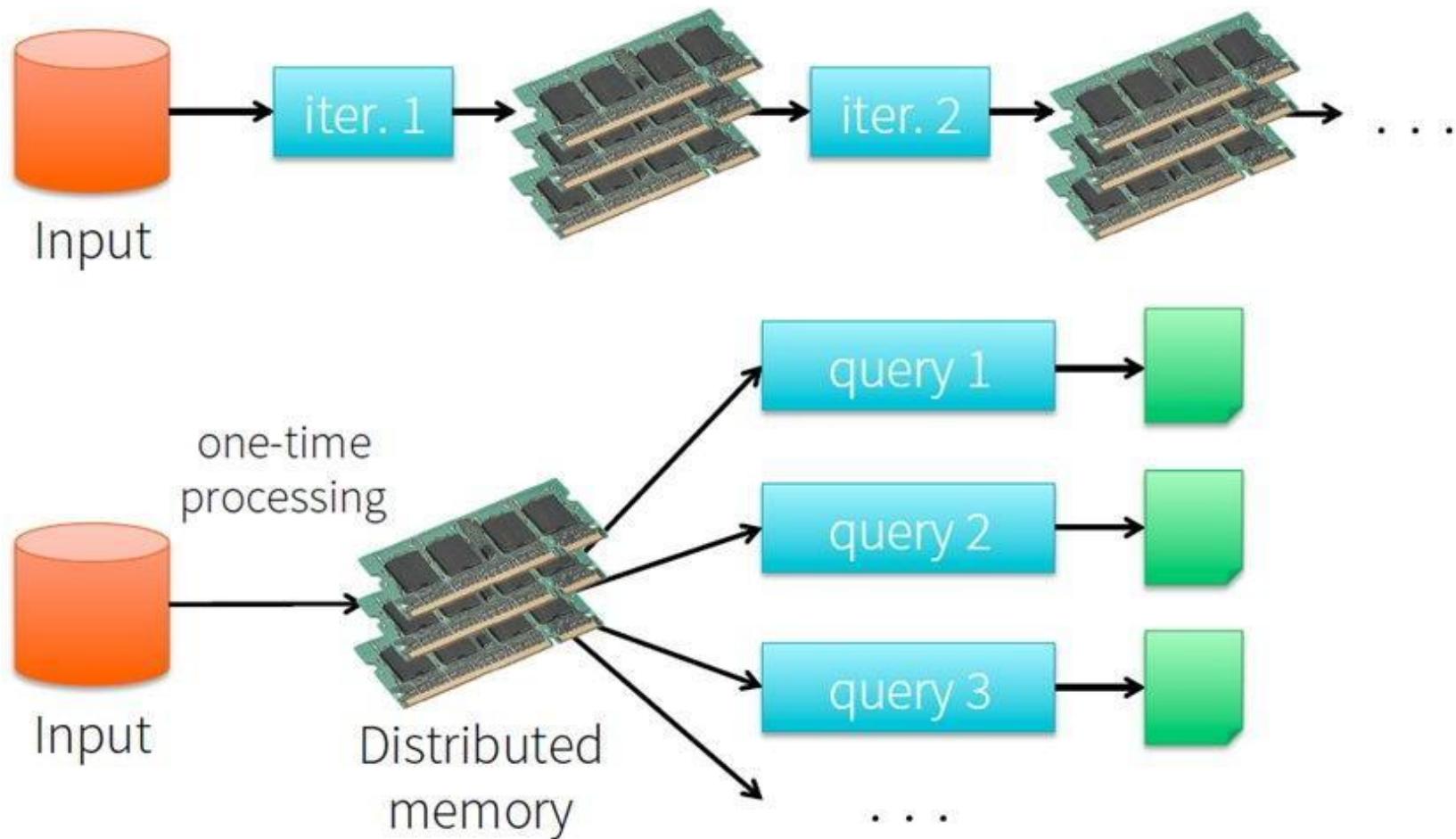


Data Sharing in MapReduce



Slow due to replication and disk I/O

What we'd like



10-100x faster than network and disk

First epoch (2009-2012)

- Key observations
 - Underutilization of cluster memory
 - For many companies data can fit into memory either now, or soon
 - Memory prices were decreasing year-over-year at that time
 - Redundant disk I/O
 - Especially in iterative MR jobs
- Key outcomes
 - RDD abstraction with rich API
 - In-memory distributed computation platform

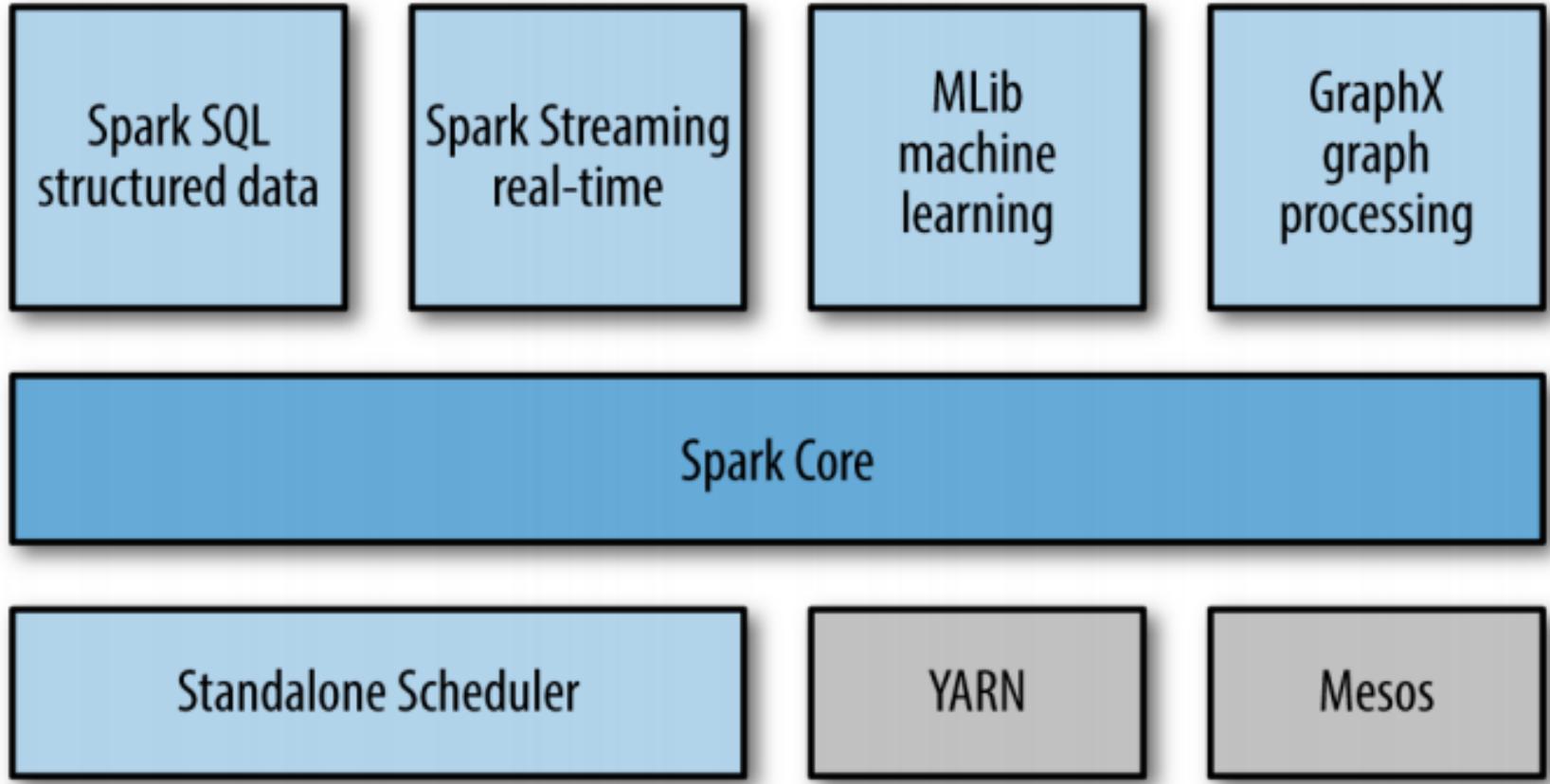
Second epoch (2012-2014)

- Key observations
 - No ‘one system to rule them all’
 - Increasing demand for interactive queries and stream processing
- Key outcomes
 - Separation of Spark Core and Applications on top of the core:
 - Spark SQL
 - Spark Streaming
 - Spark GraphX
 - Spark MLlib

