



Advanced Operating Systems: Three Easy Pieces

Hadoop: HDFS and MapReduce

Outline

■ Hadoop

☐ Hadoop Architecture Overview

☐ HDFS

- ☐ GFS Motivation

- ☐ Workload

- ☐ Architecture

☐ MapReduce:

- ☐ Motivation

- ☐ What is M/R

- ☐ Programming Model

- ☐ **Hadoop I/O**: Shuffling Optimization

- ☐ MapReduce Operations & Configuration



1. Hadoop Architecture Overview

1. Hadoop Architecture Overview

■ Motivation

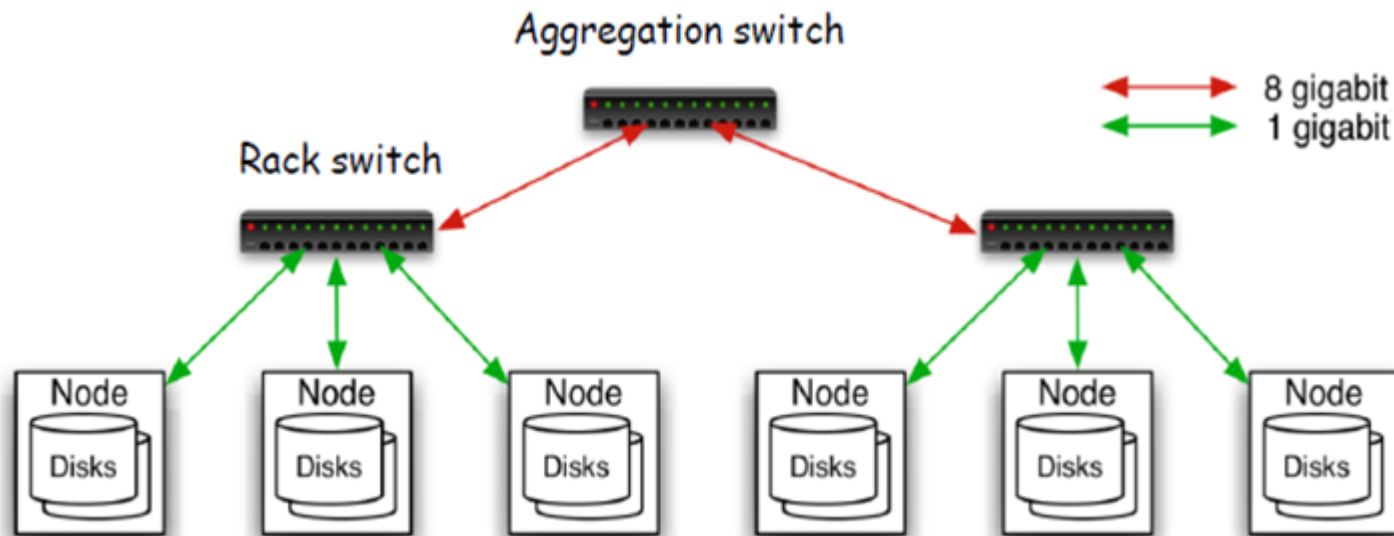
- ❑ **Amount of data is increasing** rapidly but the rate of data that can be read from disk drives have not kept up: **thirty years ago** (1990) drive stored **1.37GB** and the IO rate was around **4.4 MB/sec** (read the drive in **5 minutes**). Today drive is around **1TB** (ratio is 700+) and IO rate is around **100 MB/sec** (ratio is ~25) - (read the drive in **2.5 hrs**)!
- ❑ **With parallel processing** the probability of errors/failures increase
- ❑ **Most of the time** in the analysis process; we will need to combine data from different disk drives
- ❑ **MapReduce provides a programming model** that abstracts the problem from “R/W to disk” to “**computation over sets of Keys and Values (K/V pairs)**.”

■ **In summary**, Hadoop provides a reliable shared storage and analysis platform, storage is provided by **HDFS** and analysis by **MapReduce**.

■ **Good tutorial**: <http://developer.yahoo.com/hadoop/tutorial/module1.html> https://www.tutorialspoint.com/hadoop/hadoop_introduction.htm

1. Hadoop Architecture Overview

■ Typical Hadoop Cluster



- 40 nodes/rack, 1000-4000 nodes in a cluster (25-100 racks)
- 1 Gbps bandwidth within rack, 8 Gbps out of rack
- **Node:** 8 x 2GHZ cores, 8 GB RAM, 4 disks (4 TB)

1. Hadoop Architecture Overview

■ Hadoop Components:

□ **HDFS – Distributed File System:**

- ❖ **Combines cluster's local storage** into a single name space
- ❖ **All data is replicated X3 (default)** to multiple machines
- ❖ **Provides locality information to clients' service**, i.e., let client access the closest copy to them

□ **M/R – MapReduce:**

- ❖ **Batch computation framework** – throughput and not rpt
- ❖ **Tasks are re-executed** on failure
- ❖ **Optimizes for data locality** of input

1. Hadoop Architecture Overview

■ Comparison with Traditional MPP (OLTP) Databases

Opeartion	MPP OLTP Databases	MapReduce
Data size	GB - TB	Peta - Zeta Bytes
Access	Interactive and Batch	Batch
Update	Read and write many times	Write once, read many times
Structure	Static schema	Dynamic schema
Structure Enforcement	Schema on write	Schema on read (Hive)
Integrity	High	Low
Scaling	Nonlinear	Linear

1. Hadoop Architecture Overview

■ Features Supported by Different Hadoop releases

Feature	1.X (12/27/2011)	0.22 (12/10/2011)	2.X (2013)
Secure authentication	Yes	No	Yes
Old configuration names	Yes	Deprecated	Deprecated
New configuration names	No	Yes	Yes
Old M/R API	Yes	Yes	Yes
New M/R API	Yes (with some missing libs)	Yes	Yes
M/R 1 runtime (Classic)	Yes	Yes	No
M/R 2 runtime (YARN)	No	No	Yes
HDFS Federation	No	No	Yes
HDFS High-availability	No	No	Yes

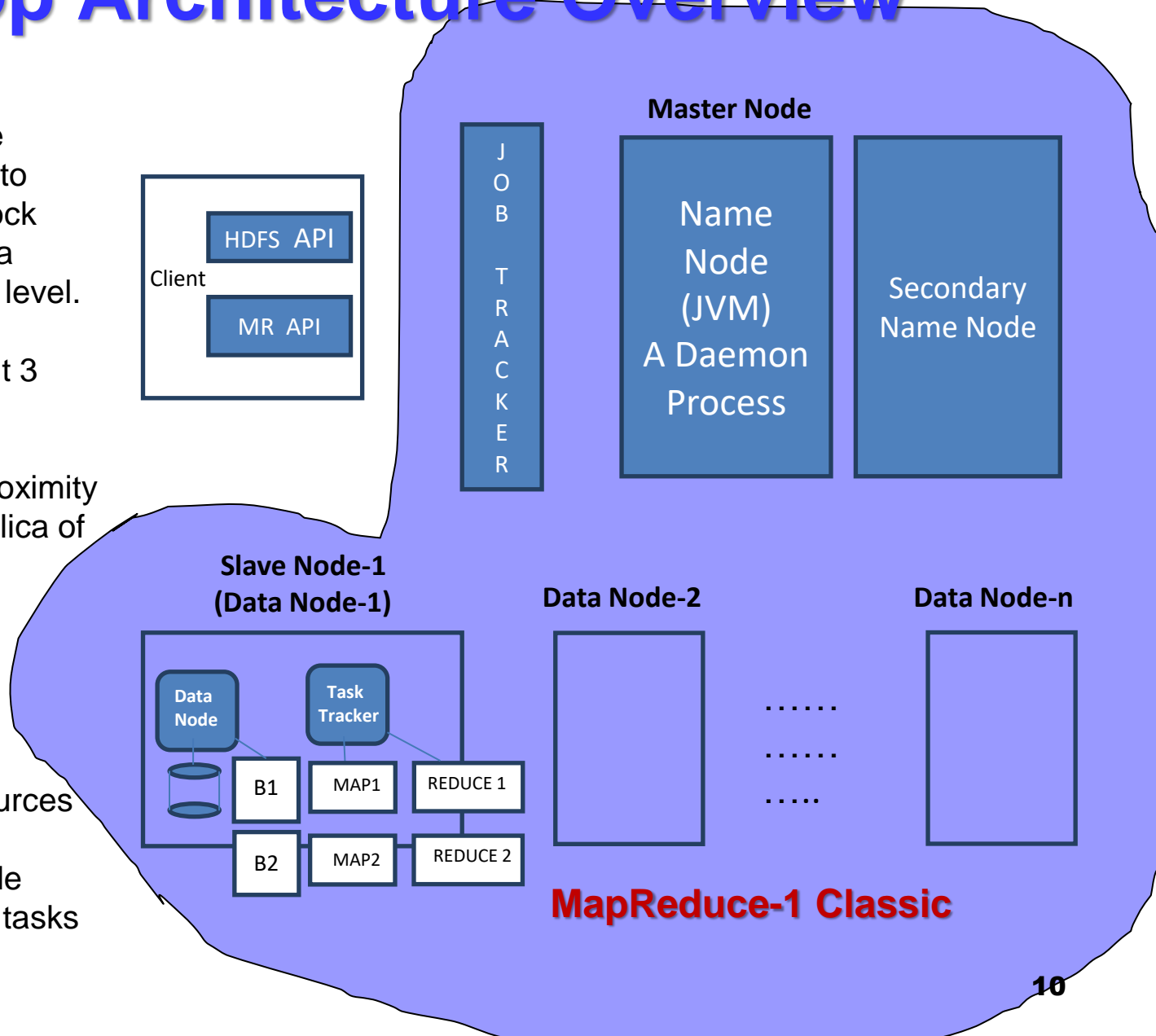
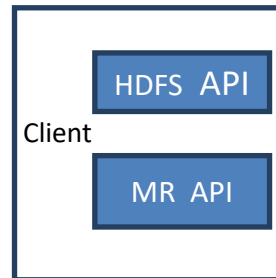
Hadoop Releases: <http://hadoop.apache.org/releases.html>
<https://hadoop.apache.org/old/releases.html>

1. Hadoop Architecture Overview

- **Configuration: configuration property names have changed after 1.x release (current release is 3.x):**
 - ❑ **dfs.namenode prefix:** dfs.name.dir → dfs.namenode.name.dir
 - ❑ **mapreduce prefix:** mapred.job.name → mapreduce.job.name
- **MapReduce APIs:**
 - ❑ **there are two Java MapReduce APIs:** Classic (API v1) and YARN (API v2)
- **Compatibility:**
 - ❑ When moving from one release to another, you need to consider the upgrade steps needed, i.e., no backward compatibility guaranteed.
 - ❑ **There are several aspects of compatibility:** API compatibility (between user code and published Hadoop APIs), data compatibility (related to persistent data and metadata format), and wire compatibility (protocol between client and server (RPC and HTTP))

1. Hadoop Architecture Overview

- **Name Node**: keep the mapping of HDFS file to blocks where each block (**64MB**) is mapped to a physical file at the OS level.
- **Each file** has a default 3 copies
- **Name node** knows proximity of client to a given replica of a given block in a file
- **Job Tracker** function as resource manager + manages an application life cycle + manages cluster resources
- **Task Tracker** Per-node agent + Manage local tasks



