5042259201
Big Data Analytics

1 Week Part 1
Introduction to Big Data Mining
& Hadoop

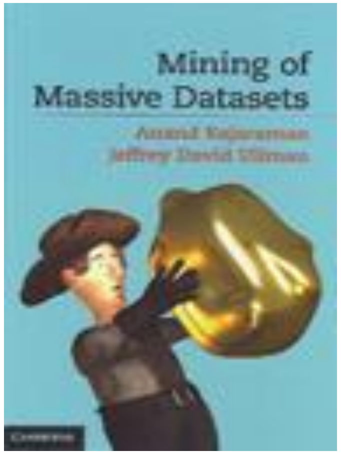
Introduction to Class (1/3)

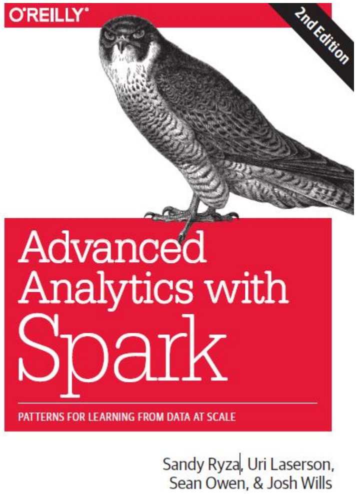
Big Data Analytics

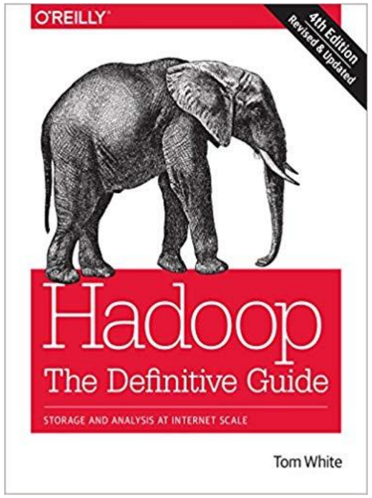
- Lecture Web site ; Smart Campus
  - Soongsil campus lecture web site
- Lecture Objectives
  - To study Big data Analytics algorithms, tools, and programming.
  - To discuss application of Big data Analytics algorithms to real world problems

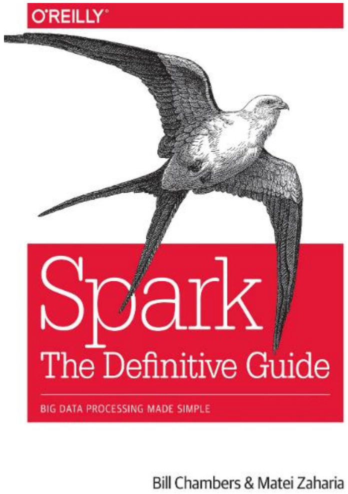
Introduction to Class(2/3)

- Textbook;
  - "Mining of Massive Datasets," J. Leskovec, et al.
    http://www.mmds.org
- Auxiliary Textbooks
  - "Advanced Analytics with Spark," S. Ryza, et al., O'Reilly
  - "Hadoop: The Definitive Guide," Tom White, 4th ed., O'Reilly
  - "Spark: The Definitive Guide, " B. chambers, et. al., O'Reilly









Introduction to Class(3/3)

- Class Style
  - Students' Presentations (+ Lecture)

- Evaluation

  Presentation(50%) – Present Chosen Subjects from Textbooks

  Final Project(50%)

Tentative Lecture Schedule

Week	Keyword	Description
01	Data Mining, MapReduce, Spark Programming (Scala, Pyspark), Tensorflow	Introduction to Data Mining, Statistical Modeling, Machine Learning, MapReduce algorithms, Spark, PySpark
02	Spark Toolsets, Spark Structured API Spark Machine Learning	RDD, DataFrame, Dataset, Spark MLlib, Spark ML
03	Similarity	Finding Similar Items, Locality-Sensitive Functions, Hashing
04	Mining Data Streams	Stream Data Model, Sampling Data in a Stream, Filtering Streams
05	Link Analysis	PageRank, Link Spam, Hubs and Authorities
06	Frequent Itemsets	Market-Basket Model, A-Priori Algorithm, Limited-Pass Algorithms
07	Clustering	Hierarchical Clustering, K-Means Clustering, CUR Algorithm
08	Advertising on the Web	Issues in On-Line Advertising, Matching Problem, Adwords Problem
09	Recommendation System	Collaborative Filtering, Dimensionality Reduction, Netflix Challenge
10	Mining Social-Network Graphs	Mining Social-Network, Graphs
11	Dimensionality Reduction	PCA, SVD, CUR Decomposition
12	Large-Scale Machine Learning	SVM. Decision Tree
13	Project Presentations	Project Presentations
14	Project Presentations	Project Presentations
15	Project Presentations	Project Presentations

Contents of 1st Week

- What is Data Mining?
- Big Data Computing Framework
- Distributed File System
- MapReduce
- Introduction to Apache Hadoop
- Hadoop: HDFS
- Hadoop: MapReduce
- Apache YARN
- Introduction to Apache Spark

What is Data Mining?
Knowledge discovery from data

SOONGSIL UNIVERSITY

Chapter 1,

"Mining of Massive Datasets, "

Jure Leskovec, Anand Rajaraman,

Jeff Ullman Stanford University

http://www.mmds.org

What is Data Mining?

- Given lots of data
- Discover patterns and models that are:
  - **Valid:** hold on new data with some certainty
  - **Useful:** should be possible to act on the item
  - **Unexpected:** non-obvious to the system
  - **Understandable:** humans should be able to interpret the pattern

Data Mining

- But to extract the knowledge data needs to be
  - Stored
  - Managed
  - and Analyzed ← this class

Data Mining ≈ Big Data ≈ Predictive Analytics ≈ Data Science

Data Mining Tasks -Business Analytics

- **Descriptive Analytics**
  - Llooks at past performance and understands that performance by mining historical data to look for the reasons behind past success or failure.
- **Predictive Analytics**
  - Historical data is combined with rules, algorithms, and occasionally external data to determine the probable future outcome of an event or the likelihood of a situation occurring.
- **Prescriptive Analytics**
  - suggests decision options on how to take advantage of a future opportunity or mitigate a future risk and shows the implication of each decision option.

Meaningfulness of Analytic Answers

- A risk with "Data mining" is that an analyst can "discover" patterns that are meaningless
- Statisticians call it Bonferroni's principle:
  - Roughly, if you look in more places for interesting patterns than your amount of data will support, you are bound to find crap

Bonferroni's Principle

- Informal presentation of a statistical theorem
  - If your method of finding significant items returns significantly more items that you would expect in the actual population, you can assume most of the items you find with it are bogus.
  - This essentially means that an algorithm or method we think is useful for finding a particular set of data actually returns more false positives as it returns larger portions of the data than should be within that category.

Ex. of Bonferroni’ s principle

- Assume you are trying to identify people who are cheating on their spouses within a certain population, and you know that the percentage in the population who cheat on their spouses is 5%.
- If you decide that people who claim to go out with coworkers more than three times a month are most likely actually cheating on their spouses, but discover that 20% of people in the population qualify with your method, then you know in the very best case only one quarter of the people you identify will actually be cheaters.
- Furthermore, if there are any false negatives (cheaters who aren't identified as cheaters), an even higher percentage of the "cheaters" identified with the system would be false positives.

Uses

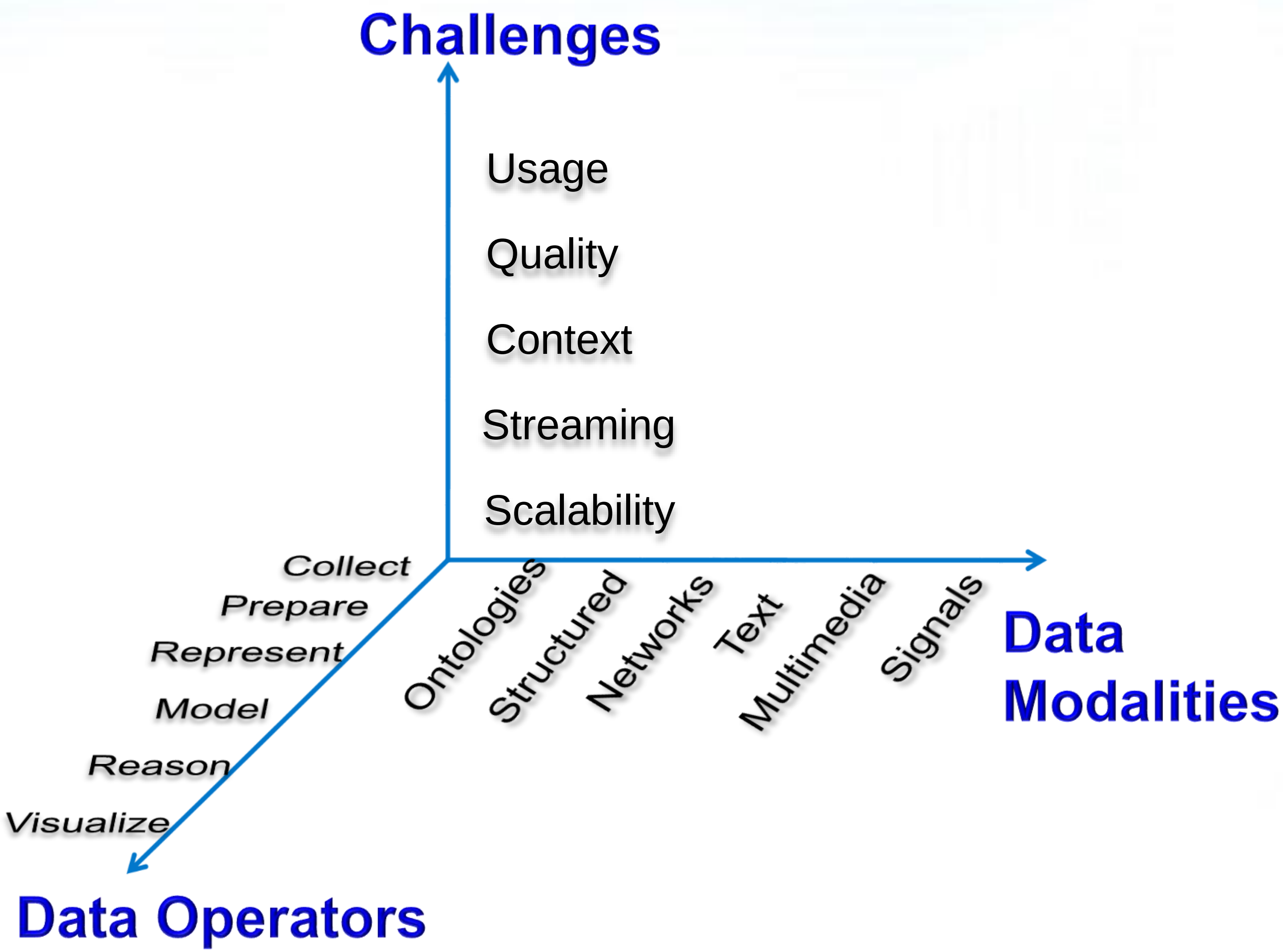
- Applying Bonferroni's Principle to an algorithm or system for identifying or classifying data gives you an upper bound on the accuracy of your methods. If you determine that you match significantly more data or less data than you should expect, then you in the best case have too many false positives or false negatives, respectively.
- The principle is especially useful in debunking individuals who use cold reading techniques. You may think they are using some sort of psychic power to accurately identify a single person, but if it turns out that 90% of the audience can identify with something they are saying and it's likely there's at least one person who can identify with most of what they are saying in every audience, their powers become much less impressive.

Meaningfulness of Analytic Answers

Example:

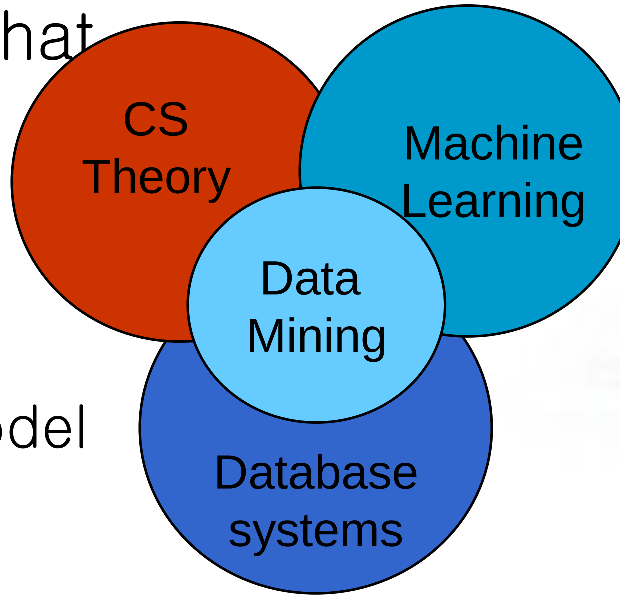
- We want to find (unrelated) people who at least twice have stayed at the same hotel on the same day
  - 10^9 people being tracked
  - 1,000 days
  - Each person stays in a hotel 1% of time (1 day out of 100)
  - Hotels hold 100 people (so 10^5 hotels)
  - If everyone behaves randomly (i.e., no terrorists) will the data mining detect anything suspicious?
- Expected number of "suspicious" pairs of people:
  - 250,000
  - … too many combinations to check – we need to have some additional evidence to find "suspicious" pairs of people in some more efficient way

What matters when dealing with data?



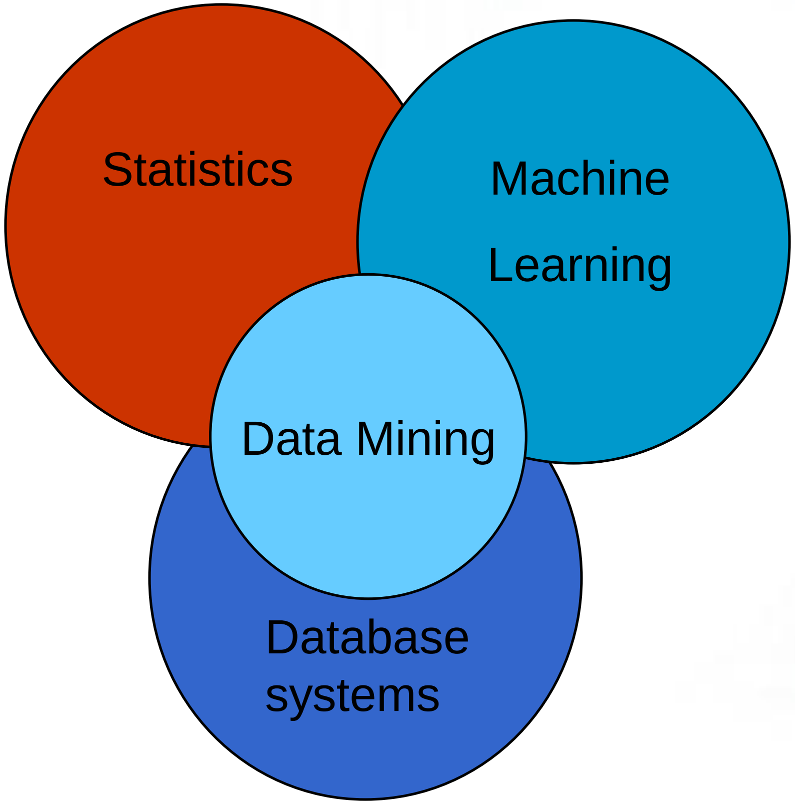
Data Mining: Cultures

- Data mining overlaps with:
  - **Databases:** Large-scale data, simple queries
  - **Machine learning:** Small data, Complex models
  - **CS Theory:** (Randomized) Algorithms
- Different cultures:
  - To a DB person, data mining is an extreme form of analytic processing – queries that examine large amounts of data
    - Result is the query answer
  - To a ML person, data-mining is the inference of models
    - Result is the parameters of the model
- In this class we will do both!



This Class

- This class overlaps with machine learning, statistics, artificial intelligence, databases but more stress on
  - Scalability (big data)
  - Algorithms
  - Computing architectures
  - Automation for handling large data



What will we learn?

- We will learn to mine different types of data:
  - Data is high dimensional
  - Data is a graph
  - Data is infinite/never-ending
  - Data is labeled
- We will learn to use different models of computation:
  - MapReduce
  - Streams and online algorithms
  - Single machine in-memory

What will we learn?

- We will learn to solve real-world problems:
  - Recommender systems
  - Market Basket Analysis
  - Spam detection
  - Duplicate document detection
- We will learn various “tools”:
  - Linear algebra (SVD, Rec. Sys., Communities)
  - Optimization (stochastic gradient descent)
  - Dynamic programming (frequent itemsets)
  - Hashing (LSH, Bloom filters)

How It All Fits Together

High dim. data	Graph data	Infinite data	Machine learning	Apps
Locality sensitive hashing	PageRank, SimRank	Filtering data streams	SVM	Recommen der systems
Clustering	Community Detection	Web advertising	Decision Trees	Association Rules
Dimensional ity reduction	Spam Detection	Queries on streams	Perceptron, kNN	Duplicate document detection

How do you want that data?

Big Data Computing Framework

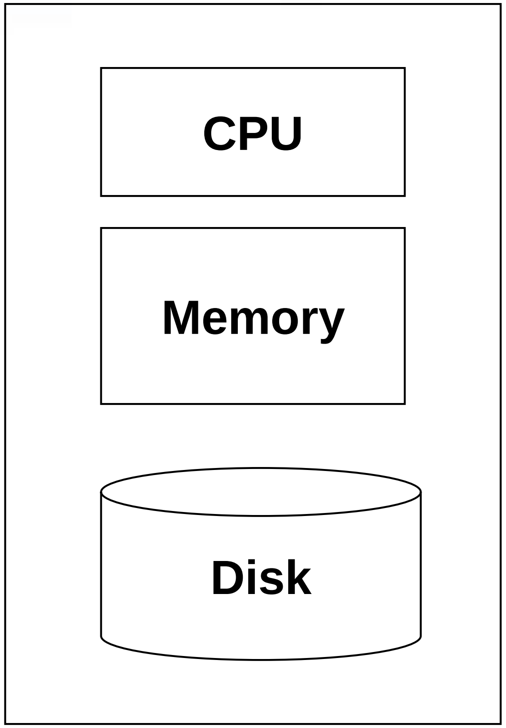
Section 2.1,

"Mining of Massive Datasets, "

Jure Leskovec, Anand Rajaraman,
Jeff Ullman Stanford University

http://www.mmds.org

Single Node Architecture



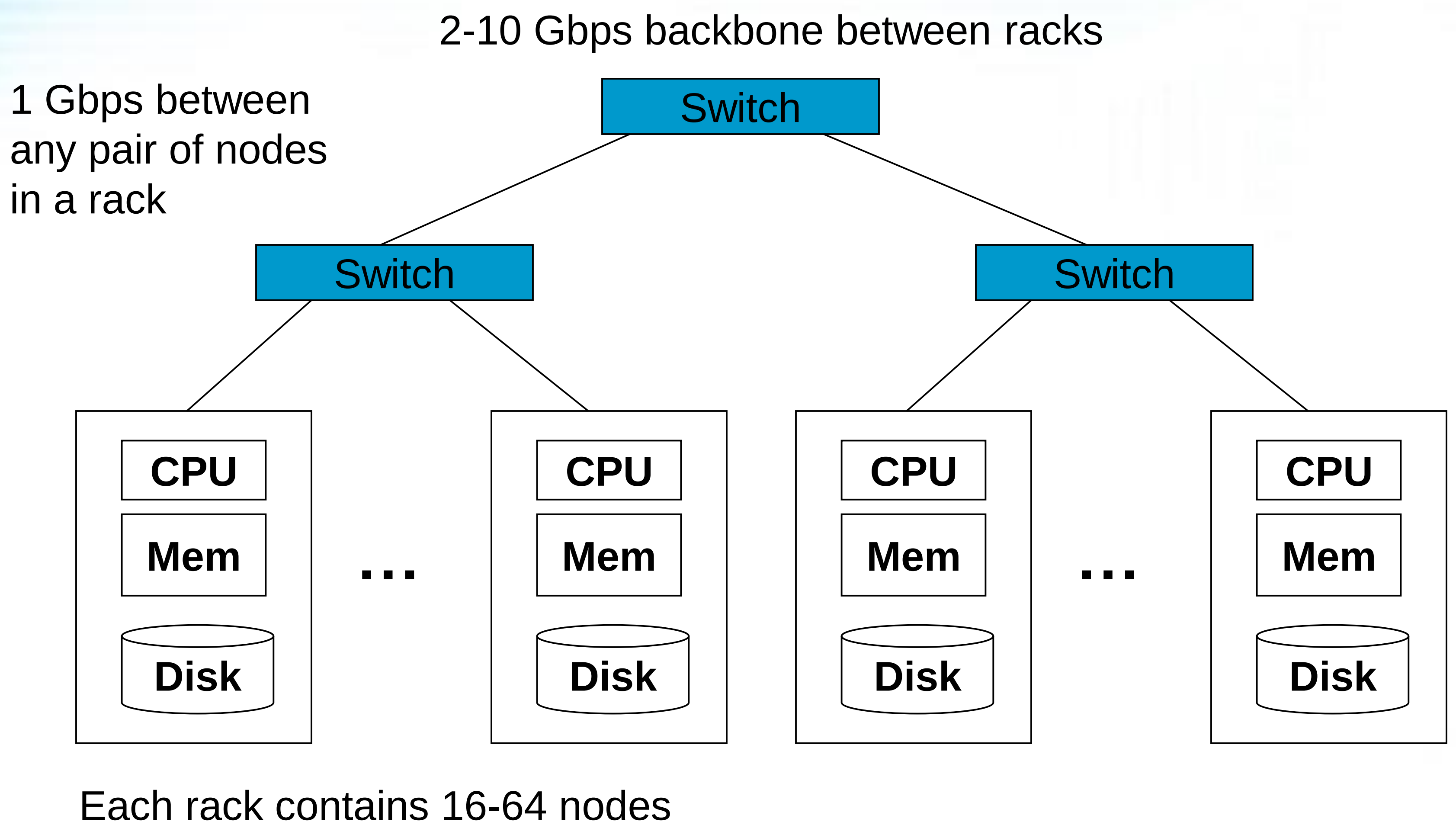
Machine Learning, Statistics

"Classical" Data Mining

Motivation: Google Example

- 20+ billion web pages x 20KB = 400+ TB
- 1 computer reads 30-35 MB/sec from disk
  - ~4 months to read the web
- ~1,000 hard drives to store the web
- Takes even more to do something useful with the data!
- Today, a standard architecture for such probl ems is emerging:
  - Cluster of commodity Linux nodes
  - Commodity network (ethernet) to connect them

Cluster Architecture



In 2011 it was guestimated that Google had 1M machines, http://bit.ly/Shh0RO

Large-scale Computing

- Large-scale computing for data mining problems on commodity hardware
- Challenges:
  - How do you distribute computation?
  - How can we make it easy to write distributed programs?
  - Machines fail:
    - One server may stay up 3 years (1,000 days)
    - If you have 1,000 servers, expect to loose 1/day
    - People estimated Google had ~1M machines in 2011
      - 1,000 machines fail every day!

