Lecture 2: Overview of Hadoop: MapReduce and HDFS

CSCI4180 (Fall 2013)

Patrick P. C. Lee

Questions

- Google is one pioneer in the cloud computing technology
- How does Google perform analytics?
- How does Google store big data in a scalable and reliable way?

Outline

- Hadoop
- MapReduce
- HDFS
  - Design
  - Data flow
  - Basic operations

Big Data

- **Big data** is a term applied to data sets whose size is beyond the ability of commonly used software tools to capture, manage, and process the data within a tolerable elapsed time.
- Big data sizes are a constantly moving target currently ranging from a few dozen terabytes to many petabytes of data in a single data set.

From Wikipedia: http://en.wikipedia.org/wiki/Big_data

Big Data: Examples (as of 2008)

- The New York Stock Exchange generates about one terabyte of new trade data per day
- Facebook hosts approximately 10 billion photos, taking up one petabyte of storage
- The Internet Archive stores around 2 petabytes of data, and is growing at a rate of 20 terabytes per month.

Big Data: Challenges

Store in multiple disks

- Disk failures are common
  - Multiple disks are needed to store big data. Chance of having one failed disk is high
  - Failures may occur during analysis
- Most analysis tasks need to combine the data in some way
  - How to resolve data dependency is a big issue

Hadoop

- **Hadoop** is a reliable shared storage and analysis system. The storage is provided by Hadoop Distributed File System (HDFS) and analysis by MapReduce.
  - Note that Hadoop also has other components such as Pig, Hive, HBase, ZooKeeper

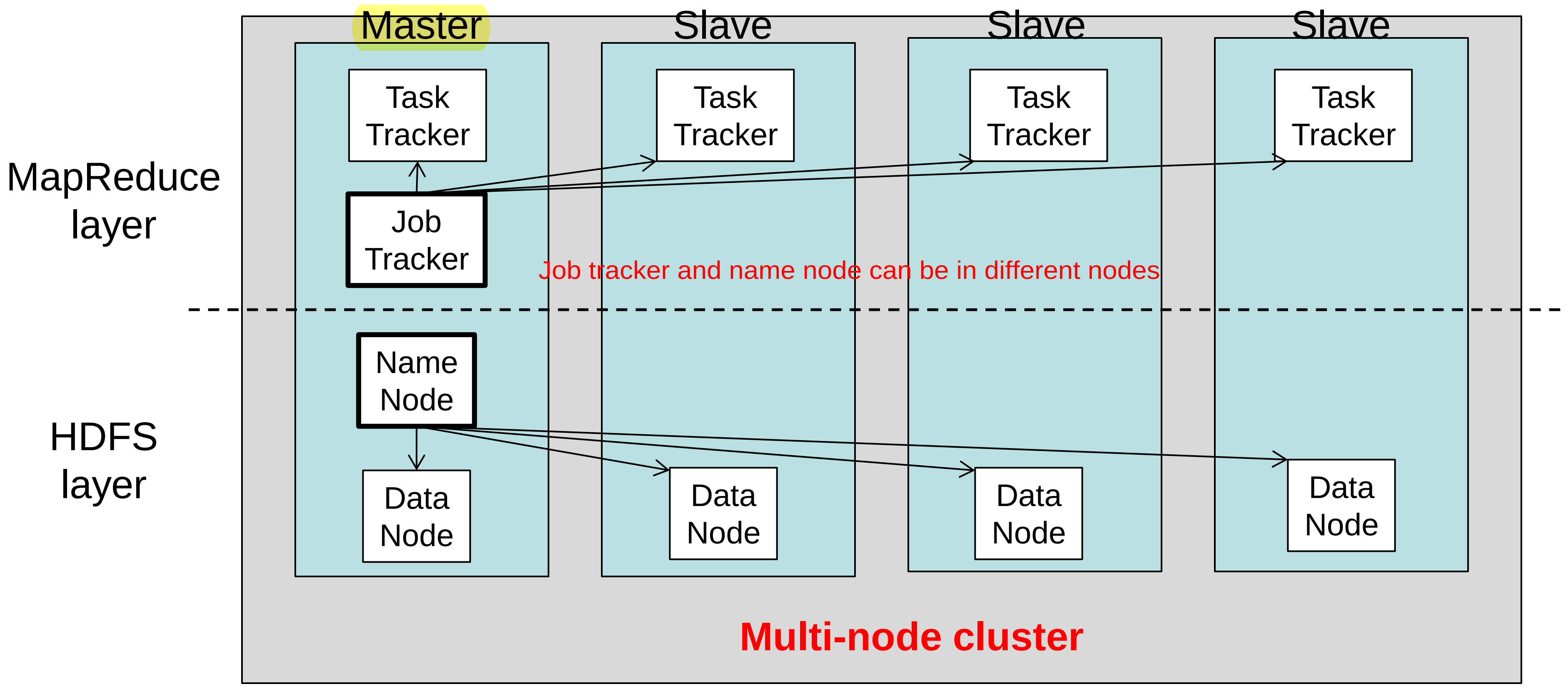
- Hadoop was created by Doug Cutting
- The name Hadoop is not an ancronym; it’s a made-up name

History of Hadoop

- Nutch, a web crawler and search system, was started in 2002 as part of the Apache Lucene project
  - The original goal is to index a billion of web pages; it's is realized that the goal cannot be met.
- Google published Google File System (GFS) and MapReduce papers in 2003 and 2004, respectively
- MapReduce was ported to Nutch in 2005
- Doug Cutting joints Yahoo! in January 2006
- Apache Hadoop project began in February 2006
- Hit web scale in early 2008

http://research.yahoo.com/files/cutting.pdf

Hadoop vs. Cloud



- Each node could be a physical machine, or a VM. The cluster forms a cloud environment.