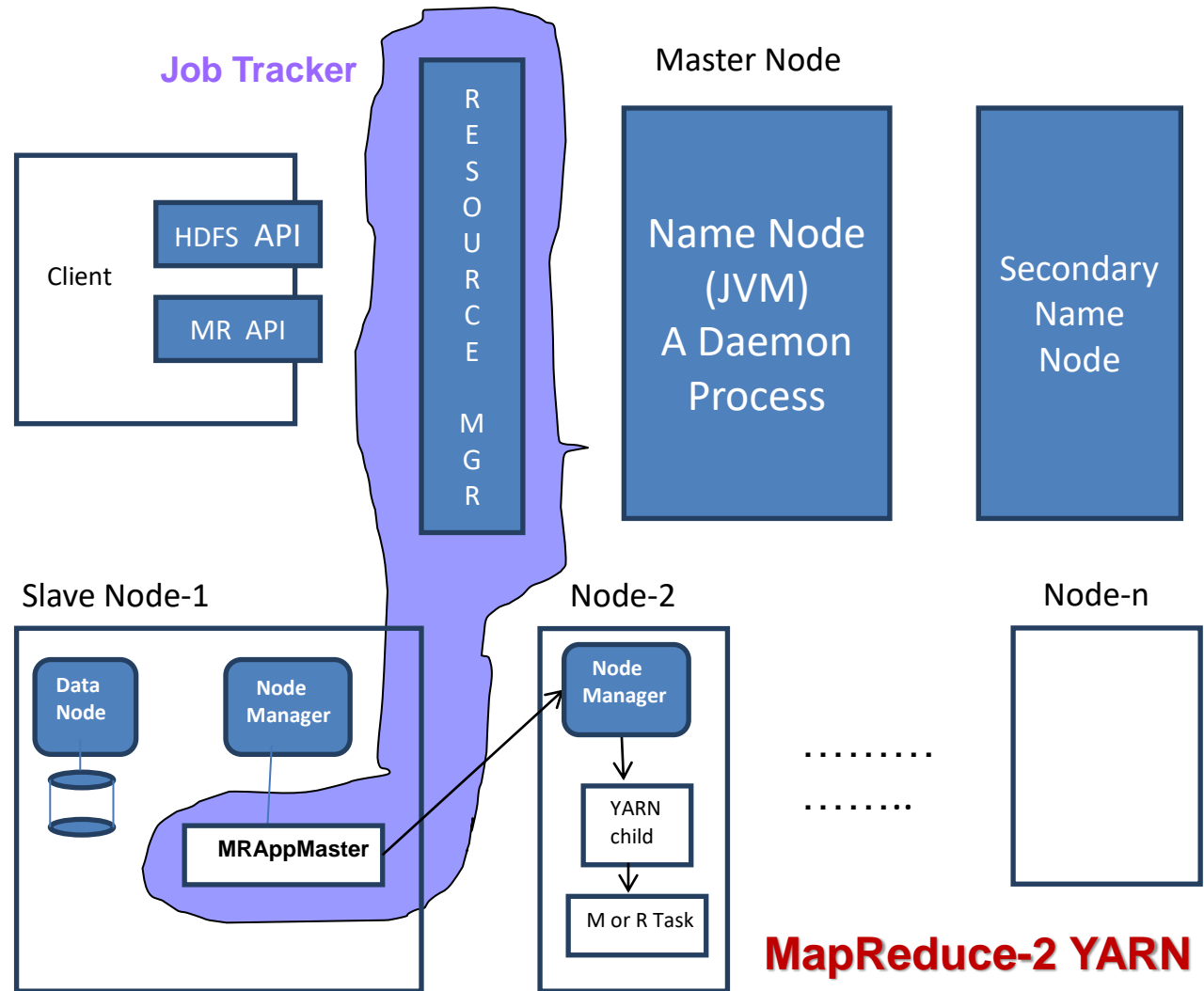
1. Hadoop Architecture Overview

■ Name Node

- ❑ **Maintains metadata (dfs.name.dir property):** consists of **FsImage** (initial copy of the metadata from last checkpoint) in memory and **EditLog** (record every change to the metadata) on disk.
- ❑ **On checkpoint or reboot** (which forces checkpoint) the name node flush metadata from memory to disk “**FsImage**” after merging **FsImage** with the **EditLog** into new version of the “**FsImage**.” On reboot the primary node (name node) will start with the merged metadata (**FsImage**) and with clean **EditLog**.
- ❑ **Problem:** if you do not do checkpoint or reboot for a while and the name node dies, you will lose data (because you lost the EditLog).
- ❑ **Secondary Name node (not an HA solution)** but rather it takes periodically (every 1 hr) the responsibility to merge EditLog with the latest “FsImage” file, store new “FsImage” to disk, and re-upload new image and clean “EditLog” to the primary name node. On primary node reboot it will have latest **FsImage** file.
- ❑ **Name node does not communicate with any;** instead data node, secondary name node, and clients will communicate with the primary name node.
- ❑ **Master Nodes:** consists of Primary NameNode, Secondary NameNode, and Checkpoint node
- ❑ **Data nodes send heartbeat every 3 seconds to the name node.** If name node do not receive heartbeat from a specific data node in **10 minutes** that data node is assumed down and it starts creating replacement data node.

1. Hadoop Architecture v2 Overview

- **YARN** (Yet Another Resource Negotiator): is intended to support multiple app paradigms in addition to M/R on the same cluster.
- **Job Tracker** is being replaced with: **Resource Mgr** (manage global resources) + **MRAppMaster** (manages the Application life cycle).



1. Hadoop Architecture Overview

■ Hadoop Scheduling:

□ **Default Hadoop Scheduler:**

- ❖ FIFO queueing of MR jobs
- ❖ Each job gets entire cluster
- ❖ Hadoop will select # of mappers, # of reducers, or you can select (over 100 parameters!)

□ **Fair scheduler / Capacity scheduler:**

- ❖ Jobs can run concurrently
- ❖ Provide short response times to small jobs
- ❖ Improve utilization and throughput

1. Hadoop Architecture Overview

■ More on Fair Scheduler Basics:

□ Group jobs into “pools”:

- ❖ Guaranteed minimum Map slots (# of Mappers)
- ❖ Guaranteed minimum Reduce slots (# of Reducers)
- ❖ Pool is set of resources shared across jobs

□ Limits on # of running jobs:

- ❖ Per user
- ❖ Per pool



2. Hadoop Distributed File System: HDFS



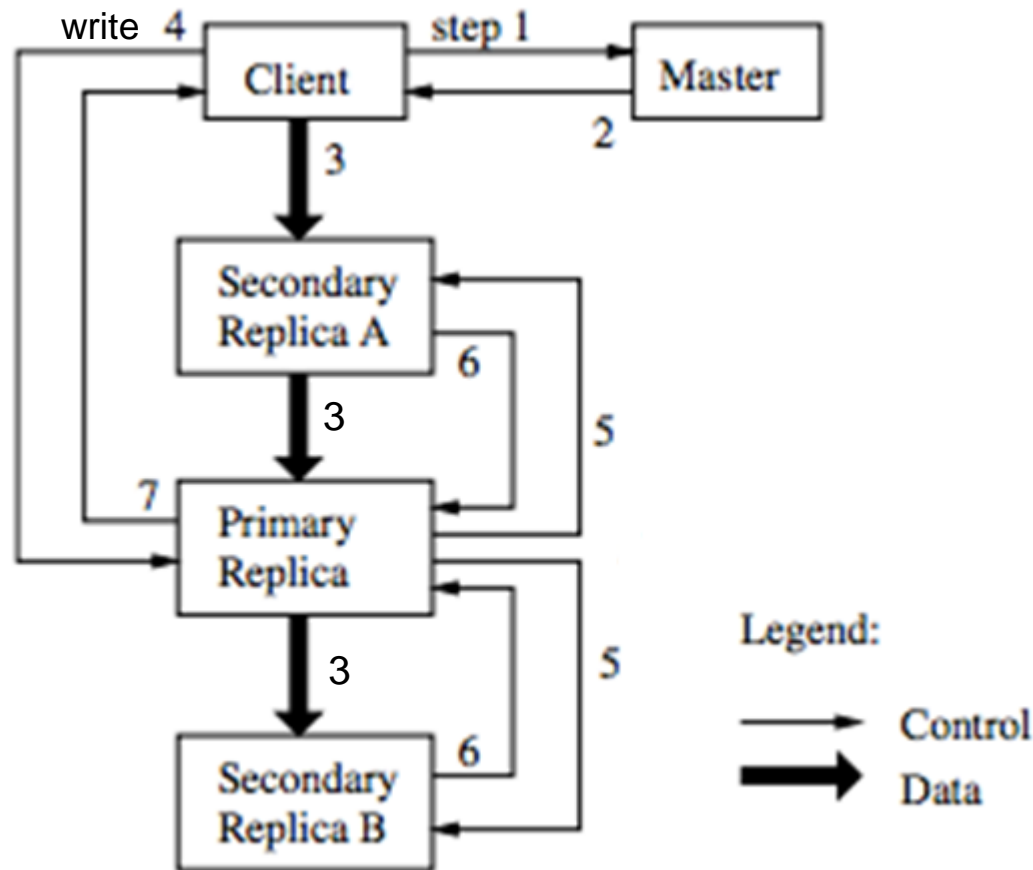
GFS Motivation

2. GFS* Motivation

- ❑ Measure then build
- ❑ **Google workload characteristics:**
 - ❑ huge files (GBs); usually read in their entirety
 - ❑ almost all writes are appends
 - ❑ concurrent appends are common
 - ❑ high throughput is valuable
 - ❑ low latency (rpt) is not
- ❑ **Computing environment:**
 - ❑ 1000s of machines
 - ❑ Machines sometimes fail (both permanently and temporarily)

* https://en.wikipedia.org/wiki/Google_File_System

2. GFS Overview: Steps of GFS write



- Client asks master (1) for identity of primary and secondary replicas
- Client pushes (3) data to memory at all replicas via a replica-to-replica “chain”
- Client sends write request (4) to primary
- **Primary (5): assign seq num to each write**
- Primary orders concurrent requests (5+6), and triggers disk writes at all replicas
- Primary reports (7) success or failure to client

Performance Optimization: Data flows with most efficient network path

Correctness: Control flow ensures data committed in same order

2. Why not use NFS?

1. **Scalability:** Must store > 100s of Terabytes of file data:
 - ❖ **NFS only exports** a local FS on one machine to other clients
 - ❖ **GFS solution:** store data on many server machines
2. **Failures:** Must handle temporary and permanent failures:
 - ❖ **NFS only recovers from temporary failure**
 - not permanent disk/server failure – no replication
 - recovery means making reboot invisible
 - **technique:** retry (stateless and idempotent protocol helps)
3. **GFS solution:** replication and failover (like RAID)