Idea and Solution

- Issue: Copying data over a network takes time
- Idea:
  - Bring computation close to the data
  - Store files multiple times for reliability
- MapReduce addresses these problems
  - Google's computational/data manipulation model
  - Elegant way to work with big data
  - Storage Infrastructure – File system
    - Google: GFS. Hadoop: HDFS
  - Programming model
    - MapReduce

Distributed File System

Section 2.1,

"Mining of Massive Datasets, "

Jure Leskovec, Anand Rajaraman,

Jeff Ullman Stanford University

http://www.mmds.org

SOONGSIL UNIVERSITY

Storage Infrastructure

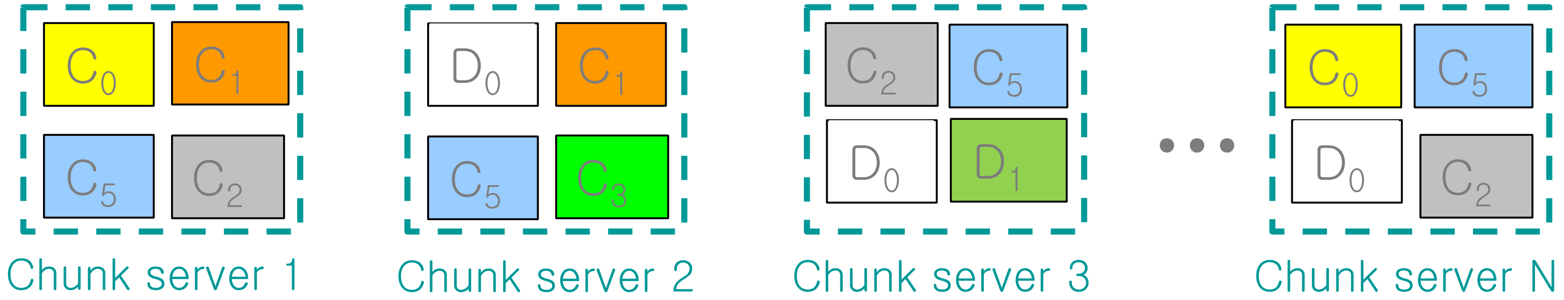
- Problem:
  - If nodes fail, how to store data persistently?
- Answer:
  - Distributed File System:
    - Provides global file namespace
    - Google GFS; Hadoop HDFS;
- Typical usage pattern
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common

Distributed File System

- Chunk servers
  - File is split into contiguous chunks
  - Typically each chunk is 16-64MB
  - Each chunk replicated (usually 2x or 3x)
  - Try to keep replicas in different racks
- Master node
  - a.k.a. Name Node in Hadoop's HDFS
  - Stores metadata about where files are stored
  - Might be replicated
- Client library for file access
  - Talks to master to find chunk servers
  - Connects directly to chunk servers to access d ata

Distributed File System

- Reliable distributed file system
- Data kept in "chunks" spread across machines
- Each chunk replicated on different machines
  - Seamless recovery from disk or machine failure



Bring computation directly to the data!

Chunk servers also serve as compute servers

Distributed File System

- Don’t move data to workers… move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- Why?
  - Not enough RAM to hold all the data in memory
  - Disk access is slow, but disk throughput is reasonable
- A distributed file system is the answer
  - GFS (Google File System) for Google’s MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

The Need for GFS

- Component failures normal
  - Due to clustered computing
- Files are huge
  - By traditional standards (many TB)
- Most mutations are mutations
  - Not random access overwrite
- Co-Designing apps & file system
- Typical: 1000 nodes & 300 TB

Things to be desired

- Must monitor & recover from comp failures
- Modest number of large files
- Workload
  - Large streaming reads + small random reads
  - Many large sequential writes
    - Random access overwrites don't need to be efficient
- Need semantics for concurrent appends
- High sustained bandwidth
  - More important than low latency

GFS: Assumptions

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

GFS: Design Decisions

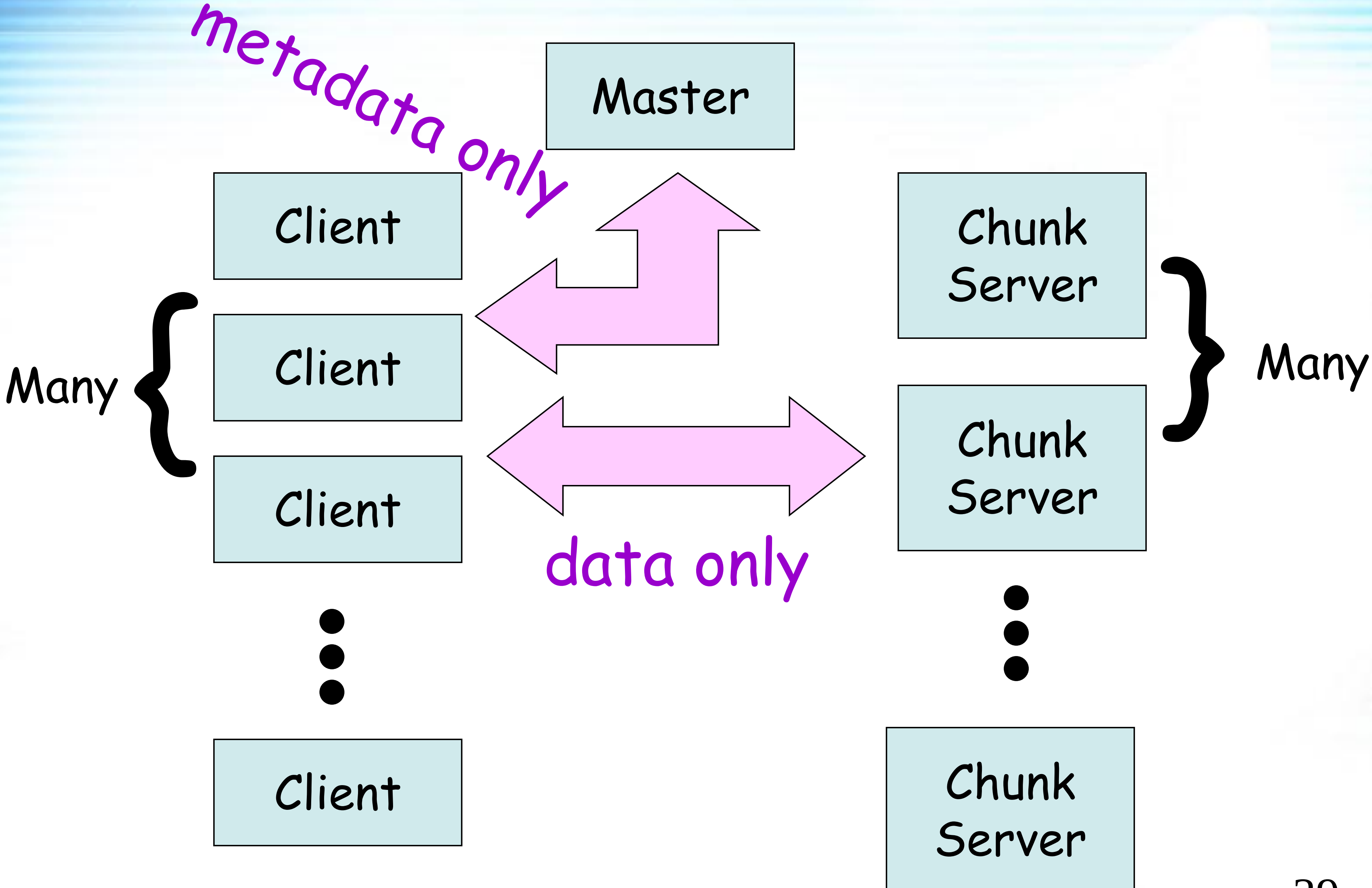
- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ chunkservers
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large datasets, streaming reads
- Simplify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

HDFS = GFS clone (same basic ideas)

Interface

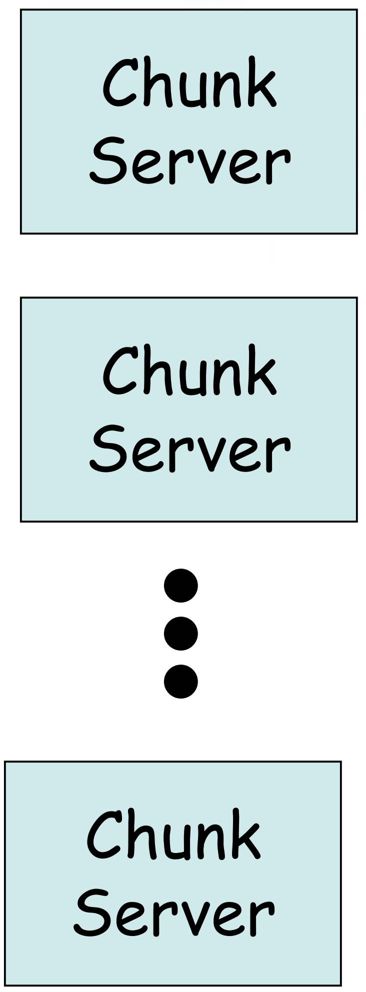
- Familiar ; CRUD
  - Create, Read, Update(write), Delete, Close
- Novel
  - Snapshot
    - Low cost
  - Record append
    - Atomicity with multiple concurrent writes

Architecture



Architecture

- Store all files
  - In fixed-size chucks
    - 64 MB
    - 64 bit unique handle
- Triple redundancy

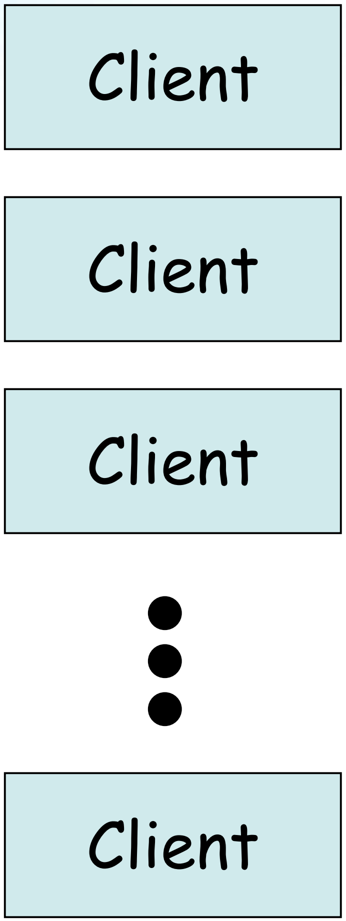


Architecture

Master

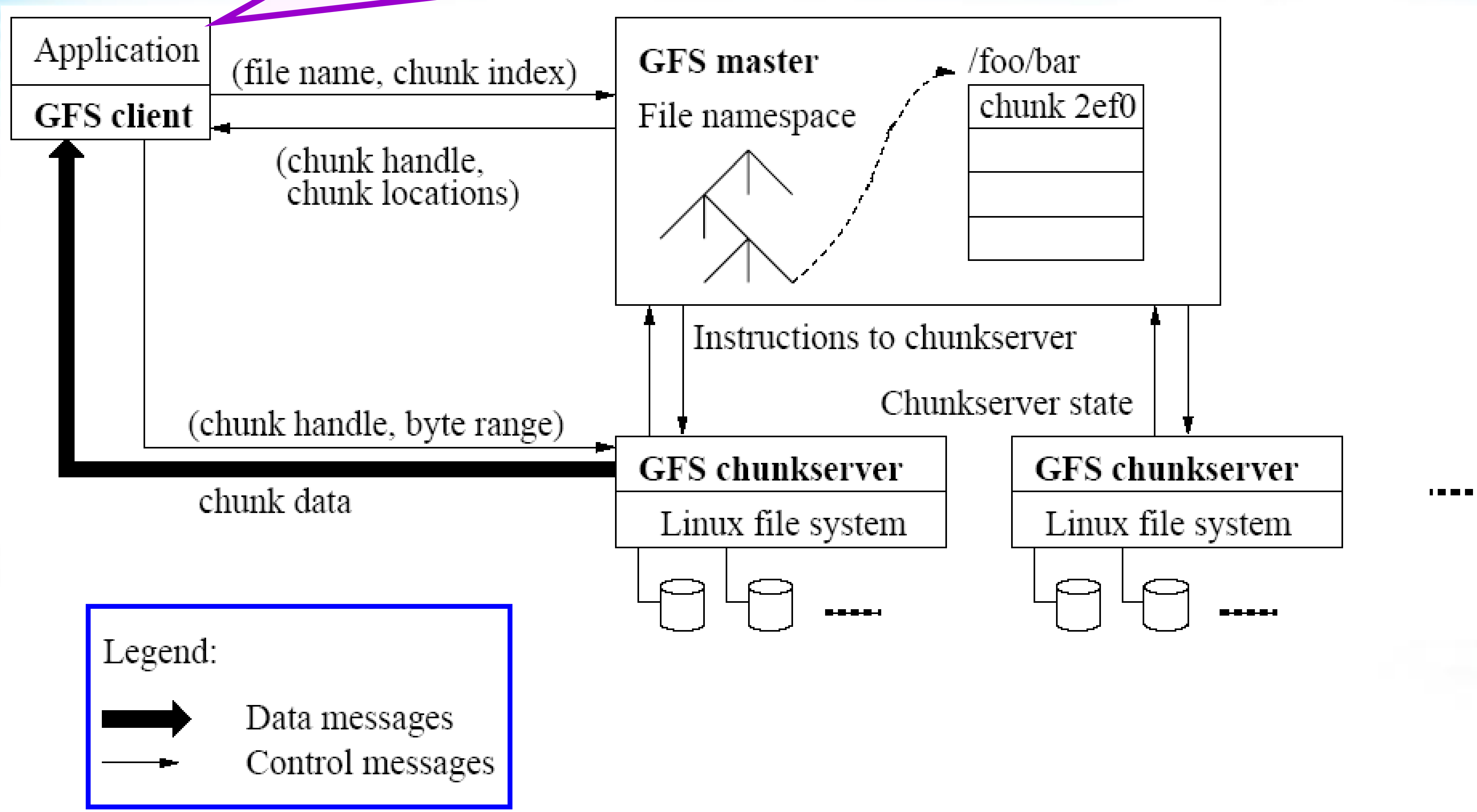
- Stores all metadata
  - Namespace
  - Access-control information
  - Chunk locations
  - 'Lease' management
- Heartbeats
- Having one master → global knowledge
  - Allows better placement / replication
  - Simplifies design

Architecture

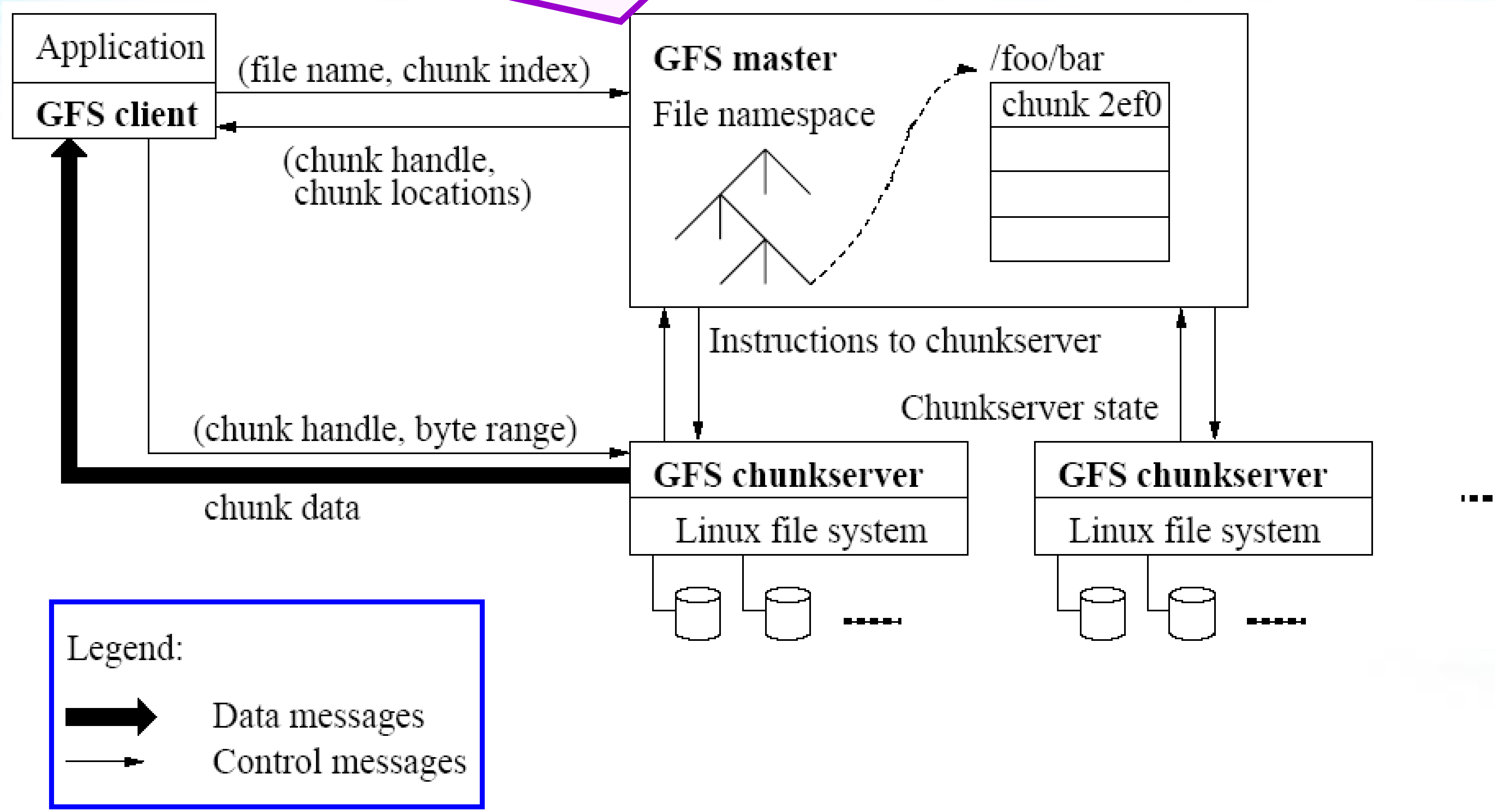


- GFS code implements API
- Cache only metadata

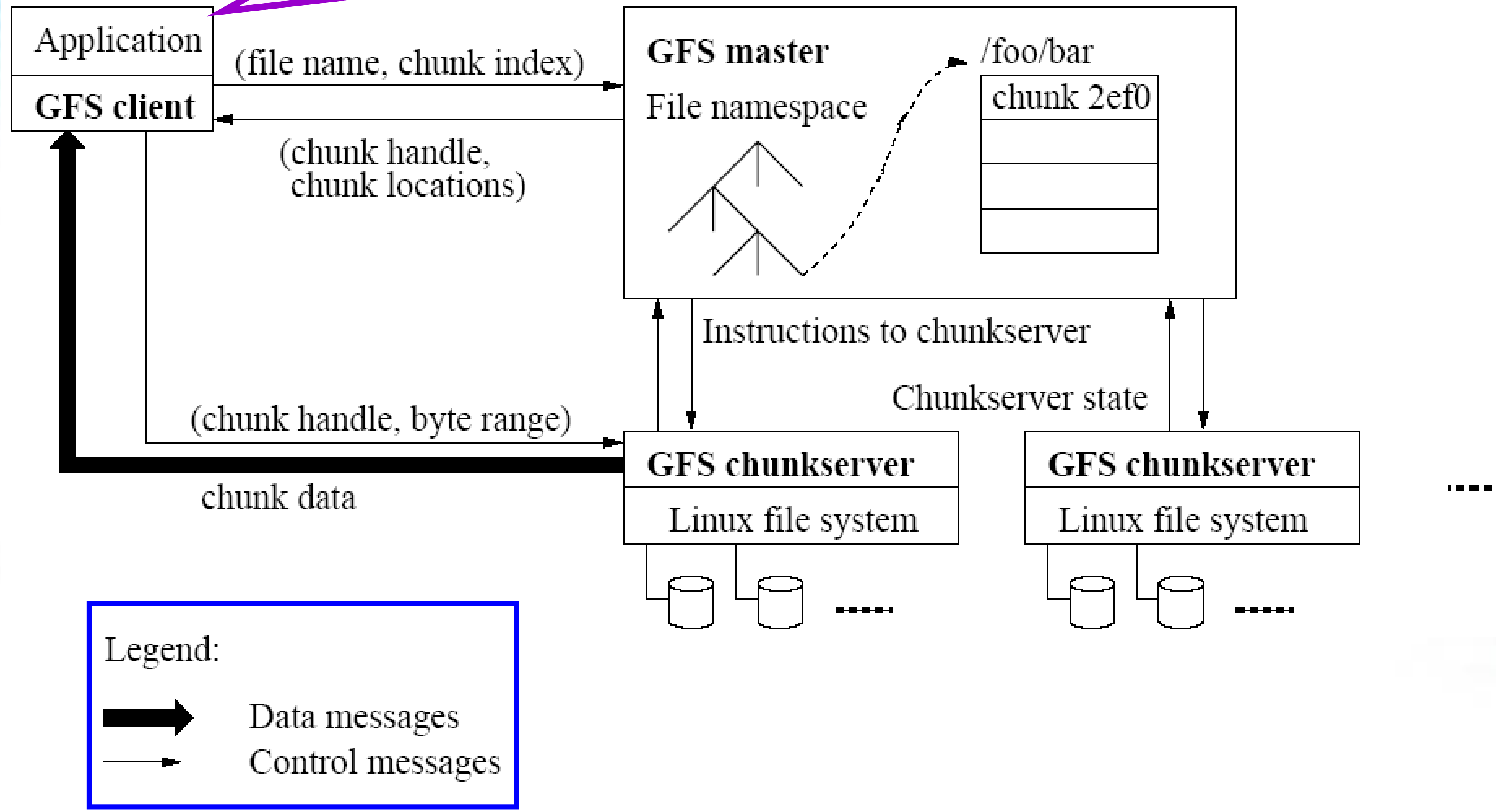
Using fixed chunk size, translate filename & byte offset to chunk index.
Send request to master