Outline

- Hadoop
- MapReduce
- HDFS

Motivation

- Google needs to process large amounts of raw data
- Challenges: how to parallelize computation and distribute data over hundreds or thousands of machines in reasonable time
- **MapReduce** is a programming model that gives:
    - Automatic parallelization and distribution
    - Fault-tolerance
    - I/O scheduling
    - Status and monitoring

Dean and Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, OSDI ‘04

Programming Model

- Input and output: each a set of key/value pairs
- Two functions implemented by users

- **Map** `(k1,v1) → list(k2,v2)`
  - Takes an input key/value pair
  - Produces a set of intermediate key/value pairs

- **Reduce** `(k2,list(v2)) → list(k3,v3)`
  - Takes a set of values for an intermediate key
  - Produces a set of output values

Word Count Example

- Example: counting occurrences of each word in a large collection of documents

Memory problem on single problem

```
map(String key, String value)
  // key: document name
  // value: document content
  for each word w in value:
    Emit(w, "1")
```

Usually not string but an object

Not necessarily 1, can be do a batch operations and then give2,3,...

Same keys arrive to same reducer at the same time

Shuffle in sort

```
reduce(String key, Iterator values)
  // key: a word
  // values: a list of counts
  int result = 0
  for each v in values:
    result += ParseInt(v)
  Emit(key, AsString(result))
```

Start this part of the code when after all mappers finished the documents.
However, preparation work of reducer can be run parallel with the mapper!

Word Count Example

Document name

- ➢ The mapper takes an input key-value pair, tokenizes the document, and emits an intermediate key-value pair for each word
  Document content
  - word → key, integer 1 → value

Same word,go to same reducer, appear at same time

- ➢ The reducer sums up all counts associated each word (key), and emits final key-value pairs with the word as the key
  - All values of the same word (key) will go to the same reducer. This "group by" operation is handled by the execution framework called Shuffle and Sort.

at same time

More Examples

- **Distributed Grep**: map() emits a line if it matches an input pattern; reduce() copies intermediate data to output
- **Count of URL Access Frequency**: map() takes the logs and outputs (URL, 1); reduce() adds all values for same URL and output (URL, total)
- **Reverse Web-Link Graph**: map() outputs (target, source); reduce() emits list of sources for each target (target, list(source))

More Examples

- **Term-Vector (set of important words) per Host**: map() emits (hostname, term vector) for each input document; reduce() aggregates all documents and outputs (hostname, term vector)

- **Inverted Index**: map() emits (word, document ID); reduce outputs (word, list(document ID))

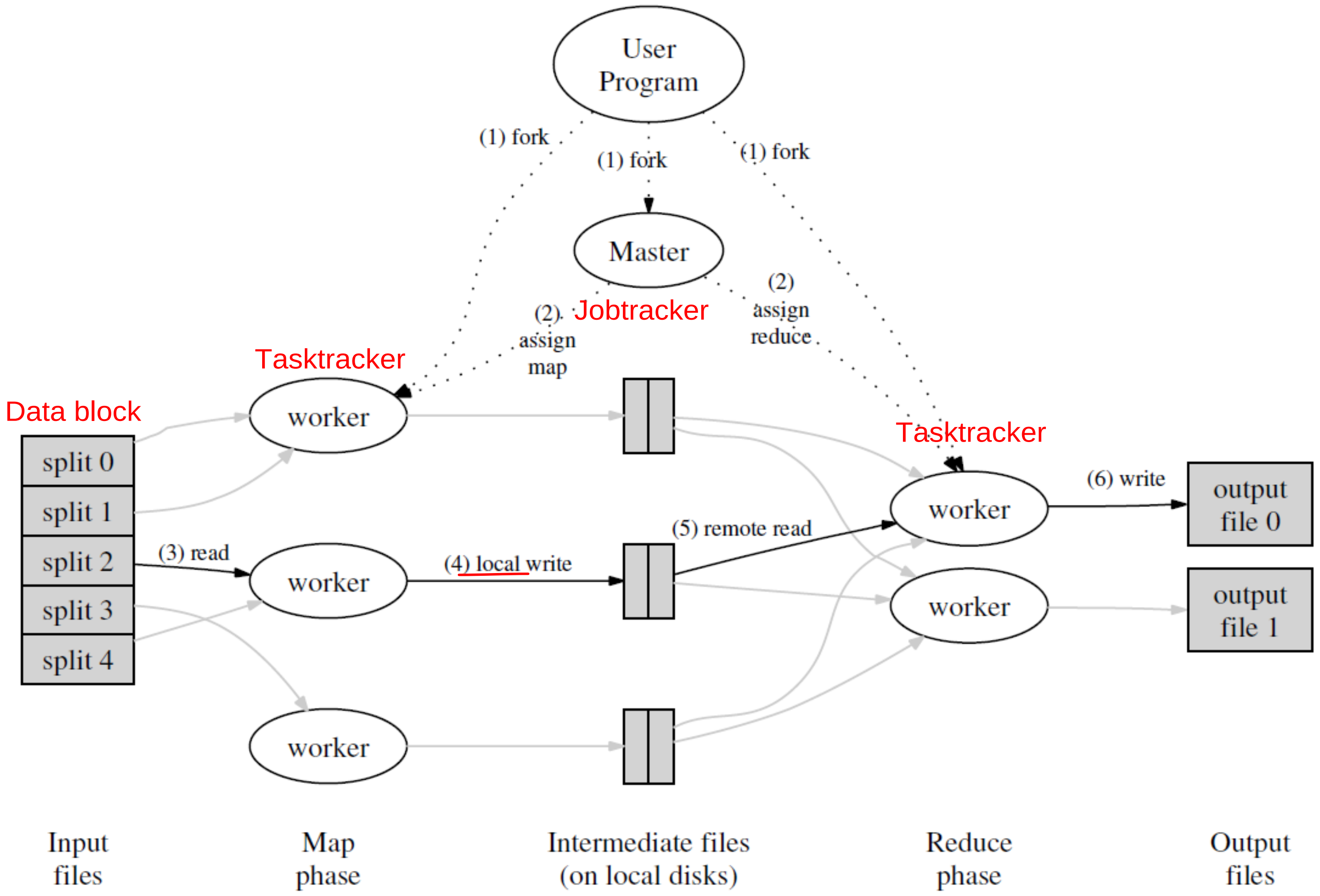
- **Distributed Sort**: map() extracts and emits each (key, record) pair, reduce() outputs all (key,record) pairs
  - will be discussed later