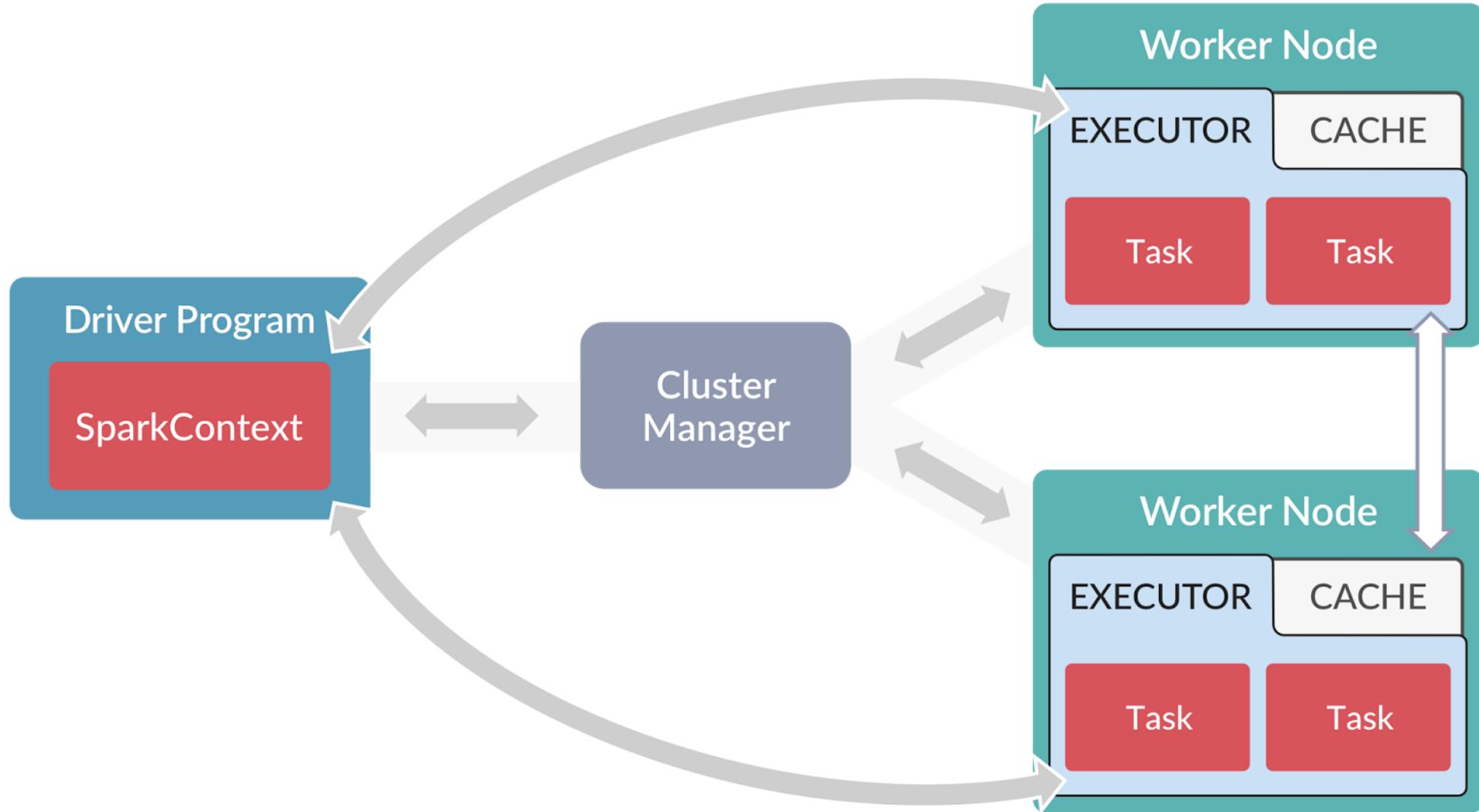
Third epoch (2014-now)

- Key observations
 - Increasing use of machine learning
 - Increasing demand for integration with other software (Python, R, Julia)
- Key outcomes
 - Focus on ease-of-use
 - Spark Dataframes as first-class citizens

Apache Spark Stack



Spark Components



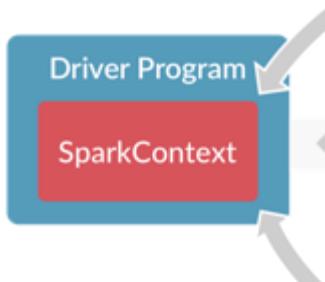
Spark Components: SparkContext, Driver Program

SparkContext

- Main entry point for Spark functionality
- Represents the connection to a Spark cluster
- Tells Spark how & where to access a cluster
- Can be used to create RDDs, accumulators and broadcast variables on that cluster

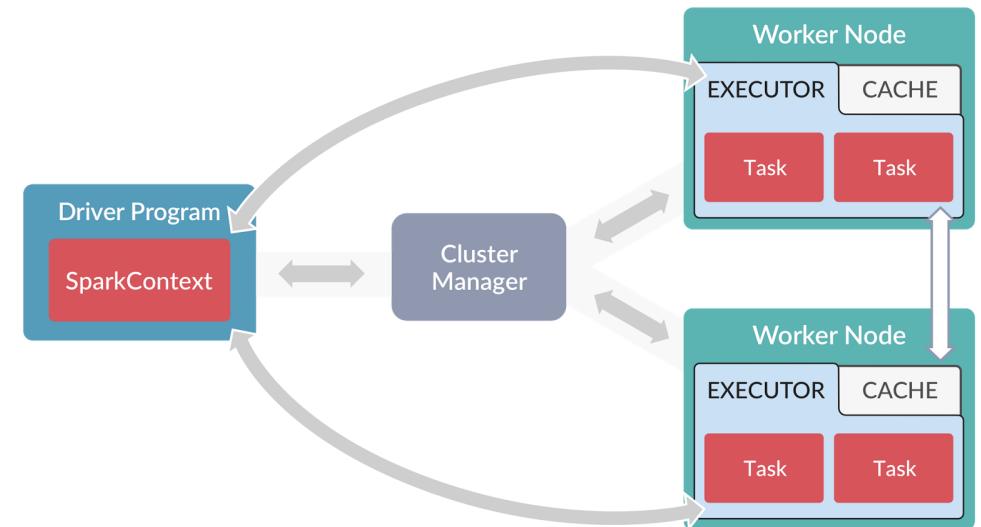
Driver Program

- “Main” process coordinated by the SparkContext object
- Allows to configure any spark process with specific parameters
- Spark actions are executed in the Driver
 - Spark-shell
 - Application ® driver program + executor



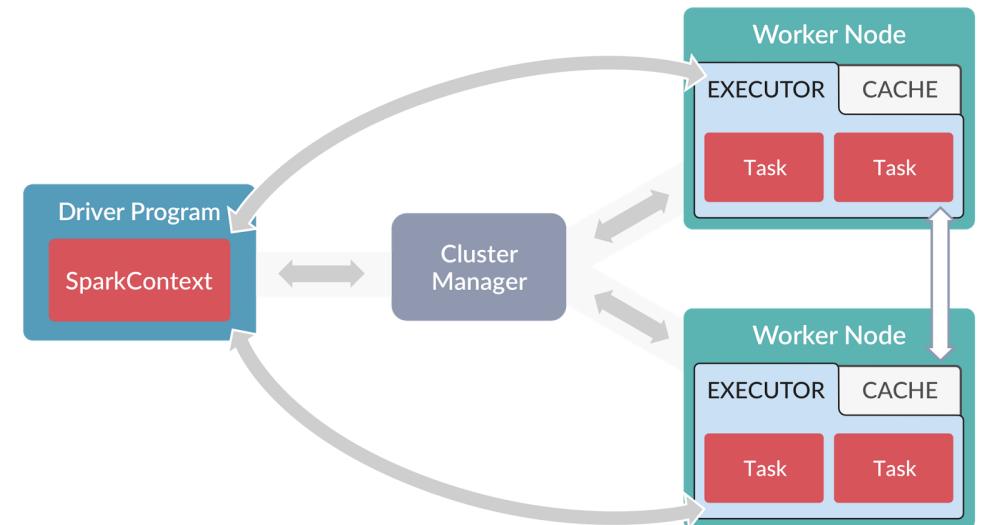
Spark Components: Cluster Manager

- External service for acquiring resources on the cluster
- Variety of cluster managers
 - Local
 - Standalone
 - YARN
 - Mesos
- Deploy mode:
 - Cluster ® framework launches the driver inside of the cluster
 - Client ® submitter launches the driver outside of the cluster



Spark Components: Worker

- Any node that can run application code in the cluster
- Key Terms
 - Executor: A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors
 - Task: Unit of work that will be sent to one executor
 - Job: A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. save collect)
 - Stage: smaller set of tasks inside any job