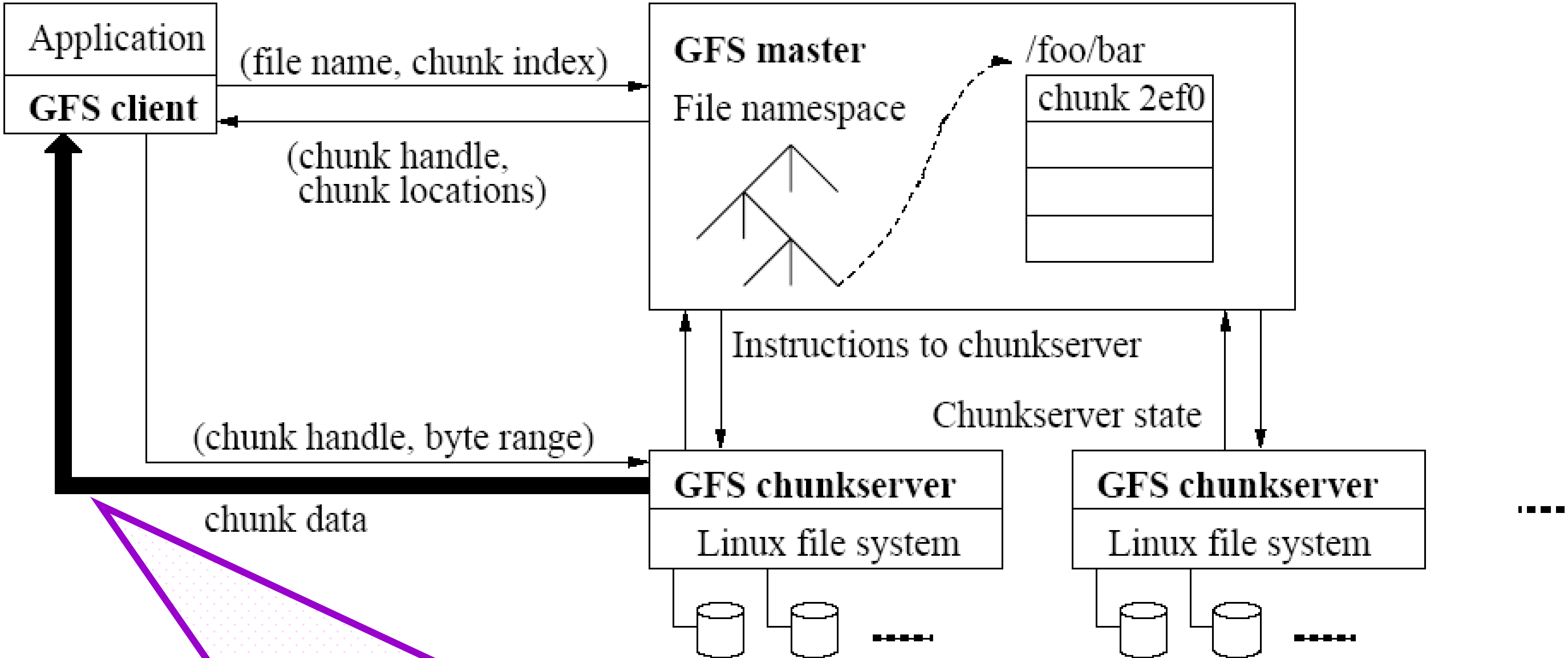


Replies with chunk handle & location of chunkserver replicas (including which is 'primary')



Cache info
using filename & chunk index as key





Request data from nearest chunkserver
"chunkhandle & index into chunk"

Metadata

- Master stores three types
  - File & chunk namespaces
  - Mapping from files → chunks
  - Location of chunk replicas

- Stored in memory

- Kept persistent thru logging

From GFS to HDFS

- Terminology differences:
  - GFS master = Hadoop namenode
  - GFS chunkservers = Hadoop datanodes
- Functional differences:
  - HDFS performance is (likely) slower

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

MapReduce

SOONGSIL UNIVERSITY

Section 2.2,

"Mining of Massive Datasets, "

Jure Leskovec, Anand Rajaraman,

Jeff Ullman Stanford University

http://www.mmds.org

MapReduce

- Much of the course will be devoted to large scale computing for data mining
- Challenges:
  - How to distribute computation?
  - Distributed/parallel programming is hard

- MapReduce addresses all of the above
  - Google's computational/data manipulation model
  - Elegant way to work with big data

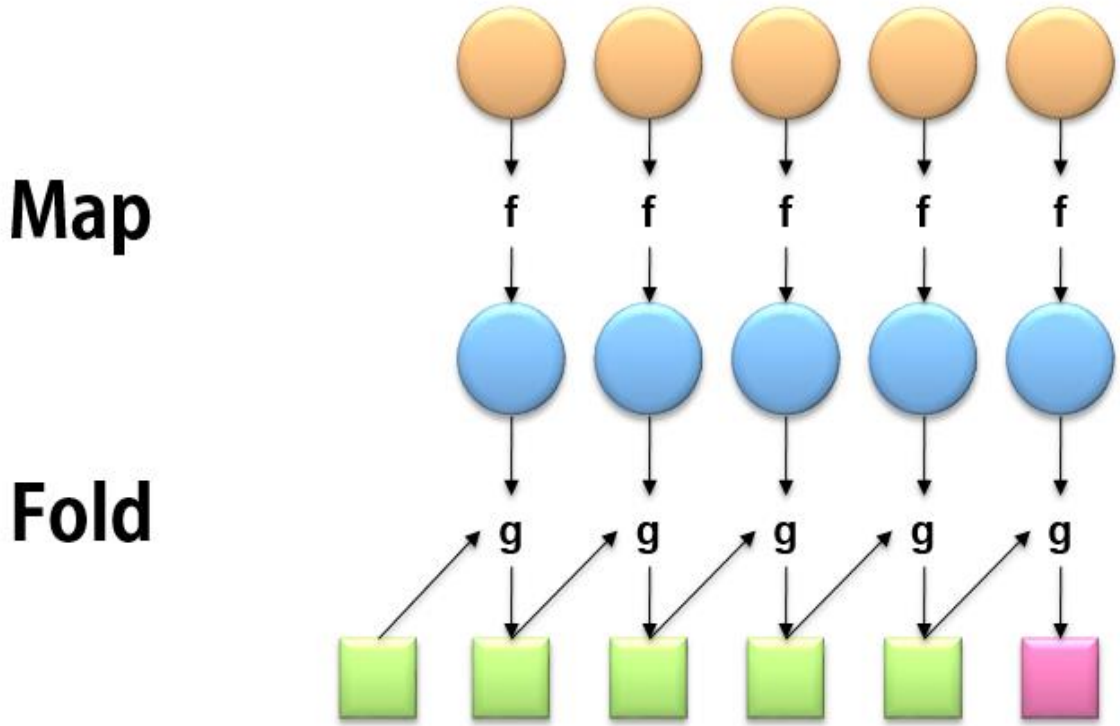
Typical Large-Data Problem

Map

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

Key idea: provide a functional abstraction for these two operations

Roots in Functional Programming



Functional programming + distributed computing!

```
items = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, items))
from functools import reduce
summation=reduce((lambda x, y: x+y), squared)
print(summation)
```

Functional Programming

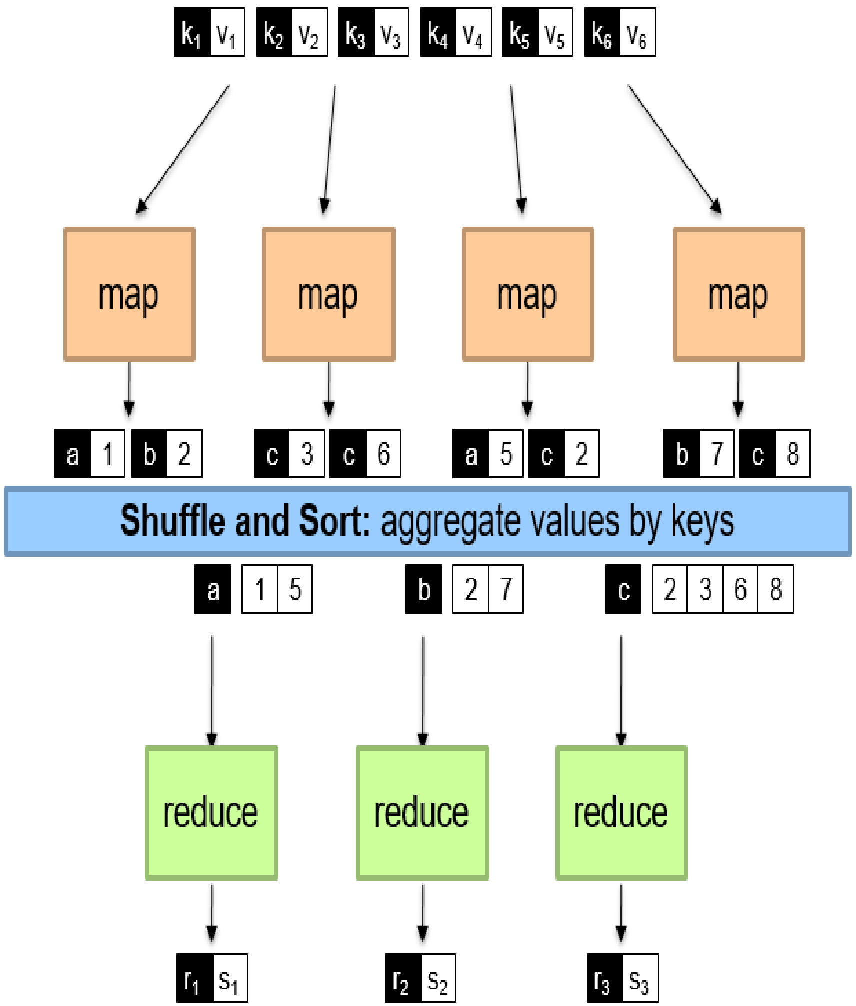
- A programming paradigm, *i.e.* a style of com puter programming that treats computation as the evaluation of mathematical functions
- Emphasizes functions that produce results th at depend only on their inputs and not on the program state
  - *i.e.* pure mathematical functions
- In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* both times

MapReduce

- A programming model for processing large data sets with a parallel, distributed algorithm on a cluster with massively parallel architecture
  - Map() procedure that performs functions such as filtering or sorting to each element of a container(e.g. a list), returning a container of results in the same order.
  - Reduce() procedure that performs a summary operation
    - *e.g.* counting the number of students in each queue, yielding surname frequencies

MapReduce

- Programmers specify two functions:
  - map $(k_1, v_1) \rightarrow [<k_2, v_2>]$
  - reduce $(k_2, [v_2]) \rightarrow [<k_3, v_3>]$
    - All values with the same key are sent to the same reducer
- The execution framework handles everything else…



Task: Word Count

Case 1:

- File too large for memory, but all <word, count> pairs fit in memory

Case 2:

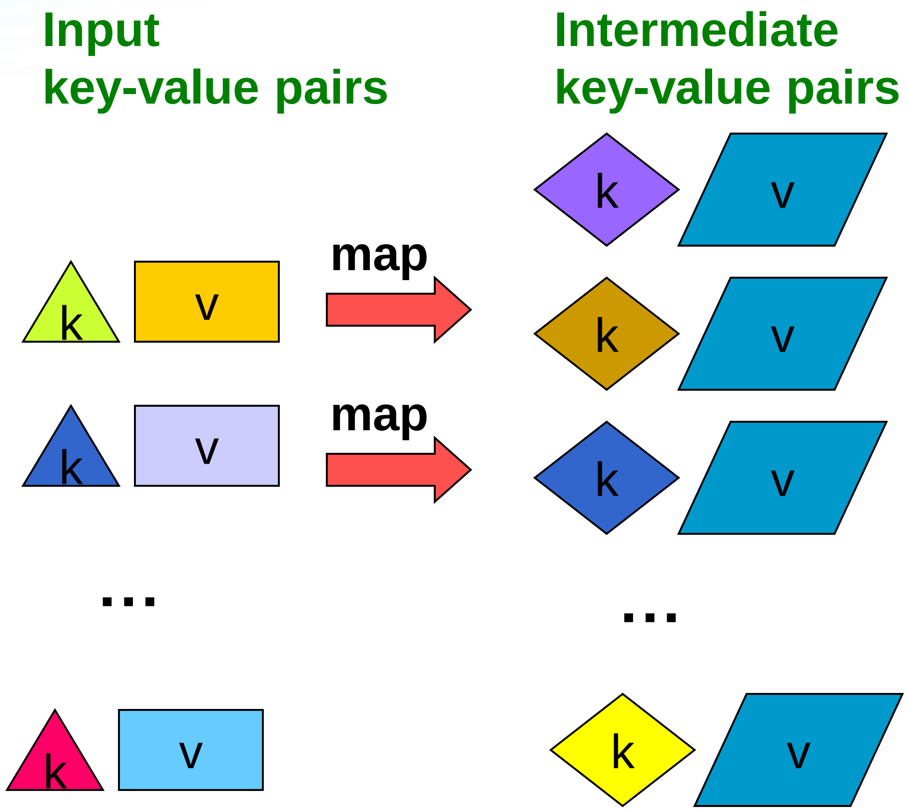
- Count occurrences of words:
  - `words(doc.txt) | sort | uniq -c`
    - where `words` takes a file and outputs the words in it, one per a line
- Case 2 captures the essence of MapReduce
  - Great thing is that it is naturally parallelizable

MapReduce: Overview

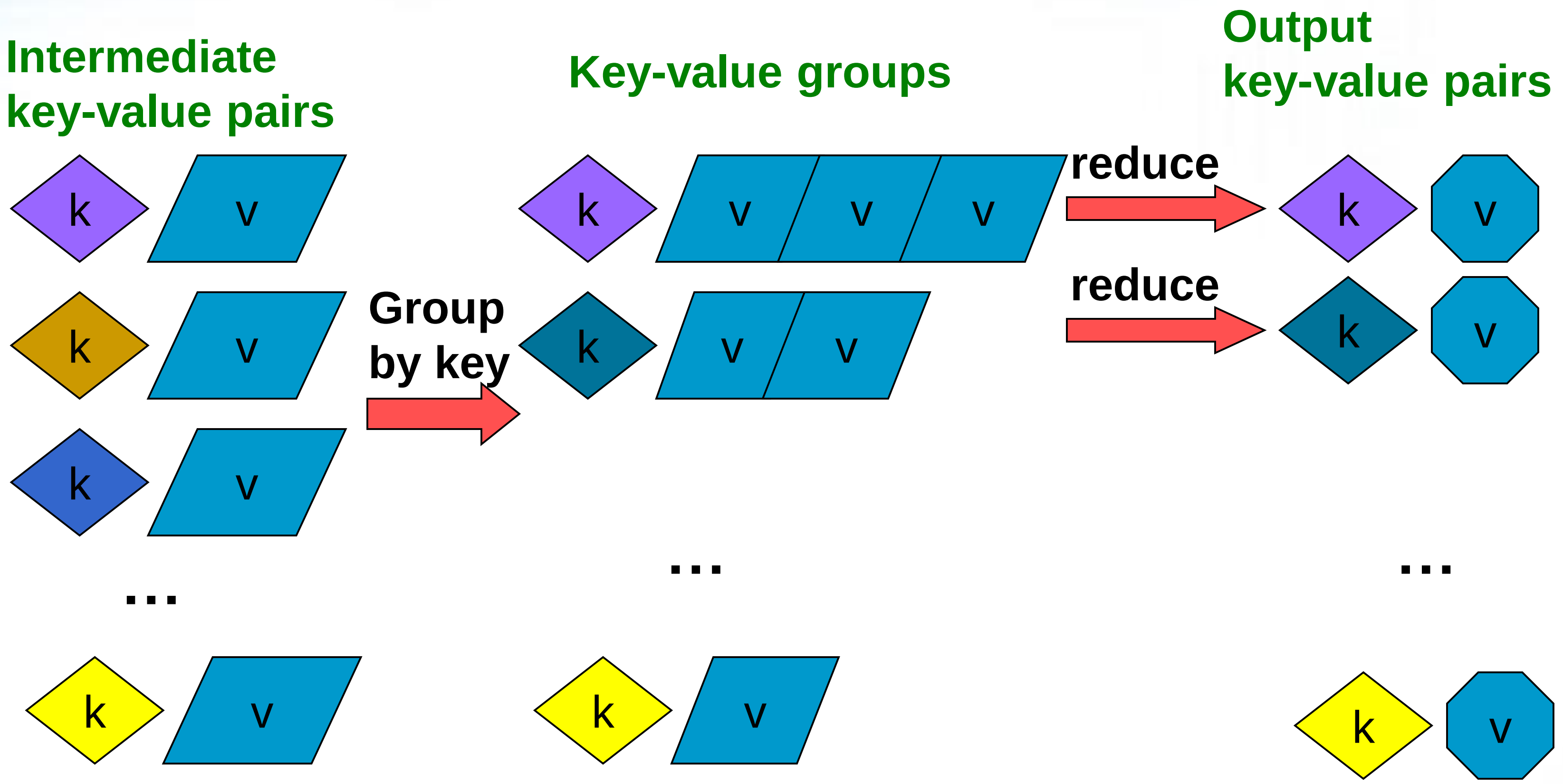
- Sequentially read a lot of data
- Map:
  - Extract something you care about
- Group by key: Sort and Shuffle
- Reduce:
  - Aggregate, summarize, filter or transform
- Write the result

Outline stays the same, **Map** and **Reduce** change to fit the problem

MapReduce: The Map Step



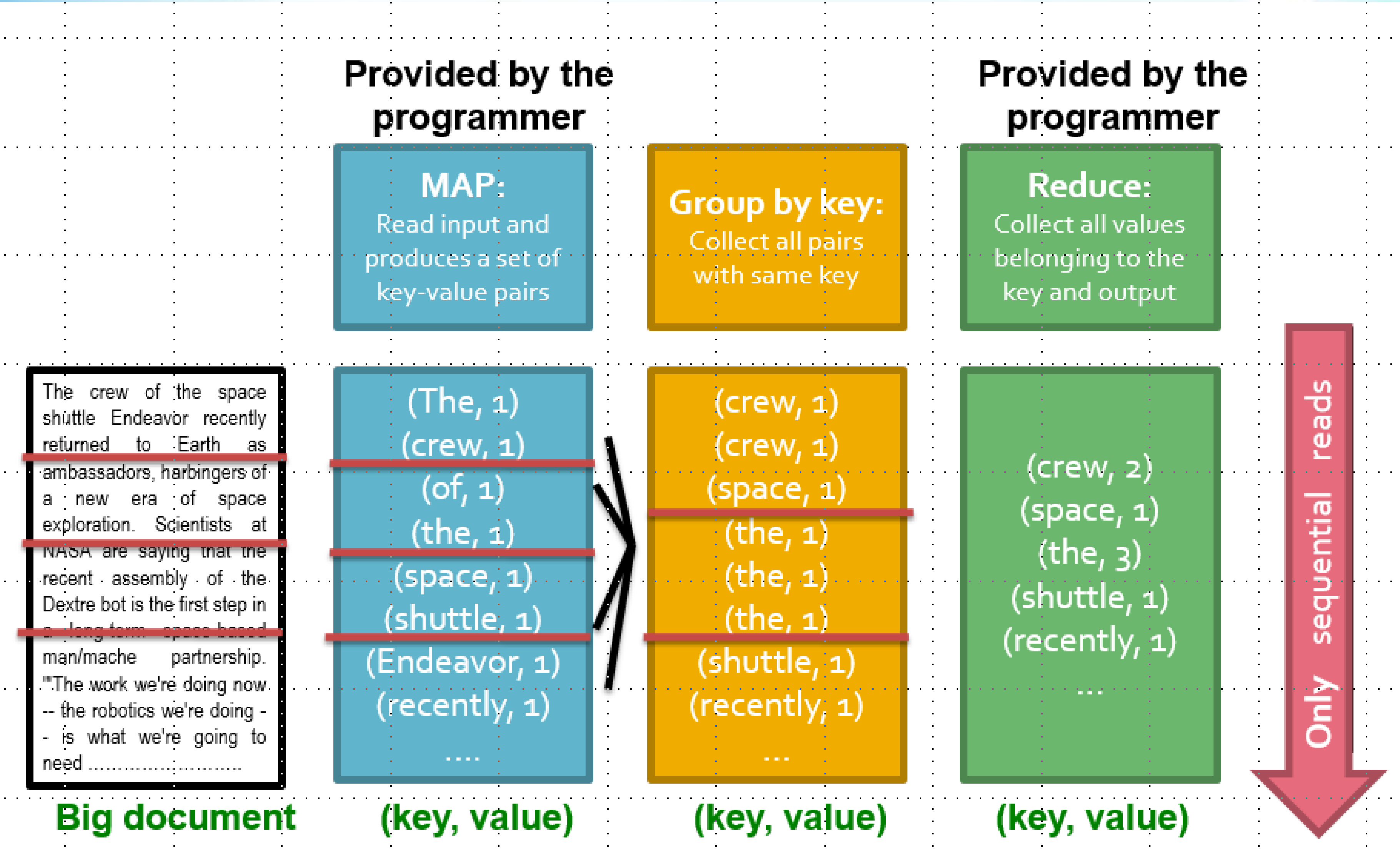
MapReduce: The Reduce Step



More Specifically

- Input: a set of key-value pairs
- Programmer specifies two methods:
  - **Map(k, v)** → <k', v'>*
    - Takes a key-value pair and outputs a set of key-value pairs
      - E.g., key is the filename, value is a single line in the file
    - There is one Map call for every *(k,v)* pair
  - **Reduce(k', <v'>*)** → <k', v''>*
    - All values *v'* with same key *k'* are reduced together and processed in *v'* order
    - There is one Reduce function call per unique key *k'*

MapReduce: Word Counting



Word Count Using MapReduce

```
map(key, value):
// key: document name; value: text of the document
  for each word w in value:
      emit(w, 1)


reduce(key, values):
// key: a word; value: an iterator over counts
      result = 0
      for each count v in values:
            result += v
      emit(key, result)
```

MapReduce: Environment

MapReduce environment takes care of:

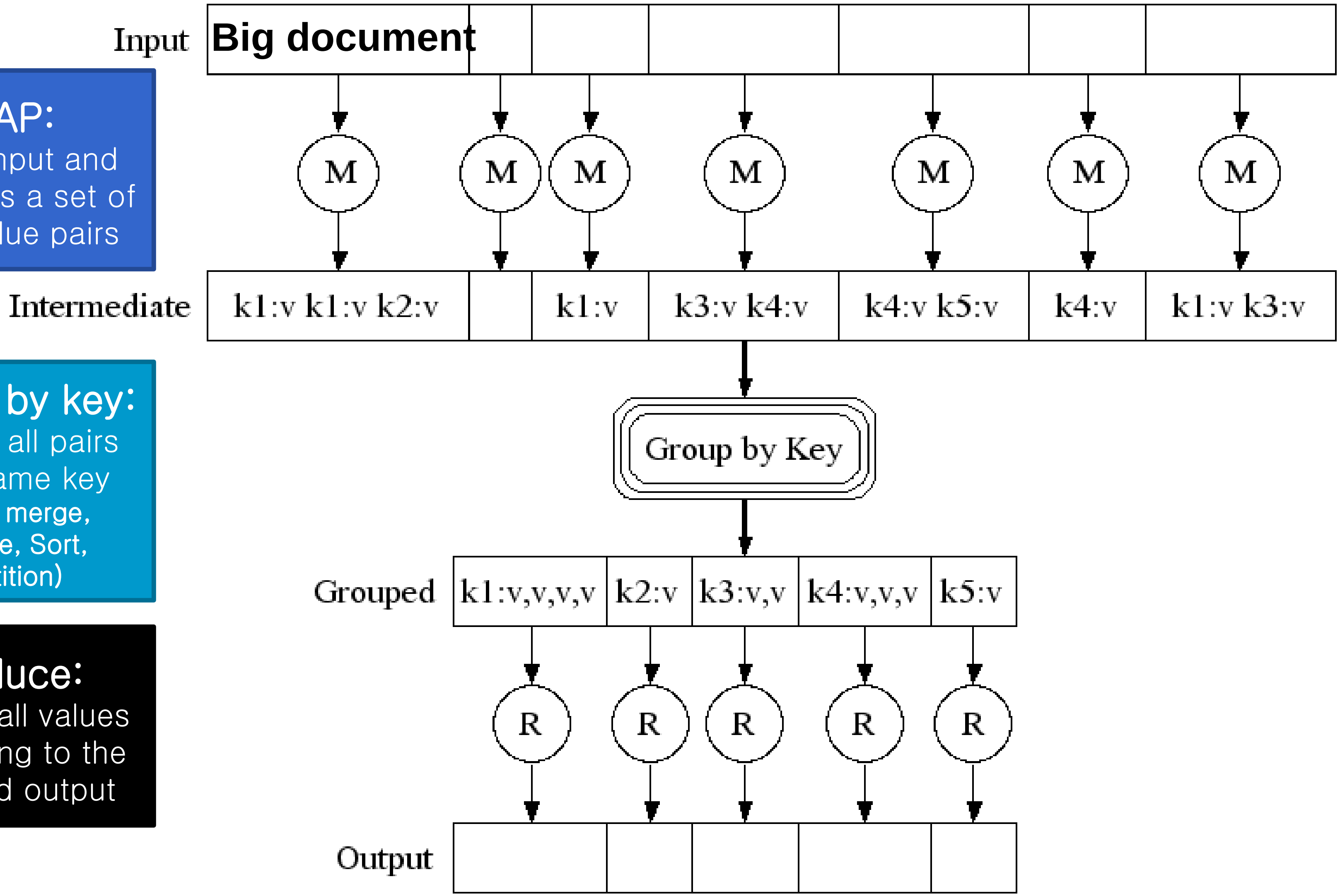
- Partitioning the input data
- Scheduling the program's execution across a set of machines
- Performing the group by key step
- Handling machine failures
- Managing required inter-machine communication

MapReduce: A diagram

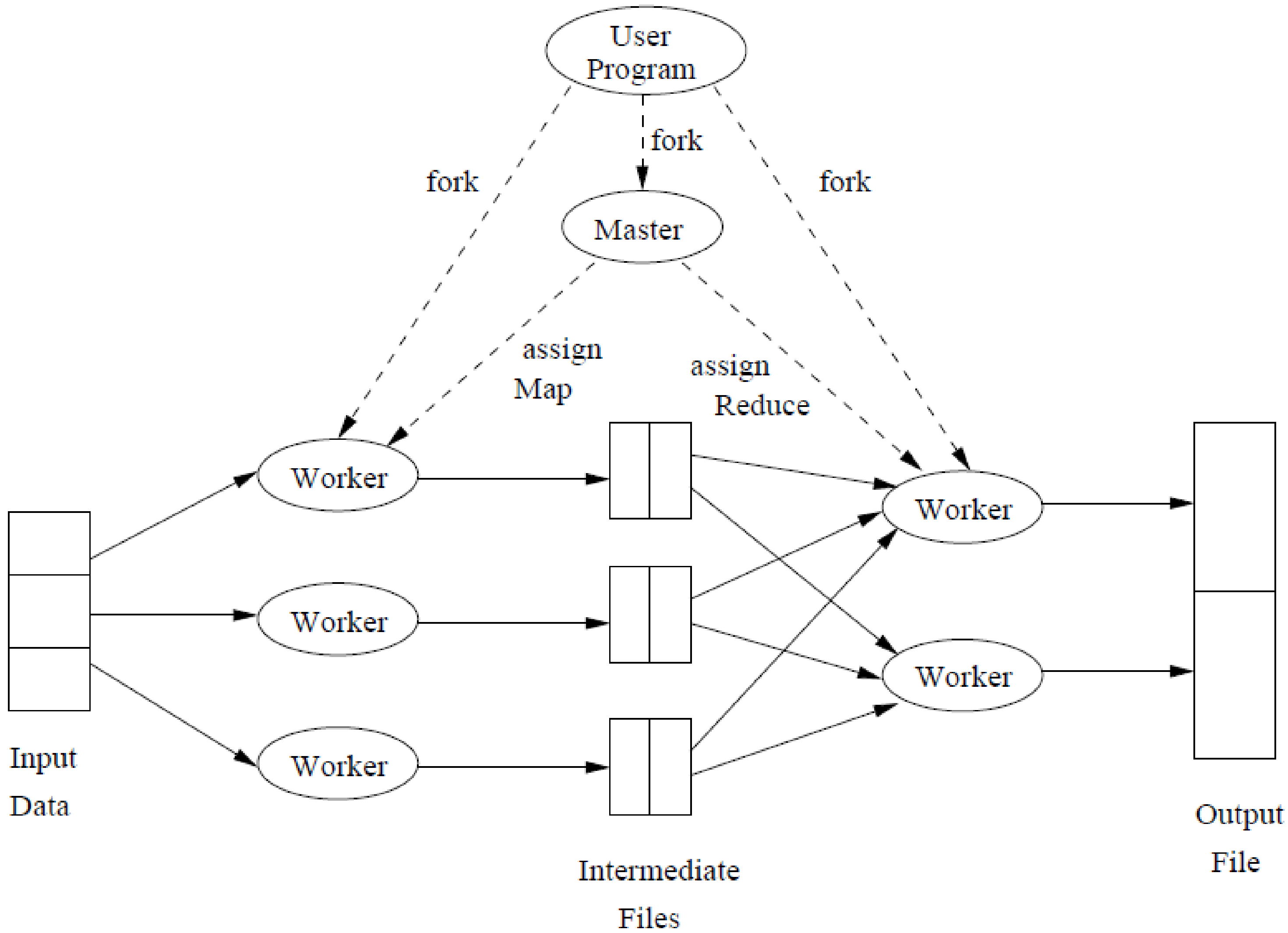
MAP:
Read input and produces a set of key-value pairs

Group by key:
Collect all pairs with same key
(Hash merge, Shuffle, Sort, Partition)

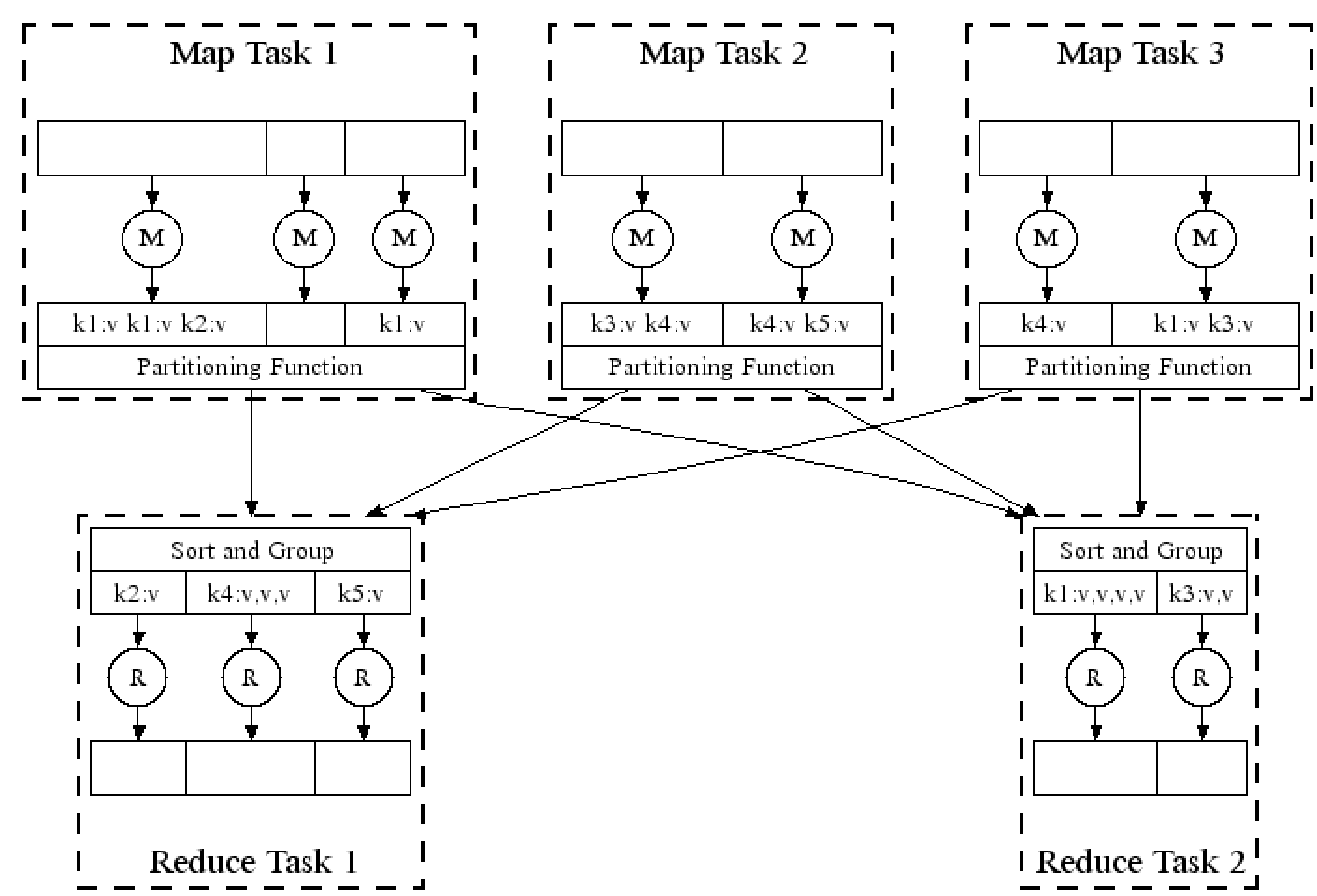
Reduce:
Collect all values belonging to the key and output



Overview of the execution of a MapReduce program

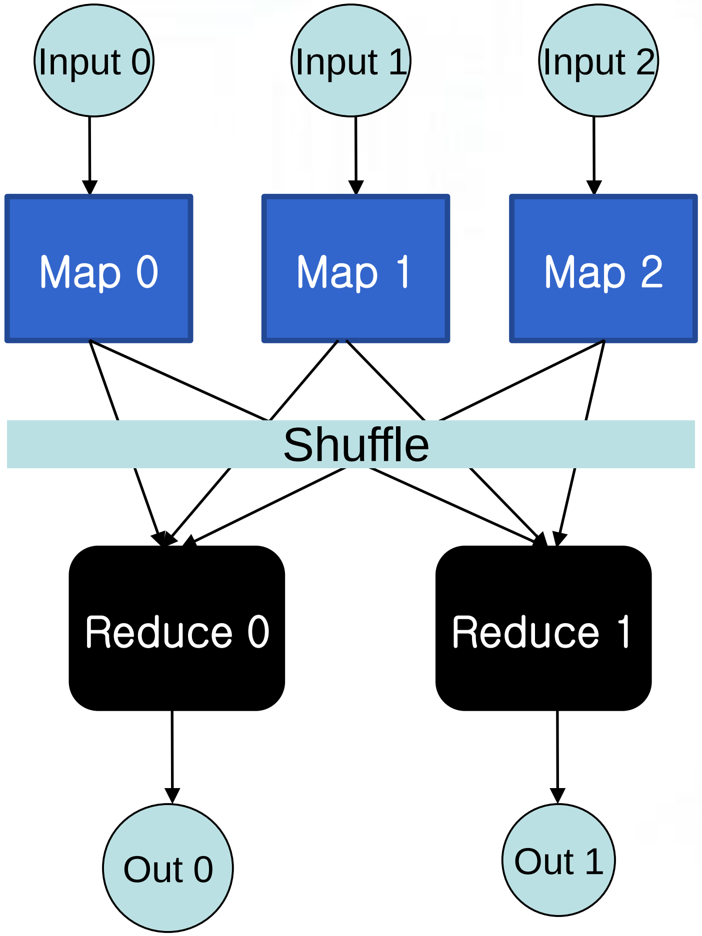


MapReduce: In Parallel



MapReduce

- Programmer specifies:
  - Map and Reduce and input files
- Workflow:
  - Read inputs as a set of key-value-p airs
  - Map transforms input kv-pairs into a new set of k'v'-pairs
  - Sorts & Shuffles the k'v'-pairs to out put nodes
  - All k'v'-pairs with a given k' are sent to the same **reduce**
  - **Reduce** processes all k'v'-pairs grou ped by key into new k''v''-pairs
  - Write the resulting pairs to files
- All phases are distributed with man y tasks doing the work



Data Flow

- Input and final output are stored on a distributed file system (FS):
  - Scheduler tries to schedule map tasks "close" to physical storage location of input data
- Intermediate results are stored on local FS of Map and Reduce workers
- Output is often input to another MapReduce task

Coordination: Master

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

Dealing with Failures

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

How many Map and Reduce jobs?

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

Task Granularity & Pipelining

- Fine granularity tasks: map tasks >> machines
  - Minimizes time for fault recovery
  - Can do pipeline shuffling with map execution
  - Better dynamic load balancing

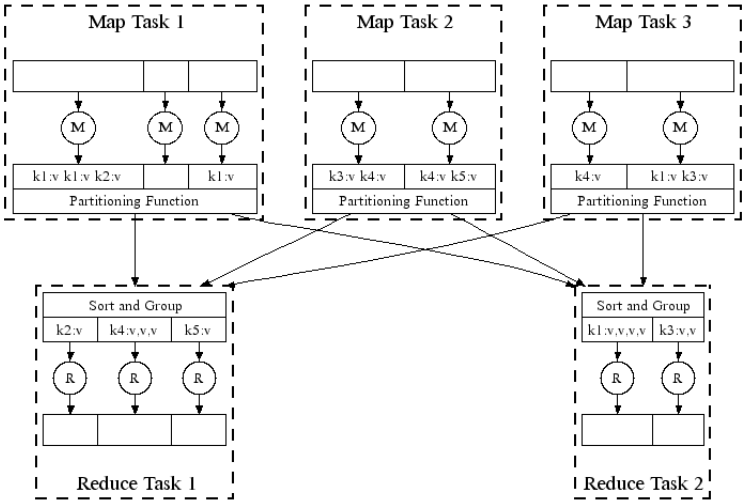
Process	Time ---------------------->
User Program	MapReduce() ... wait ...
Master	Assign tasks to worker machines...
Worker 1	Map 1 | Map 3
Worker 2	Map 2
Worker 3	Read 1.1 | Read 1.3 | Read 1.2 | Reduce 1
Worker 4	Read 2.1 | Read 2.2 | Read 2.3 | Reduce 2

Refinements: Backup Tasks

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

Refinement: Combiners

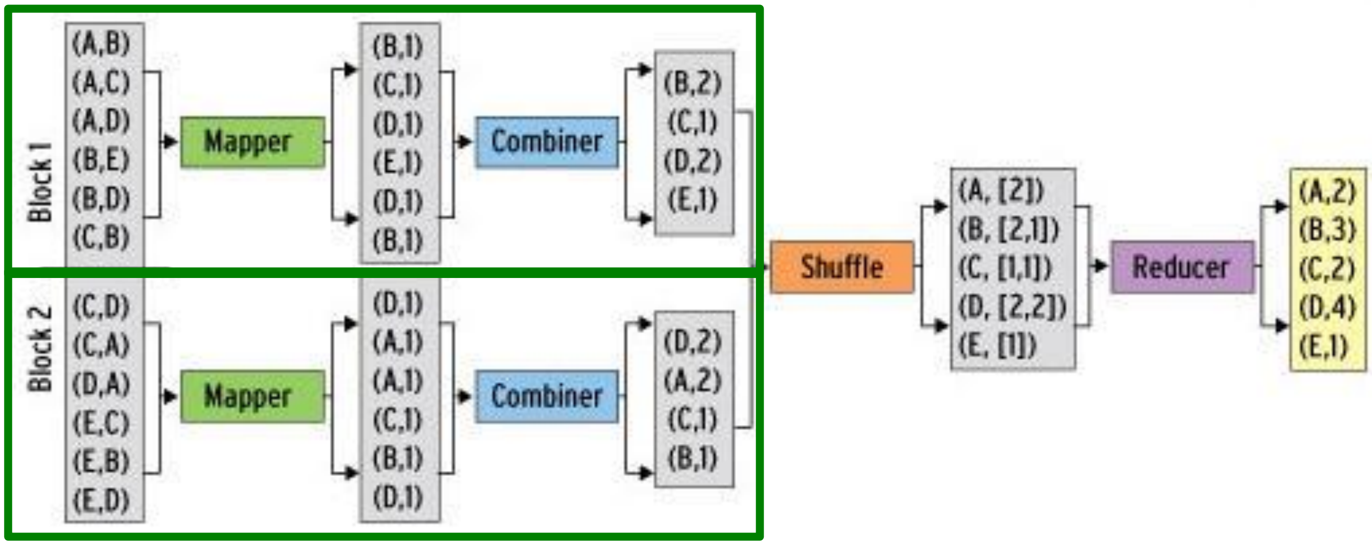
- Often a Map task will produce many pairs of the form (k,v_1), (k,v_2), … for the same key k
  - E.g., popular words in the word count example
- Can save network time by



  - combine(k, list(v_1)) → v_2
  - Combiner is usually same as the reduce function
- Works only if reduce function is commutative and associative

Refinement: Combiners

- Back to our word counting example:
  - Combiner combines the values of all keys of a single mapper (single machine):



  - Much less data needs to be copied and shuffled!

Refinement: Partition Function

- Want to control how keys get partitioned
  - Inputs to map tasks are created by contiguous splits of input file
  - Reduce needs to ensure that records with the same intermediate key end up at the same worker
- System uses a default partition function:
  - **hash(key) mod *R***

- Sometimes useful to override the hash function:
  - E.g., **hash(hostname(URL)) mod *R*** ensures URLs from a host end up in the same output file

Introduction to Apache Hadoop

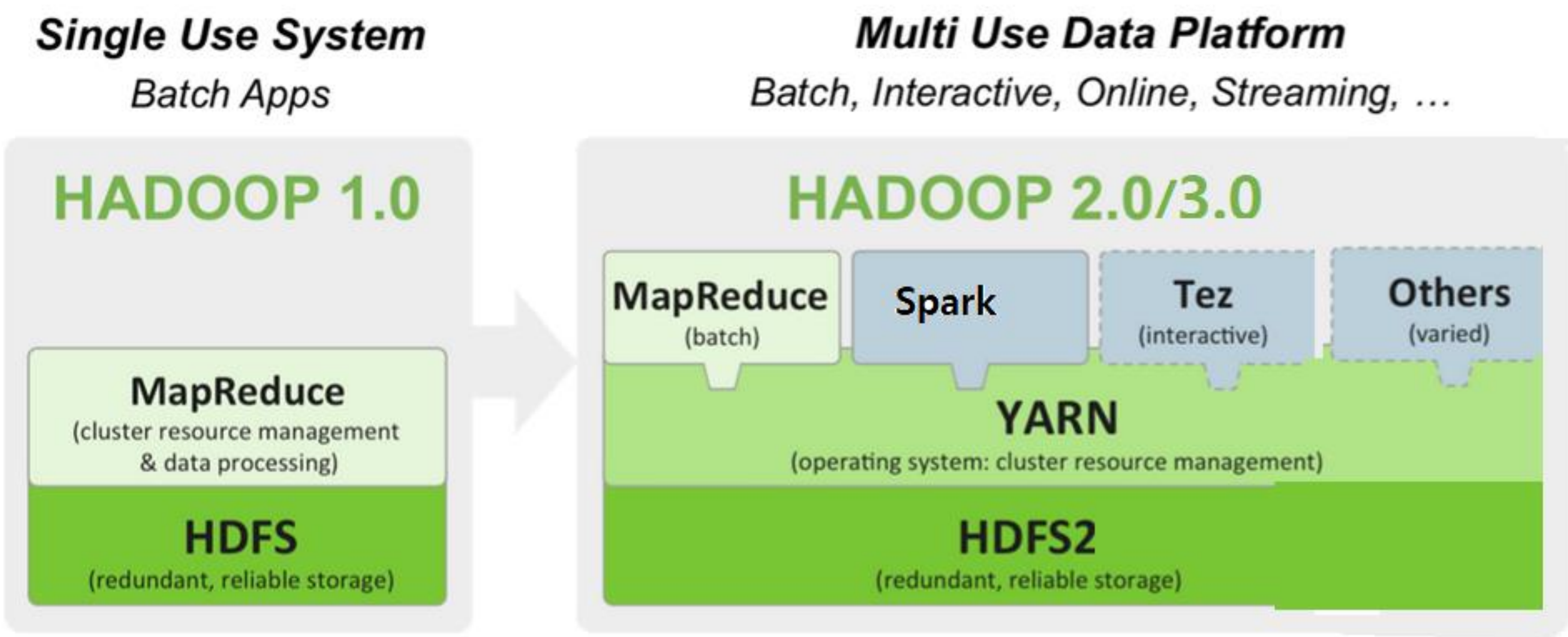
SOONGSIL UNIVERSITY

Apache Hadoop

- Scalable fault-tolerant distributed system for Big Data: Current version 3.2.1
  - Data Storage
  - Data Processing
  - A virtual Big Data machine
  - Borrowed concepts/Ideas from Google; Open source under the Apache license

- Core Hadoop has the following main systems:
  - *Hadoop Common* – contains libraries and utilities needed by other Hadoop modules;
  - *Hadoop Distributed File System (HDFS)* – a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster;
  - *Hadoop YARN* – (introduced in 2012) a platform responsible for managing computing resources in clusters and using them for scheduling users' applications;[10][11]
  - *Hadoop MapReduce* – an implementation of the MapReduce programming model for large-scale data processing.