Implementation

Unreliable hardware, easy to have failure.

- MapReduce is deployable in a commodity environment:
  - Machines with commodity CPU (e.g., x86) and a few GB of memory (e.g., 2-4GB)
  - Commodity network hardware (e.g., 100Mb/s, 1Gb/s)
  - Hundreds/thousands of machines
  - Storage with inexpensive disks
- More sophisticated platforms (e.g., GPUs) are also allowed

Execution Overview



Adapted from OSDI'04 paper

MapReduce Data Flow in Hadoop

Parent process

- A MapReduce job is a unit of work that the client wants to be performed
  - with input data, the MapReduce program, and configuration information
- Hadoop runs the job by dividing it into tasks
  - two types of tasks: map tasks and reduce tasks

Physical server

- Two types of nodes that control the job execution process: a jobtracker (or master) and a number of tasktrackers (or workers/slaves)

Multiple slaves architecture

1 physical server: job tracker (master)
Others: tasktrackerssalaves(slaves)

  - jobtracker: coordinates all jobs
  - tasktracker: runs tasks and sends reports to jobtracker

MapReduce Data Flow in Hadoop

- Hadoop divides the input to a MapReduce job into fixed-size pieces called splits
  - Hadoop creates one map task for each split
  - map function runs each record in the split
- Split size:
  - too large: lacks parallelization
  - Then split information
  - too small: increases management overhead
  - A good split size = HDFS block (default = 64MB)

Every split can lower the accuracy
But the lower of accuracy, compare with the amount of the data will be processed, is insignificant.

MapReduce Data Flow in Hadoop

Move computation to data has a higher priority to move data to computation

➢ Data locality optimization

- Hadoop tries to run the map task on a node where the input data resides in HDFS Read data locally
- Such as Ext4
- Map tasks write their output to the local disk (not in HDFS)

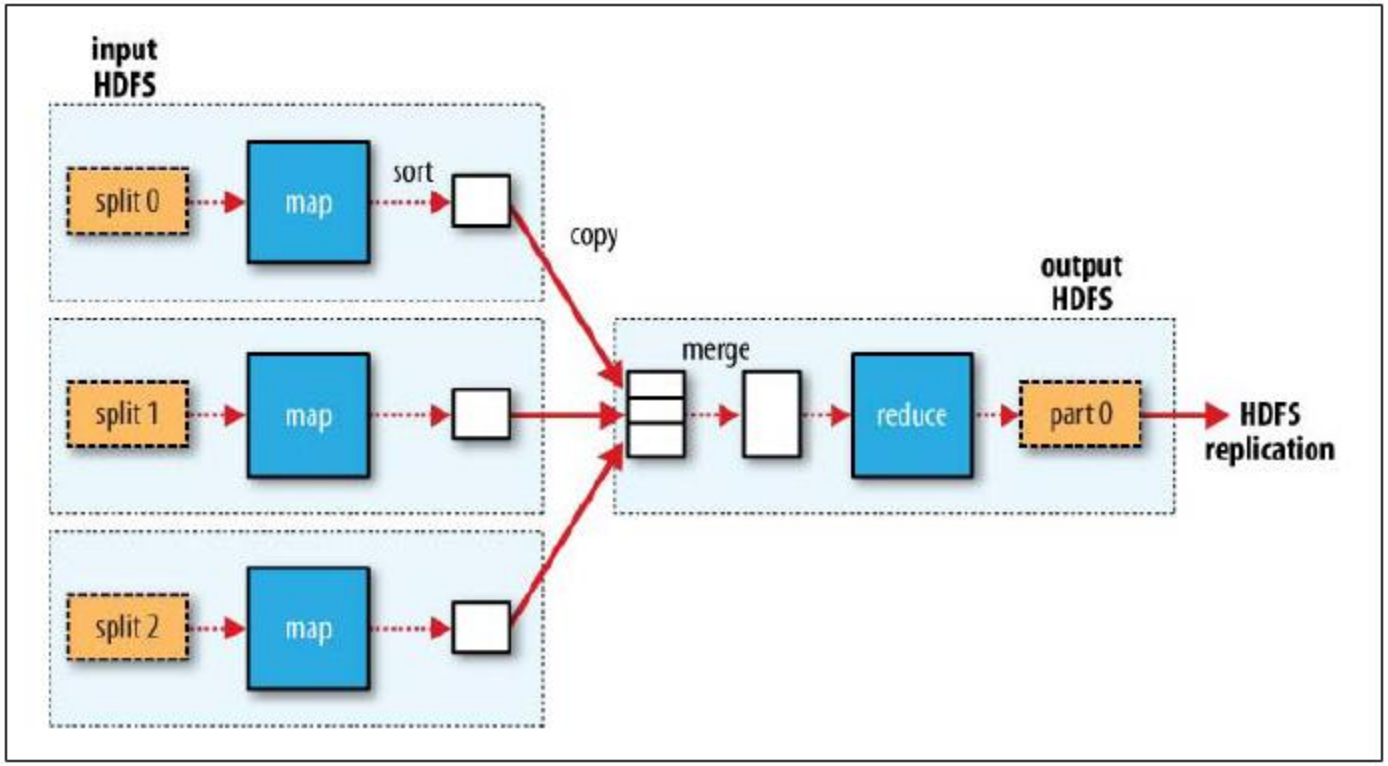
HDFS Will be slower because it is a layer on top of the local disk