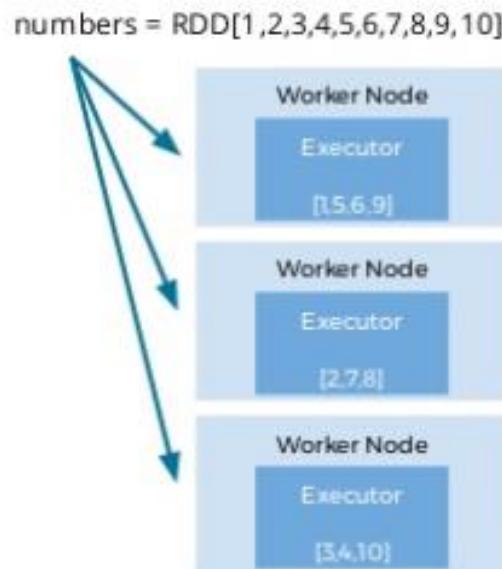
Summary of Spark runtime architecture

1. The user submits an application using spark-submit.
2. spark-submit launches the driver program and invokes the main() method specified by the user.
3. The driver program contacts the cluster manager to ask for resources to launch executors.
4. The cluster manager launches executors on behalf of the driver program.
5. The driver process runs through the user application. Based on the RDD actions and transformations in the program, the driver sends work to executors in the form of tasks.
6. Tasks are run on executor processes to compute and save results.
7. If the driver's main() method exits or it calls `SparkContext.stop()`, it will terminate the executors and release resources from the cluster manager.

Resilient Distributed Datasets (RDD)

Resilient Distributed Datasets (RDD)

- Collection of objects stored in RAM that is distributed across nodes in a cluster
- Data Operations are performed on RDD
- Once created, RDD are immutable
- RDD can be persisted in memory or on disk
- Fault Tolerant



Example of shell commands to create RDD and perform operations on RDD:

```
>>> lines = sc.textFile("README.md") # Create an RDD called lines
>>> lines.count() # Count the number of items in this RDD
127
>>> lines.first() # First item in this RDD, i.e. first line of README.md
u'# Apache Spark'
```

Transformations

- Creates new dataset from existing one
- Lazy evaluated (Transformed RDD executed only when action runs on it)
- Example: filter(), map(), flatMap()

Example 3-2. Calling the filter() transformation

```
>>> pythonLines = lines.filter(lambda line: "Python" in line)
```

Actions

- Return a value to driver program after computation on dataset
- Example: count(), reduce(), take(), collect()

Example 3-3. Calling the first() action

```
>>> pythonLines.first()  
u'## Interactive Python Shell'
```

RDD: Lazy evaluation

Execution in Spark does not start until an action is triggered.

Consider the example below: we want to create an RDD of strings (example 3-1) and then derive the lines that include *Python* (example 3-2)

Example 3-1. Creating an RDD of strings with `textFile()` in Python

```
>>> lines = sc.textFile("README.md")
```

If Spark were to load and store all the lines in the files at this point, it would waste a lot of storage space, given that we immediately filter out many lines.

Example 3-2. Calling the `filter()` transformation

```
>>> pythonLines = lines.filter(lambda line: "Python" in line)
```

Instead, once Spark sees the whole chain of transformations, it can compute just the data needed for its result

Example 3-3. Calling the `first()` action

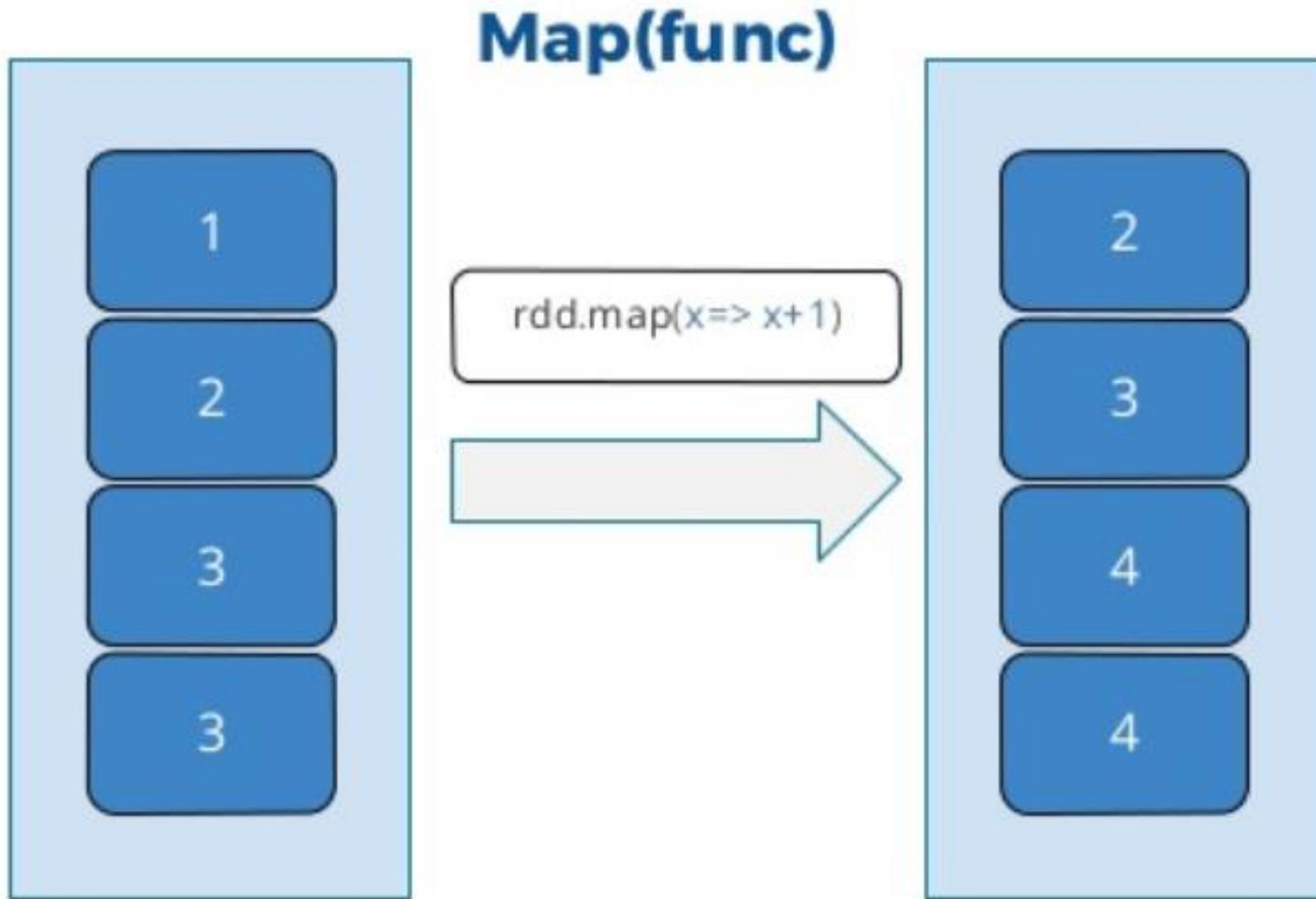
```
>>> pythonLines.first()  
u'## Interactive Python Shell'
```

Spark scans the file only until it finds the first matching line; it doesn't even read the whole file

