# Data-intensive computing systems

## Hadoop

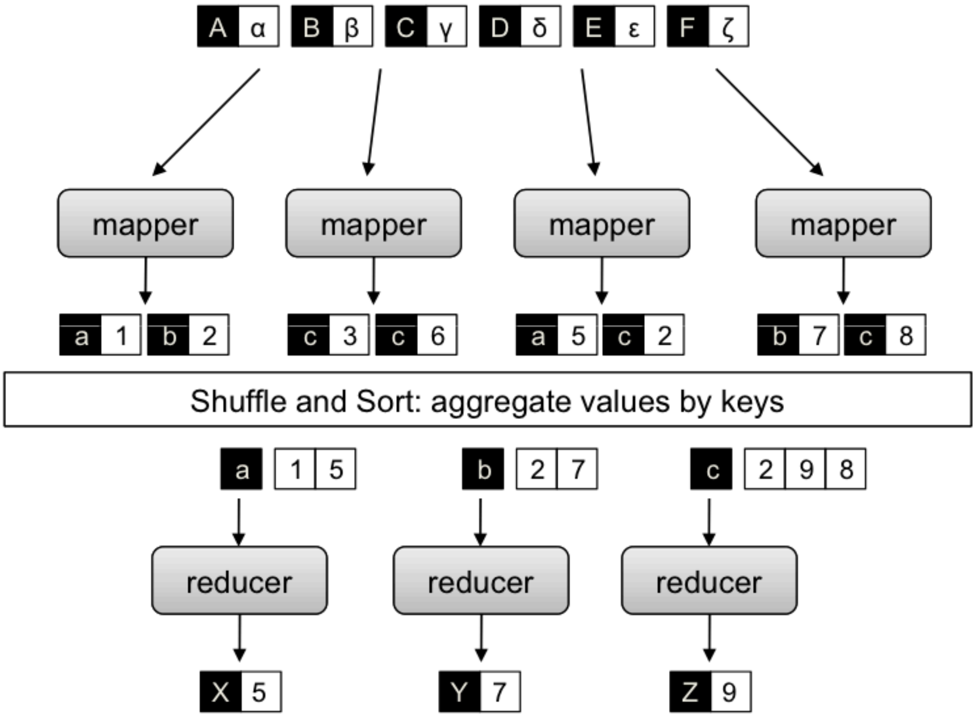
Universtity of Verona
Computer Science Department

Damiano Carra

## Acknowledgements

- *Credits*
  - *Part of the course material is based on slides provided by the following authors*
    - *Pietro Michiardi, Jimmy Lin*

# A Simplified view of MapReduce






# From theory to practice

- The story so far
  - MapReduce programming model
  - High level view of the execution framework
- Next, we'll see
  - Implementation of MapReduce: Hadoop
    - Implementation details
    - Types and Formats
- Before this, we present the special file-system used in Hadoop

# Hadoop Distributed File-System (HDFS)



## Data and computation colocation

- As dataset sizes increase, more computing capacity is required for processing
- As compute capacity grows, the link between the compute nodes and the storage nodes becomes a bottleneck
  - One could think of special-purpose interconnects for high-performance networking
  - This is often a costly solution as cost does not increase linearly with performance
- Key idea: abandon the separation between compute and storage nodes
  - This is exactly what happens in current implementations of the MapReduce framework
  - A distributed filesystem is not mandatory, but highly desirable

# The Hadoop Distributed Filesystem

- Large dataset(s) outgrowing the storage capacity of a single physical machine
  - Need to partition it across a number of separate machines
  - Network-based system, with all its complications
  - Tolerate failures of machines
- Distributed filesystems are not new!
  - HDFS builds upon previous results, tailored to the specific requirements of MapReduce
  - Write once, read many workloads
  - Does not handle concurrency, but allow replication
  - Optimized for throughput, not latency
- Hadoop Distributed Filesystem
  - Very large files
  - Streaming data access
  - Commodity hardware