  - interemediate results are later retrieved by reduce tasks
- Reduce tasks don't have the advantage of data locality -- the input to a single reduce task is normally the output from all mappers

MapReduce Data Flow in Hadoop

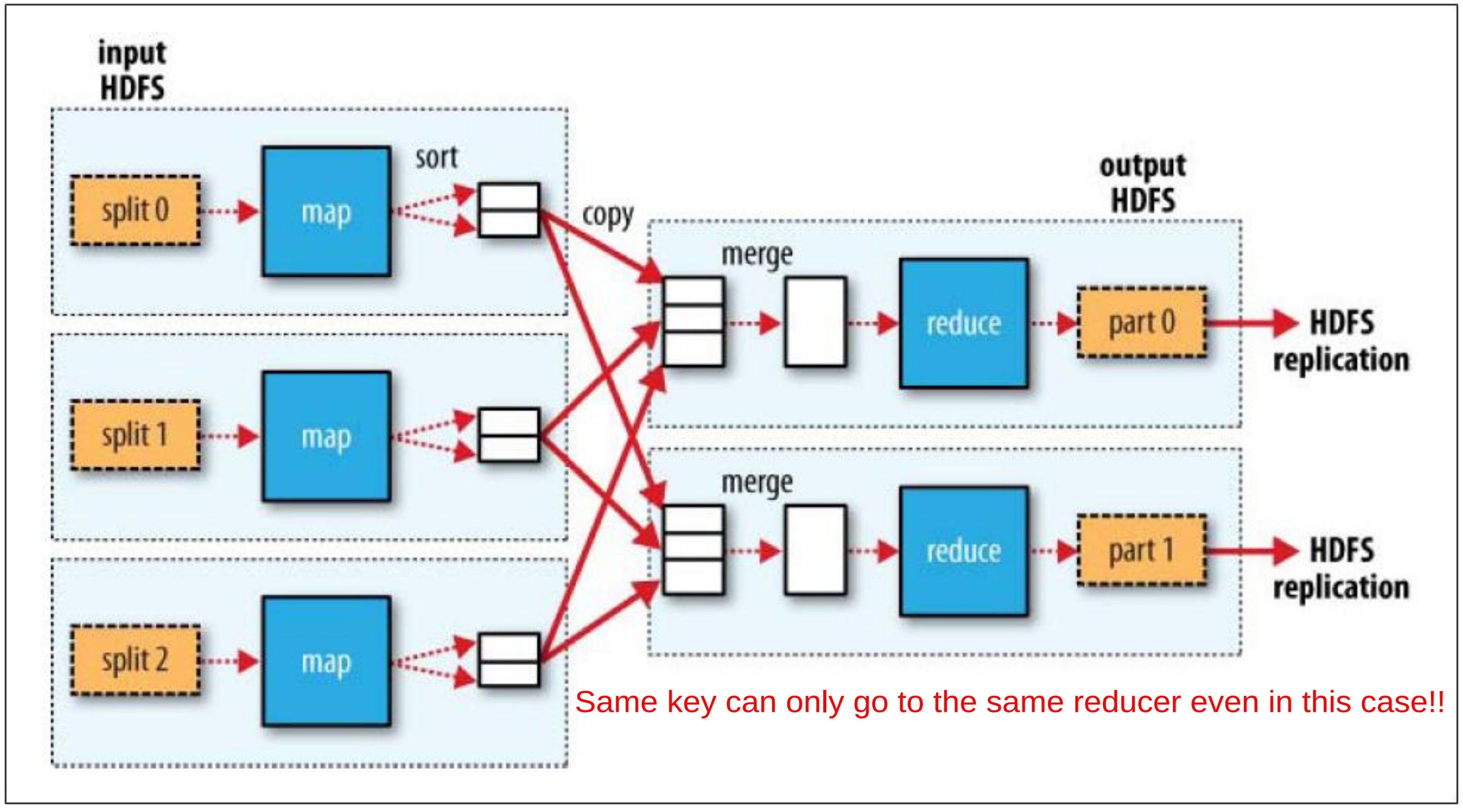
- Fault tolerance
  - If the node running the map task fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun the map task on another node to re-create the map output. Just quit the task, but not the whole program.
  - The output of reduce is normally stored in HDFS for reliability
    - for each HDFS block of the reduce output, the first replica is stored on the local node, with other replicas being stored on off-rack nodes.

Data Flow Illustration



MapReduce data flow with a single reduce task

Data Flow Illustration



Same key can only go to the same reducer even in this case!!