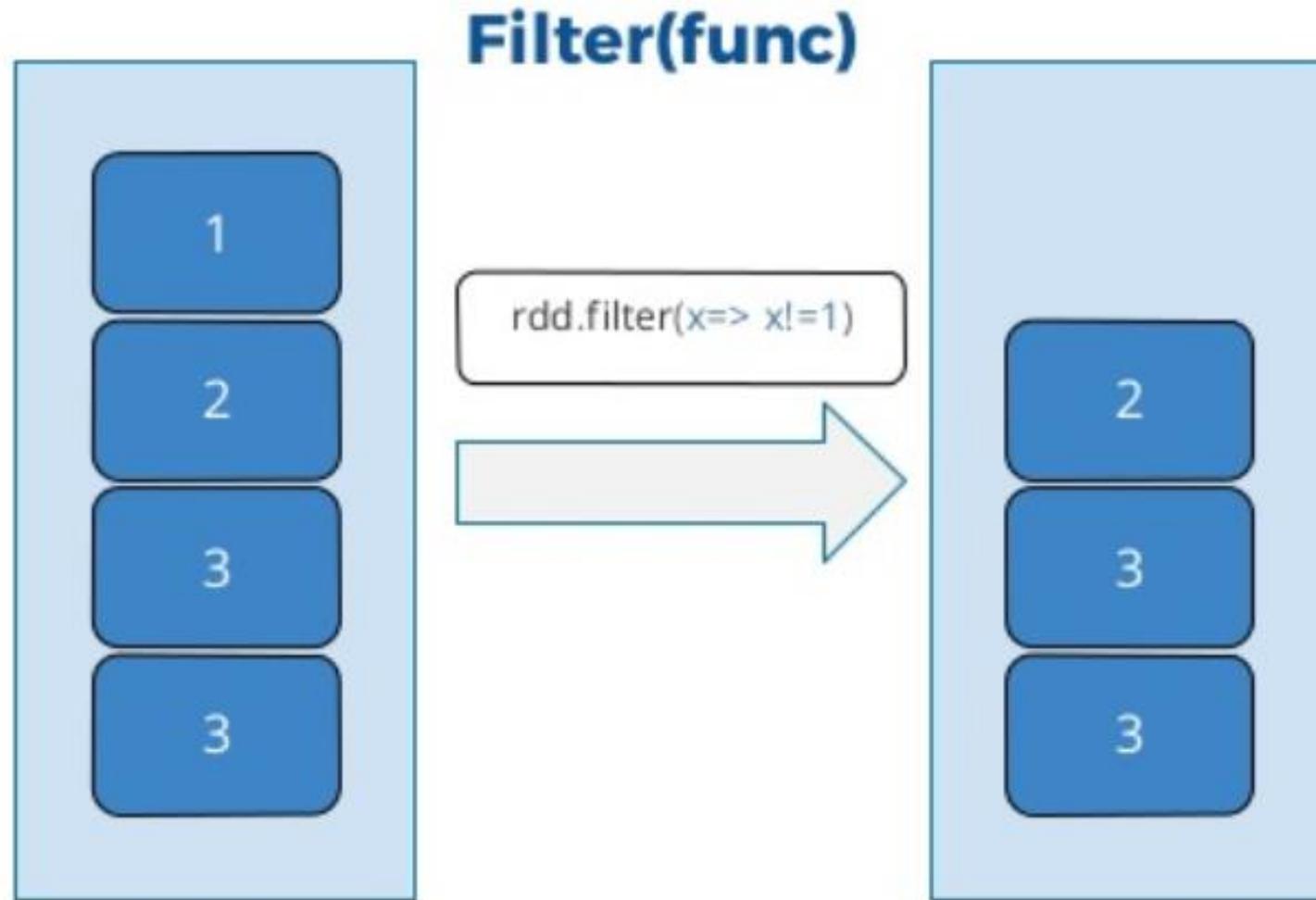
Common Transformations

Commonly Used Transformations	
Map(func)	Return a new distributed dataset formed by passing each element of the source through a function func
Filter(func)	Return a new dataset formed by selecting those elements of the source on which func returns true
FlatMap(func)	Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item)
Distinct	Return a new dataset that contains the distinct elements of the source dataset

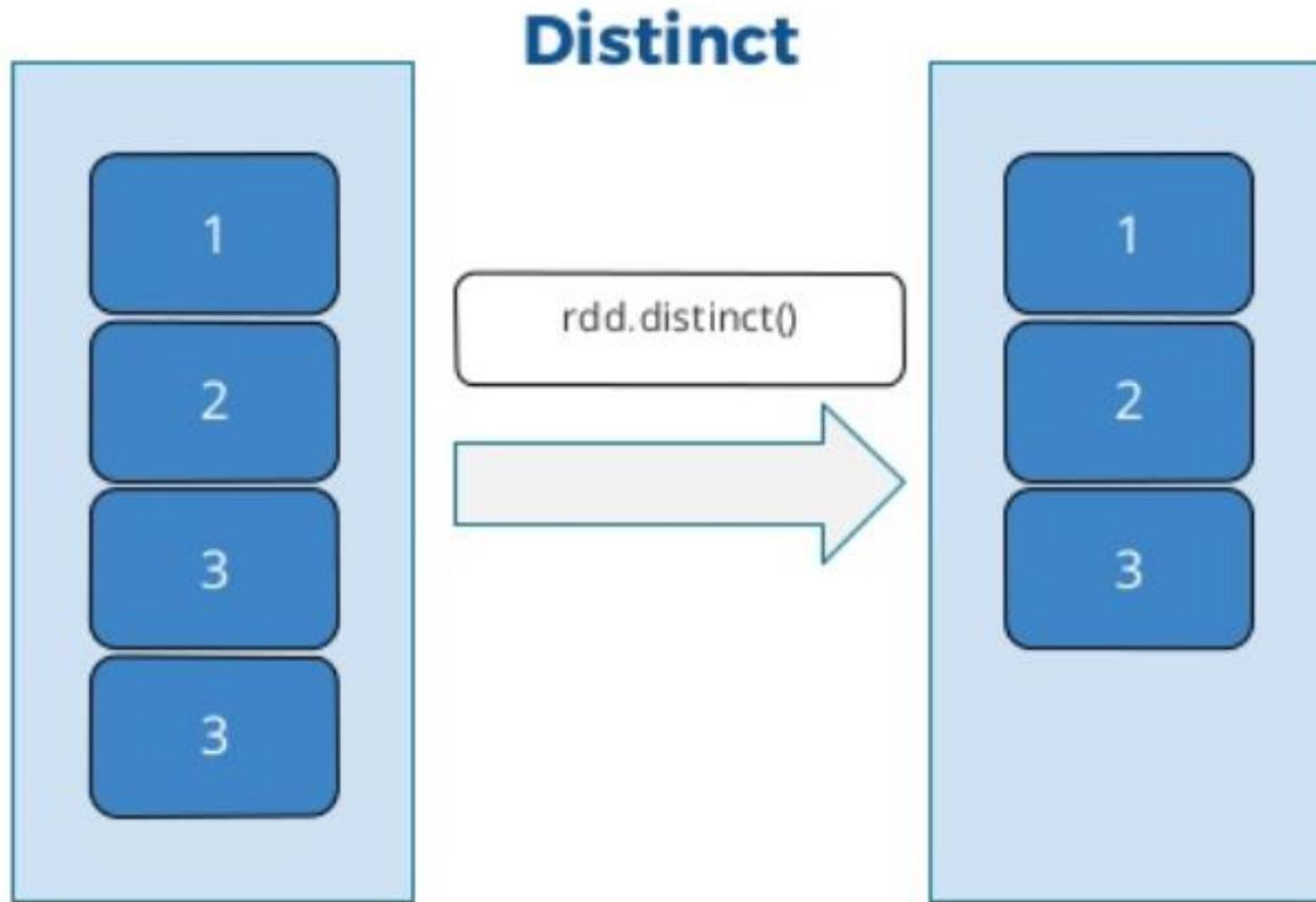
Transformations: Map(func)



Transformations: Filter(func)