# HDFS Blocks

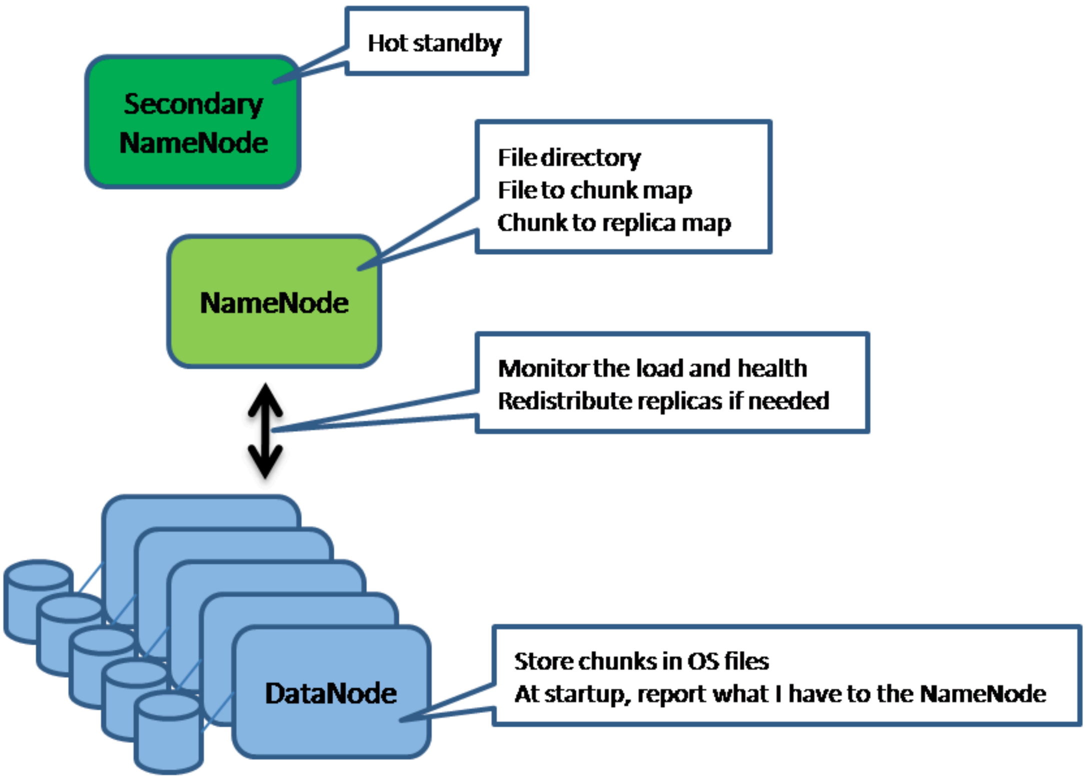
- (Big) files are broken into block-sized chunks
  - E.g, 64 MB or 128 MB
  - NOTE: A file that is smaller than a single block does not occupy a full block's worth of underlying storage
- Blocks are stored on independent machines
  - Replicate across the local disks of nodes in the cluster
  - Reliability and parallel access
  - Replication is handled by storage nodes themselves (similar to chain replication)
- Why is a block so large?
  - Make transfer times larger than seek latency
  - E.g.: Assume seek time is 10ms and the transfer rate is 100 MB/s, if you want seek time to be 1% of transfer time, then the block size should be 100MB