GFS Workload

2. GFS Workloads

- **Most files are mutated** (modified) by appending new data – large sequential writes
- **Random writes** are very uncommon
- **Files are written once**, then they are only read many times
- **Reads are sequential**
- **Large streaming reads** and small random reads
- **High bandwidth** (throughput) is more important than low latency (rpt)
- **Google applications:**
 - ❑ **Data analysis programs** that scan through data repositories
 - ❑ **Data streaming** applications
 - ❑ **Archiving**
 - ❑ **Applications producing** (intermediate) search results



HDFS Architecture

2. HDFS: Overview

- FS: <http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/fs/>

- **Single PetaByte file system**

- ❑ Files split into 64 MB blocks (**shards**)
- ❑ Blocks are replicated across several data nodes (x3 default)

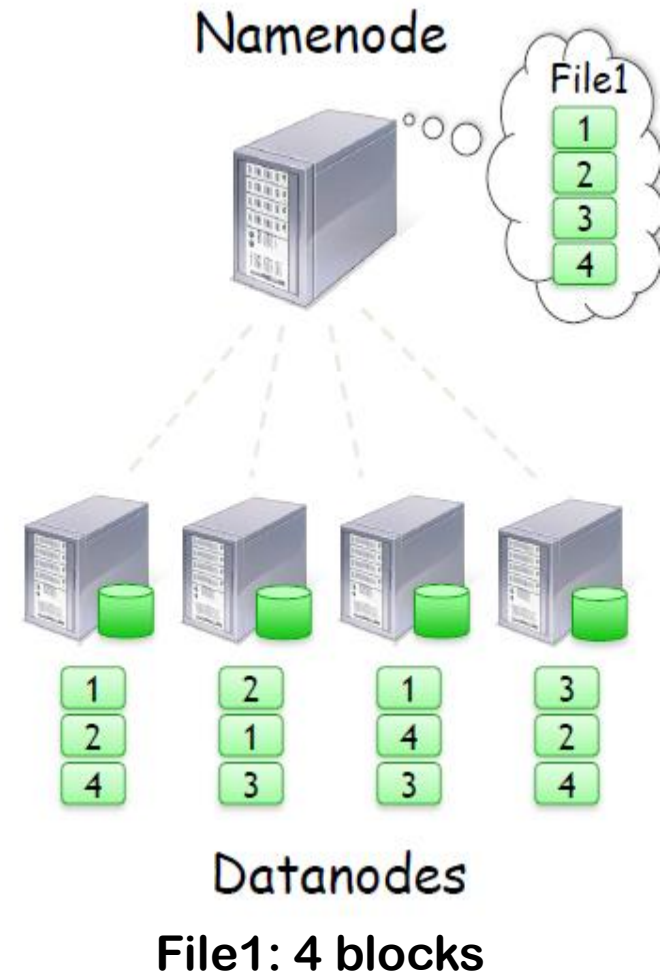
- **Single namenode** stores metadata (file names, block locations, etc.)

- **Optimized** for large files, sequential reads

- **Files are append-only** (no random modification)

- **Robust to failures**; no need for backup (i.e., replication)

- **Multiple sources/replicas** for any one piece of data



2. HDFS: Overview

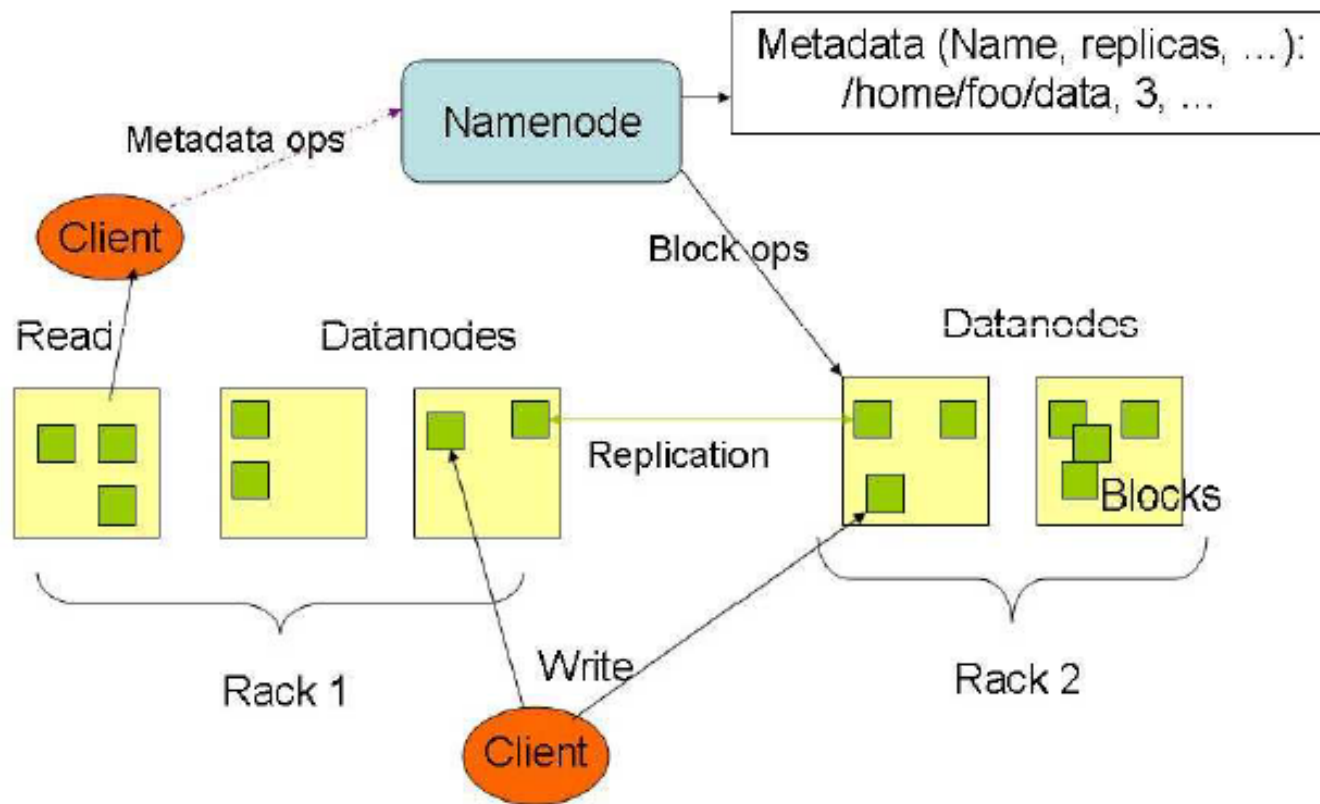
- FS: <http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/fs/>

- **Single Namespace** for the entire cluster
- **Data Coherency**
 - ❑ Write-once-read-many access model
 - ❑ Client can only append to existing files
- **Files are broken up into blocks**
 - ❑ Default is 64 MB block size
 - ❑ Each block replicated on multiple DataNodes
- **Intelligent Client**
 - ❑ Client can find location of blocks from the Name Node
 - ❑ Client accesses data directly from DataNode

2. HDFS: Operations

- FS: <http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/fs/>

■ HDFS Architecture:



Client → NameNode → Client → DataNode

2. HDFS: Overview

- FS: <http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/fs/>

■ NameNode Metadata:

- ❑ The entire metadata is (resident) in main memory
- ❑ No demand paging of metadata

■ Types of Metadata:

- ❑ List of files
- ❑ List of Blocks for each file
- ❑ List of DataNodes for each block and its replicas
- ❑ File attributes, e.g., creation time, replication factor, etc.

■ A Transaction (Edit) Log:

- ❑ Records file creation, file deletion, operations on the file, etc.

2. HDFS: Overview

- FS: <http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/fs/>

■ DataNode

□ A Block Server – to clients:

- ❖ 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 – to Name Node:

- ❖ Periodically sends a report of all existing blocks to the NameNode (primary)

□ Facilitates Pipelining of Data:

- ❖ Forwards data to other specified DataNodes during write operation

2. HDFS: Overview

- FS: <http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/fs/>

■ Block Placement:

□ Current Strategy:

- ❖ One replica on local node
- ❖ Second replica on a remote rack
- ❖ Third replica is randomly placed
- **Different blocks** from a given file in one rack are placed close to each other for highest bandwidth (minimum seek).
- **Clients read** from nearest replica
- Would like to make this policy pluggable!

■ Heartbeats:

- DataNodes send heartbeat to the NameNode; **once every 3 seconds**
- NameNode use heartbeats (timeout = 10 minutes) to detect DataNode failure

2. HDFS: Overview

- FS: <http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/fs/>

■ High Availability leveraging Replication:

□ **NameNode detects DataNode failure:**

- ❖ Chooses new DataNodes for new replica
- ❖ Balances disk usage
- ❖ Balances' communication traffic to DataNodes

■ Data Correctness:

□ **Use checksum to validate data for a block**

- ❖ Use CRC32

□ **File creation**

- ❖ Client computes checksum per 512 bytes (disk sector)
- ❖ DataNode stores the checksum in the DataNode/block

□ **File access**

- ❖ Client retrieves the data and checksum from DataNode
- ❖ If validation fails, client tries other replicas

2. HDFS: Overview

- FS: <http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/fs/>

■ NameNode Failure:

- ❑ **A single point of failure**
- ❑ **Transaction/Edit Log** is 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 solution for the NameNode!**

■ Data Pipelining – for write:

- ❑ Client retrieves a list of DataNodes (e.g., 3 nodes), from the NameNode, on which to place replicas of a block
- ❑ Client writes block to the first (closest) DataNode
- ❑ The first DataNode forwards the data and the remaining nodes list to the next DataNode in the pipeline
- ❑ When all replicas are written, the client moves on to write the next block in the file.

