Distributed Computing

A-14. Hadoop Design

Apache Hadoop

- If you don't work at Google, Hadoop is the software suite you're likely to use if you have a large dataset
- Free-Open Source, Apache License
- Handled by the Apache Foundation
- Based on Java
- A large ecosystem
- We'll see the parts that deal with MapReduce

Credits

- Again, thanks to Pietro Michiardi of EURECOM. Many diagrams thanks to him.

HDFS:
the Hadoop Distributed FileSystem

Move Computation to the Data

- We have seen that MapReduce is based on
  - Performing a **map** phase wherever data is read
  - A **shuffle** phase to move around processed data
  - A **reduce** phase to aggregate it
- HDFS is designed to **enable** this kind of computation
  - For nodes that do **both** storage and computation
  - Inspired by GFS, the Google correspondent
  - See the paper for more information

HDFS Principles

- Large datasets, that can’t be stored on a single machine
  - Each “file” is partitioned in several machines
  - Network-based, with all the complications
  - Failure-tolerant
- One distributed filesystem design, tailored to
  - Read-intensive workloads (many reads for a write)
  - Throughput, not latency (sequential reads)
  - Commodity hardware

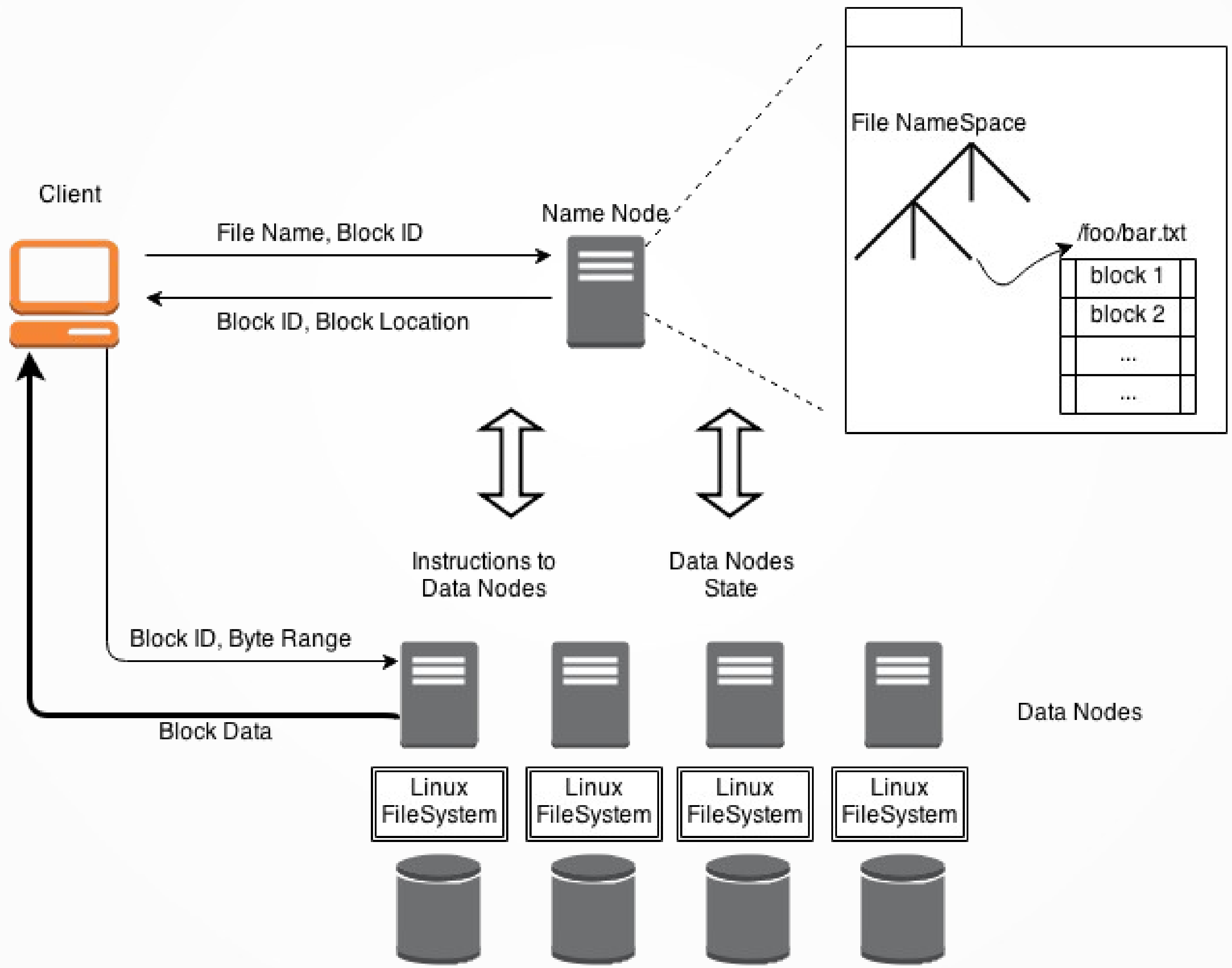
HDFS Blocks

- Big files are broken in chunks
  - Big chunks! Default is 128 MB
  - Unrelated to space used on disks (a 1MB file doesn’t use 128 MB): blocks are stored as files on the native filesystem
- Blocks are **replicated** in different machines
  - Q: Why not using **erasure coding**?
  - A: Because you can run processing right away (map!)
- Q: Why are blocks so large?
  - A1: To make seek times small compared to read. Consider 10ms seek and 100MB/s read, for a 100MB block seek time is 1%
  - A2: To ease handling metadata. We’ll see right away.

HDFS Nodes

- **NameNode**: keeps metadata in RAM
  - Directory tree, and index of blocks per file (around 150B/block)
  - Metadata is around **1M** times smaller than the dataset–1GB of RAM can index 1PB of data
  - The load of NameNode is kept manageable exactly because there aren't that many blocks
  - Writes are written in an atomic and synchronous way on a **journal**
  - It's a good idea to put this journal somewhere on the net
- **Secondary NameNode**
  - Receives copies of the edit log from the NameNode
  - When the primary is down, the system uses it and stays read-only
  - If the journal is on the network, we can switch the secondary to primary
- **DataNode:** store data, heartbeat to the NameNode with the list of their blocks