## NameNodes and DataNodes

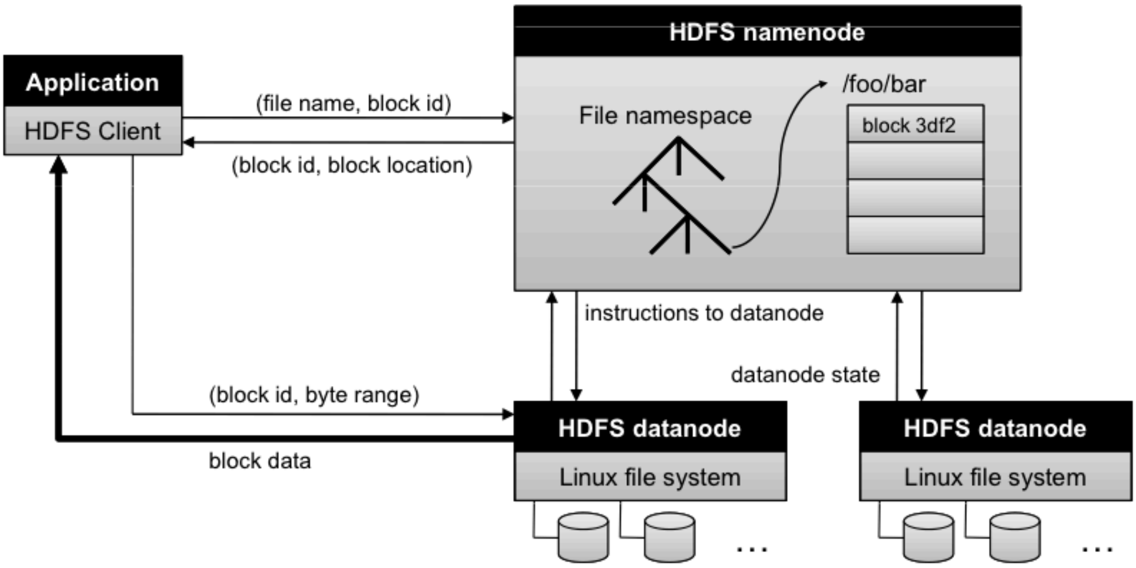
- NameNode
  - Keeps metadata in RAM
  - Each block information occupies roughly 150 bytes of memory
  - Without NameNode, the filesystem cannot be used
    - Persistence of metadata: synchronous and atomic writes to NFS
- Secondary NameNode
  - Merges the namespace with the edit log
  - A useful trick to recover from a failure of the NameNode is to use the NFS copy of metadata and switch the secondary to primary
- DataNode
  - They store data and talk to clients
  - They report periodically to the NameNode the list of blocks they hold



## Architecture



# Architecture






# Anatomy of a File Read

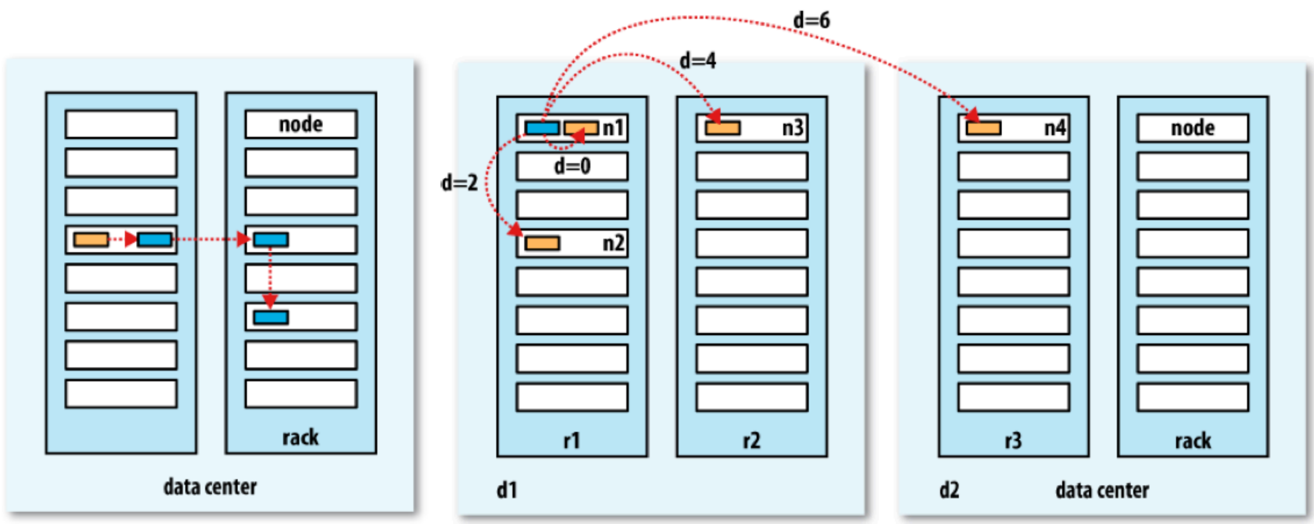
- `NameNode` is only used to get block location
  - Unresponsive `DataNode` are discarded by clients
  - Batch reading of blocks is allowed
- "External" clients
  - For each block, the `NameNode` returns a set of `DataNodes` holding a copy thereof
  - `DataNodes` are sorted according to their proximity to the client
- "MapReduce" clients
  - `TaskTracker` and `DataNodes` are colocated
  - For each block, the `NameNode` usually returns the local `DataNode`