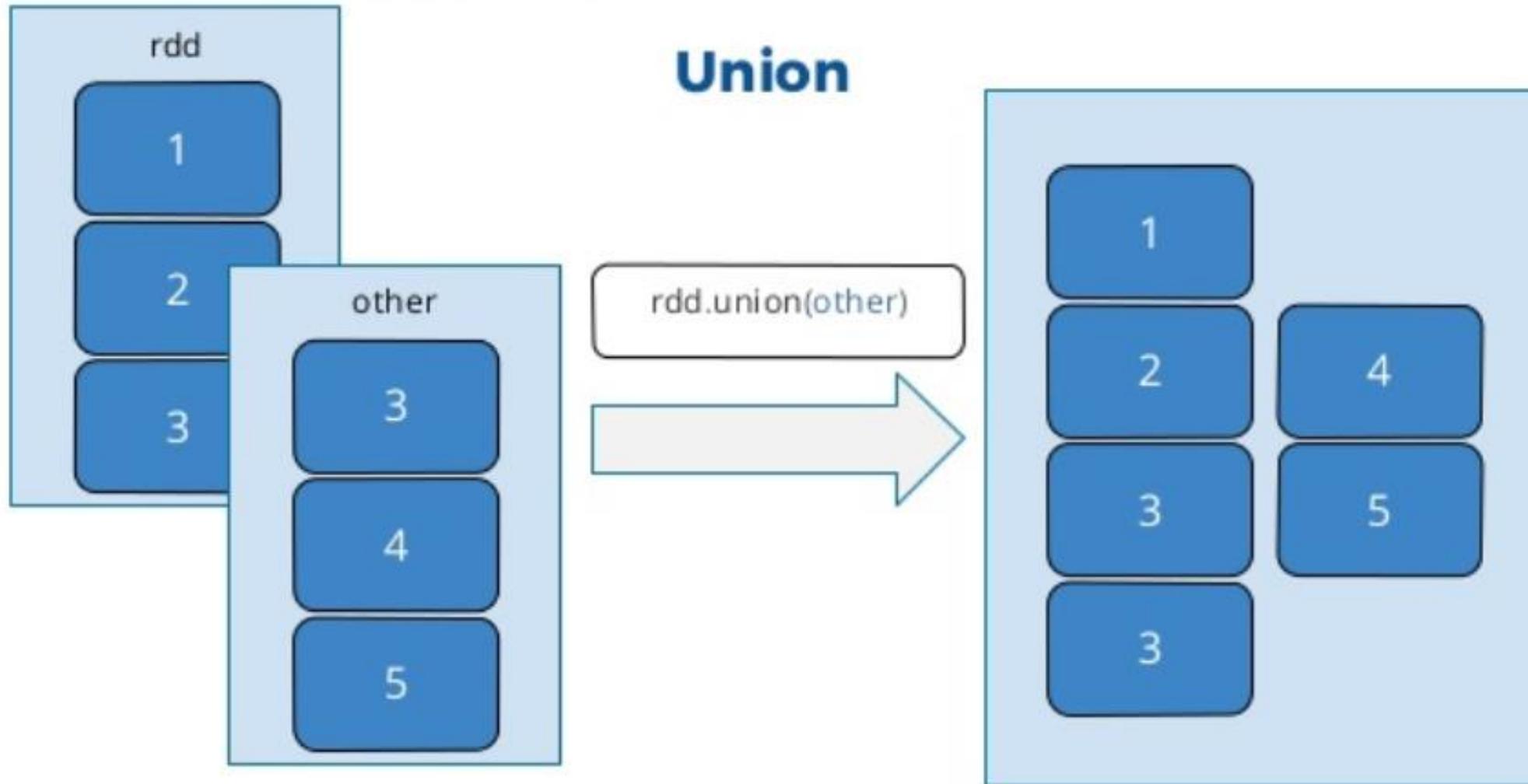
Transformations: Distinct



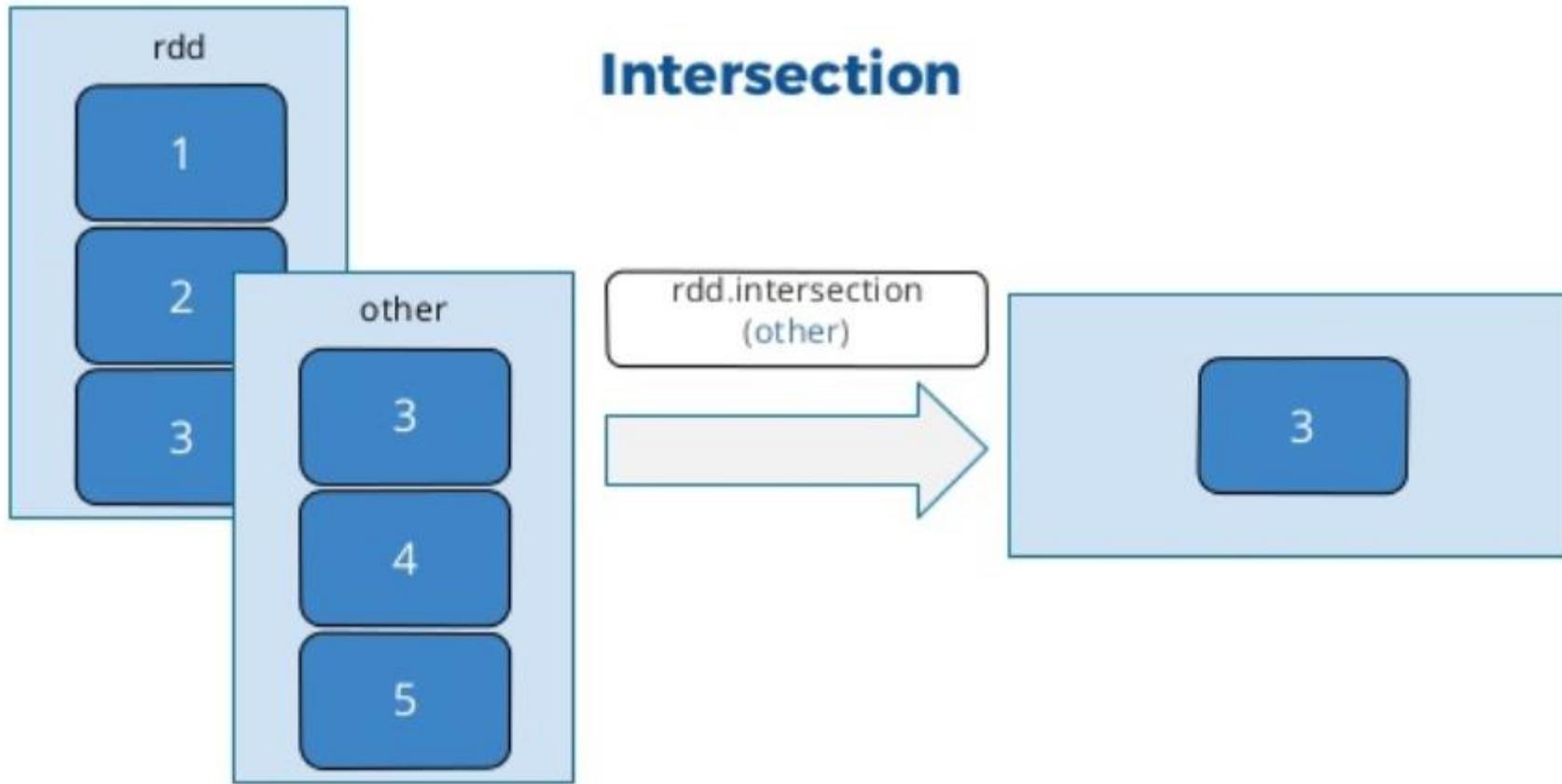
Common Transformations: Operations of mathematical sets

Union (other RDD)	Return a new RDD that contains the union of the elements in the source dataset and the argument
Intersection (otherRDD)	Return a new RDD that contains the intersection of elements in the source dataset and the argument