HDFS Architecture

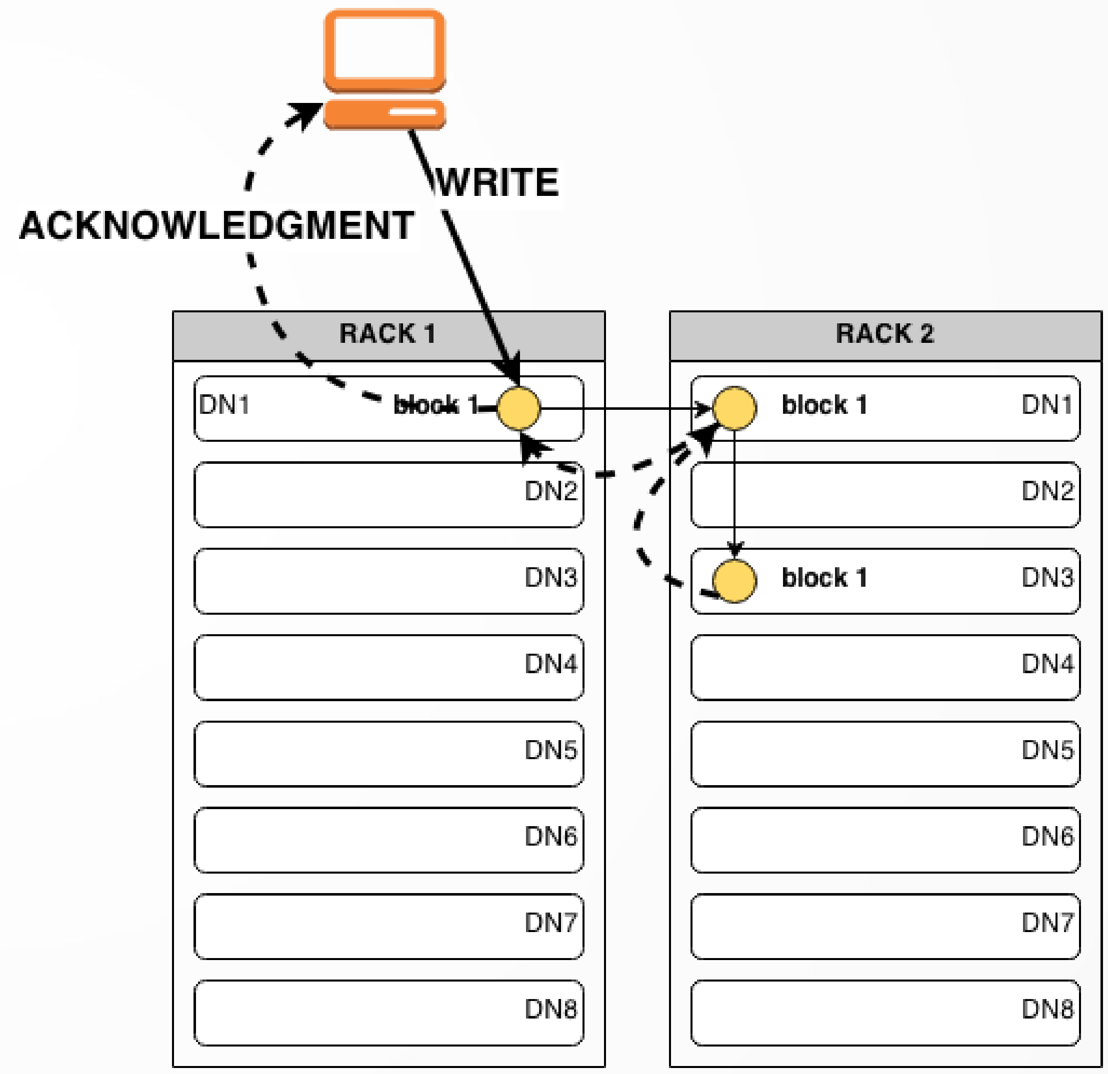


History of a File Read

- Note: the client is often a machine **in the same cluster**
- Get the block locations from the NameNode
- Obtain a set of DataNodes, sorted by proximity to the client
  - i.e.: first the same node, then the same rack
- If MapReduce is reading, the data will be in the very same machine
  - Data is read in Map tasks
  - There are corner-case exceptions

History of a File Write

- Client asks the NameNode for *k* Datanodes (default *k=3*)
- **Pipeline** replication: the first datanode will make a copy to the second, and that one to the third