2. HDFS: Overview

- **FS:** <http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/fs/>

■ Rebalancer:

- ❑ **Goal:** the percentage of disk utilization (full) on DataNodes should be similar/comparable:
 - ❖ **Usually, Rebalancer** is run **when new DataNodes are added**
 - ❖ **Cluster is online** when Rebalancer is active
 - ❖ **Rebalancer is throttled** to avoid network congestion
 - ❖ **Available also** as a **Command line tool** (i.e., executed on-demand)

■ Secondary NameNode:

- ❑ **Copies** FileSystem Image (FsImage) and Transaction Log (EditLog) from primary NameNode to temporary directory
- ❑ **Merges** FileSystem Image and Transaction Log into a new FileSystem Image (FsImage) in a temporary directory
- ❑ **Uploads new FileSystem Image** to the primary NameNode and to disk:
 - ❖ **Transaction (Edit) Log** on primary NameNode is purged/empty

2. HDFS: Commands Shell

<http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSCommands.html>

■ Execute:

- ❑ Echo alias hfs = 'hadoop fs ' >> .bashrc
- ❑ Source .bashrc // executed alias hfs = 'hadoop fs '

■ Common Hadoop FS shell commands:

- ❑ hfs // See available commands
- ❑ hfs -Help // more command details

// change ownership recursively to all files in the directory

- ❑ hfs -chown [-R] [new-owner] <dir>
- ❑ hfs -ls

2. HDFS: Commands Shell

<http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSCommands.html>

■ Remote access commands:

- ❑ `hfs -cat <src>` // Cat content to stdout
- ❑ `hfs -copyFromLocal <localsrc> <dst>` // Copy from Linux to HDFS
- ❑ `hfs -copyToLocal <src> <localdst>` // Copy from HDFS to Linux

- The file system is browsable.
- Local is Linux FS

2. Local - HDFS: Commands Shell

<https://developer.yahoo.com/hadoop/tutorial/module2.html#commonops>

■ Moving/copying file between Local FS and Hadoop HDFS:

□ Pulling files out from my Linux home directory to HDFS:

- `hadoop fs -CopyFromLocal /home/$USERNAME/out ~/myfiles_out`
- `hadoop fs -MoveFromLocal /home/$USERNAME/out
hdfs://dc-hadoop-name1.hadoop.dc.engr.scu.edu:8020/user/aezzat/myfiles_out`
- `hadoop fs -Put /home/$USERNAME/out ~/myfiles_out`

□ Pulling files out from HDFS into my Linux home directory:

- `hadoop fs -CopyToLocal /user/$USERNAME/out ~/myfiles_out`
- `hadoop fs -MoveToLocal /user/$USERNAME/out
/home/aezzat/out/myfiles_out`
- `hadoop fs -Get /user/$USERNAME/out ~/myfiles_out`

□ Moving files between file systems is not permitted:

- `hadoop fs -mv URI [URI...] <dest>`
- `hadoop fs -mv hdfs://dc-hadoop-name1.hadoop.dc.engr.scu.edu:8020/user/aezzat/myfile1
hdfs://dc-hadoop-name1.hadoop.dc.engr.scu.edu:8020/user/aezzat/myfile2
hdfs://dc-hadoop-name1.hadoop.dc.engr.scu.edu:8020/user/aezzat/dir`

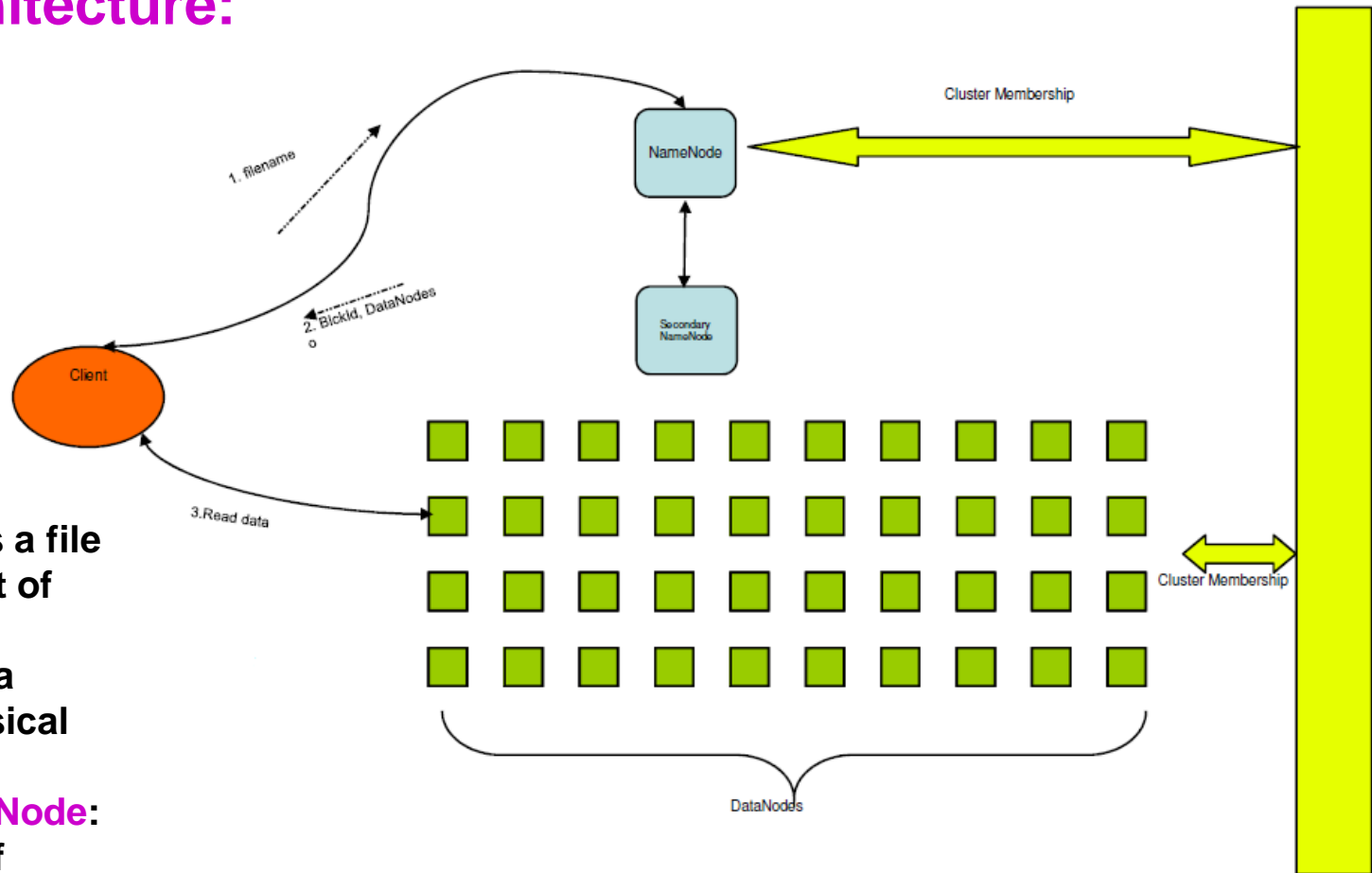
2. Local - HDFS: Commands Shell

<https://developer.yahoo.com/hadoop/tutorial/module2.html#commonops>

- **Moving/copying file between Local FS and Hadoop HDFS:**
 - ❑ **MoveFromLocal:** copy file from Local/Linux to HDFS then delete the copy in the local/Linux FS
 - ❑ **Put:** copy file from local file system/Linux to HDFS. At the end we have two copies of the same file in different file systems
 - ❑ **CopyFromLocal** is similar-to **Put**; however, source is limited to Local FS. So you can do with Put all what you can do with CopyFromLocal but not vice-versa. **Put** can copy from HDFS to HDFS, i.e., source does not have to be local file system.
 - ❑ **MoveToLocal:** copy file from HDFS to Local/Linux then delete the copy in HDFS
 - ❑ **Get:** copy file from HDFS to local file system/Linux. At the end we have two copies of the same file in different file systems
 - ❑ **CopyToLocal** is similar to **Get**; destination is limited to Local FS so you can do with Get all what you can do with CopyToLocal but not vice-versa. **Get** can copy from HDFS to HDFS. With **Get** the destination does not have to be Local.

3. HDFS: Operations

HDFS Architecture:



NameNode: Maps a file to a file-id and list of DataNodes.

DataNode: Maps a block-id to a physical location on disk

Secondary NameNode: Periodic merge of transaction log



3. MapReduce

3. MapReduce Overview: Motivation

- **Data-Intensive Scalable Computing (DISC) has arrived:**
 - ❑ Both user-facing (interactive) services and batch data processing
 - ❑ Data analysis is key
- **Need massive Horizontal scalability and easy parallelism:**
 - ❑ PB's of data, millions of files, 1000's of nodes, millions of users.
- **Need to do this cost effectively and reliably:**
 - ❑ Use commodity hardware where failure is the norm
 - ❑ Share resources among multiple projects/applications

MapReduce to the rescue!

3. MapReduce Overview: What is MR

- **Simple data-parallel programming model and framework:**
 - ❑ Designed for scalability and fault-tolerance
- **Pioneered/invented by Google**
 - ❑ Process **20 PB** of data per day!
- **Popularized by open-source Hadoop project**
 - ❑ Used at Yahoo!, Facebook, Amazon, etc.
- **Applications:**
 - ❑ **Google:** index construction for search, article clustering for Google news, statistical machine translation
 - ❑ **Yahoo!:** web-map and spam detection for Yahoo! mail
 - ❑ **Facebook:** Ad optimization and spam detection

3. MapReduce Overview: What is MR

- ❑ **Data cleaning:**

- ❖ Preprocess data in order to reduce noise and handle missing values – goal is to improve data quality → learning

- ❑ **Relevance analysis:**

- ❖ Remove the irrelevant or redundant attributes using correlation analysis

- ❑ **Data transformation and reduction:**

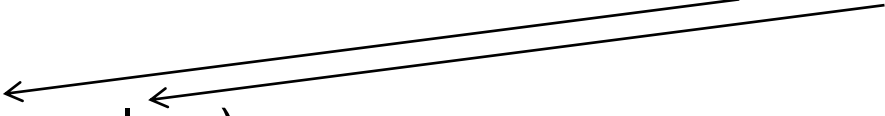
- ❖ Generalize to higher-level concepts, and/or normalize data (an attribute value is scaled to be between 0.0 – 1.0), especially if neural networks or distance measurements are used in the learning step. Data can be Reduced by applying methods ranging from wavelet transformation to discretization techniques

3. MapReduce Overview: Programming Model

- **APIs:** <http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/mapred/>

- **Input data type:** file of K/V pair records
- **Map function:** $(K_{in}, V_{in}) \rightarrow \text{list}(K_{inter}, V_{inter})$
- **Reduce function:** $(K_{inter}, \text{list}(V_{inter})) \rightarrow \text{list}(K_{out}, V_{out})$
- **Example:**

```
def mapper(line):  
    foreach word in line.split();  
        output(word, 1);           // (Kinter, Vinter)  
  
def reducer(key, values):  
    output(key, sum(values));      // list(Kout, Vout)
```