MapReduce data flow with multiple reduce tasks

Partition and Combiner

Function

- Two additional elements in MapReduce model
- **Partition**
  - Map outputs are partitioned to multiple reduce tasks
  - Often a simple hash key “H(k) mod R”
  - Can be user-defined
- **Combiner**
  - Map outputs are reduced (or combined) before being sent to reducers
  - Used as an optimization to reduce network traffic

Outline

- Hadoop
- MapReduce
- HDFS

Distributed Filesystem

- Motivation:
  - How to store data so that the MapReduce framework can process efficiently?
- Data is stored in multiple disks in a number of separate machines
- We need a distributed filesystem that manages the storage across a network of machines
- Need to deal with the complication of network programming, different from disk-based filesystem

Hadoop Distributed Filesystem (HDFS)

- HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware
  - Very large files: files are hundreds of megabytes, gigabytes, or even terabytes in size
  - WORM model
    Streaming data access: write-once, read-many-times pattern; designed for batch processing rather than interactive patterns
    Random access
  - Commodity hardware: low-cost hardware is used; hardware failure is a norm rather than exception

Hadoop Distributed Filesystem (HDFS)

- More design goals of HDFS:
  - Simple coherency model: data won't be changed; simplifies synchronization overhead
    Assume storage space is unlimited
  - Moving computation is cheaper than moving data: migrate the computation closer to where the data is located rather than moving the data to where the application is running
  - Portability across heterogeneous hardware and software platforms: facilitates widespread adoption of HDFS

Not a Good Fit for HDFS

- Some applications are not suitable for HDFS:
  - Requires low-latency data access Vs streaming data access
  - Lots of small files Vs large files
  - Multiple writers, arbitrary file modifications Vs WORM model

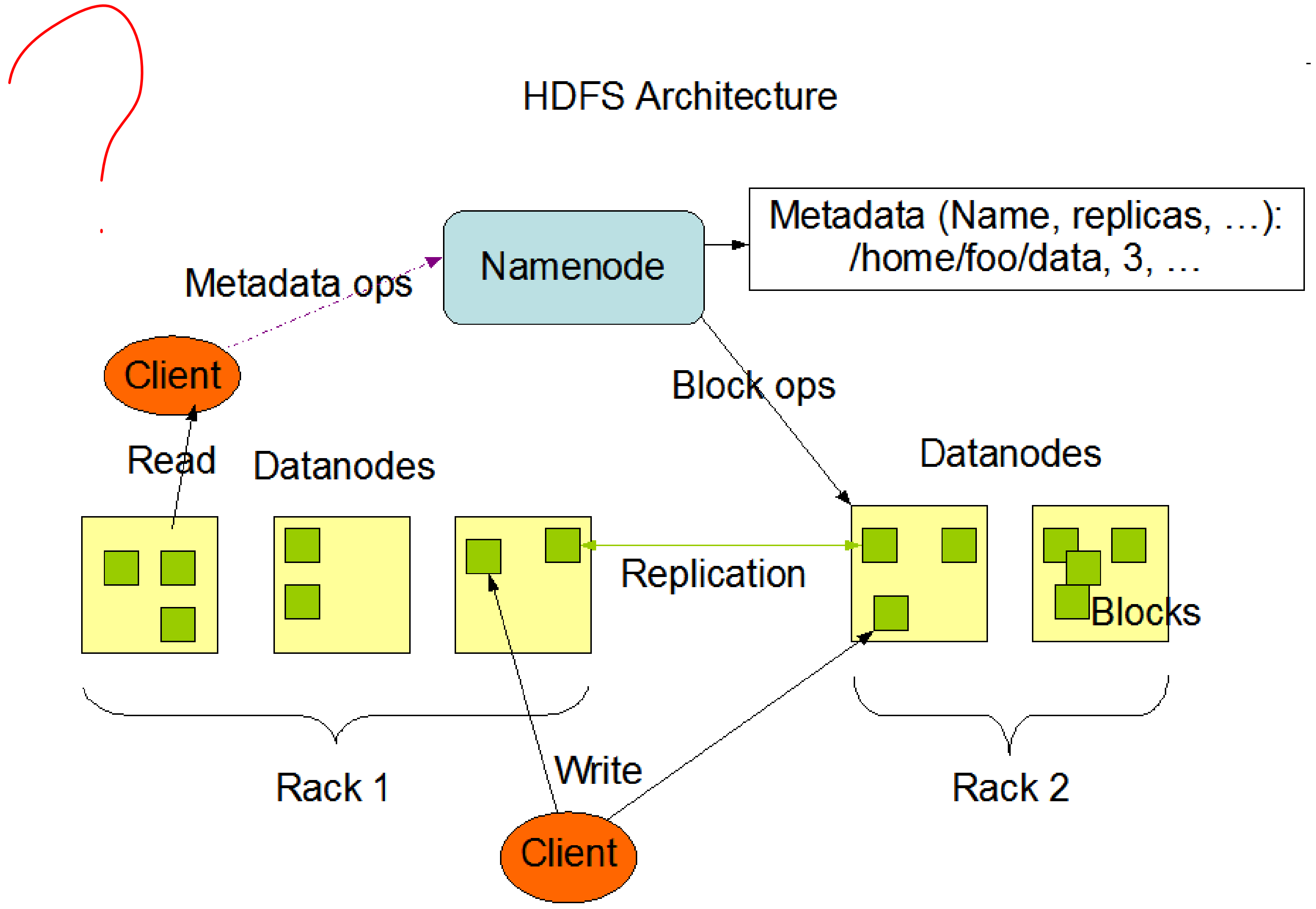
HDFS Architectural Overview

- HDFS has a master/slave architecture:

  Divided meta data and data, stored in two different machines

  - A single namenode, a master server that manages the file system namespace and regulates access to files by clients
  - Multiple datanodes, each manages storage attached to the nodes, and serves read/write data requests from clients
- Namenode and Datanode are pieces of software designed to run on commodity machines.