- Default: first replica off-rack, second replica in the same rack of the first
  - Tradeoff between reliability and cluster bandwidth



Scheduling

A Job Is Made of Tasks

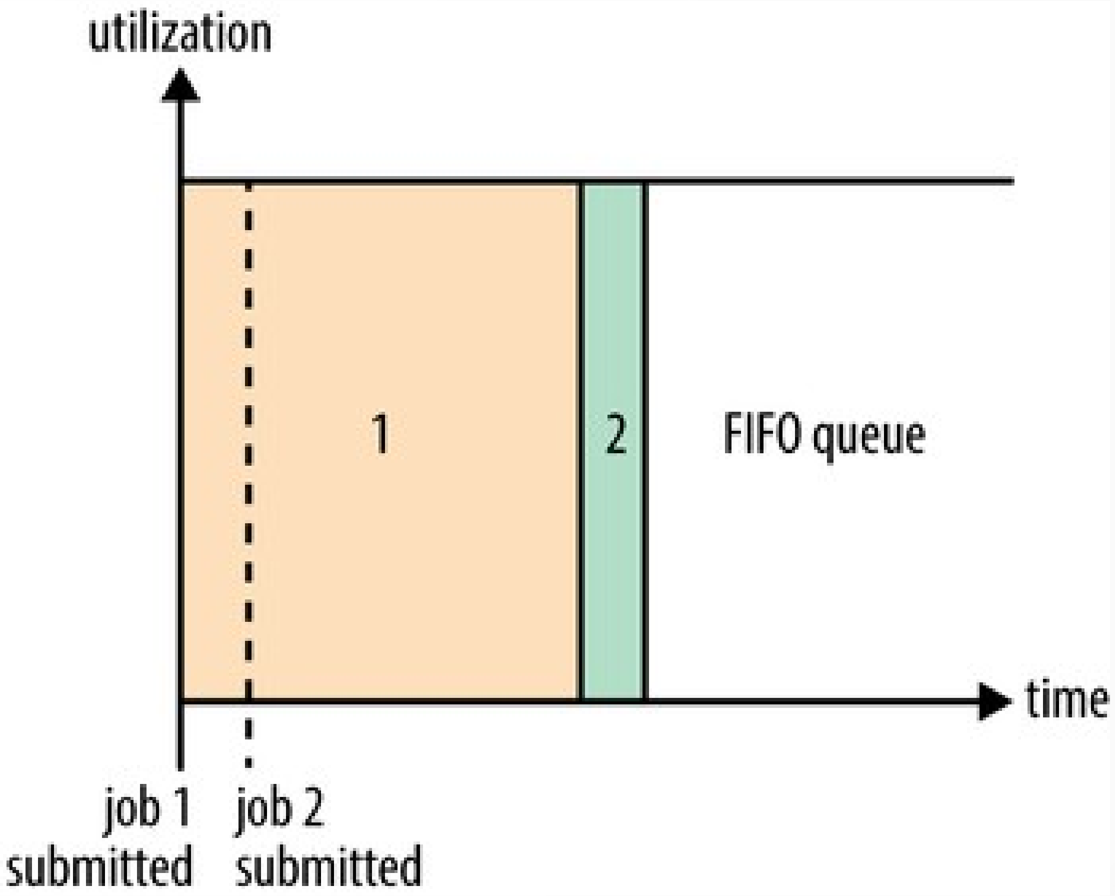
- A **MapReduce job**
  - Runs on a set of blocks specified by the programmer
  - Has one phase each of map, shuffle & reduce
- Map and reduce phases are divided in **tasks**
  - Tasks are single-machine, independent job
  - The only “holistic” part is in the shuffle phase
- Each machine runs a configurable number of tasks
  - Often: one task per CPU, so they don’t slow each other down