# Anatomy of a File Write

- Details on replication
  - Clients ask `NameNode` for a list of suitable `DataNodes`
  - This list forms a *pipeline*: first `DataNode` stores a copy of a block, then forwards it to the second, and so on
- Replica Placement
  - Tradeoff between reliability and bandwidth
  - Default placement:
    - First copy on the "same" node of the client, second replica is off-rack, third replica is on the same rack as the second but on a different node
    - Since Hadoop 0.21, replica placement can be customized



# Network Topology and HDFS



## HDFS Coherency Model

- Read your writes is not guaranteed
  - The namespace is updated
  - Block contents may not be visible after a write is finished
  - Application design (other than MapReduce) should use `sync()` to force synchronization
  - `sync()` involves some overhead: tradeoff between robustness/consistency and throughput
- Multiple writers (for the same block) are not supported
  - Instead, different blocks can be written in parallel (using MapReduce)



## Hadoop MapReduce: the Execution Framework

# Disclaimer

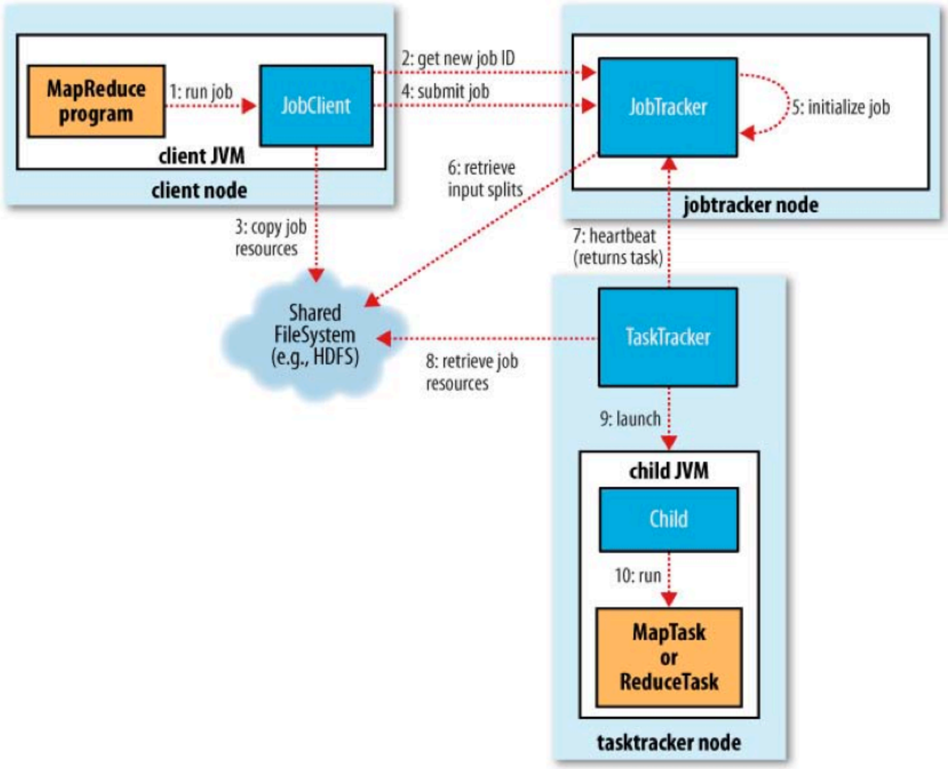
- ❑ MapReduce APIs
  - Fast evolving
  - Sometimes confusing
- ❑ Do NOT reply on these slides as a reference
  - Use appropriate API docs



# Terminology

- ❑ MapReduce:
  - `Job`: an execution of a Mapper and Reducer across a data set
  - `Task`: an execution of a Mapper or a Reducer on a slice of data
  - `Task Attempt`: instance of an attempt to execute a task
  - Example:
    - Running “Word Count” across 20 files is one job
    - 20 files to be mapped = 20 map tasks + some number of reduce tasks
    - At least 20 attempts will be performed... more if a machine crashes
- ❑ Task Attempts
  - Task attempted at least once, possibly more
  - Multiple crashes on input imply discarding it
  - Multiple attempts may occur in parallel (speculative execution)