2. MapReduce Overview: Programming Model

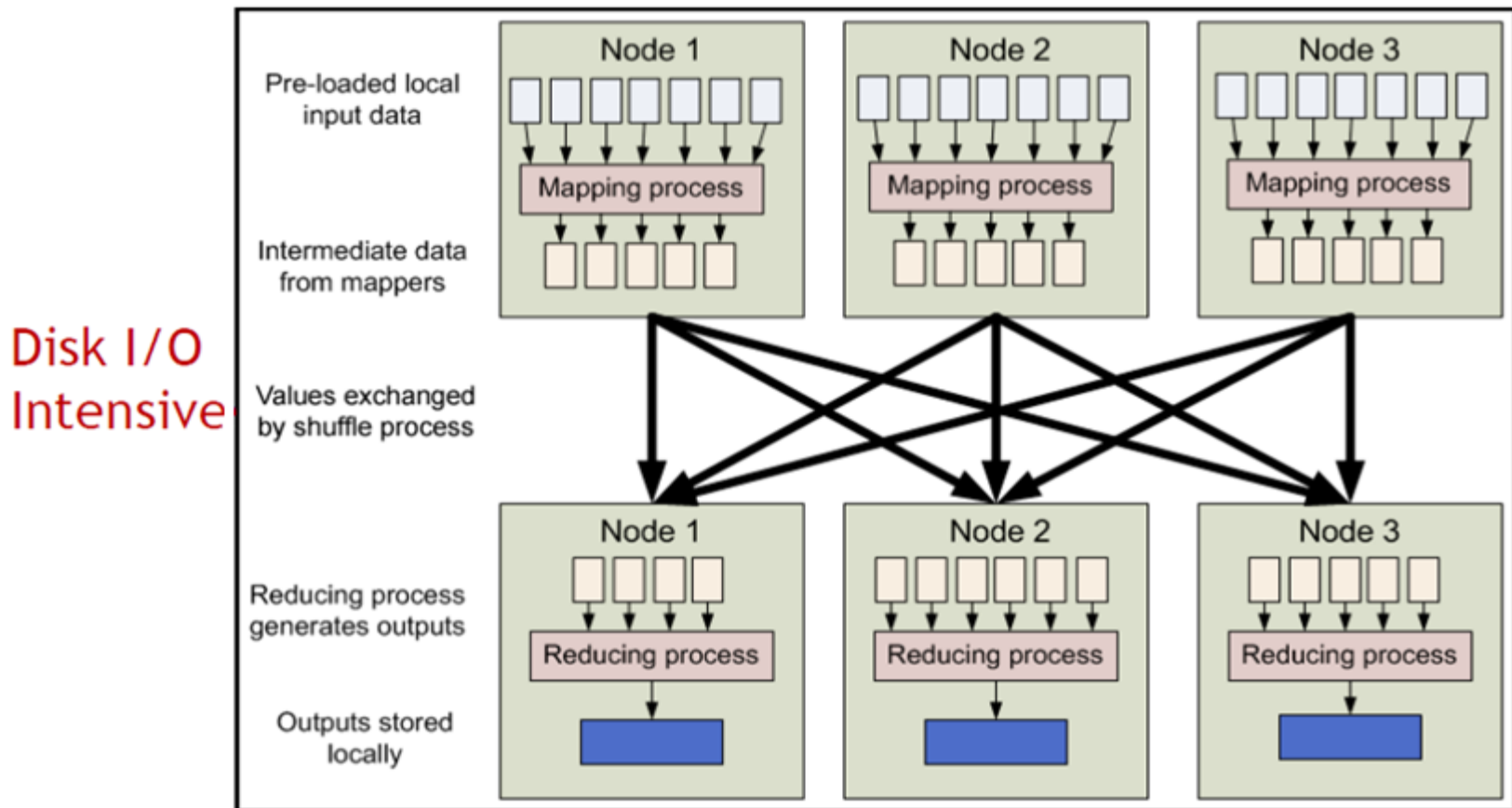
- **APIs:** <http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/mapred/>

- **Push:** input split into large chunks and are placed on local disks of cluster nodes
- **Map:** chunks (data to map tasks) are served to “mapper”
 - Prefer mapper that has data locally
 - Mappers save outputs (intermediate result) to local (Linux) disk before passing them to reducers' memory; allows efficient recovery
- **Reduce:** reducers execute reduce tasks only when all the map phase is completed

3. MapReduce Overview: Programming Model

- APIs: <http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/mapred/>

The Big Picture



High degree of parallelism: Master/Slave architecture

3. MapReduce Overview: Programming Model

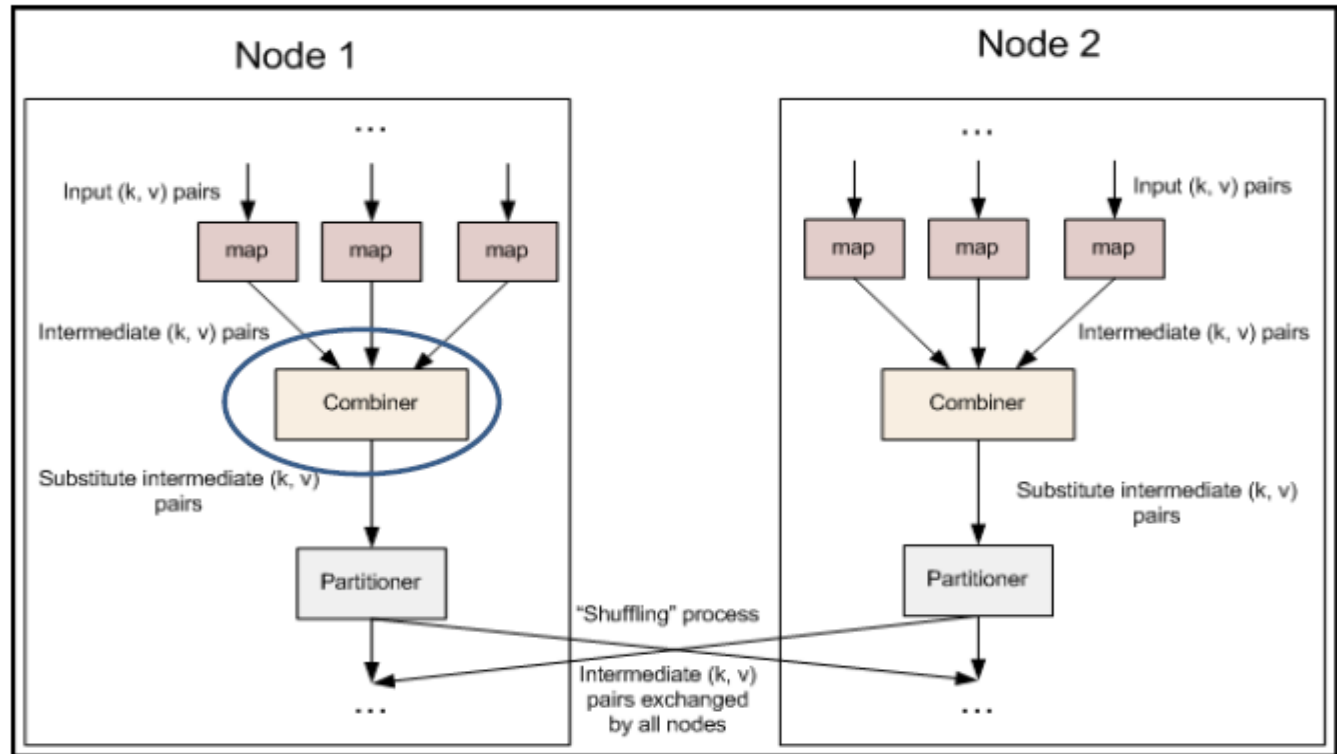
- **APIs:** <http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/mapred/>

- **Partitioning/Shuffling:** divide intermediate key space across reducers:
 - ❑ K reduce tasks → K partitions (simple hash function = %k)
 - ❑ E.g., K (# of reducers) = 3, keys {3,6}, {1,4}, {2,5}
- **Shuffle/exchange:**
 - ❑ Since all mappers typically have all intermediate key values
 - ❑ It is all-to-all communication between mappers and reducers
- **Serial workflow by default:**
 - ❑ Research groups have explored pipelining

3. MapReduce Overview: Programming Model

- APIs: <http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/mapred/>

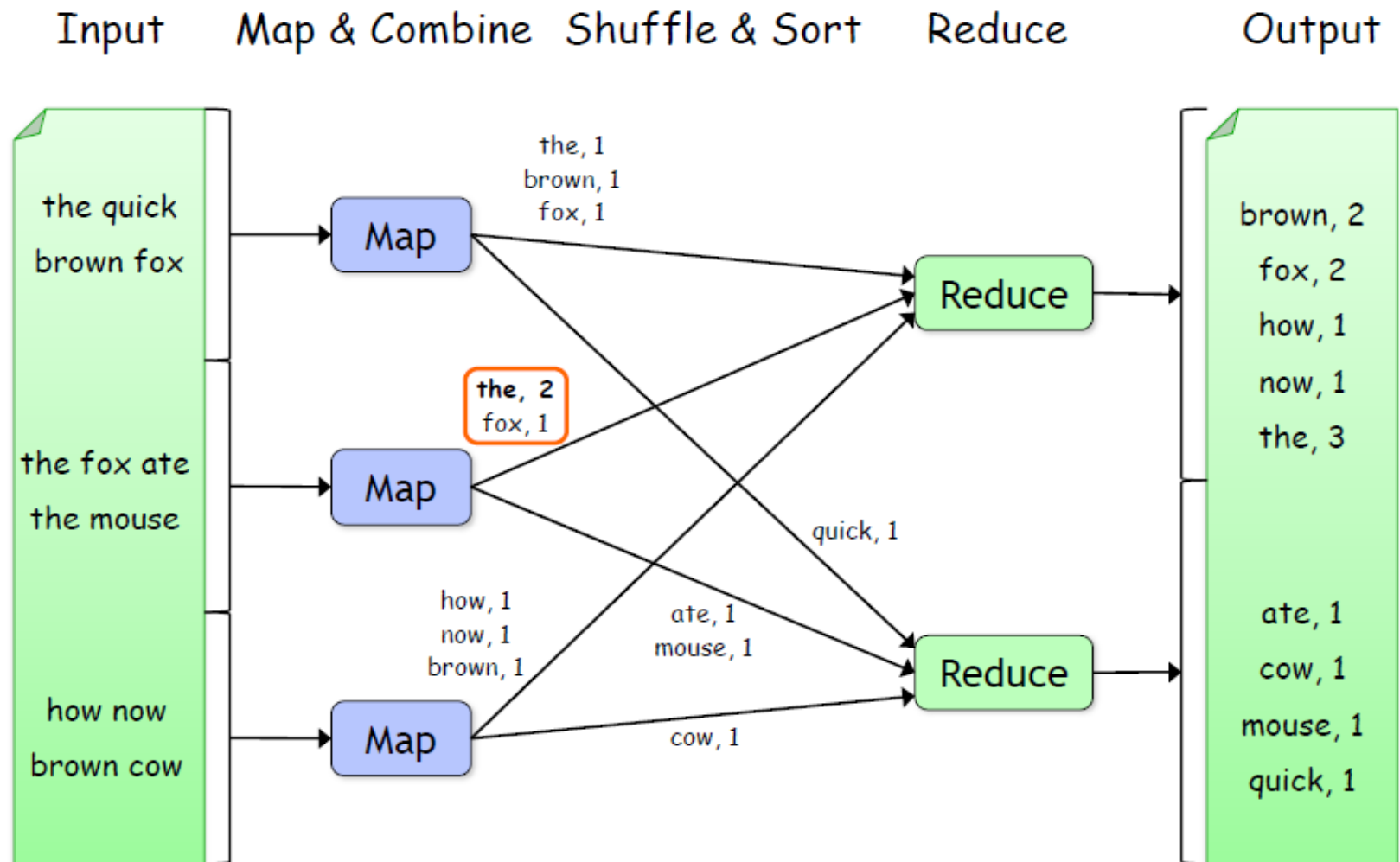
■ Combiner:



- A **combiner** is a local aggregation function for repeated keys produced by the same mapper ← **reduce shuffling**

3. MapReduce Overview: Programming Model

- **APIs:** <http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/mapred/>



Word Count with Combiner

3. MapReduce Overview: Programming Model

- **APIs:** <http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/mapred/>

■ Fault Tolerance in MapReduce:

□ If a task crashes:

❖ Retry on another node

- OK for a map because it has no dependencies
- OK for reducer because map outputs are on disk

□ If a node crashes:

- ❖ Re-launch its current tasks on other nodes
- ❖ Re-run any mappers the node was running when crashed to get output intermediate data

□ If a task is going slowly:

- ❖ Launch second copy of task on another node (“speculative execution”)

3. MapReduce Overview: Programming Model

- **APIs:** <http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/mapred/>

■ Example: Inverted Index

Regular Index: find word in a given file

Inverted Index: given a word find list of files containing the word

❑ **Input:** (filename, text) records

❑ **Output:** list of files containing each word <word : list of files>

❑ **Map:**

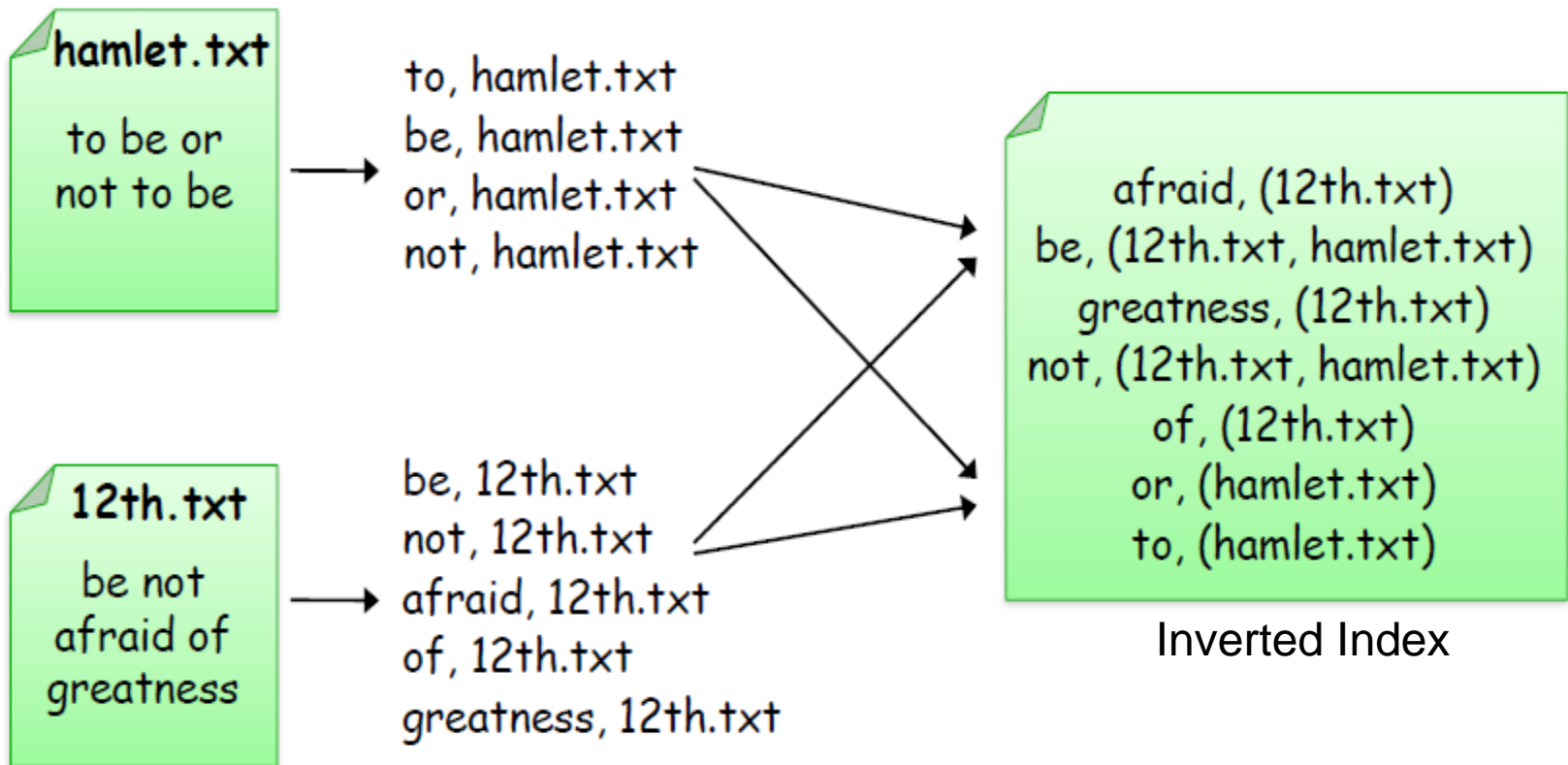
```
foreach word in text.split():  
    output(word, filename)
```

❑ **Reduce:**

```
def reduce(word, filenames):  
    output(word, sort(filenames))
```


3. MapReduce Overview: Programming Model

- **APIs:** <http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/mapred/>



Inverted Index Example

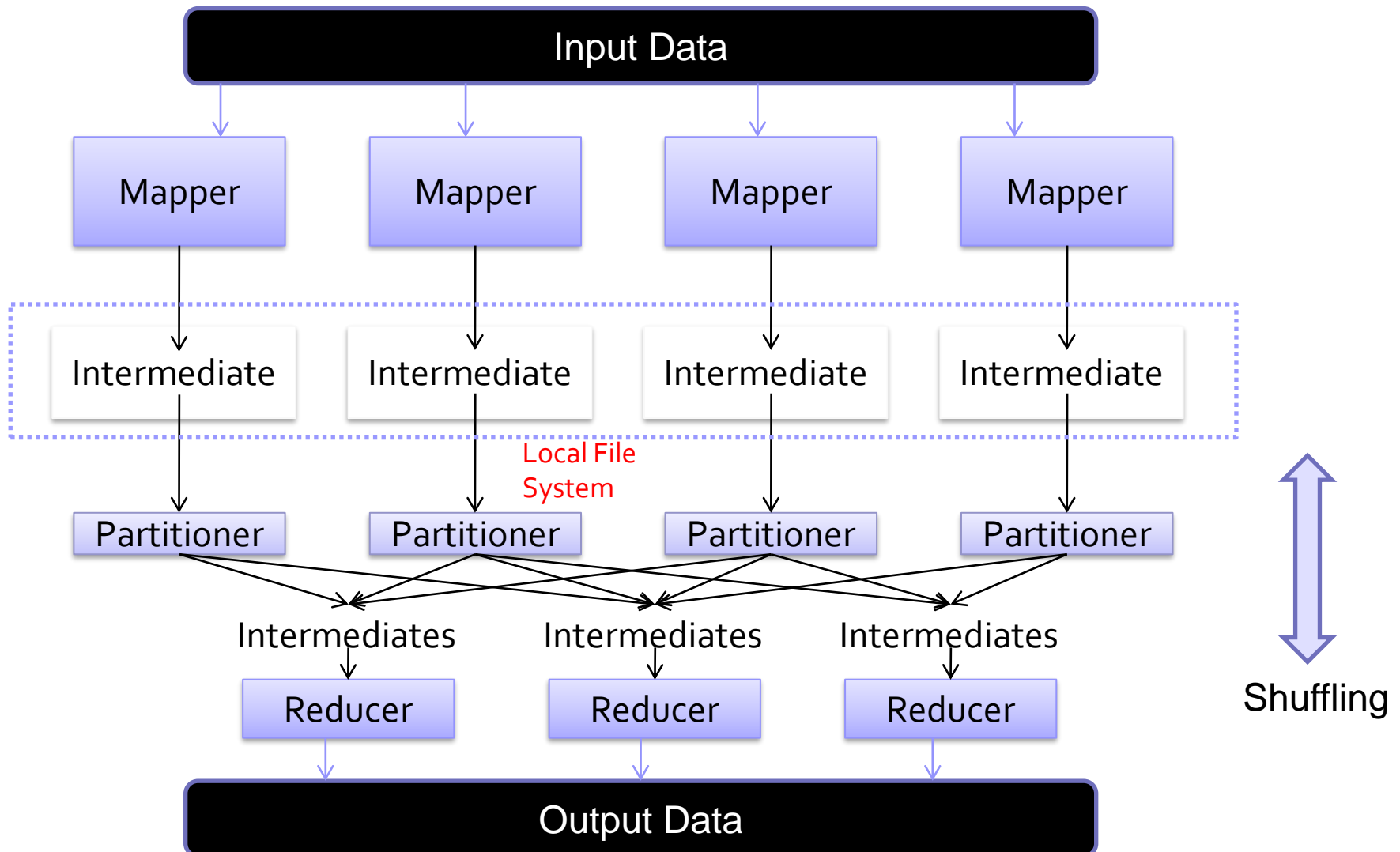
3. MapReduce Overview: Programming Model

- **APIs:** <http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/mapred/>

■ What applications may perform well on MR?

- ❑ Modest computing/Processing relative to data, i.e., it is DISC (**Data Intensive Scalable Computing**) platform.
- ❑ Data-independent processing of maps
- ❑ Data-independent processing of keys
- ❑ Smaller ballooning of map output relative to input

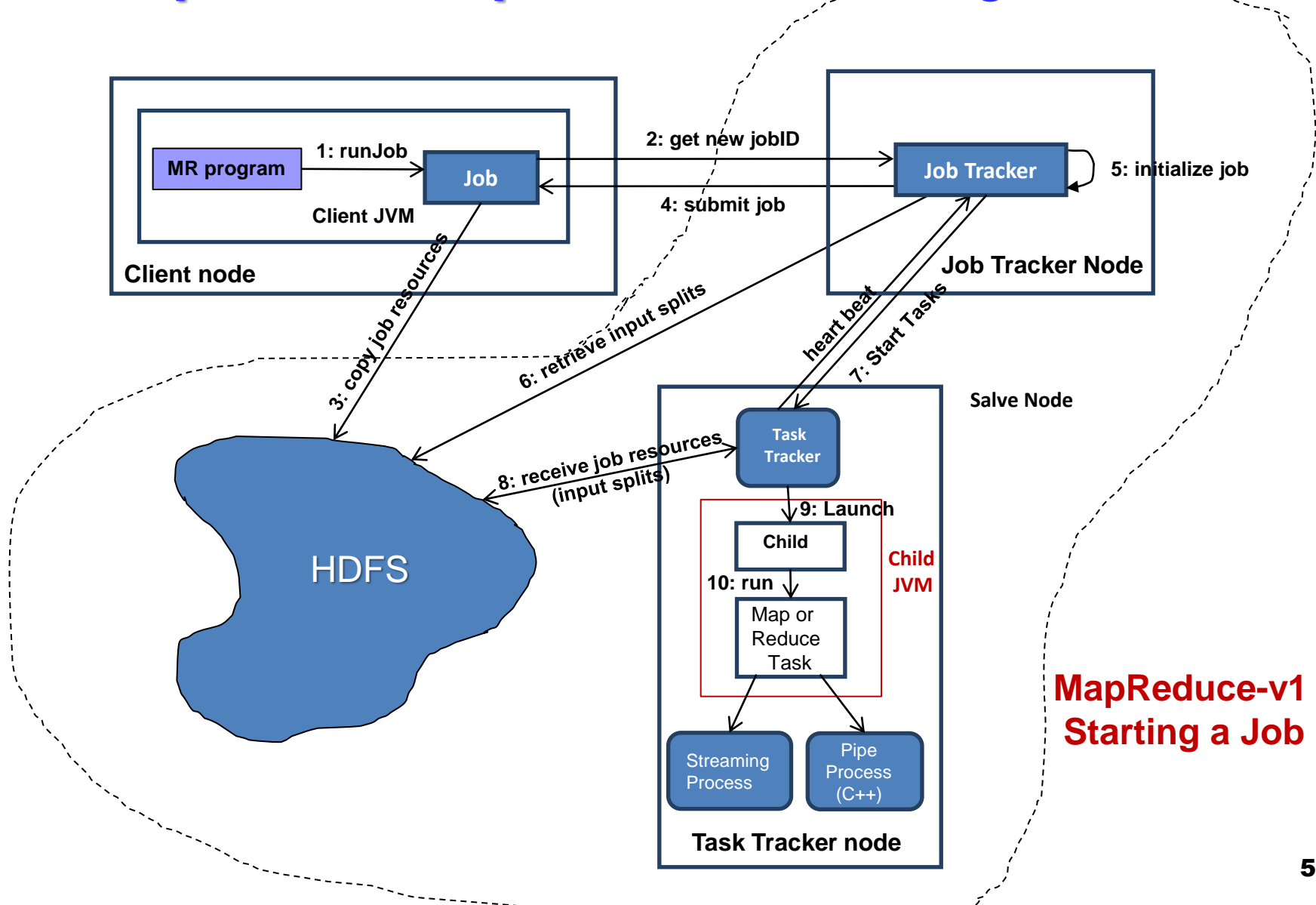
3. Hadoop IO: Shuffling Optimization in Hadoop MR