Map and Reduce Tasks

- **Map**: by default, **1 HDFS block → 1 *input split* → 1 task**
  - The scheduler will do whatever is possible to run the tasks on a machine having that block
  - Map tasks are usually quick (a few seconds), unless they perform unusually large computation
- **Reduce**: Number of reduce tasks is user-specified
  - Keys are partitioned **randomly** based on hash values: one task will handle several keys
  - Users can override this and write a **custom partitioner** (useful to handle **skewed** data)
  - Reduce tasks have very variable runtime, depending on what they do

Schedulers: FIFO

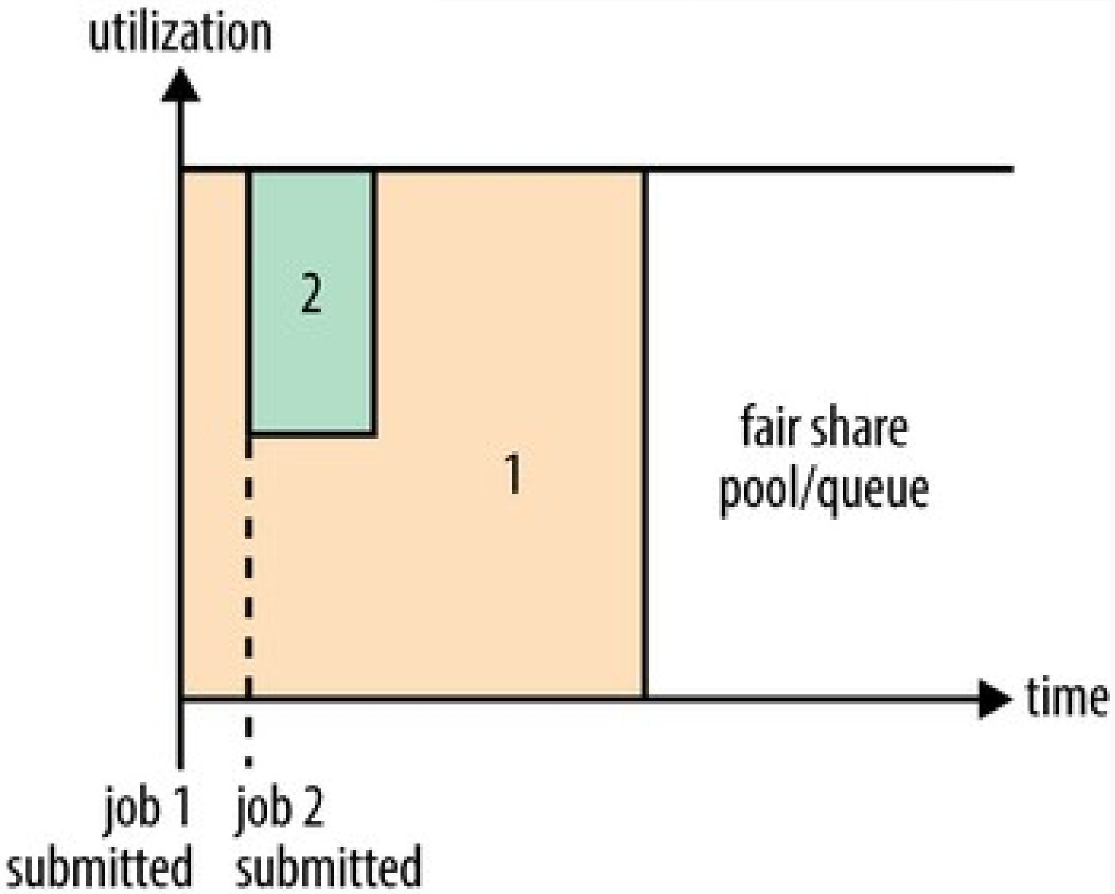
- Priority of jobs is their arrival time
- As soon as a machine is free, it's given the first pending task by the first job
- Penalizes small jobs which can **wait forever** when very large jobs are there