# Anatomy of a MapReduce Job Run






# Job Submission

- ❑ `JobClient` class
  - The `runJob()` method creates a new instance of a **JobClient**
  - Then it calls the `submitJob()` on this class
- ❑ Simple verifications on the Job
  - Is there an output directory?
  - Are there any input splits?
  - Can I copy the JAR of the job to HDFS?
- ❑ Note: the JAR of the job is replicated 10 times

## Job Initialization

- The JobTracker is responsible for:
  - Create an object for the job
  - Encapsulate its tasks
  - Bookkeeping with the tasks' status and progress
- This is where the scheduling happens
  - JobTracker performs scheduling by maintaining a queue
  - Queueing disciplines are pluggable
- Compute mappers and reducers
  - JobTracker retrieves input splits (computed by JobClient)
  - Determines the number of Mappers based on the number of input splits
  - Reads the configuration file to set the number of Reducers



## Task Assignment

- Hearbeat-based mechanism
  - TaskTrackers periodically send heartbeats to the JobTracker
    - TaskTrackers is alive
    - Heartbeat contains information on availability of the TaskTrackers to execute a task
  - JobTracker piggybacks a task if TaskTracker is available
- Selecting a task
  - JobTracker first needs to select a job (i.e. job scheduling)
  - TaskTrackers have a fixed number of slots for map and reduce tasks
  - JobTracker gives priority to map tasks (WHY?)
- Data locality
  - JobTracker is topology aware
    - Useful for map tasks, unused for reduce tasks (WHY?)

# Task Execution

- ❑ Task Assignment is done, now `TaskTrackers` can execute
  - Copy the JAR from the HDFS
  - Create a local working directory
  - Create an instance of `TaskRunner`
- ❑ TaskRunner launches a child JVM
  - This prevents bugs from stalling the `TaskTracker`
  - A new child JVM is created per `InputSplit`
    - Can be overridden by specifying JVM Reuse option, which is very useful for custom, in-memory, combiners
- ❑ Streaming and Pipes
  - User-defined map and reduce methods need not to be in Java
  - Streaming and Pipes allow C++ or python mappers and reducers



# Scheduling

- ❑ FIFO Scheduler (default behavior)
  - Each job uses the whole cluster
  - Not suitable for shared production-level cluster
    - Long jobs monopolize the cluster
    - Short jobs can hold back and have no guarantees on execution time
- ❑ Fair Scheduler
  - Every user gets a fair share of the cluster capacity over time
  - Jobs are placed in to pools, one for each user
    - Users that submit more jobs have no more resources than others
    - Can guarantee minimum capacity per pool
  - Supports preemption
- ❑ Capacity Scheduler
  - Hierarchical queues (mimic an organization)
  - FIFO scheduling in each queue
  - Supports priority

# Handling Failures

***In the real world, code is buggy, processes crash and machines fail***

- Task Failure
  - Case 1: map or reduce task throws a runtime exception
    - The child JVM reports back to the parent `TaskTracker`
    - `TaskTracker` logs the error and marks the TaskAttempt as failed
    - `TaskTracker` frees up a slot to run another task
  - Case 2: Hanging tasks
    - `TaskTracker` notices no progress updates (timeout = 10 minutes)
    - `TaskTracker` kills the child JVM
  - `JobTracker` is notified of a failed task
    - Avoids rescheduling the task on the same `TaskTracker`
    - If a task fails 4 times, it is not re-scheduled
    - Default behavior: if any task fails 4 times, the job fails



# Handling Failures (cont'd)

- `TaskTracker` Failure
  - Types: crash, running very slowly
  - Heartbeats will not be sent to `JobTracker`
  - `JobTracker` waits for a timeout (10 minutes), then it removes the `TaskTracker` from its scheduling pool
  - `JobTracker` needs to reschedule even completed tasks (WHY?)
  - `JobTracker` needs to reschedule tasks in progress
  - `JobTracker` may even blacklist a `TaskTracker` if too many tasks failed