HDFS Architectural Overview



- *Note: Data doesn't go through the namenode*

Otherwise, the name node will then become the bottleneck

Data Blocks

Linux is 4 kilobytes

- HDFS arranges data in blocks (chunks) of very large size – 64MB by default

  Balance paallelization and management overhead

  - similar to disk-based file systems, but uses a large block size
  - designed for very large files
  - a file in HDFS that is smaller than a single block does not occupy a full block’s worth of underlying storage

  Meta data overhead

- Why large blocks?
  - to minimize the cost of disk seeks

HDFS Namespace

- HDFS supports a traditional hierarchical file organization (like many filesystems)
  - one can create and remove files, move a file from one directory to another, or rename a file
  - *does not support*: user quotas, access permissions, hard/soft links This will introduce additional overhead
- Filesystem namespace is maintained at the namenode

Data Replication

- ➢ Blocks of a file are replicated for fault tolerance
  - Replication means to create identical copies
- ➢ Block size and replication factor are configurable per file
  - Replication factor: number of copies of a file
- ➢ The Namenode makes all decisions regarding replication of blocks. It periodically receives a Heartbeat and a Blockreport from each of the Datanodes
  - Heartbeat: keep-alive message
  - Blockreport: a list of all blocks

Replica can safe the data. And, it boost the performance and fault tolerance.

Data Replication

- Default replication factor = 3
- Replication placement is crucial to HDFS reliability and performance
- Rack-aware placement – balance reliability and write performance
  - One replica in one node in the local rack
  - One replica in another node in the local rack
  - One replica in a different node in a different rack
- HDFS tries to satisfy a read request from a replica that is closest to the reader (decided by the namenode)
- The optimal placement policy remains an open issue
- RAID can be used: http://wiki.apache.org/hadoop/HDFS-RAID

Communication Protocols

- All HDFS communication protocols are layered on top of the TCP/IP protocol
- A Remote Procedure Call (RPC) abstraction wraps the communication protocols
- Namenode never initiates any RPCs. Instead, it only responds to RPC requests issued by Datanodes or clients

Robustness

- Data disk failures
  - namenode receives no heartbeats → datanode fail
  - re-replicate data to new datanodes
- Cluster rebalancing
  - HDFS can dynamically generate new replicas and rebalance other data in the cluster
    - Data nodes are of heterogeneous types; some get full sooner than others
- Data integrity
  - Checksum is attached to each block

Robustness

Problems in HDFS

- Namenode failure
  - Currently it’s a single point of failure in HDFS
  - Needs manual intervention
- Snapshots
  - Currently not supported

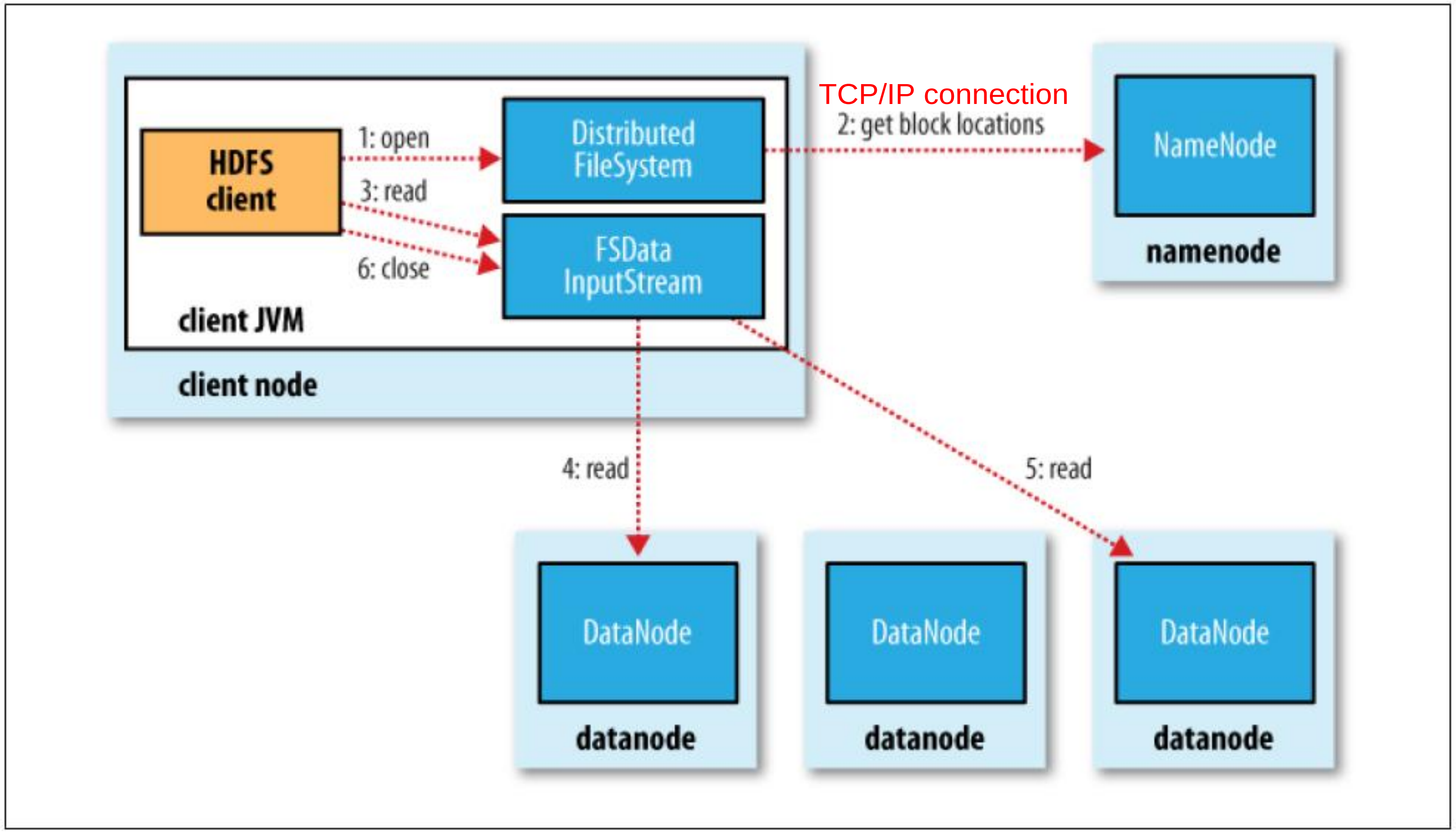
Recap: Putting It All Together

- Design features of HDFS:
  - One namenode, multiple datanodes
  - Hierarchical filesystem namespace
  - Large data blocks
  - Data replication
  - Communication over TCP/IP
  - Robustness issues addressed
- Next question: how is data actually read/written from/to HDFS?