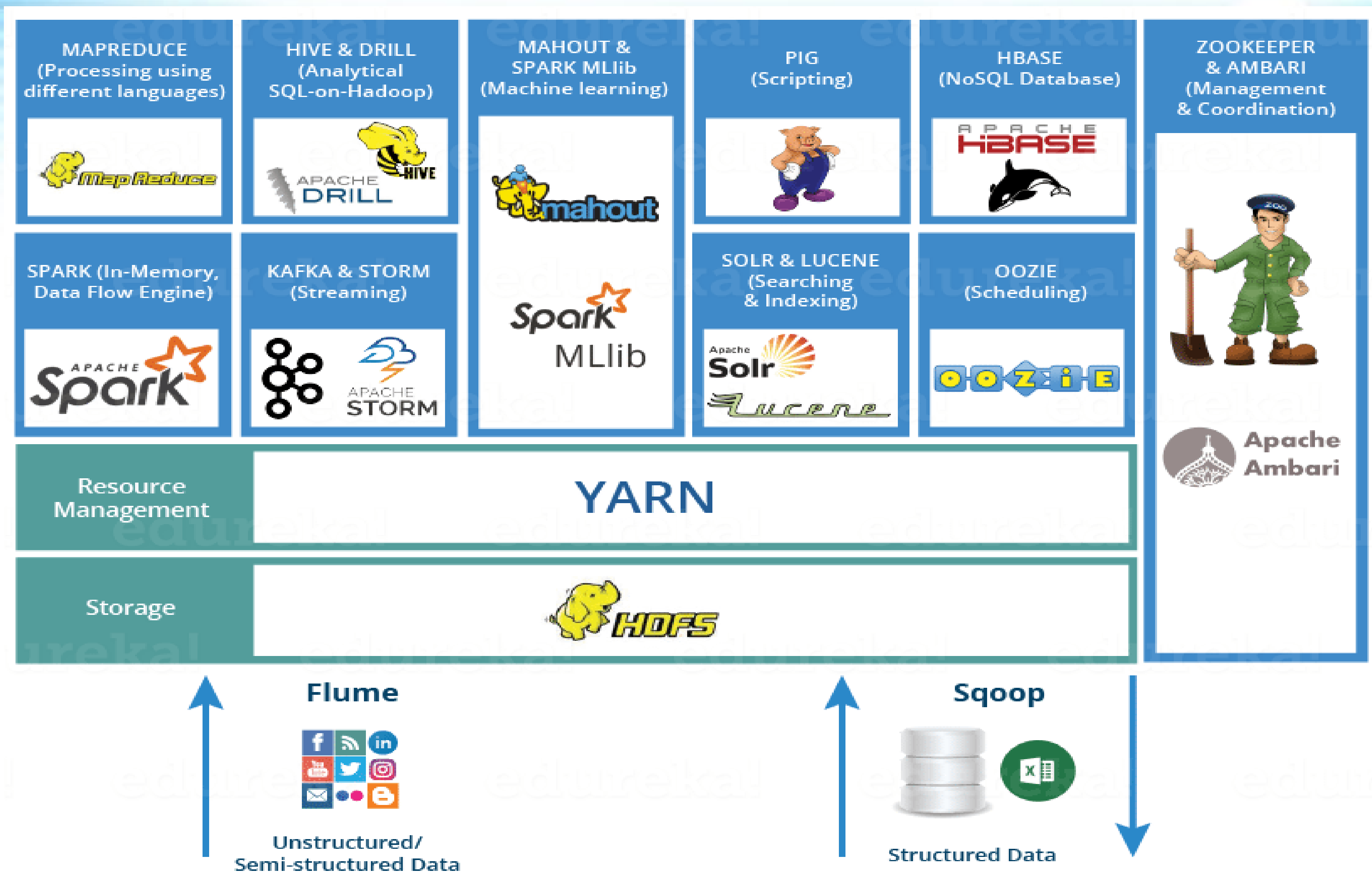
Hadoop 1st vs 2nd/3rd



Hadoop 2.x vs 3.x

Features	Hadoop 2.x	Hadoop 3.x
Min Java Version Required	Java 7	Java 8
Fault Tolerance	Via replication	Via erasure coding
Storage Scheme	3x replication factor for data reliability, 200% overhead	Erasure coding for data reliability, 50% overhead
Yarn Timeline Service	Scalability issues	Highly scalable and reliable
Standby NN	Supports only 1 SBNN	Supports only 2 or more SBNN
Heap Management	We need to configure HADOOP_HEAPSIZE	Provides auto-tuning of heap

Hadoop Ecosystem (Hadoop 2&3)



Hadoop: MapReduce

Chapter 2. MapReduce,

"Hadoop, The Definitive Guide," 4th Ed.,

Tom White, O'Reilly

MapReduce Data Flow

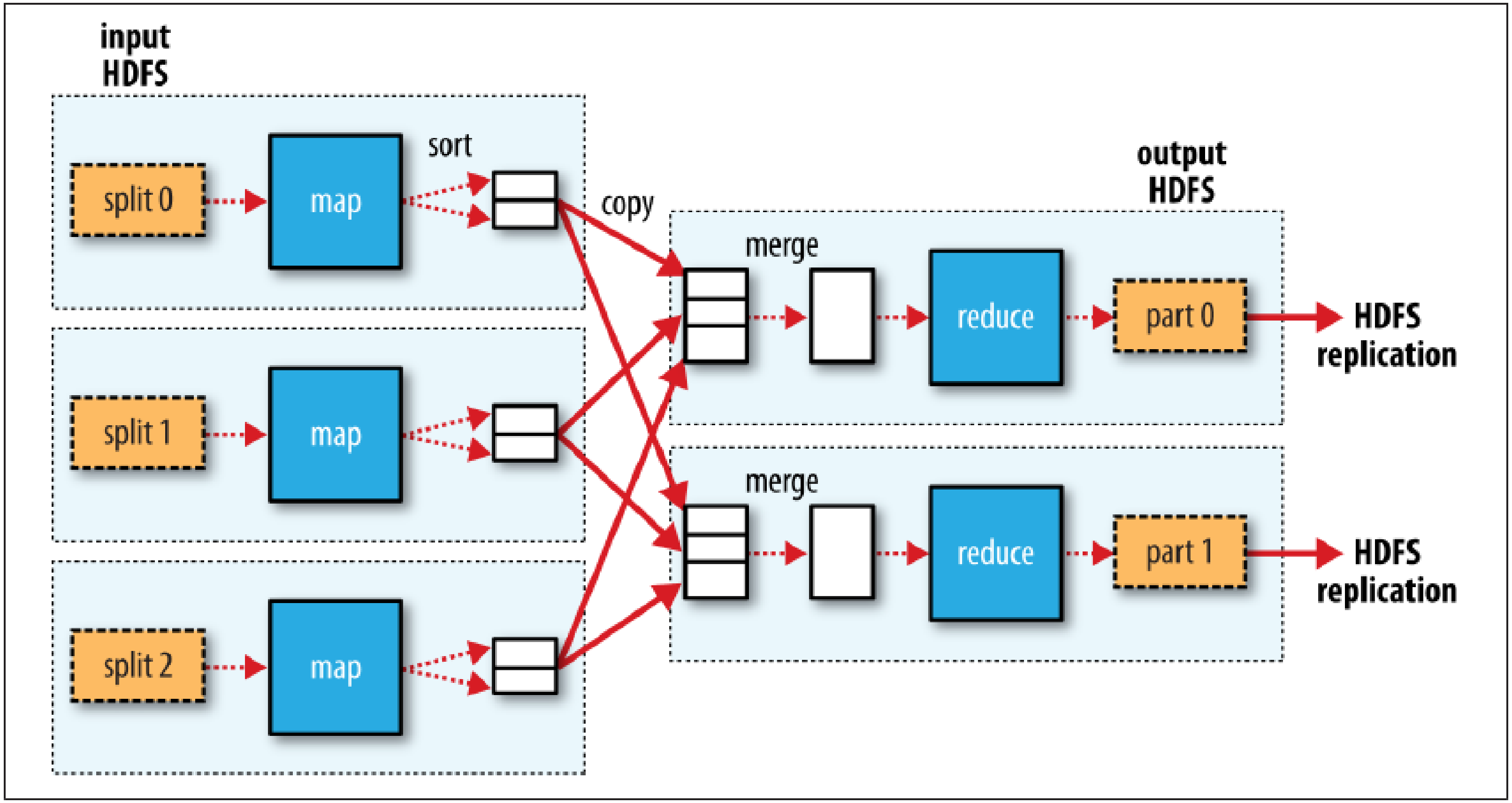
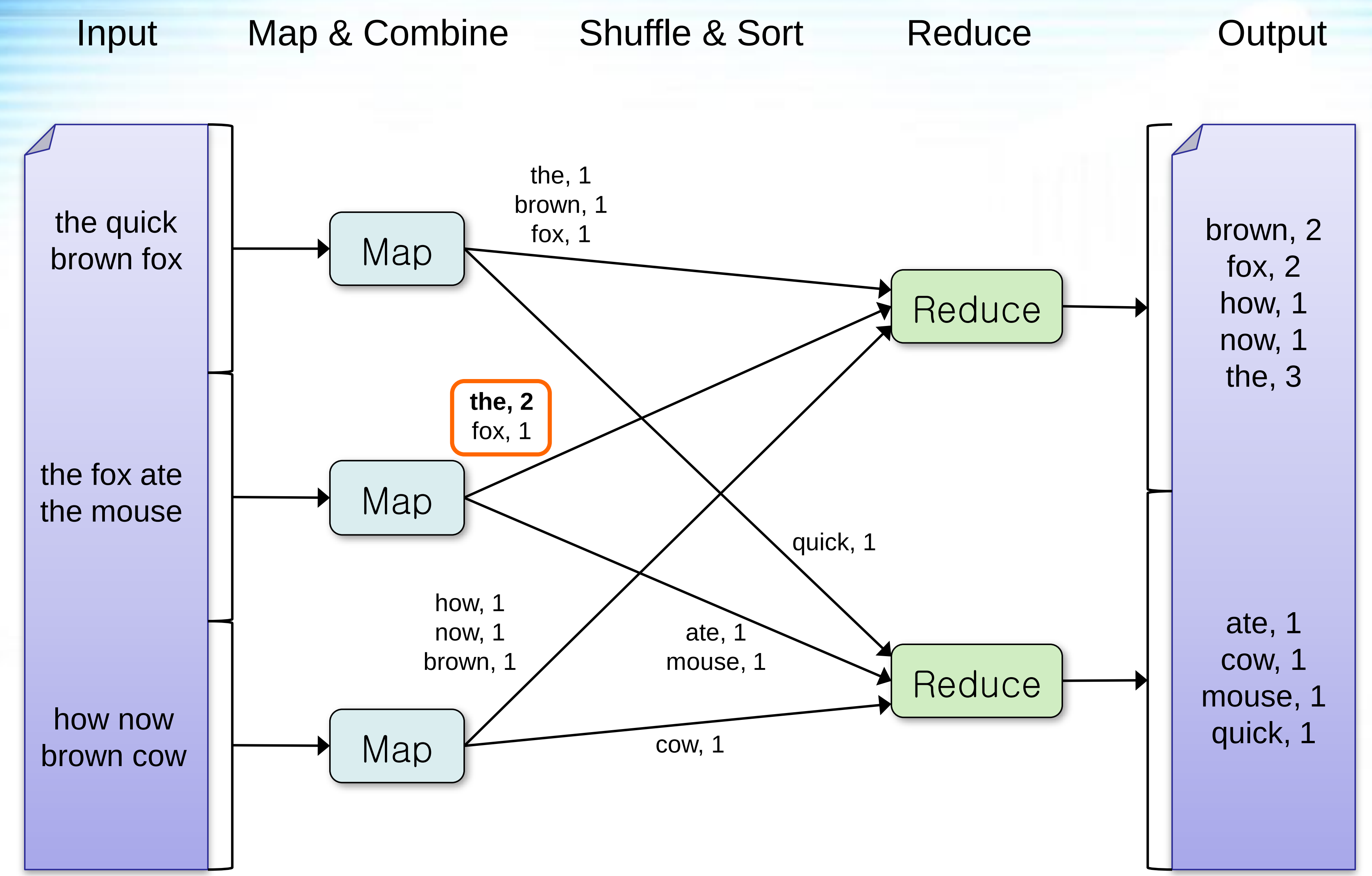


Figure 2-4. MapReduce data flow with multiple reduce tasks

Figure is from Hadoop, The Definitive Guide, 4th Edition, Tom White, O'Reilly

Word Count with Combiner



HDFS: Hadoop File System

Chapter 3. The Hadoop Distributed Filesystem, "Hadoop, The Definitive Guide," 4th Ed., Tom White, O'Reilly

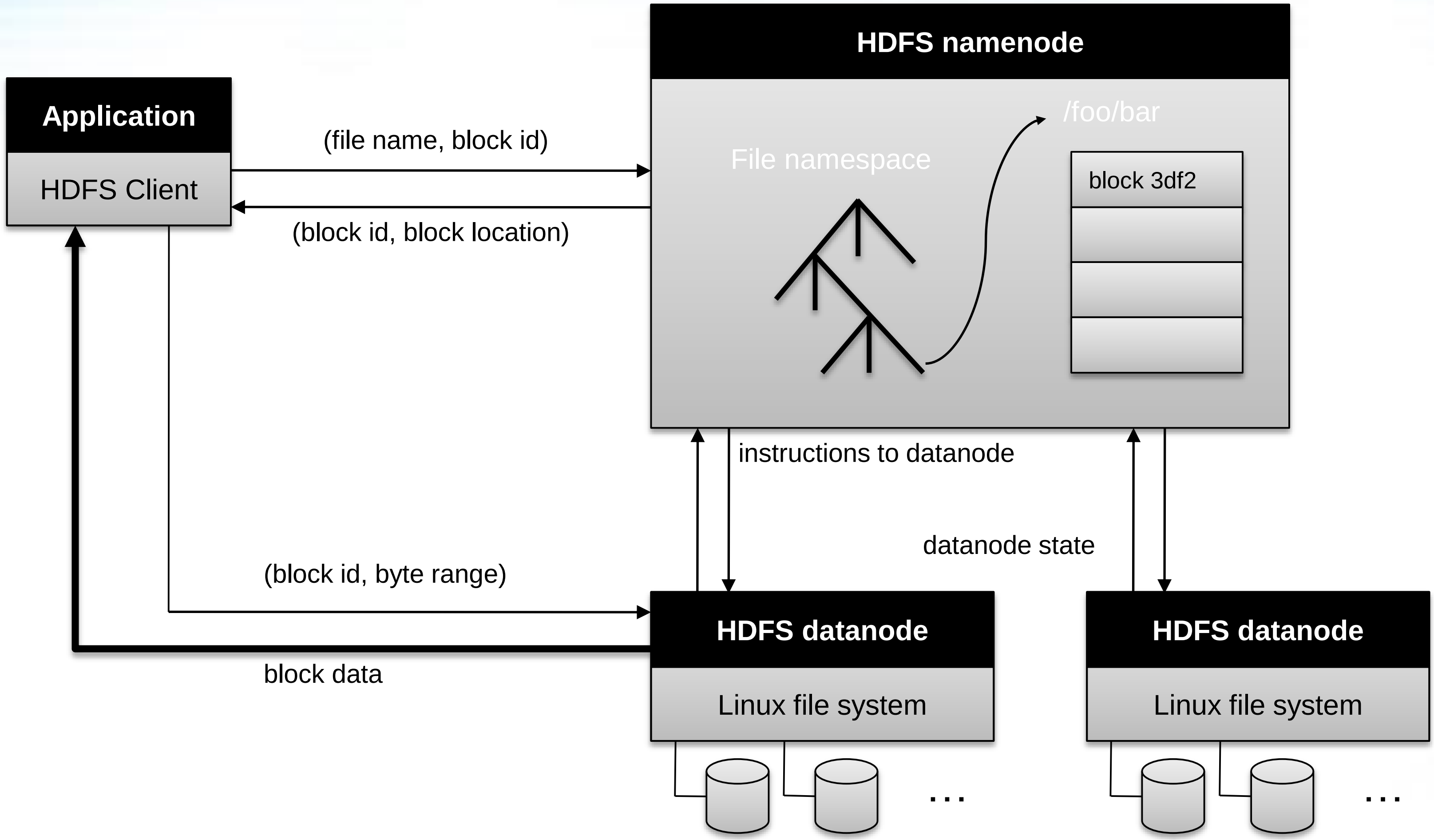
From GFS to HDFS

- Terminology differences:
  - GFS master = Hadoop namenode
  - GFS chunkservers = Hadoop datanodes

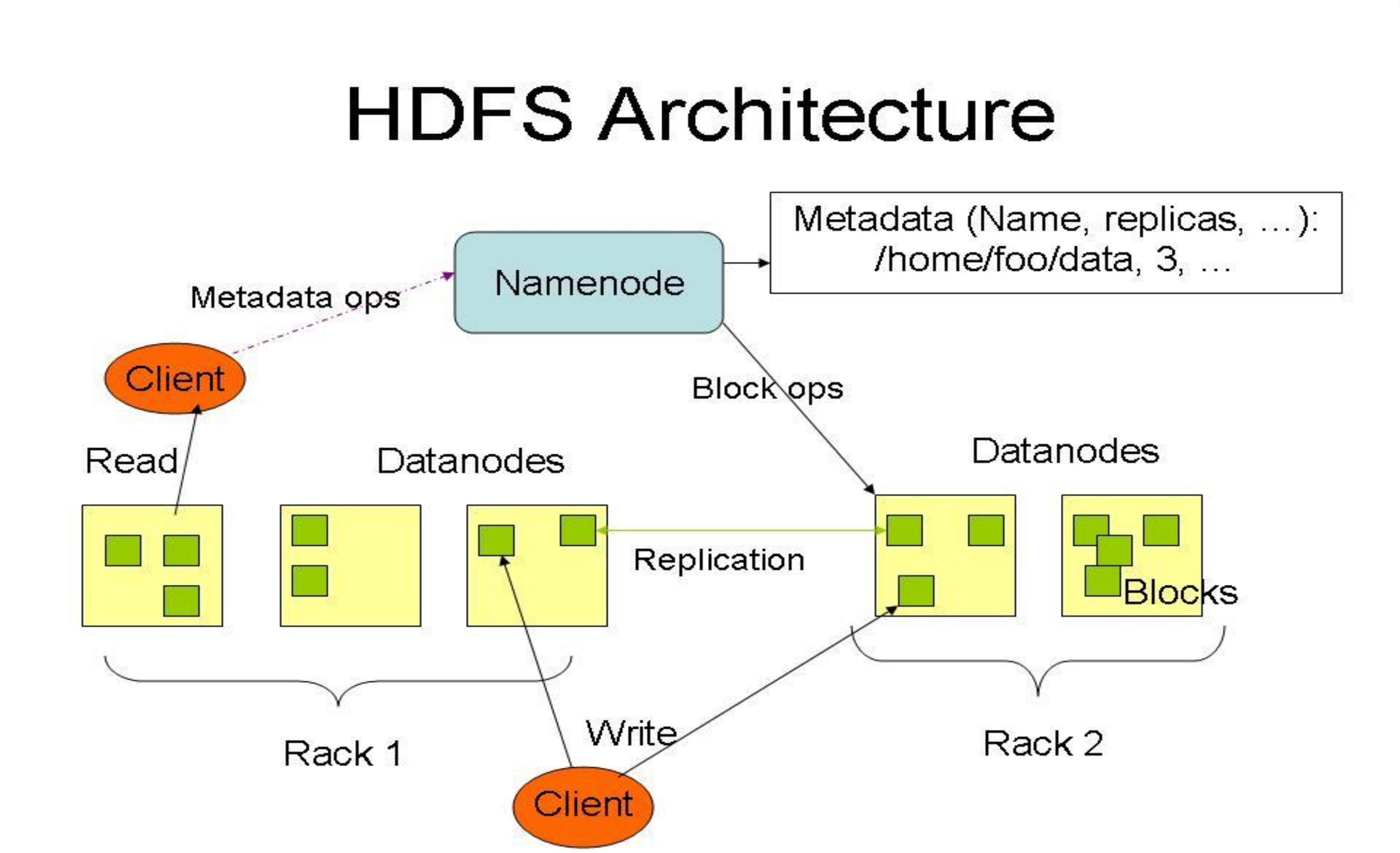
- Functional differences:
  - HDFS performance is (likely) slower

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

HDFS Working Flow



HDFS Architecture



Distributed File System

- Single Namespace for entire cluster
- Data Coherency
  - Write-once-read-many access model
  - Client can only append to existing files
- Files are broken up into blocks
  - Typically 128 MB block size
  - Each block replicated on multiple DataNodes
- Intelligent Client
  - Client can find location of blocks
  - Client accesses data directly from DataNode

NameNode Metadata

- Meta-data in Memory
  - The entire metadata is in main memory
  - No demand paging of meta-data
- Types of Metadata
  - List of files
  - List of Blocks for each file
  - List of DataNodes for each block
  - File attributes, e.g creation time, replication factor
- A Transaction Log
  - Records file creations, file deletions. etc

Namenode Responsibilities

- Managing the file system namespace:
  - Holds file/directory structure, metadata, file-to-bloc k mapping, access permissions, etc.
- Coordinating file operations:
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - Periodic communication with the datanodes
  - Block re-replication and rebalancing
  - Garbage collection

DataNode

- A Block Server
  - Stores data in the local file system (e.g. ext3)
  - Stores meta-data of a block (e.g. CRC)
  - Serves data and meta-data to Clients

- Block Report
  - Periodically sends a report of all existing blocks to the NameNode

- Facilitates Pipelining of Data
  - Forwards data to other specified DataNodes

Block Placement

- Current Strategy
  - One replica on local node
  - Second replica on a remote rack
  - Third replica on same remote rack
  - Additional replicas are randomly placed
- Clients read from nearest replica
- Would like to make this policy pluggable

Data Correctness

- Use Checksums to validate data
  - Use CRC32
- File Creation
  - Client computes checksum per 512 byte
  - DataNode stores the checksum
- File access
  - Client retrieves the data and checksum from DataNode
  - If Validation fails, Client tries other replicas