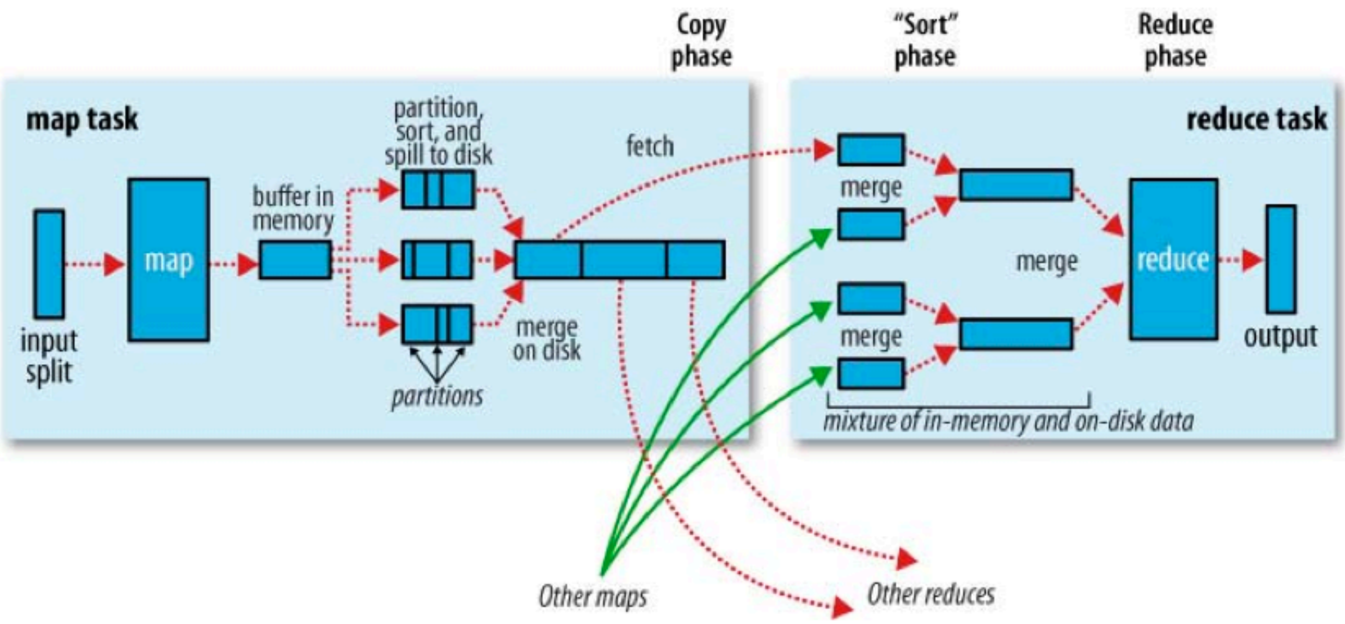
- `JobTracker` Failure
  - Currently, Hadoop has no mechanism for this kind of failure
  - In future (and commercial) releases:
    - Multiple `JobTrackers`

# Shuffle and Sort

- The MapReduce framework guarantees the input to every reducer to be sorted by key
  - The process by which the system sorts and transfers map outputs to reducers is known as shuffle
- Shuffle is the most important part of the framework, where the "magic" happens
  - Good understanding allows optimizing both the framework and the execution time of MapReduce jobs
- Subject to continuous refinements



# Shuffle and Sort: the Map Side



## Shuffle and Sort: the Map Side

- The output of a map task is not simply written to disk
  - In memory buffering
  - Pre-sorting
- Circular memory buffer
  - 100 MB by default
  - Threshold based mechanism to spill buffer content to disk
  - Map output written to the buffer while spilling to disk
  - If buffer fills up while spilling, the map task is blocked
- Disk spills
  - Written in round-robin to a local dir
  - Output data is partitioned corresponding to the reducers they will be sent to
  - Within each partition, data is sorted (in-memory)
  - Optionally, if there is a combiner, it is executed just after the sort phase



## Shuffle and Sort: the Map Side

- More on spills and memory buffer
  - Each time the buffer is full, a new spill is created
  - Once the map task finishes, there are many spills
  - Such spills are merged into a single partitioned and sorted output file
- The output file partitions are made available to reducers over HTTP
  - There are 40 (default) threads dedicated to serve the file partitions to reducers

## Shuffle and Sort: the Reduce Side

- The map output file is located on the local disk of tasktracker
- Another tasktracker (in charge of a reduce task) requires input from many other TaskTracker (that finished their map tasks)
  - How do reducers know which tasktrackers to fetch map output from?
    - When a map task finishes it notifies the parent tasktracker
    - The tasktracker notifies (with the heartbeat mechanism) the jobtracker
    - A thread in the reducer polls periodically the jobtracker
    - Tasktrackers do not delete local map output as soon as a reduce task has fetched them (WHY?)
- Copy phase: a pull approach
  - There is a small number (5) of copy threads that can fetch map outputs in parallel