Transformations: Union



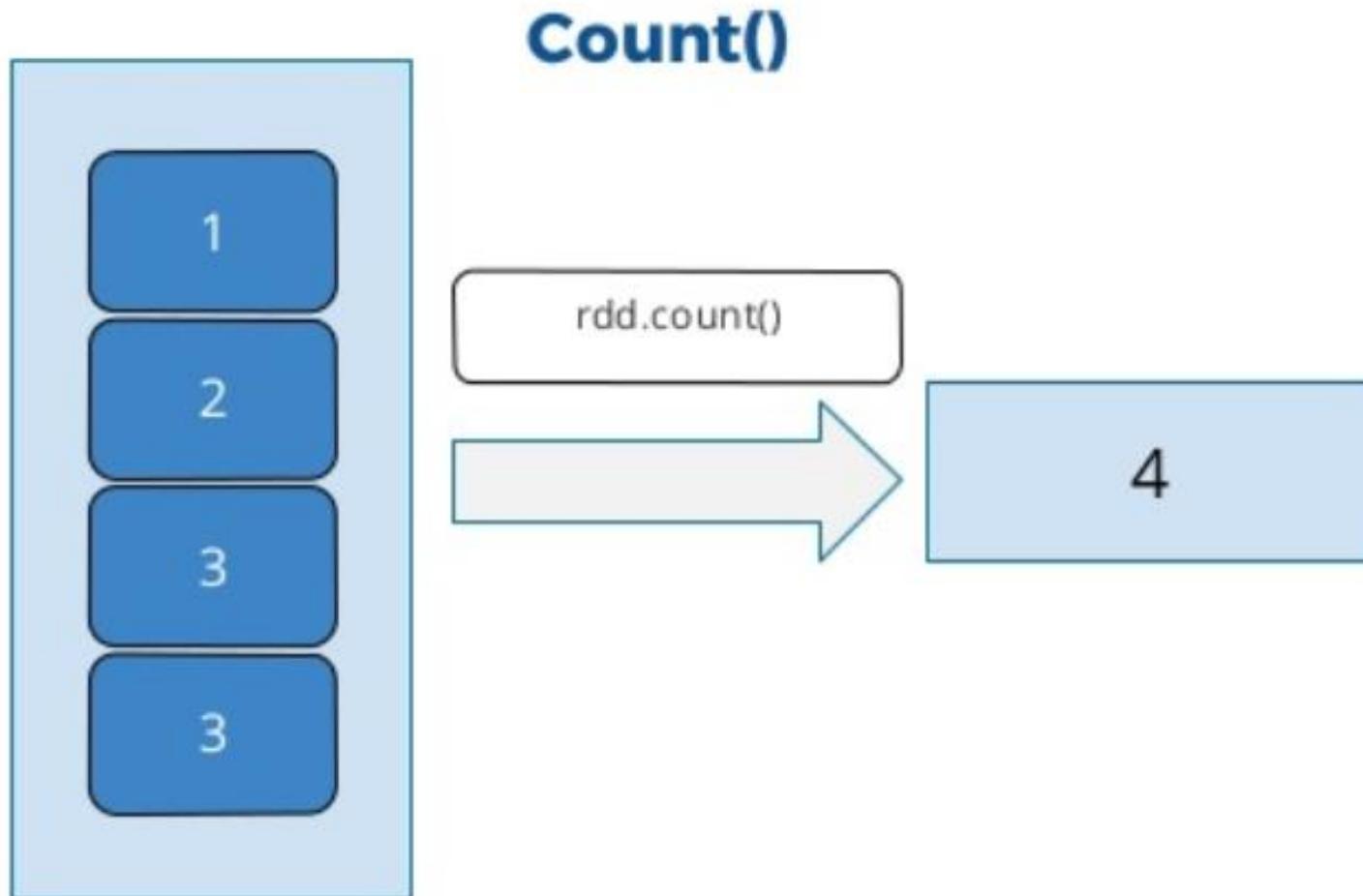
Transformations: Intersection



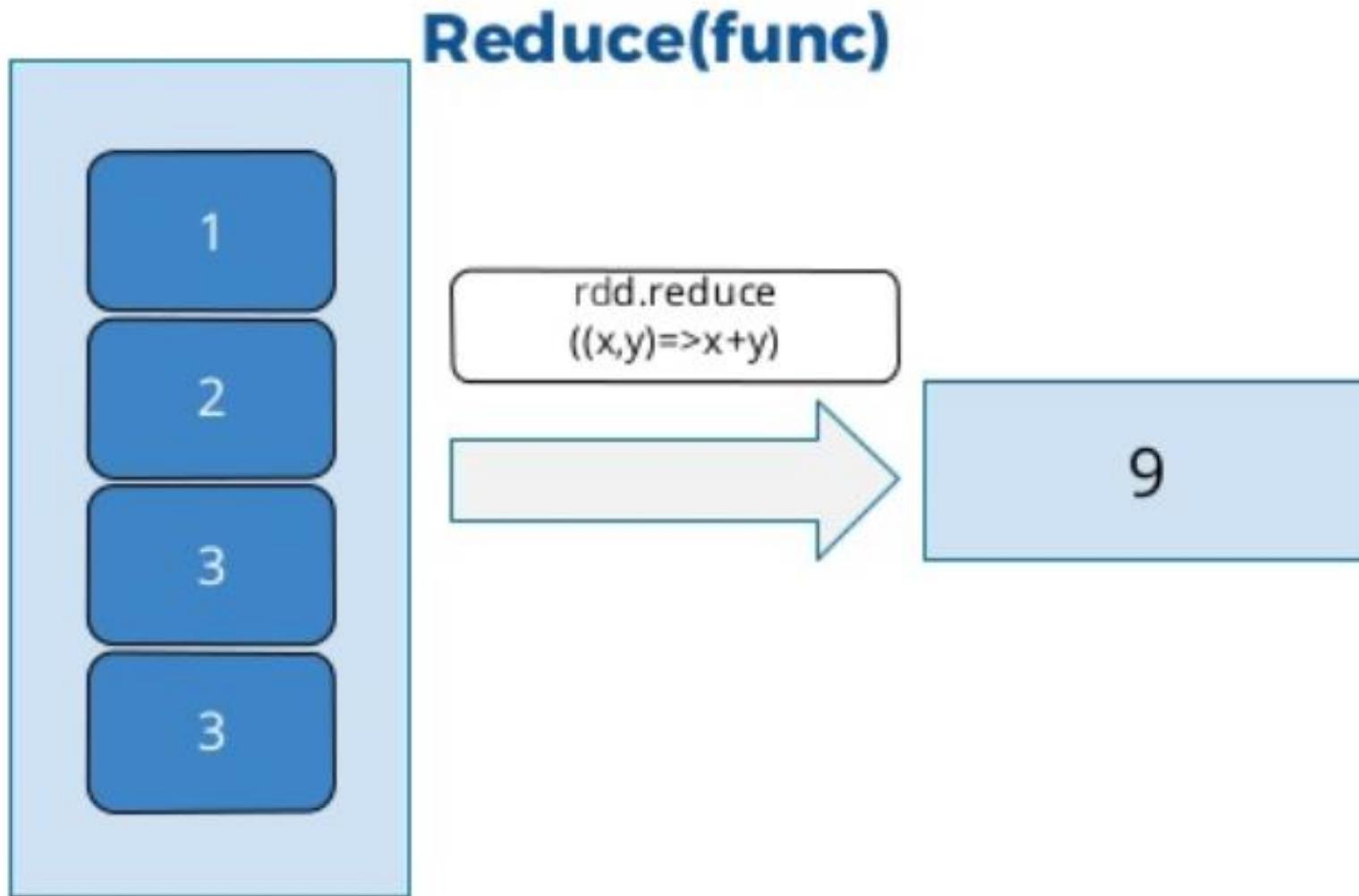
Commonly Used Actions

Count()	Returns the number of elements in the dataset
Reduce(func)	Aggregate the element of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel
Collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data
Take(n)	Returns an array with first n elements
First()	Returns the first element of the dataset
TakeOrdered(n, [ordering])	Returns first n elements of RDD using natural order or custom operator

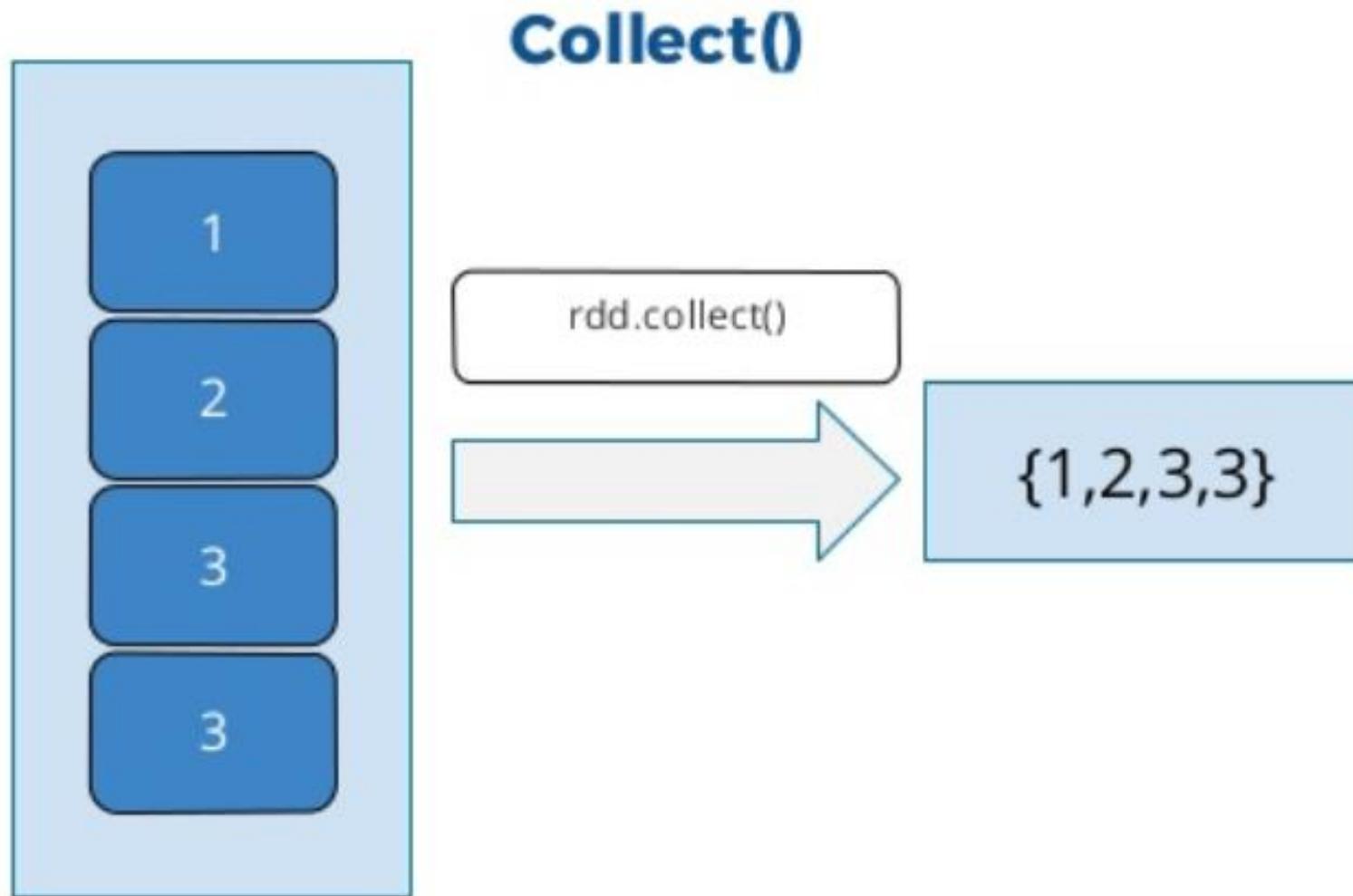
Actions: Count()