HDFS Read

- A client reading data from HDFS

Round trip



HDFS Read

- Step 1: HDFS client opens the file through the file system object.
- Step 2: The file system object calls the namenode, using RPC, to determine the locations of the blocks for the first few blocks in the file. The file system object returns an input stream.
- Step 3: The client then call the read function to retrieve data blocks through the input stream

HDFS Read

- Step 4: Data is streamed from the datanode back to the client, which calls the read function repeatedly on the input stream
- Step 5: When the end of the block is reached, the input stream will close the connection to the datanode, then find the best datanode for the next block
- Step 6: When the client has finished reading, it closes the input stream

Notes on HDFS Read

- If the input stream encounters an error while communicating with a datanode, then it will try the next “closest” one for that block.
- Note that the client contacts datanodes directly to retrieve data and is guided by the namenode to the best datanode for each block. Namenode doesn’t serve data
  - This allows scalability to a large number of clients

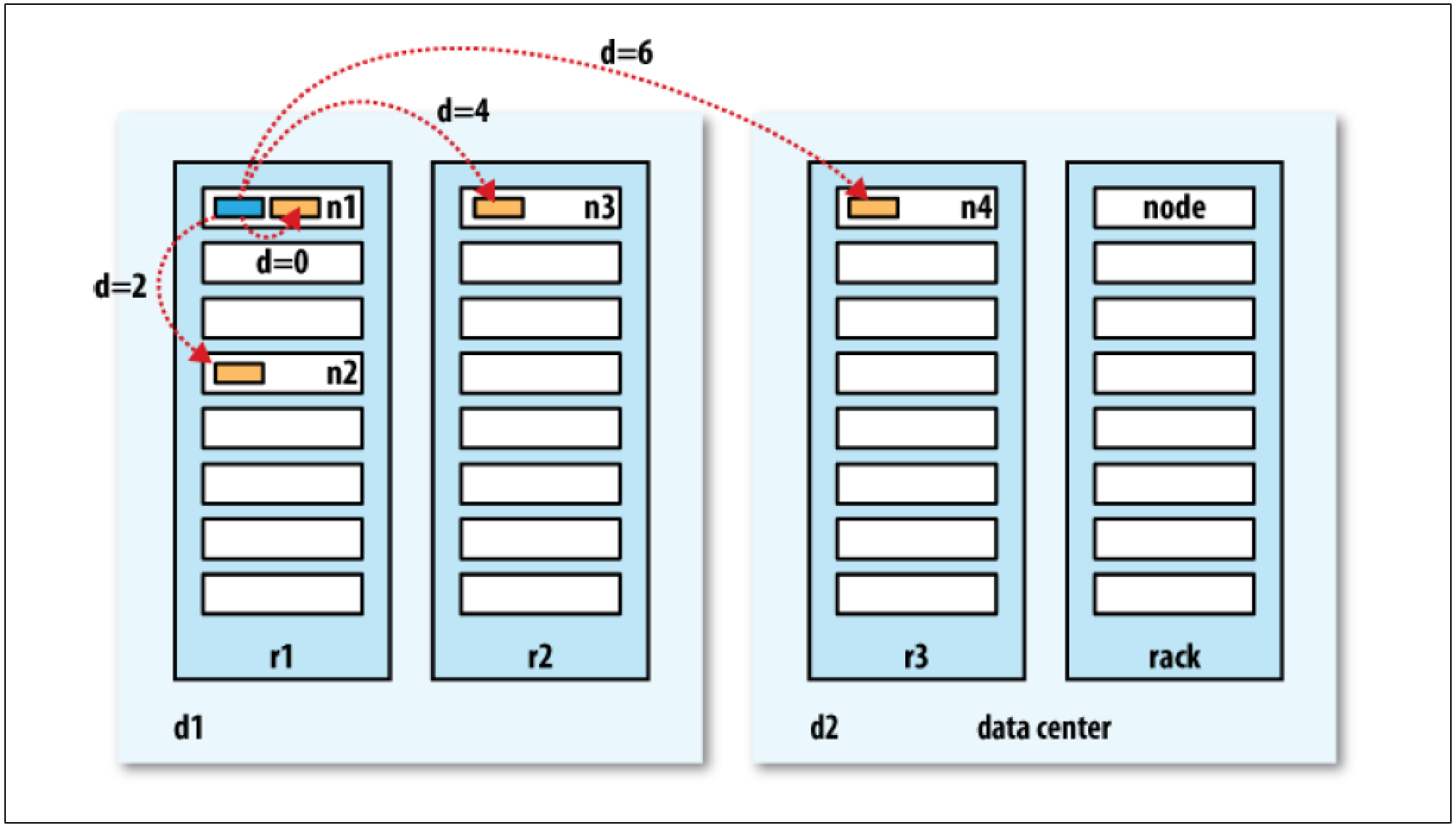
Notes on HDFS Read

- What do we mean by "closest"? Such as Dijkstra algo, but maybe so complicated
- HDFS defines distance by using hop counts, considering the network as a tree
- Levels in the tree correspond to the data center, the rack, and the node
- Examples: Assumption only
  - Distance = 0 (processes on the same node)
  - Distance = 2 (different nodes on the same rack)
  - Distance = 4 (nodes on different racks in the same data center
  - Distance = 6 (nodes in different data centers)