(from Hadoop: the Definitive Guide)

Schedulers: Fair

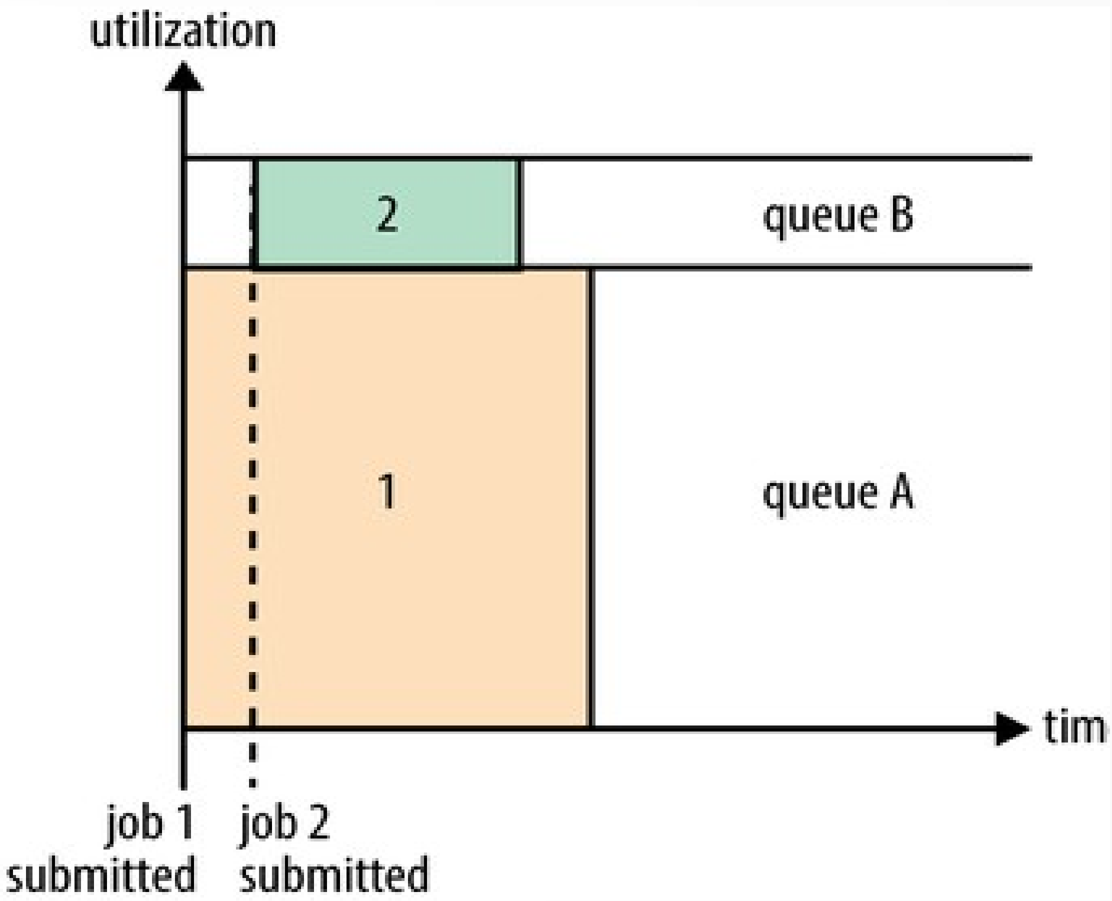
- Priority: give precedence to active jobs with **least running tasks**
- Results in each job having roughly the same amount of work done in a given moment
- Conceptually very similar to processor-sharing and/or round-robin schedulers
- Can be configured to **kill running tasks** to free up space for jobs
- You can't prioritize jobs in a queue