NameNode Failure

- A single point of failure
- Transaction Log stored in multiple directories
  - A directory on the local file system
  - A directory on a remote file system (NFS/CIFS)
- Need to develop a real HA(High Availability) solution

Anatomy of a File Read in HDFS

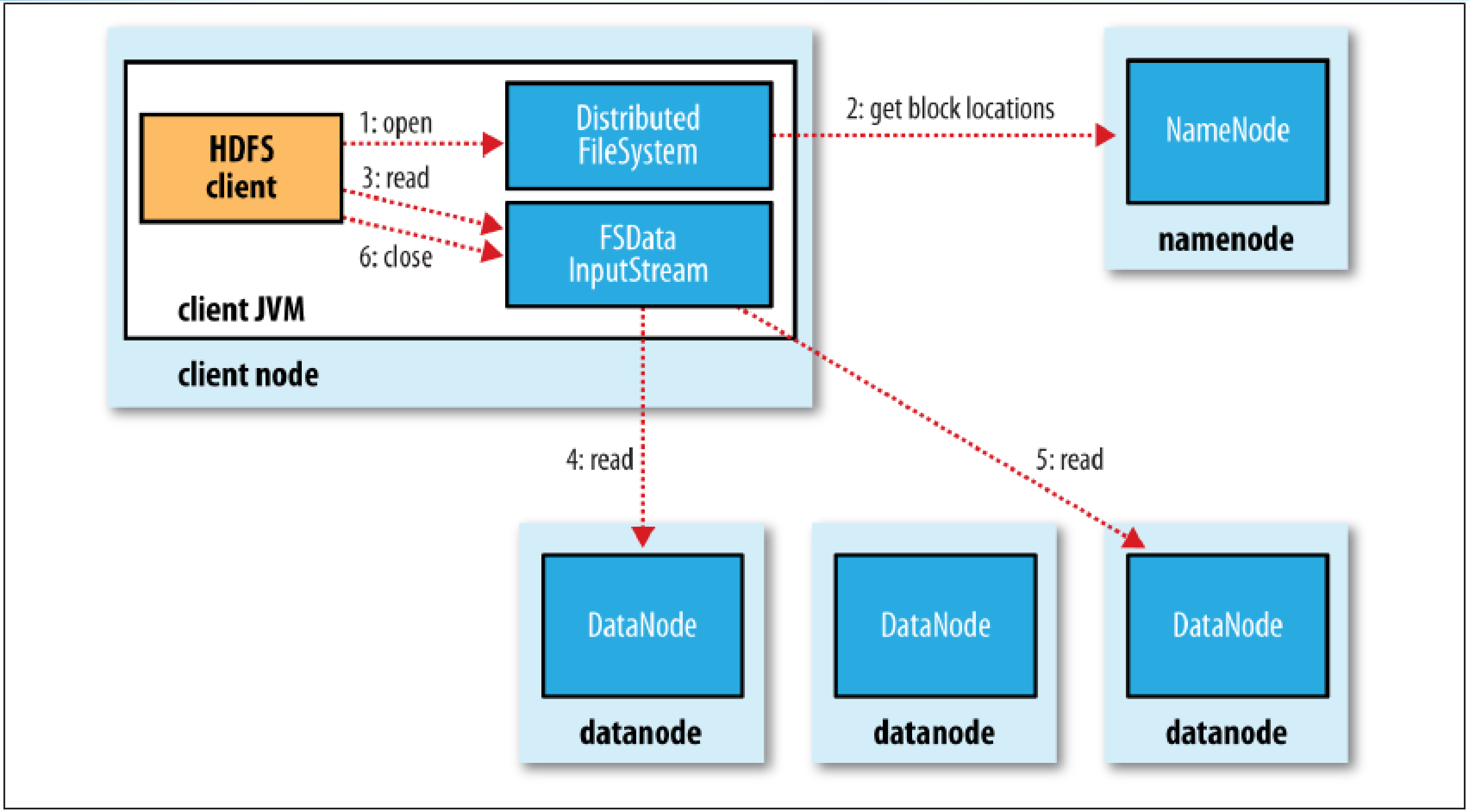


Figure 3-2. A client reading data from HDFS

Figure is from Hadoop, The Definitive Guide, 4th Edition, Tom White, O'Reilly

Network Distance in HDFS

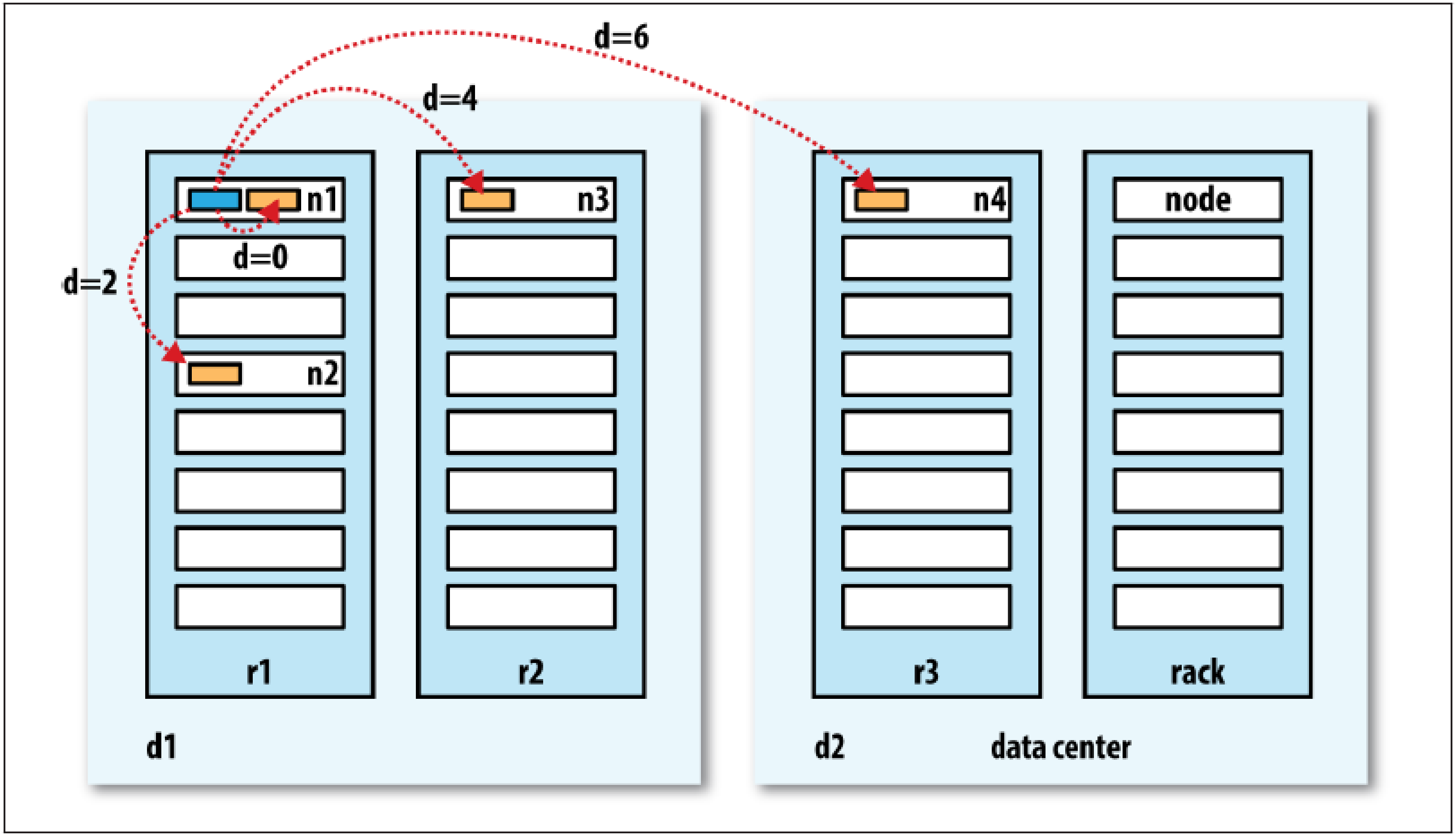


Figure 3-3. Network distance in Hadoop

Figure is from Hadoop, The Definitive Guide, 4th Edition, Tom White, O'Reilly

Anatomy of a File Write in HDFS

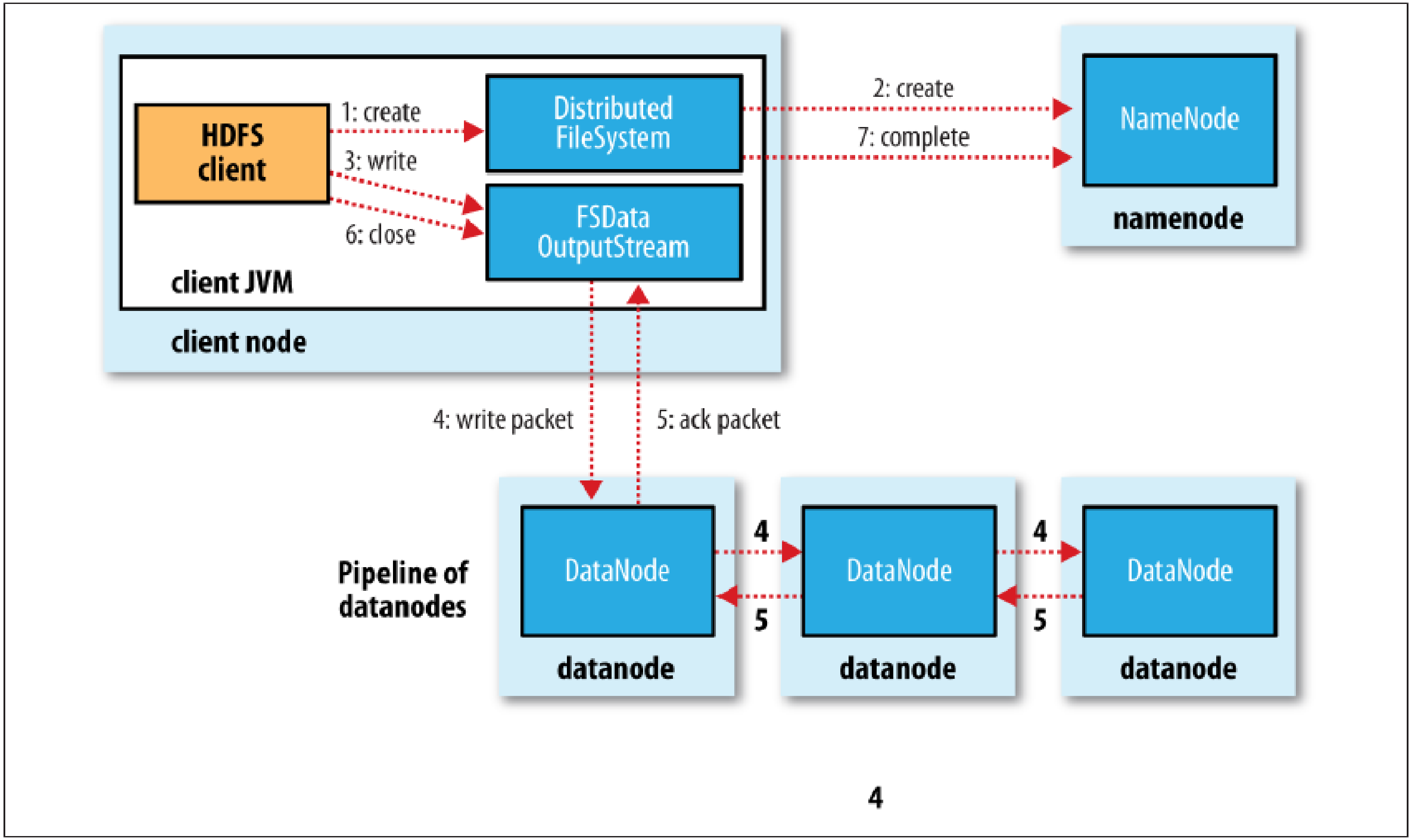


Figure 3-4. A client writing data to HDFS

Figure is from Hadoop, The Definitive Guide, 4th Edition, Tom White, O'Reilly

Replica Placement Stragies in Hadoop (Replicas = 3)

1. place the first replica on the same node as the client

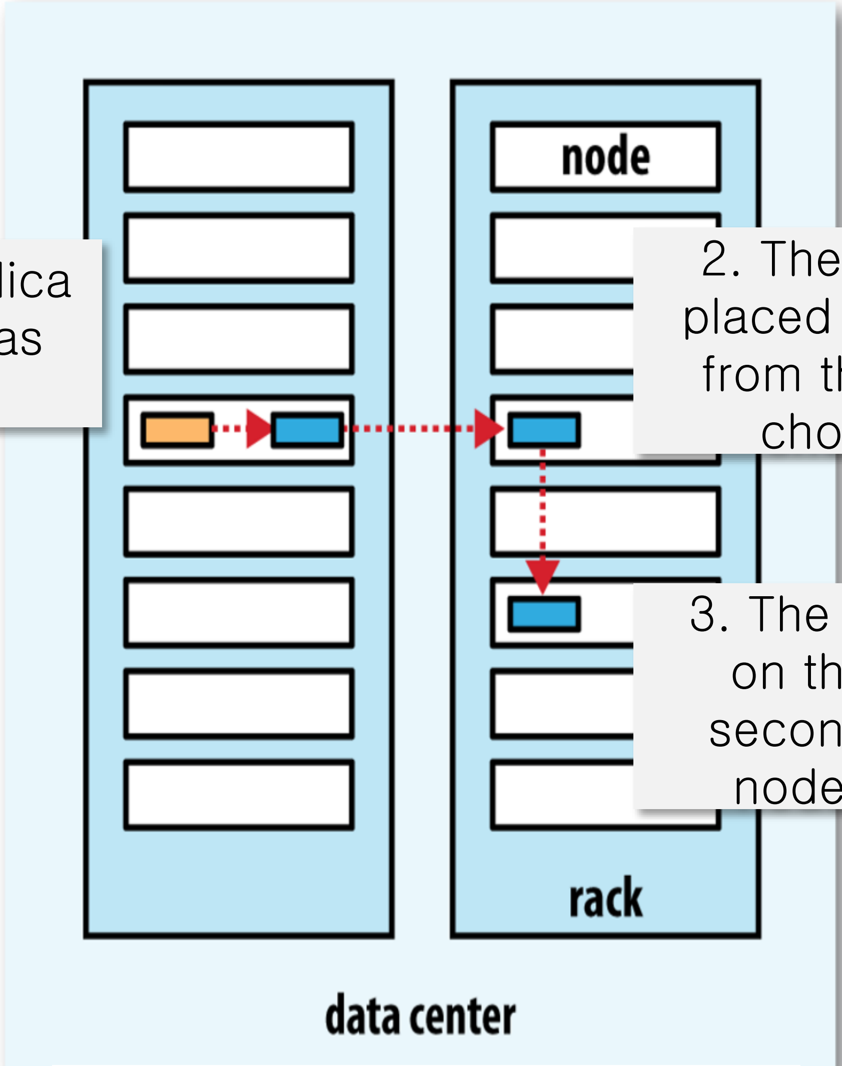


Figure 3-5. A typical replica pipeline

2. The second replica is placed on a different rack from the first (off-rack), chosen at random.

3. The third replica is placed on the same rack as the second, but on a different node chosen at random

Figure is from Hadoop, The Definitive Guide, 4th Edition, Tom White, O'Reilly

Apache Hadoop YARN: Yet Another Resource Negotiator

Chapter 4. YARN,

"Hadoop, The Definitive Guide," 4th Ed., Tom White, O'Reilly

YARN

- Scalability
- Multi-tenancy
- Serviceability
- Locality Awareness
- High Cluster Utilization
- Reliability/Availability
- Secure and auditable operation
- Support for Programming Model Diversity

Flexible Resource Model

- Hadoop: # of Map/reduce slots are fixed.
- Easy, but lower utilization

YARN

- Scalability
- Multi-tenancy
- Serviceability
- Locality Awareness
- High Cluster Utilization
- Reliability/Availability
- Secure and auditable operation
- Support for Programming Model Diversity
- Flexible Resource Model

- **Backward Compatibility**
  - The system behaves similar to the old Hadoop

YARN

- Separating resource management functions from the programming model
- MapReduce becomes just one of the application
- Dryad, …. Etc
- Binary compatible/Source compatible

YARN: Architecture

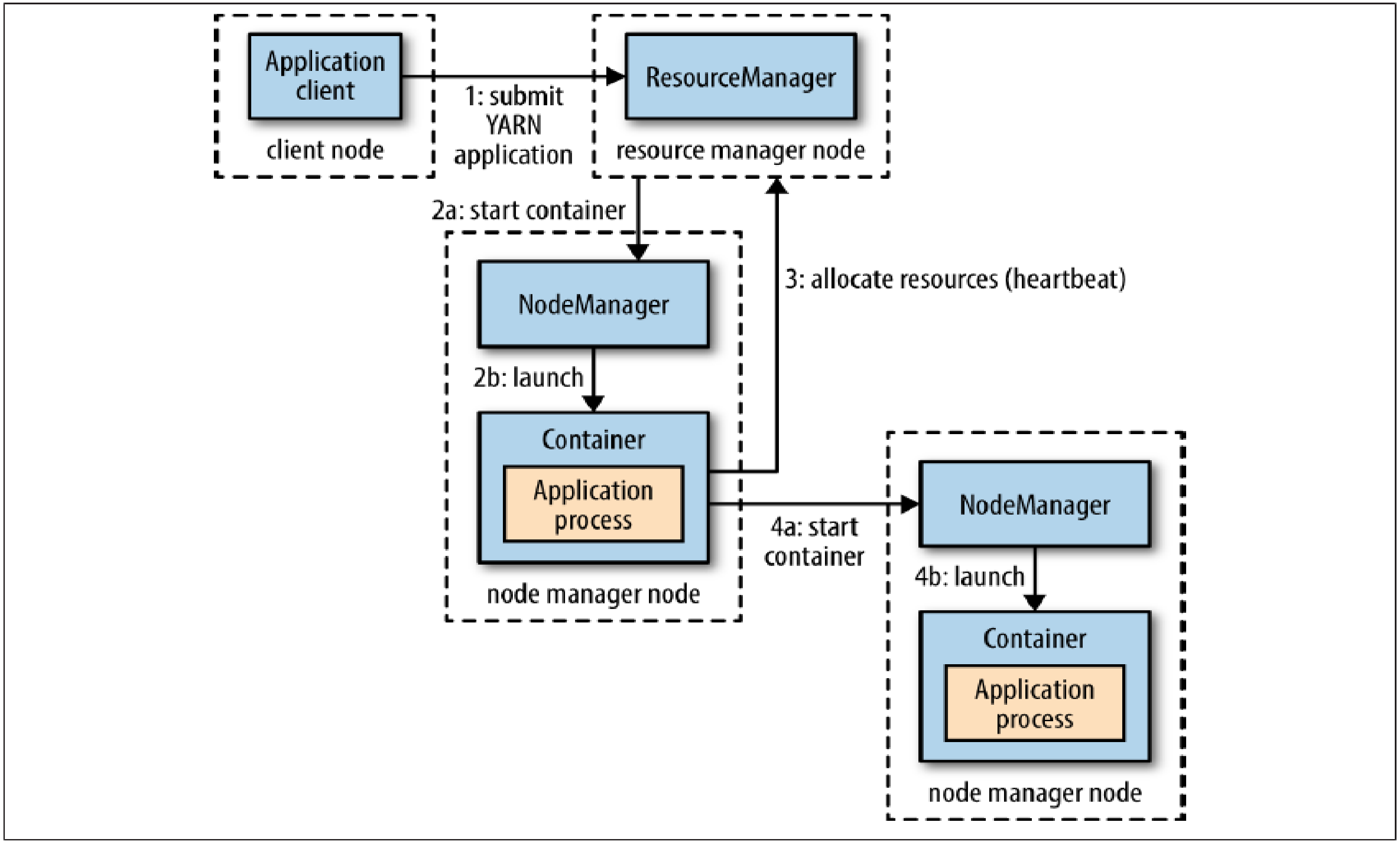


Figure 4-2. How YARN runs an application

Figure is from Hadoop, The Definitive Guide, 4th Edition, Tom White, O'Reilly

Resource Manager

- One per cluster
  - Central, global view
  - Enable global properties
    - Fairness, capacity, locality
- Container
  - Logical bundle of resources (CPU/memory)
- Job requests are submitted to RM
    - To start a job, RM finds a container to spawn AM
- No static resource partitioning

Resource Manager (cont’)

- only handles an overall resource profile for each application
  - Local optimization/internal flow is up to the application

- Preemption
  - Request resources back from an application
  - Checkpoint snapshot instead of explicitly killing jobs / migrate computation to other containers

Application Master

- The head of a job
- Runs as a container
- Request resources from RM
  - # of containers/ resource per container/ locality …
- Dynamically changing resource consumption
- Can run any user code (Dryad, MapReduce, Tez, REEF…etc)
- Requests are "*late-binding*"

MapReduce AM

- Optimizes for locality among map tasks with identical resource requirements
  - Selecting a task with input data close to the container.
- AM determines the semantics of the success or failure of the container

Node Manager

- The “worker” daemon. Registers with RM
- One per node
- Container Launch Context – env var, commands…
- Report resources (memory/CPU/etc…)
- Configure the environment for task execution
- Garbage collection/ Authentication
- Auxiliary services
  - Output intermediate data between map and reduce tasks