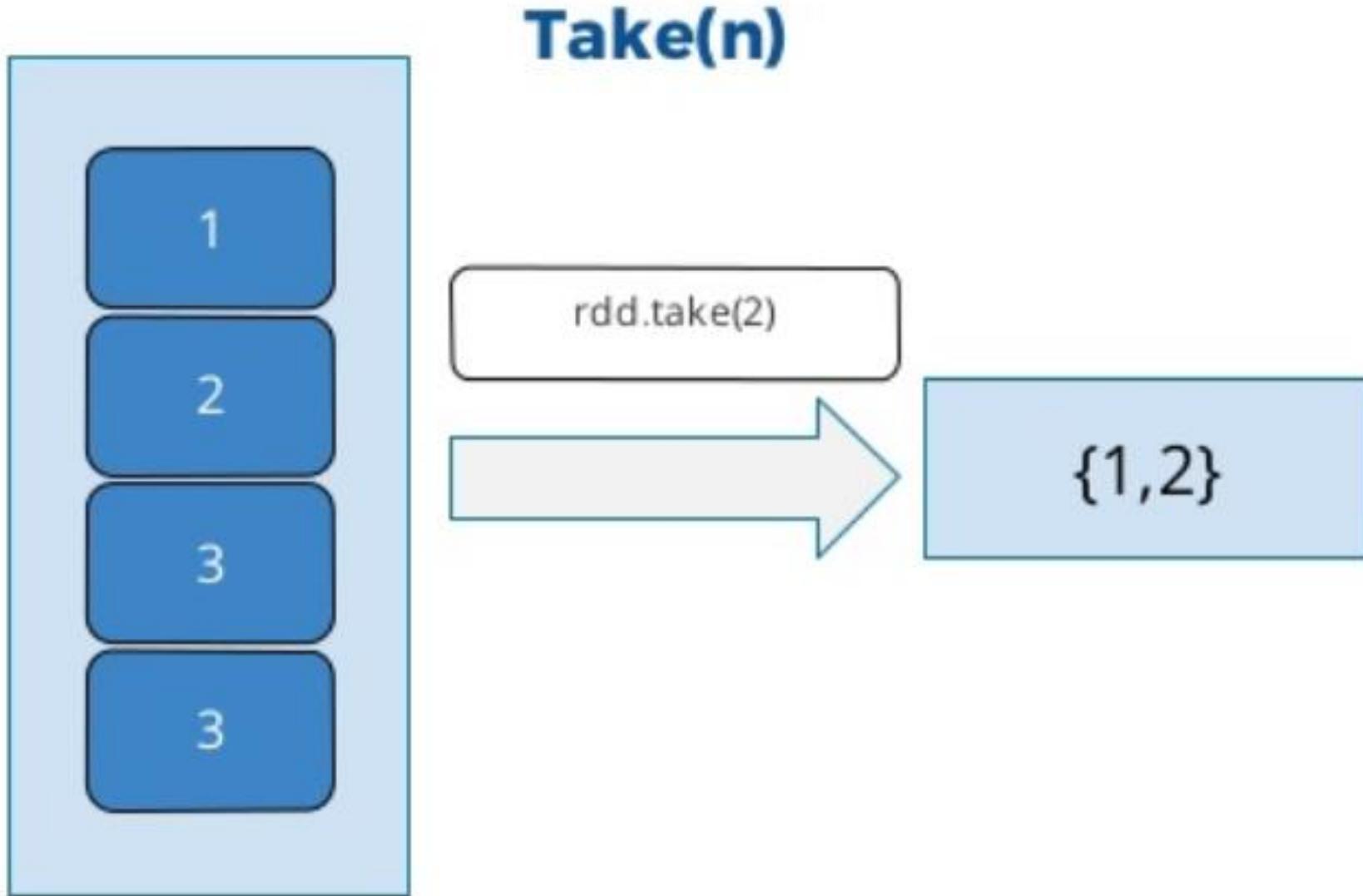
Actions: Reduce(func)



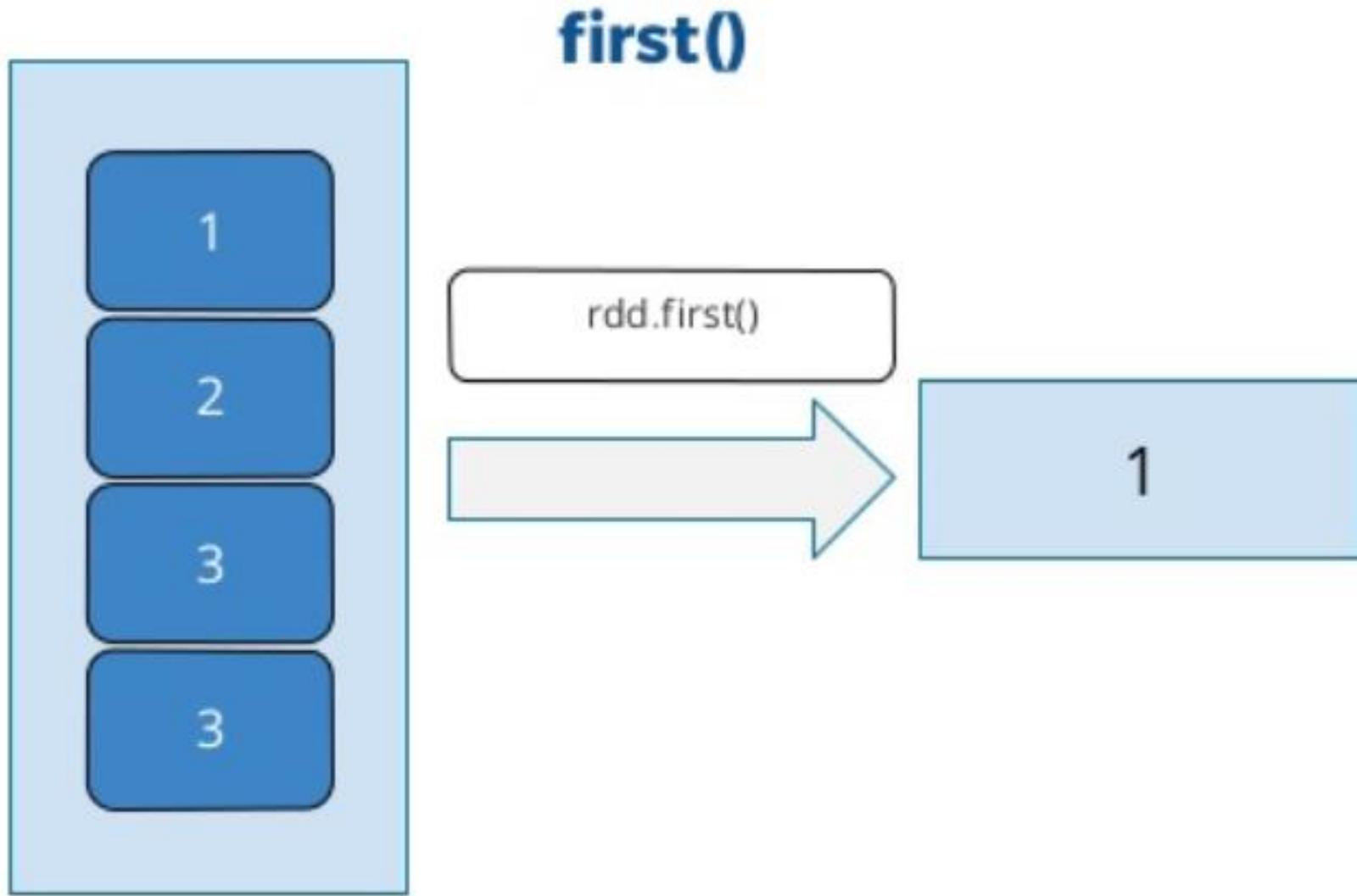
Actions: Collect()