(from *Hadoop: the Definitive Guide*)

Schedulers: Capacity

- Creates "virtual clusters" with a queue each and a dedicated amount of resources
- Can be used to make sure that organizations/application have access to a reasonable amount of computing power
- There is elasticity possible: if a queue leaves unused resources, they can be used by another queue



(from *Hadoop: the Definitive Guide*)

What About Shortest Job First?

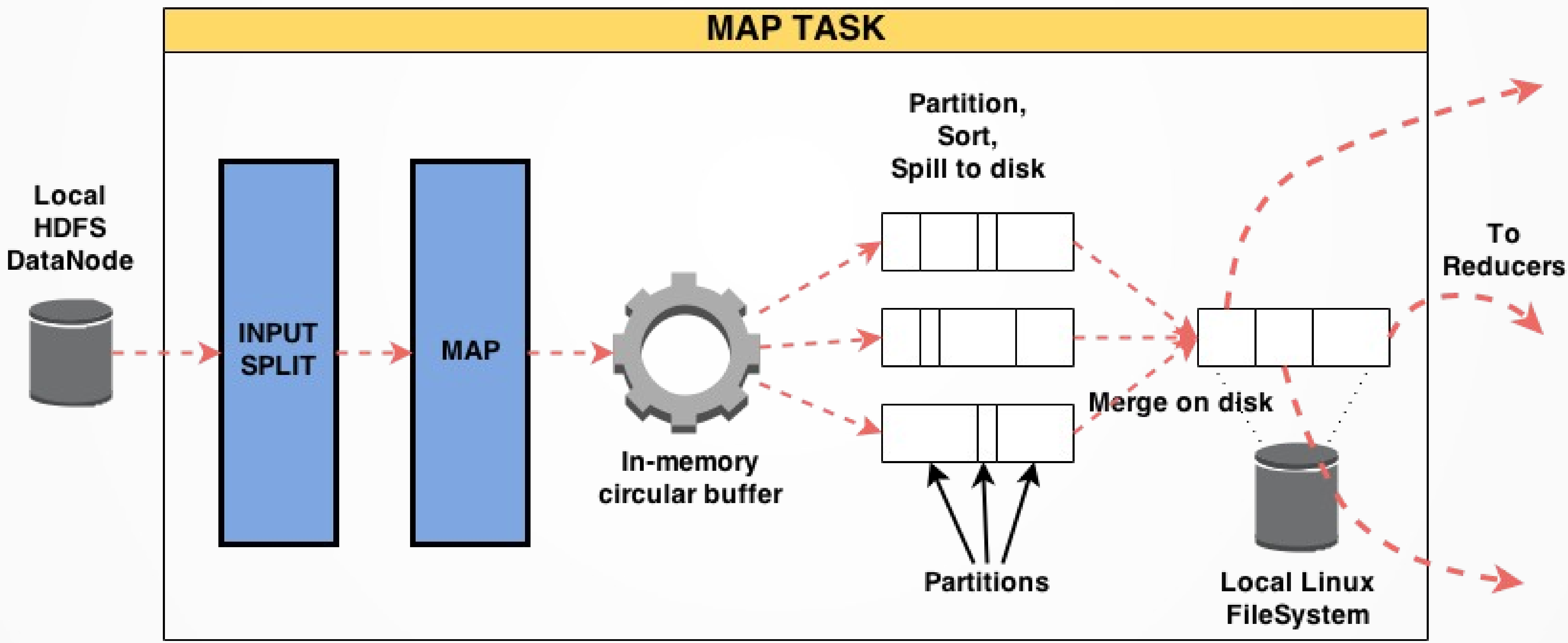
- We have said that letting shortest jobs first ahead can be great for a loaded system
  - In particular for real-world jobs
- The problem is that you generally **don’t know** how long a job will need to run
- There’s a big potential here to improve performance, but system designers are conservative people & you don’t know when the system will be problematic