## Shuffle and Sort: the Reduce Side

- The map outputs are copied to the the trasktracker running the reducer in memory (if they fit)
  - Otherwise they are copied to disk
- Input consolidation
  - A background thread merges all partial inputs into larger, sorted files
  - Note that if compression was used (for map outputs to save bandwidth), decompression will take place in memory
- Sorting the input
  - When all map outputs have been copied a merge phase starts

# Hadoop MapReduce: Types and Formats



## Data Types: Keys and Values

| | |
|---|---|
| ***Writable*** | Defines a de/serialization protocol. Every data type in Hadoop is a Writable. |
| ↑ | |
| ***WritableComprable*** | Defines a sort order. All keys must be of this type (but not values). |
| ↑ | |
| IntWritable<br>LongWritable<br>Text<br>… | Concrete classes for different data types. |
| SequenceFiles | Binary encoded of a sequence of key/value pairs |

## Map interface

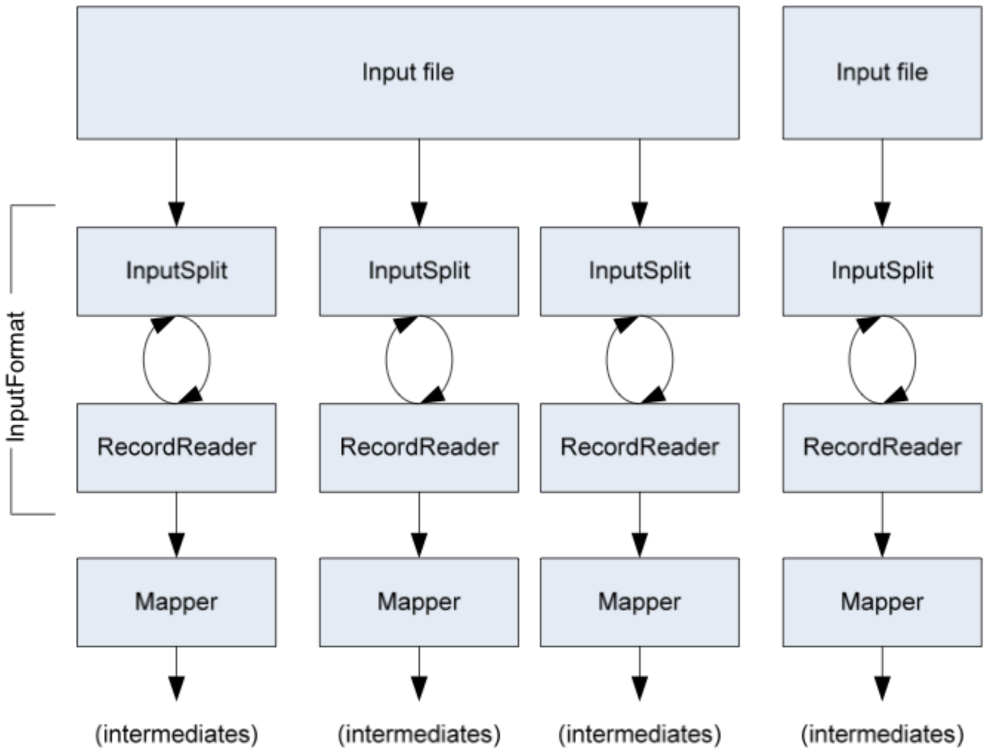
- ❑ Input / output to mappers and reducers
  - map: (k1, v1) → [(k2, v2)]
  - reduce: (k2, [v2]) → [(k3, v3)]
- ❑ In Hadoop, a mapper is created as follows:

```
void map(k1 key, v1 value, Context context)
```

- ❑ Types:
  - k1 types implement `WritableComparable`
  - v1 types implement `Writable`
- ❑ What about “context”?
  - Used to send the data to the reducers
  - context.write(k2 outKey, v2 outValue)
    - k2 implements `WritableComparable`, v2 implements `Writable`