Notes on HDFS Read

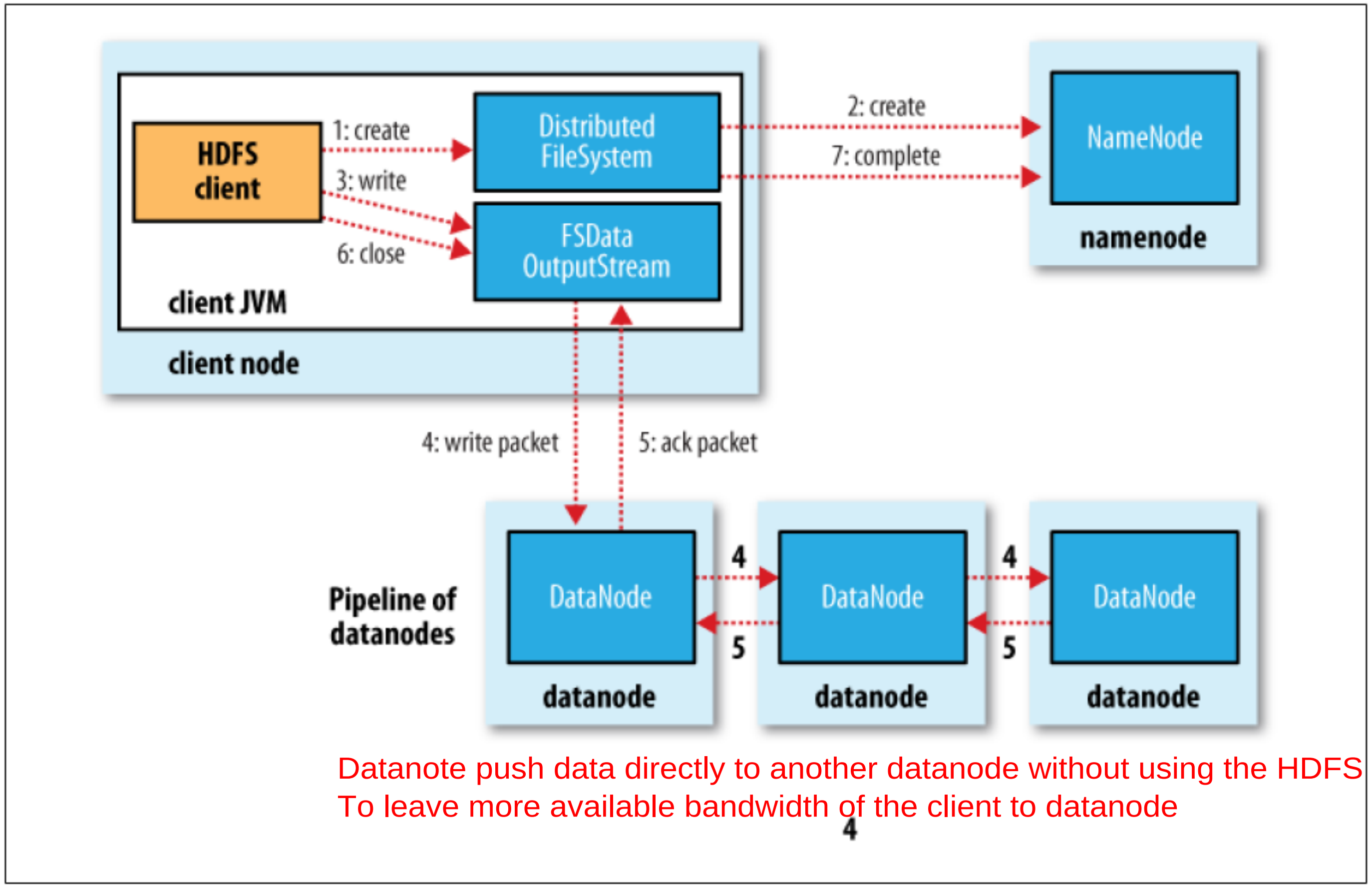
Switch Router difference

- Network distance on Hadoop



HDFS Write

- A client writing data to HDFS



Datanote push data directly to another datanode without using the HDFS client
To leave more available bandwidth of the client to datanode

HDFS Write

- Step 1: HDFS client creates the file on the file system object
- Step 2: The file system object makes an RPC call to the namenode to create a new file in the filesystem’s namespace. An output stream is created.
- Step 3: As the client writes data, the output stream splits it into packets, which it writes to an internal queue called the data queue

HDFS Write

- Step 4: The output stream streams the packets to the first datanode in the pipeline, which stores the packet and forwards it to the second datanode in the pipeline, followed by the third (and last) datanode in the pipeline
- Step 5: The output stream maintains an internal queue of packets that are waiting to be acknowledged by datanodes called the ack queue. A packet is removed from the ack queue only when it has been acknowledged by all the datanodes in the pipeline

HDFS Write

- Step 6: When the client has finished writing data, it closes the output stream
- Step 7: The client contacts the namenode to signal that the file is complete

Notes on HDFS Write

- Operations on coherency model
  - flush(): when it's called, data is flushed to operating systems (but may not be on disk yet)
  - sync(): when it's called, all datanodes are synchronized to store the same data on disk
- Without calling sync(), you may lose data during system failures
- You may call sync() regularly, but frequent sync()'s will hurt application's performance

Beyond HDFS/GFS

- ➢ HDFS is derived from Google File System (GFS) → Both designed for batch operations
- ➢ **Google's Colossus**: next-generation GFS
  - Designed for real-time search
  - Build on the structured storage system BigTable
  - Details are still unclear to public

- ➢ http://highscalability.com/blog/2010/9/11/googles-colossus-makes-search-real-time-by-dumping-mapreduce.html