Handling Failures

- Failures are **common**: software & hardware problems
- If a **task fails**, it's retried a few times (e.g., 4)
  - After that, by default the job is marked as failed
- If a **task hangs** (no progress), it's killed and retried
- If a **worker machine** fails, the scheduler notices the lack of heartbeats and removes it from the worker pool
- If the **scheduler** fails—Zookeeper can be used to set up a backup and keep it updated

Shuffle (& Sort) Phase

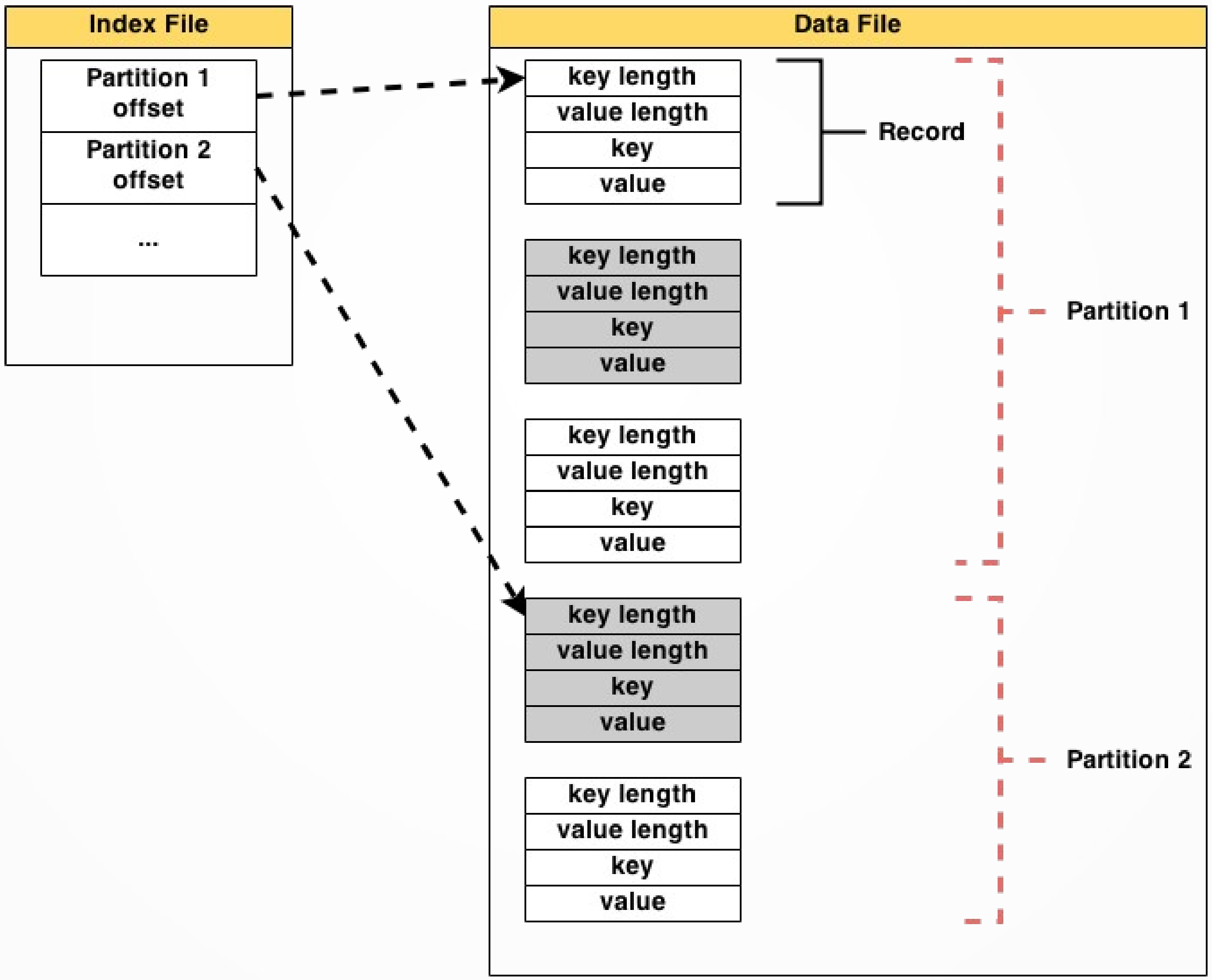
Shuffle: Map Side



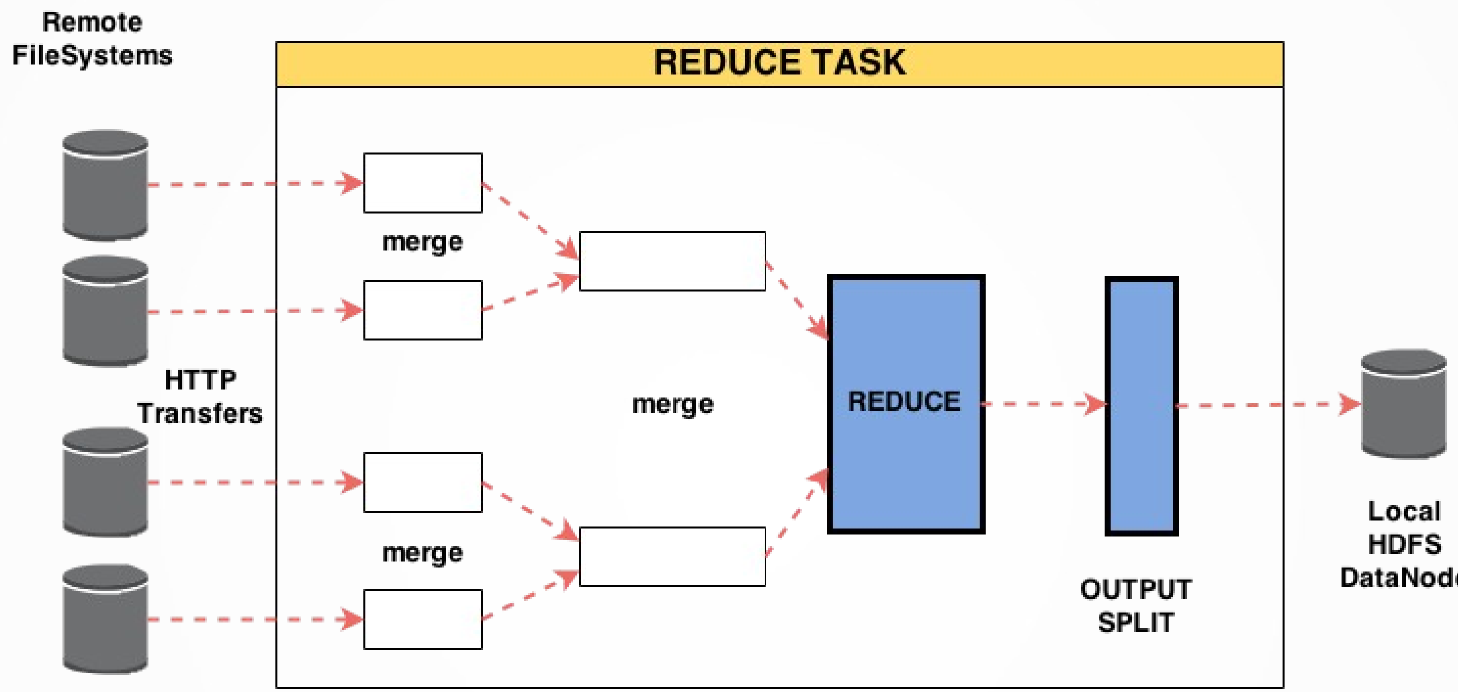
Map Side: Description

- The output of map stays in a buffer in memory
  - Default buffer size: 100 MB
- When the buffer is filled, it’s partitioned (by destination reducer), sorted and saved to disk
  - Additional guarantee in Hadoop: reduce keys are always sorted
- At the end of a map phase, spills are merged and sent to reducers
- Combiners are run right before spilling to disk. **Why?**

A Look at A Spill File



Shuffle: Reduce Side



- Reducers **fetch** data from mappers and run a merge
  - Mappers don't delete data right after it's sent to reducers. **Why?**
- We're essentially running a distributed mergesort
- Output is saved (& replicated) on HDFS