MapReduce Job in Hadoop YARN

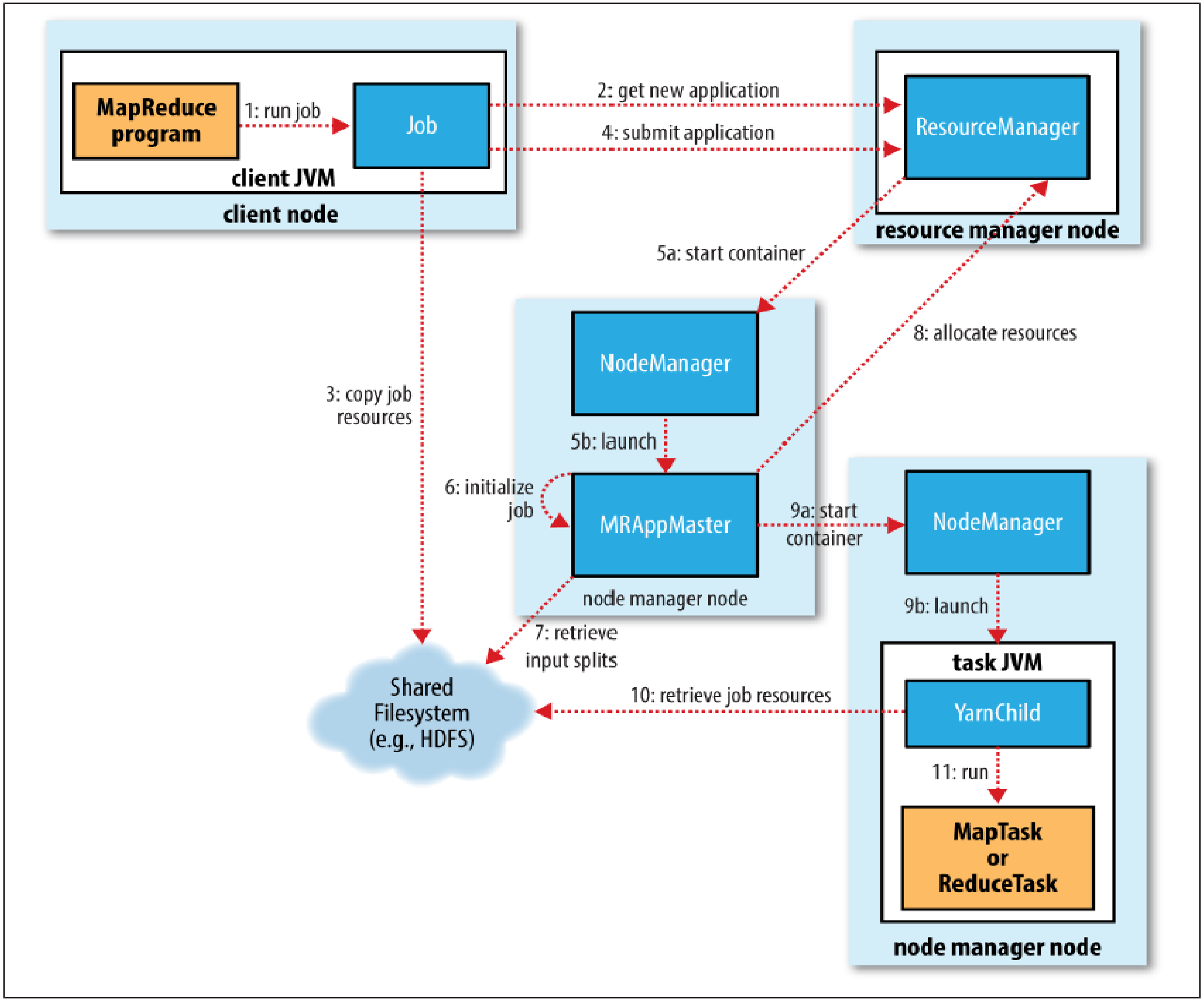


Figure 7-1. How Hadoop runs a MapReduce job

Hadoop, The Definitive Guide

Fault tolerance and availability

- RM Failure
  - Recover using persistent storage
  - Kill all containers, including AMs'
  - Relaunch AMs
- NM Failure
  - RM detects it, mark the containers as killed, report to Ams
- AM Failure
  - RM kills the container and restarts it.
- Container Failure
  - The framework is responsible for recovery

Hadoop Resources

Reading

- Jeffrey Dean and Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters
  - http://labs.google.com/papers/mapreduce.html

- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung: The Google File System
  - http://labs.google.com/papers/gfs.html

Resources

- Hadoop Wiki
  - Introduction
    - http://wiki.apache.org/lucene-hadoop/
  - Getting Started
    - http://wiki.apache.org/lucene-hadoop/GettingStartedWithHadoop
  - Map/Reduce Overview
    - http://wiki.apache.org/lucene-hadoop/HadoopMapReduce
    - http://wiki.apache.org/lucene-hadoop/HadoopMapRedClasses
  - Eclipse Environment
    - http://wiki.apache.org/lucene-hadoop/EclipseEnvironment
- Javadoc
  - http://lucene.apache.org/hadoop/docs/api/

Resources

- Releases from Apache download mirrors
  - http://www.apache.org/dyn/closer.cgi/lucene/hadoop/
- Nightly builds of source
  - http://people.apache.org/dist/lucene/hadoop/nightly/
- Source code from subversion
  - http://lucene.apache.org/hadoop/version_control.html

Introduction to Apache Spark

Shortcoming of MapReduce

- Forces your data processing into Map and Reduce
  - Other workflows missing include join, filter, flatMap, groupByKey, union, intersection, …
- Based on "Acyclic Data Flow" from Disk to Disk (HDFS)
- Read and write to Disk before and after Map and Reduce (stateless machine)
  - Not efficient for iterative tasks, i.e. Machine Learning
- Only for Batch processing
  - Interactivity, streaming data

Iteration in MapReduce

- Both Iterative and Interactive applications require faster data sharing across parallel jobs.
- Data sharing is slow in MapReduce due to replication, serialization, and disk IO. Regarding storage system, most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

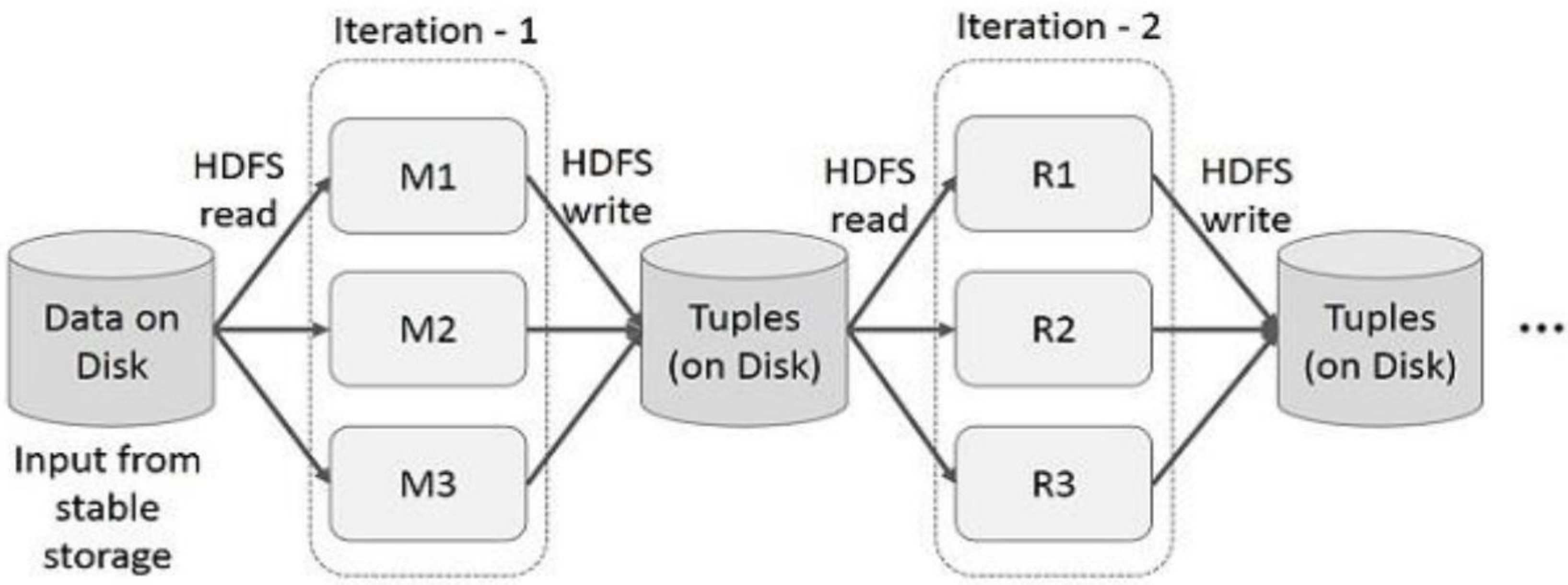


Figure: Iterative operations on MapReduce

Interactiveness in MapReduce

- Both Iterative and Interactive applications require faster data sharing across parallel jobs.
- Data sharing is slow in MapReduce due to replication, serialization, and disk IO. Regarding storage system, most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

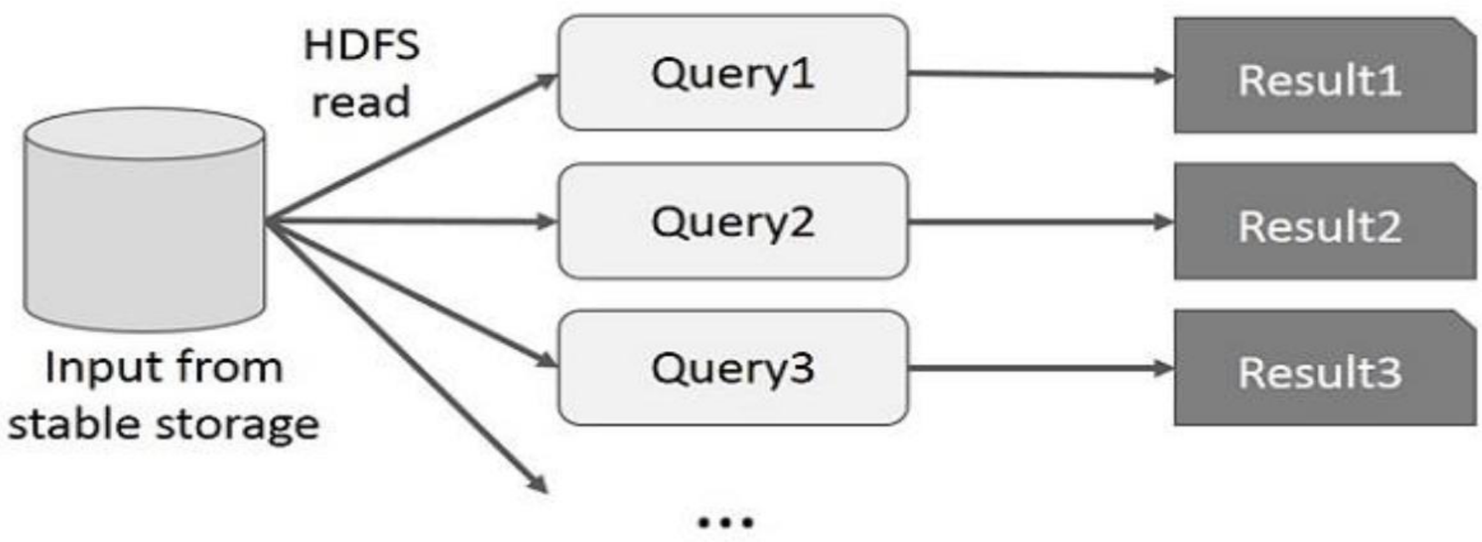


Figure: Interactive operations on MapReduce

One Solution is Apache Spark

- A new general framework, which solves many of the shortcomings of MapReduce:
  - Has many other workflows, i.e. join, filter, flatMapdistinct, groupByKey, reduceByKey, sortByKey, collect, count, first…
    - (around 30 efficient distributed operations)
  - In-memory caching of data (for iterative, graph, and machine learning algorithms, etc.)
  - Original key construct: Resilient Distributed Dataset(RDD)
  - More recently added: DataFrames & DataSets
    - Different APIs for aggregate data
- It capable of leveraging the Hadoop ecosystem, e.g. HDFS, YARN, HBase, S3, …
- Native Scala, Java, Python, and R support
- Supports interactive shells for exploratory data analysis
- Spark API is extremely simple to use
- Developed at AMPLab UC Berkeley, now by Databricks.com

Spark

- Implements Resilient Distributed Datasets (RDDs)

- Operations on RDDs
  - Transformations: defines new dataset based on previous ones
  - Actions: starts a job to execute on cluster

- Well-designed interface to represent RDDs
  - Makes it very easy to implement transformations
  - Most Spark transformation implementation < 20 LoC

Operation	Meaning
partitions()	Return a list of Partition objects
preferredLocations(*p*)	List nodes where partition *p* can be accessed faster due to data locality
dependencies()	Return a list of dependencies
iterator(*p*, *parentIters*)	Compute the elements of partition *p* given iterators for its parent partitions
partitioner()	Return metadata specifying whether the RDD is hash/range partitioned

Table 3: Interface used to represent RDDs in Spark.

121

Iteration in RDD

- The key idea of spark is Resilient Distributed Datasets (RDD); it supports in-memory processing computation. Data sharing in memory is 10 to 100 times faster than network and Disk.

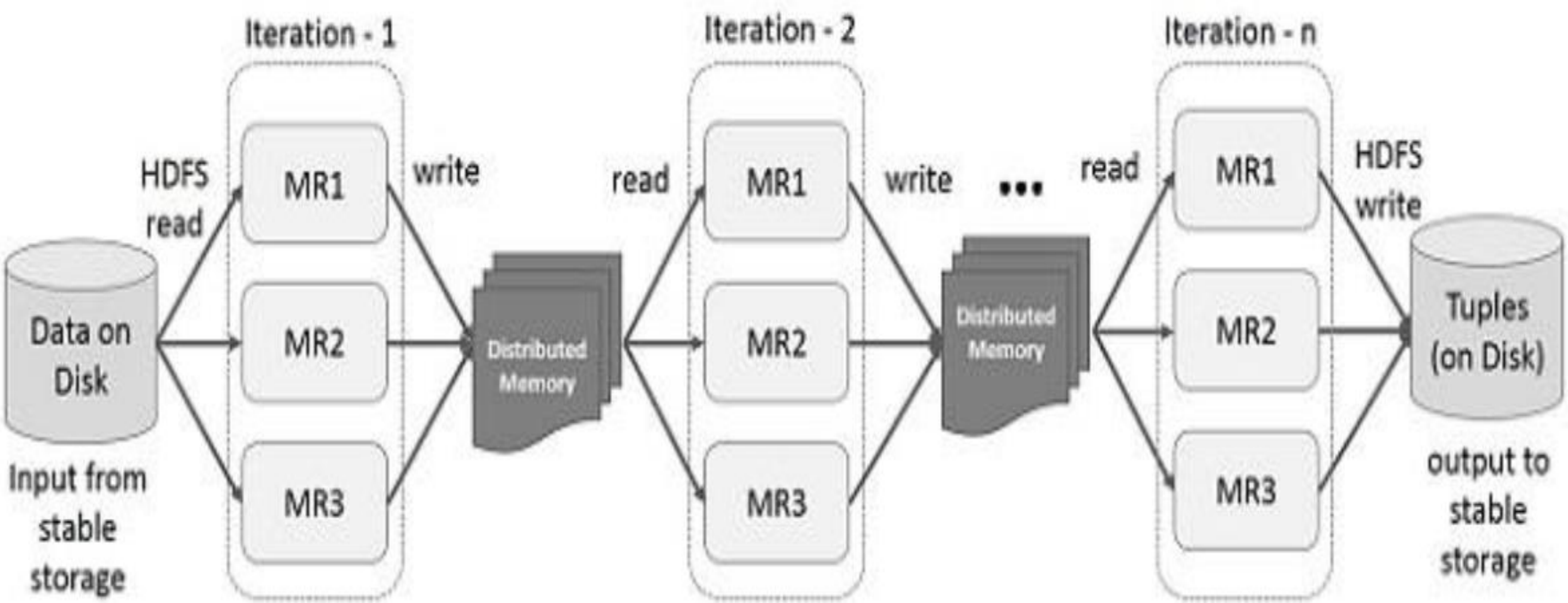


Figure: Iterative operations on Spark RDD

Interactiveness in Spark

- The key idea of spark is Resilient Distributed Datasets (RDD); it supports in-memory processing computation. Data sharing in memory is 10 to 100 times faster than network and Disk.

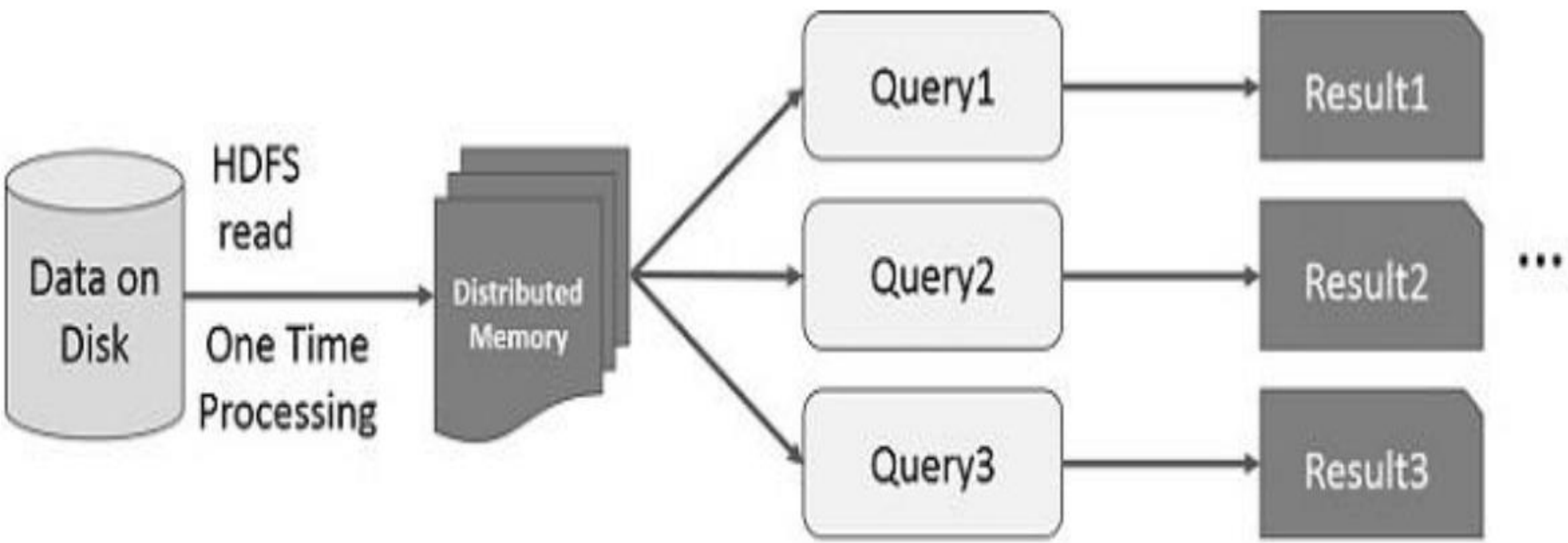


Figure: Interactive operations on Spark RDD

Spark vs. Hadoop MapReduce

"Spark vs. Hadoop MapReduce" by Xplenty (March 11, 2019)
https://www.xplenty.com/blog/apache-spark-vs-hadoop-mapreduce/

- **Performance:** Spark performs better when all the data fits in the memory, especially on dedicated clusters; Hadoop MapReduce is designed for data that doesn't fit in the memory and it can run well alongside other services.
- **Ease of use:** Spark is easier to program and includes an interactive mode; Hadoop MapReduce is more difficult to program but many tools are available to make it easier.
- **Compatibility:** Spark's compatibility to data types and data sources is the same as Hadoop MapReduce.
- **Data processing:** Spark is the Swiss army knife of data processing; Hadoop MapReduce is the commando knife of batch processing.
- **Failure Tolerance :** Spark and Hadoop MapReduce both have good failure tolerance, but Hadoop MapReduce is slightly more tolerant.
- **Security :** Spark security is still in its infancy; Hadoop MapReduce has mo re security features and projects.
- **Maturity: Spark maturing, Hadoop MapReduce mature**

Resilient Distributed Datasets (RDDs)

- Restricted form of distributed shared memory
  - Immutable, partitioned collections of records
  - Allows for coarse-grained deterministic transformations (map, filter, join, …)
  - All the different processing components in Spark share the same abstraction called RDD
  - As applications share the RDD abstraction, you can mix different kind of transformations to create new RDDs
  - The user can control its partition
  - The user can control its persistence (caching)
- Efficient fault recovery using lineage
  - Each RDD "remembers" how it was derived from other RDD / stable storage
  - Log one operation to apply to many elements
  - Recompute lost partitions on failure
  - No cost if nothing fails

RDD Actions and Transformations

- Transformations are realized when an action is called
- Transformations
  - Lazy operations applied on an RDD
  - Creates a new RDD from an existing RDD
  - Allows Spark to perform optimizations
  - e.g. map, filter, flatMap, union, intersection, distinct, reduceByKey, groupByKey
- Actions
  - Returns a value to the driver program after computation
  - e.g. reduce, collect, count, first, take, saveAsFile

Spark Operations

Transformations (create a new RDD)	map filter sample groupByKey reduceByKey sortByKey intersection	flatMap union join cogroup cross mapValues reduceByKey
Actions (return results to driver program)	collect Reduce Count takeSample take lookupKey	first take takeOrdered countByKey save foreach

Sample Spark 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
- union(otherDataset): Return a new dataset that contains the union of the elements in the source dataset and the argument.
- intersection(otherDataset): Return a new RDD that contains the intersec tion of elements in the source dataset and the argument.
- distinct([numTasks])): Return a new dataset that contains the distinct elements of the source dataset
- join(otherDataset, [numTasks]): When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of eleme nts for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.

Source: https://spark.apache.org/docs/latest/programming-guide.html

Sample Spark Actions

- reduce(func): Aggregate the elements 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 par allel.
- 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.
- count(): Return the number of elements in the dataset.

Remember: *Actions* cause calculations to be performed; *transformations* just set things up (lazy evaluation)

Source: https://spark.apache.org/docs/latest/programming-guide.html

Spark 2.0

- Note that, before Spark 2.0, the main programming interface of Spark was the Resilient Distributed Dataset (RDD).

- After Spark 2.0, RDDs are replaced by Dataset, which is strongly-typed like an RDD, but with richer optimizations under the hood. The RDD interface is still supported, and you can get a more detailed reference at the RDD programming guide.