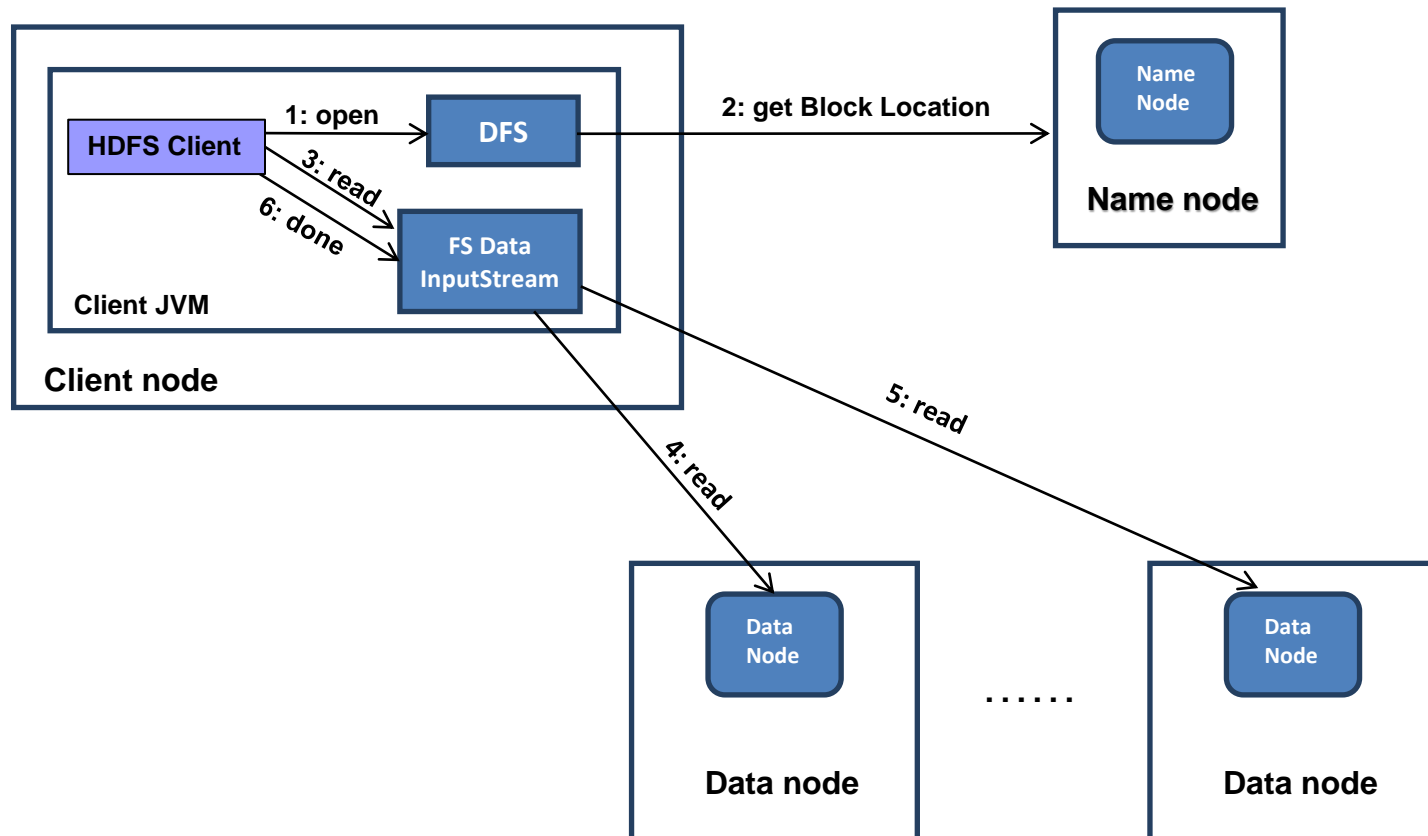
3. Hadoop IO: Shuffling Optimization in Hadoop MR

- Mapper needs to write to local file system, then reducer needs to read from the file system → **2 I/Os per block**.
- Main observation is shuffling hurts response time. Also, no reducer can start processing till all mappers are completed!
- **Optimization approach:**
 - Put MapOutputFile into memory in addition to local disk
 - Two-level intermediate data/file storage
 - ❖ In-memory
 - ❖ Local disk file (traditional) – needed for recovery
- The main idea is to pass the data from mapper output to reducer input through “memory and IPC” and in parallel still do the disk IO for recovery.
- Absence of failure, reducers can start much earlier as IPC is much faster than 2 I/Os
- **Mellanox Technologies** optimization implements the above approach and uses RDMA instead of IPC to pass the data between mapper and reducer.
http://www.mellanox.com/page/products_overview