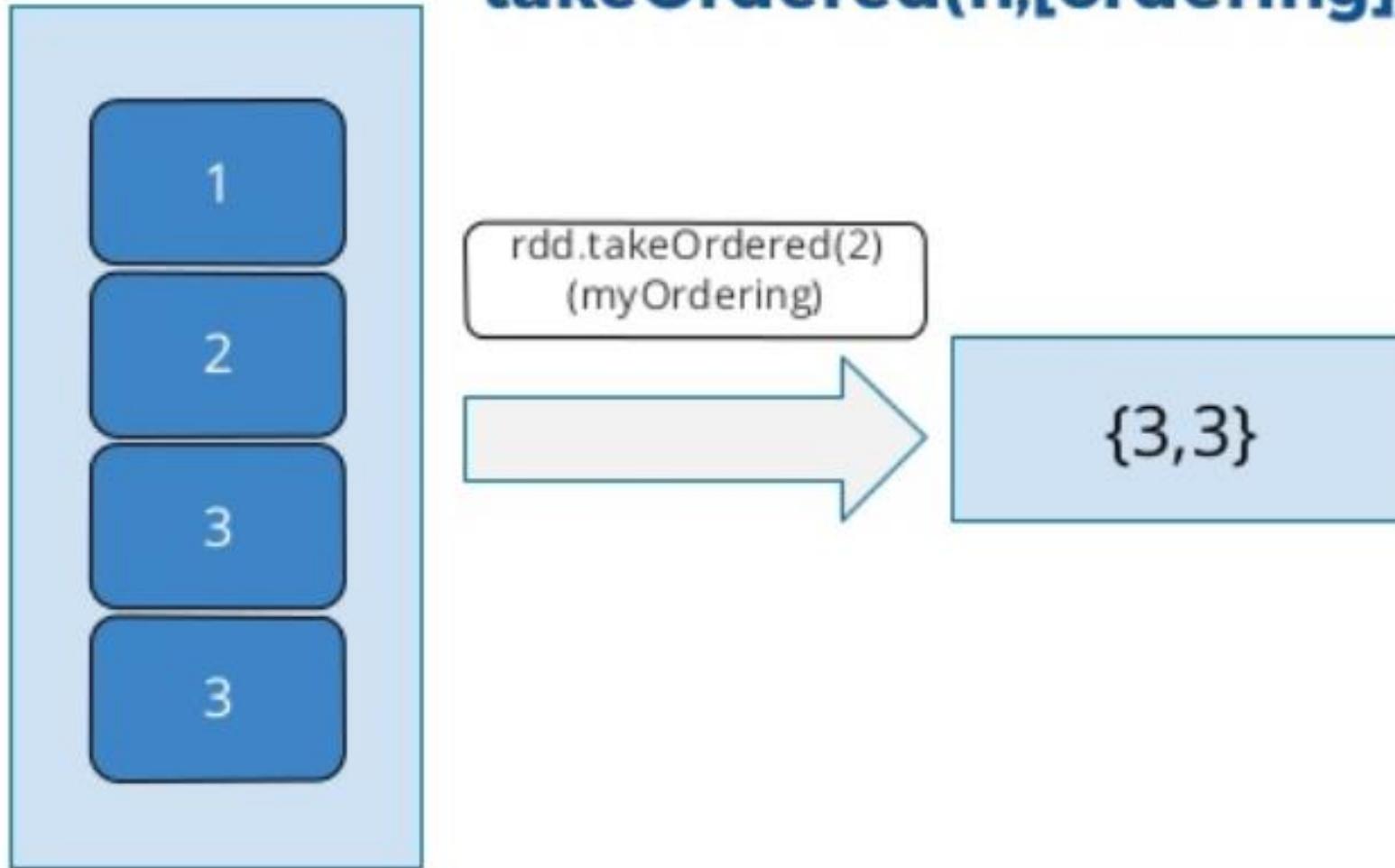
Actions: Take(n)



Actions: First()

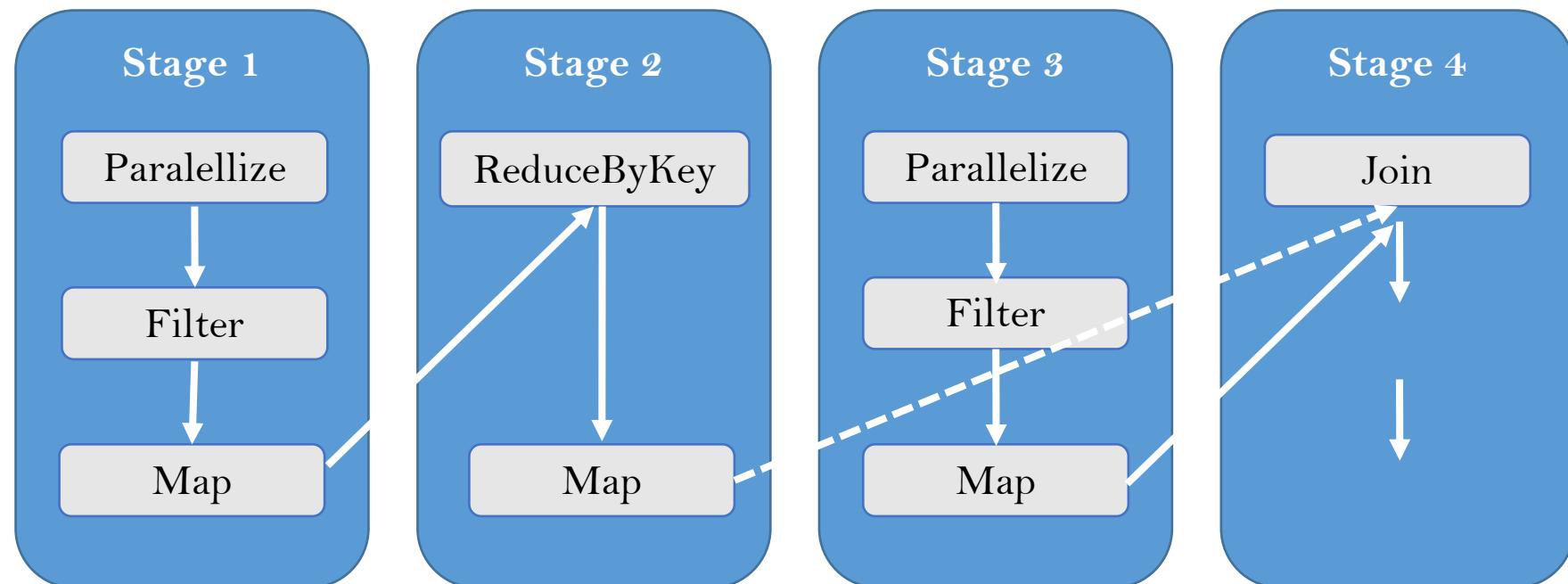


Actions: takeOrdered(n, [ordering])



Spark: Directed Acyclic Graph

(Directed Acyclic Graph) DAG in Apache Spark is a set of **Vertices** and **Edges**, where *vertices* represent the **RDDs** and the *edges* represent the **Operation to be applied on RDD**.



Key messages

- Spark is best known for its ability to keep large working datasets in memory between jobs.
- This capability allows Spark to outperform the equivalent MapReduce workflow (by an order of magnitude or more in some cases), where datasets are always loaded from disk.
- Two styles of application that benefit greatly from Spark's processing model are iterative algorithms (where a function is applied to a dataset repeatedly until an exit condition is met) and interactive analysis (where a user issues a series of ad hoc exploratory queries on a dataset).
- Spark has a DAG engine that can process arbitrary pipelines of operators and translate them into a single job for the user.
- Spark's user experience: a rich set of APIs for performing many common data processing tasks, such as joins.

Reading

Book:

Learning Spark: Lightning-Fast Data Analysis by Karau H, Konwinski A, Wendell P & Zaharia M, O'Reilly, chapters 1, 2, 3

Video:

<https://www.youtube.com/watch?v=QaoJNXW6SQo&t=174s>

https://www.youtube.com/watch?v=QaoJNXW6SQo&t=174s&ab_channel=Simplilearn

https://www.youtube.com/watch?v=znBa13Earms&ab_channel=Simplilearn

https://www.youtube.com/watch?v=ymtq8yjmD9I&ab_channel=nullQueries

https://www.youtube.com/watch?v=dRJ5HiqJZOo&ab_channel=Intellipaat

Web Resources:

- <http://sparkhub.databricks.com>
- <https://data-flair.training/blogs/how-apache-spark-works/>
- <https://mapr.com/blog/spark-101-what-it-is-and-why-it-matters/>