Spark Ecosystem

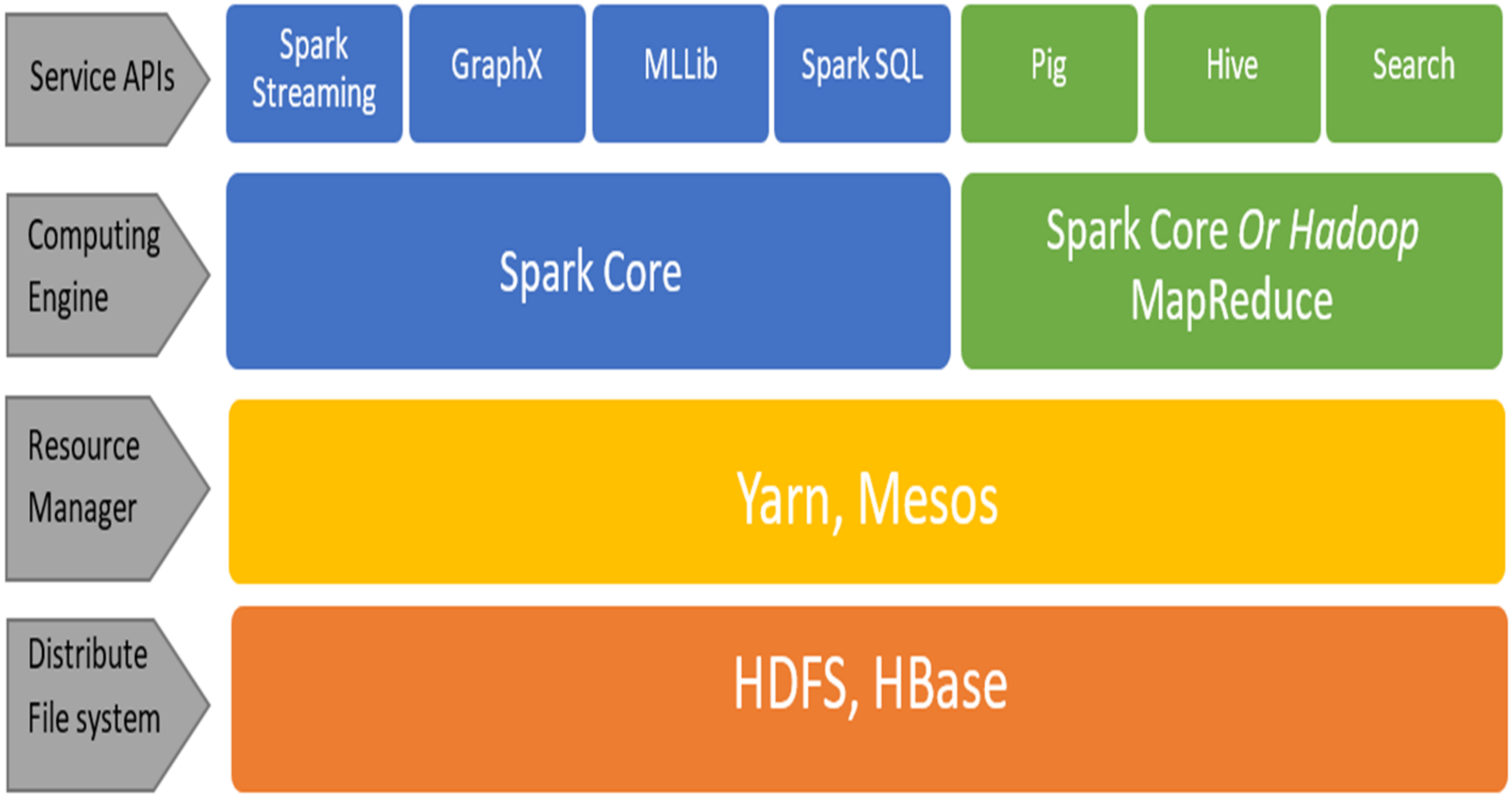


Figure 1 - Spark Context

Architecture of Spark Applications

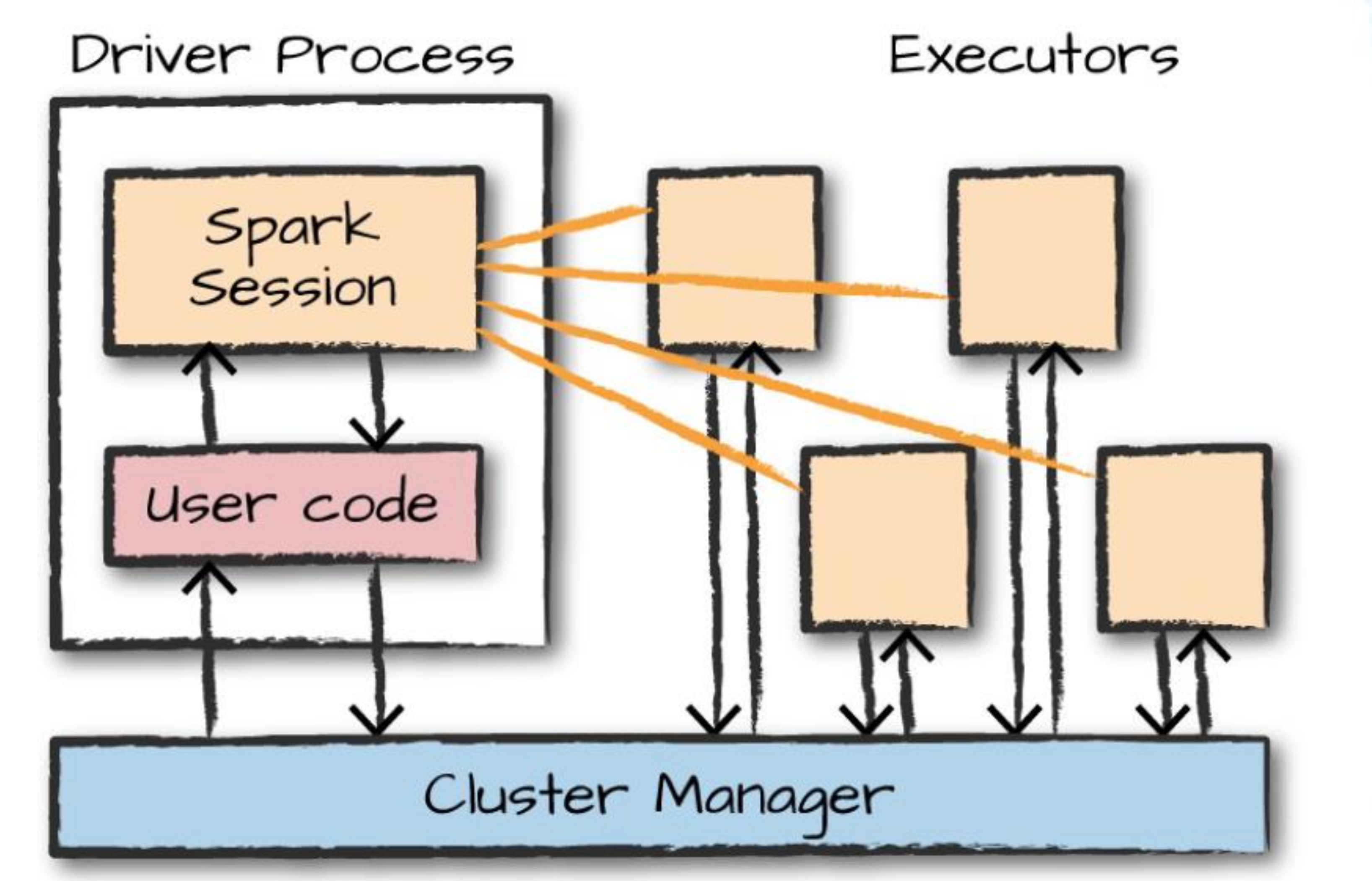
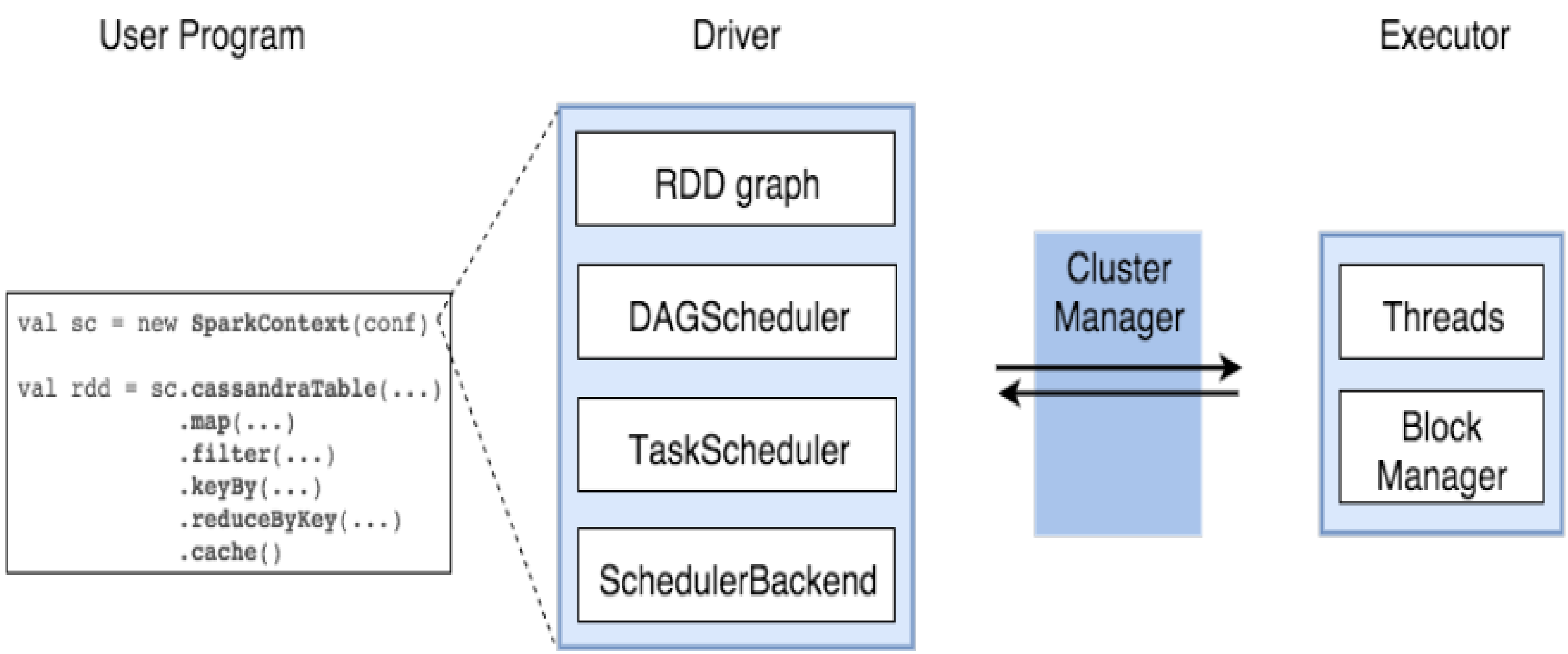
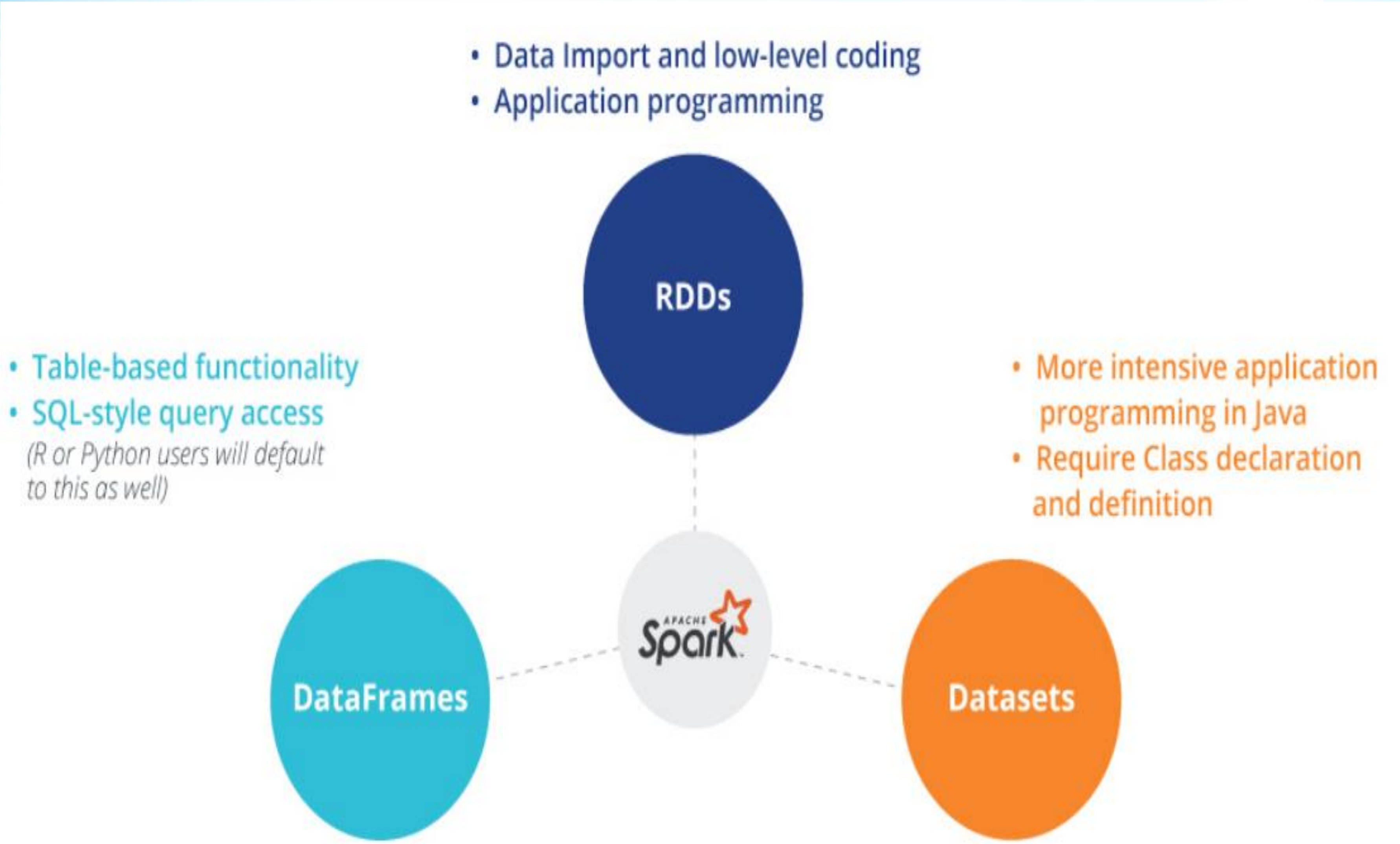


Figure 2-1. The architecture of a Spark Application


User Program
Driver
Executor
val sc = new SparkContext(conf)
val rdd = sc.cassandraTable(...)
.map(...)
.filter(...)
.keyBy(...)
.reduceByKey(...)
.cache()
RDD graph
DAGScheduler
TaskScheduler
SchedulerBackend
Cluster Manager
Threads
Block Manager

RDD, DataFrame, Dataset



DataFrame & Dataset

https://data-flair.training/blogs/apache-spark-rdd-vs-dataframe-vs-dataset/
https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

- DataFrame:
  - Unlike an RDD, data organized into named columns, e.g. a table in a relational database.
  - Imposes a structure onto a distributed collection of data, allowing higher-level abstraction
- Dataset:
  - Extension of DataFrame API which provides type-safe, object-oriented programming interface (compile-time error detection)

Both built on Spark SQL engine & use Catalyst to generate optimized logical and physical query plan; both can be converted to an RDD

Dataframe Transformation

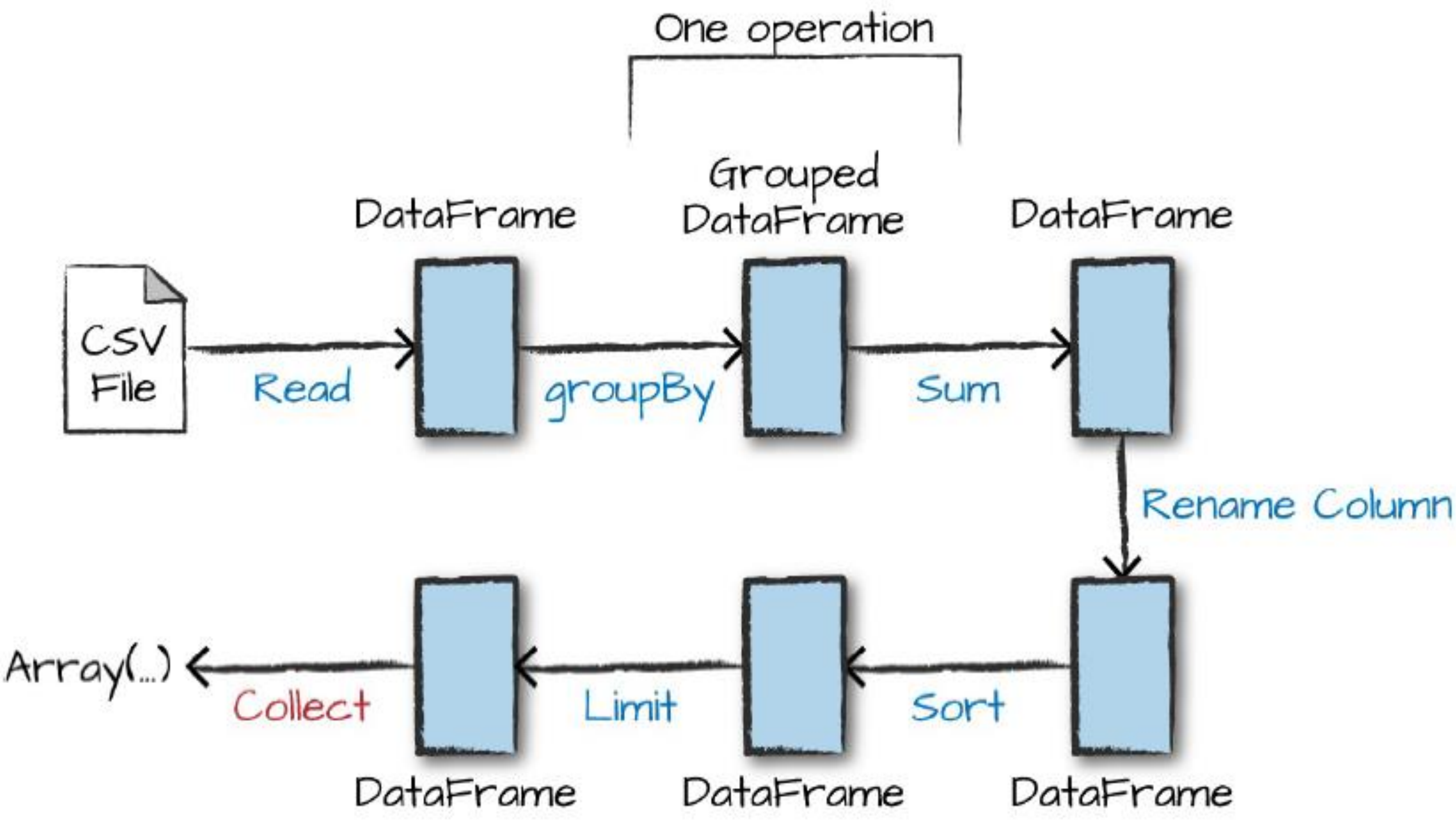
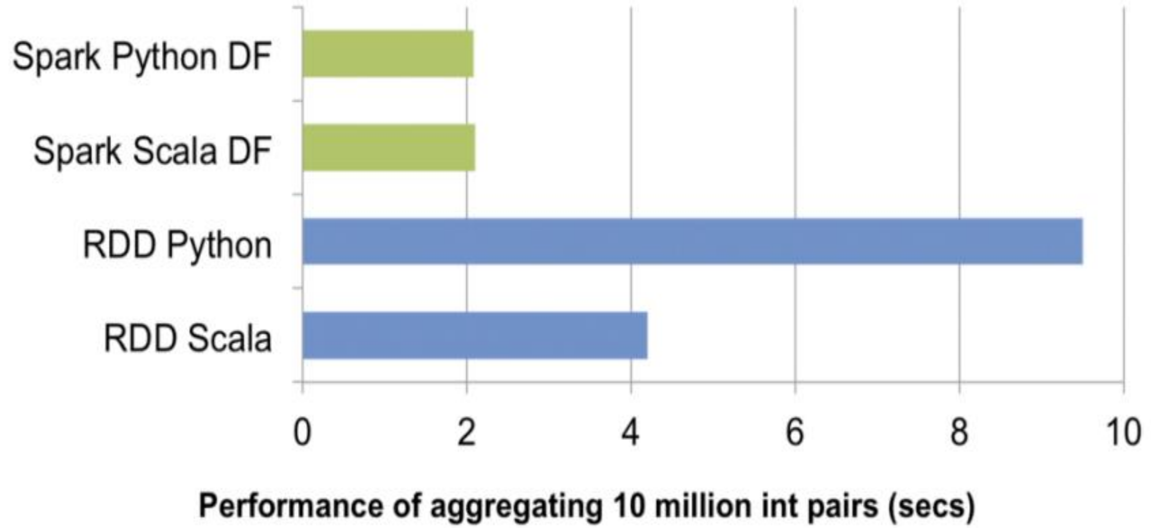


Figure 2-10. The entire DataFrame transformation flow

RDDs vs. DataFrames

- RDDs provide a low level interface into Spark
- DataFrames have a schema
- DataFrames are cached and optimized by Spark
- DataFrames are built on top of the RDDs and the core Spark API



API distinction: typing

- Python & R don't have compile-time type safety checks, so *only* support DataFrame
  - Error detection only at runtime
- Java & Scala support compile-time type safety checks, so support *both* DataSet and DataFrame
  - Dataset APIs are all expressed as lambda functions and JVM typed objects
  - any mismatch of typed-parameters will be detected at compile time.

https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

Spark in the Real World (I)

- Uber – the online taxi company gathers terabytes of event data from its mobile users every day.
  - By using Kafka, Spark Streaming, and HDFS, to build a continuous ETL pipeline
  - Convert raw unstructured event data into structured data as it is collected
  - Uses it further for more complex analytics and optimization of operations

- Pinterest – Uses a Spark ETL pipeline
  - Leverages Spark Streaming to gain immediate insight into how users all over the world are engaging with Pins—in real time.
  - Can make more relevant recommendations as people navigate the site
  - Recommends related Pins
  - Determine which products to buy, or destinations to visit

Spark in the Real World (II)

Here are Few other Real World Use Cases:

- Conviva – 4 million video feeds per month
  - This streaming video company is second only to YouTube.
  - Uses Spark to reduce customer churn by optimizing video streams and managing live video traffic
  - Maintains a consistently smooth, high quality viewing experience.
- Capital One – is using Spark and data science algorithms to understand customers in a better way.
  - Developing next generation of financial products and services
  - Find attributes and patterns of increased probability for fraud
- Netflix – leveraging Spark for insights of user viewing habits and then re commends movies to them.
  - User data is also used for content creation

Spark: when not to use

- Even though Spark is versatile, that doesn't mean Spark's in-memory capabilities are the best fit for all use cases:
  - For many simple use cases Apache MapReduce and Hive might be a more appropriate choice
  - Spark was not designed as a multi-user environment
  - Spark users are required to know that memory they have is sufficient for a dataset
  - Adding more users adds complications, since the users will have to coordinate memory usage to run code

Apache Spark Extensions "on top" of core

- Spark SQL
- Spark Streaming – stream processing of live datastreams
- GraphX – graph manipulation
  - extends Spark RDD with Graph abstraction: a directed multigraph with properties attached to each vertex and edge.
- MLlib – machine learning