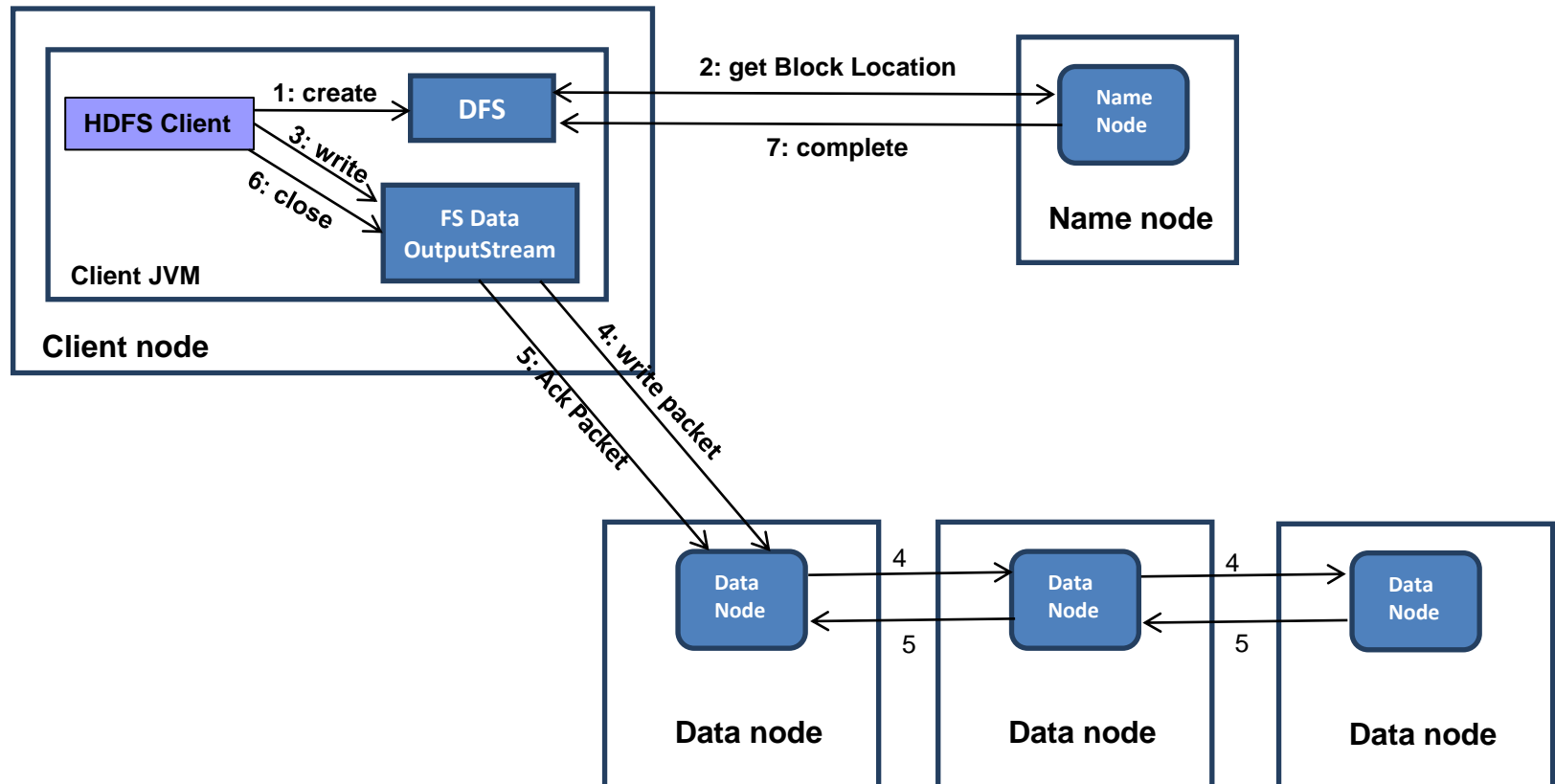
3. MapReduce Operations: Starting a Job



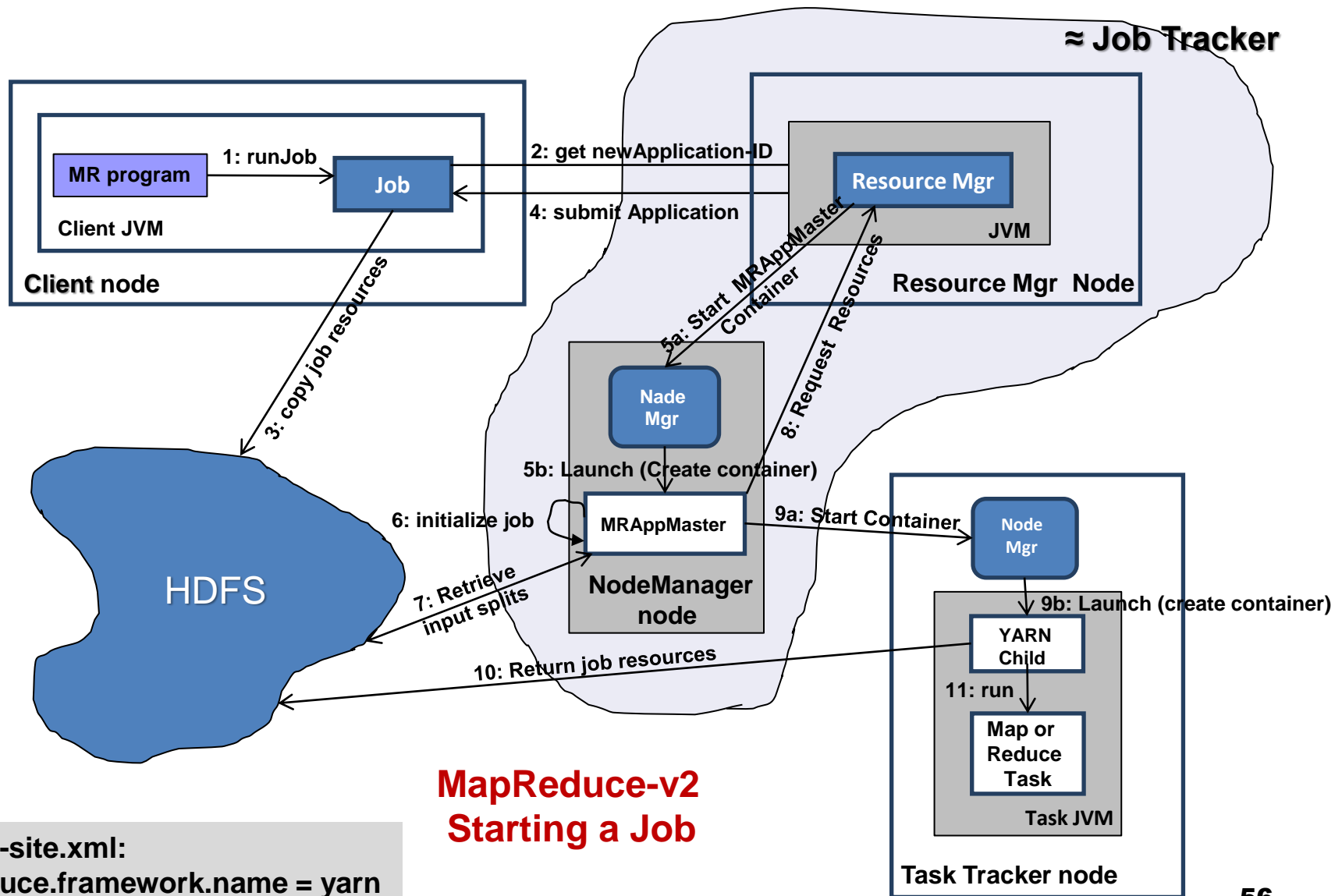
3. MapReduce Operations: Read Anatomy



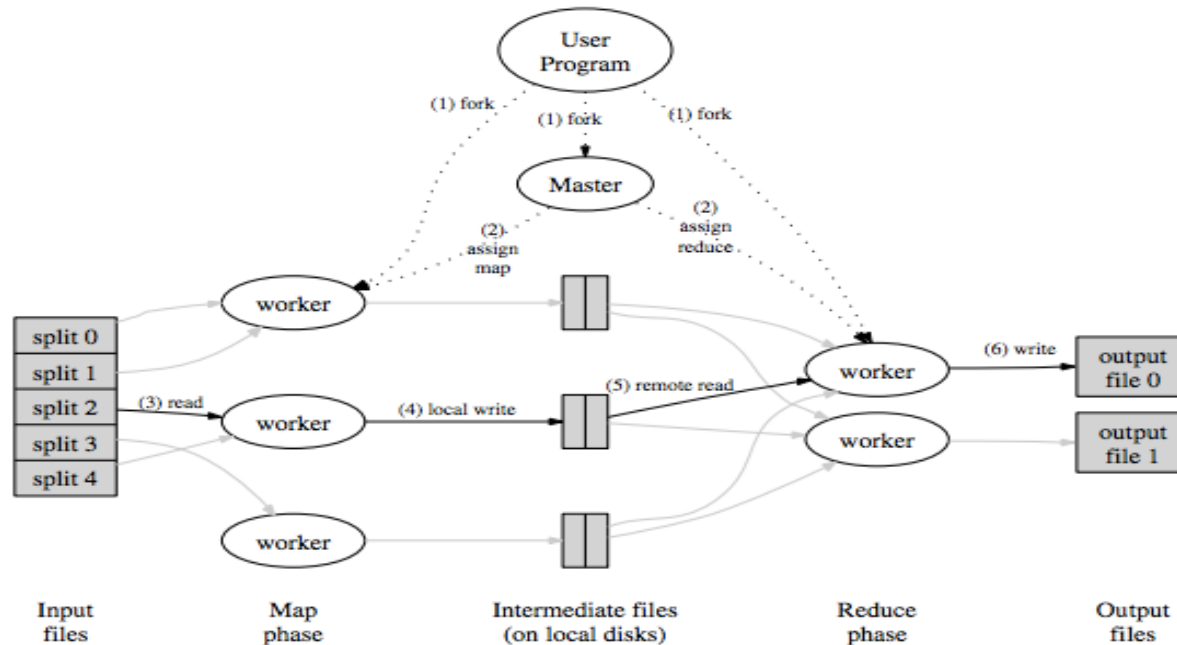
3. MapReduce Operations: Write Anatomy



3. MapReduce Operations: Starting a Job



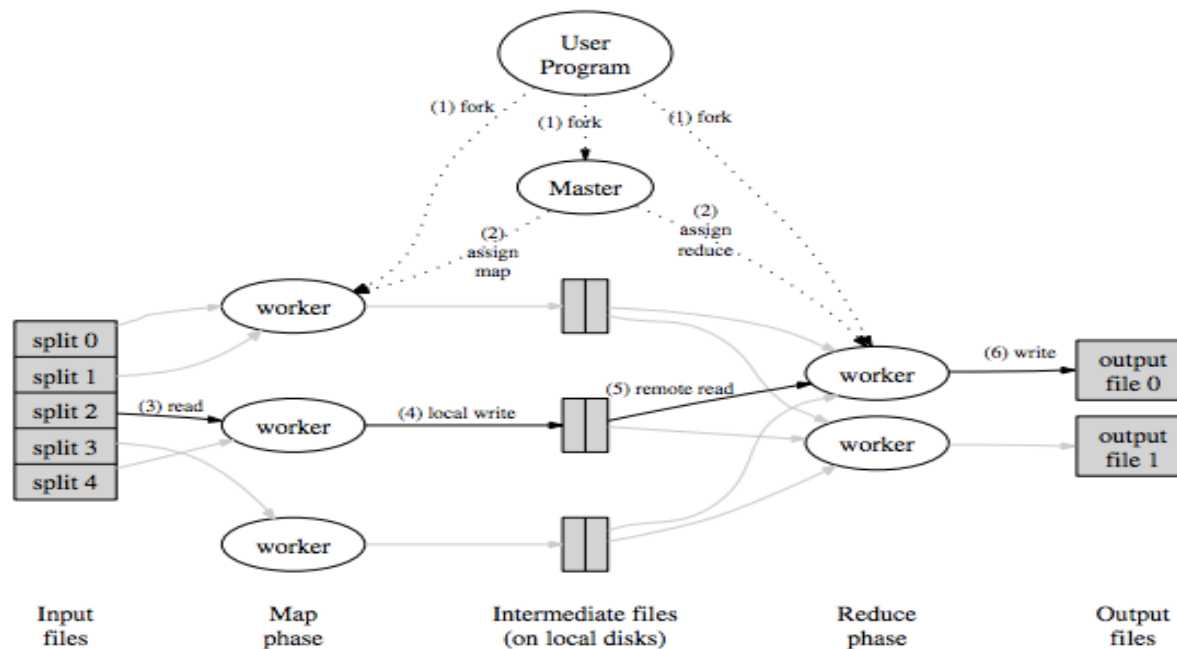
3. MapReduce Operations: Failed Tasks



- ❑ MapReduce App Master tracks status of all map and reduce tasks for an application
 - ❑ If any tasks don't respond to pings, what should master do?
 - ❖ Restart mapper or reducer tasks on new machines;
 - ❖ Possible because tasks are deterministic and idempotent
- System still has all inputs

MapReduce Configuration:

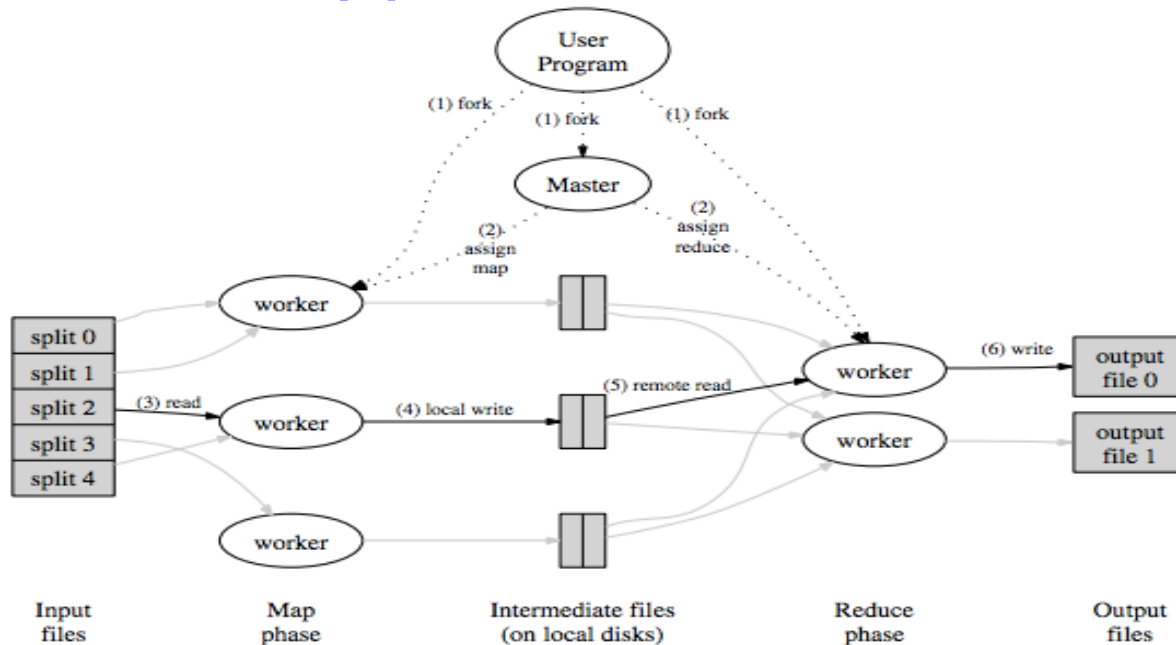
Number of Mappers and reducers



- ❑ What does the value of M (number of mappers) influence?
- ❑ What if M is too big?
- ❑ What if M is too small?
- ❑ Goal: Choose M to control size of input data...

MapReduce Configuration:

Number of Mappers and reducers



- ❑ What if R (Reducers) is too big?
- ❑ What if R is too small?
- ❑ **Goal:** M and R much larger than number of worker machines
 - Each worker performing many different tasks improves dynamic load balancing
 - Speeds up recovery if worker fails: its many completed map tasks can be allocated across many other machines



END