## How the mapper get the data?



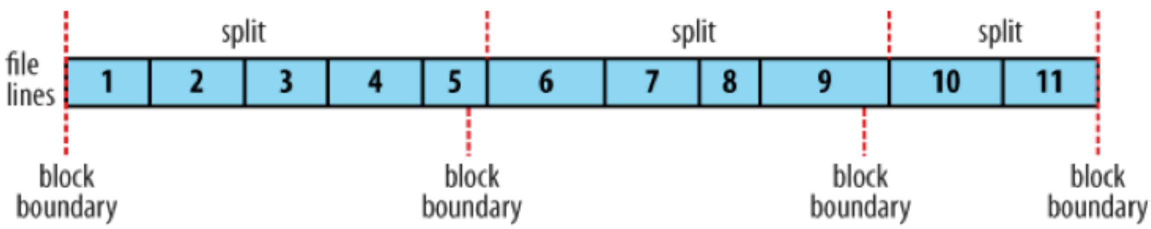
# Reading data

- Datasets are specified by `InputFormats`
  - `InputFormats` define input data (e.g. a file, a directory)
  - `InputFormats` identify partitions of the data that form an `InputSplit`
    - `InputSplit` is a (reference to a) chunk of the input processed by a single map
  - `InputFormats` is a factory for `RecordReader` objects to extract key-value records from the input source

- Splits and records are logical, they are not physically bound to a file



# Relationship between `InputSplit` and HDFS blocks



# FileInputFormat

- Base class for all implementations of `InputFormat` that use files as their data source
- It provides a method for specifying the path where the input file(s) are stored
  - The path can be a directory with many files in it
- Example of implementation: `TextInputFormat`
  - treats each newline-terminated line of a file as a value

| | | |
|---|---|---|
| On the top of the Crumpetty Tree | → | (0, On the top of the Crumpetty Tree) |
| The Quangle Wangle sat, | → | (33, The Quangle Wangle sat,) |
| But his face you could not see, | → | (57, But his face you could not see,) |
| On account of his Beaver Hat. | → | (89, On account of his Beaver Hat.) |



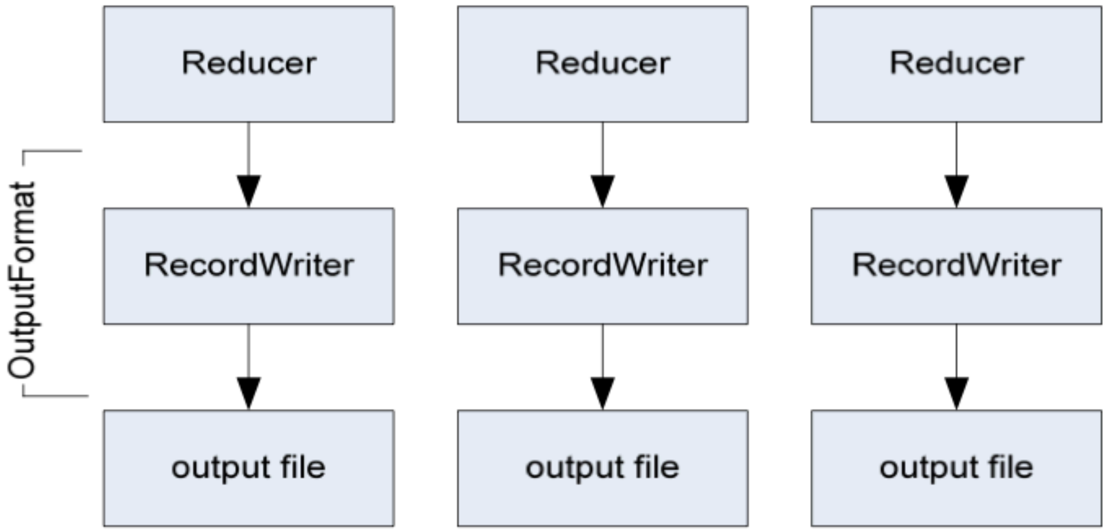
# Reduce interface

- In Hadoop, a reducer is created as follows:

```
void reduce(k2 key, iterator<v2> values, Context context)
```

- Types:
  - k2 types implement `WritableComparable`
  - v2 types implement `Writable`
  - Context is used to write data to the output

# Writing the output






# Writing the output

- Analogous to `InputFormat`
- `TextOutputFormat` writes “key value <newline>” strings to output file
- `NullOutputFormat` discards output

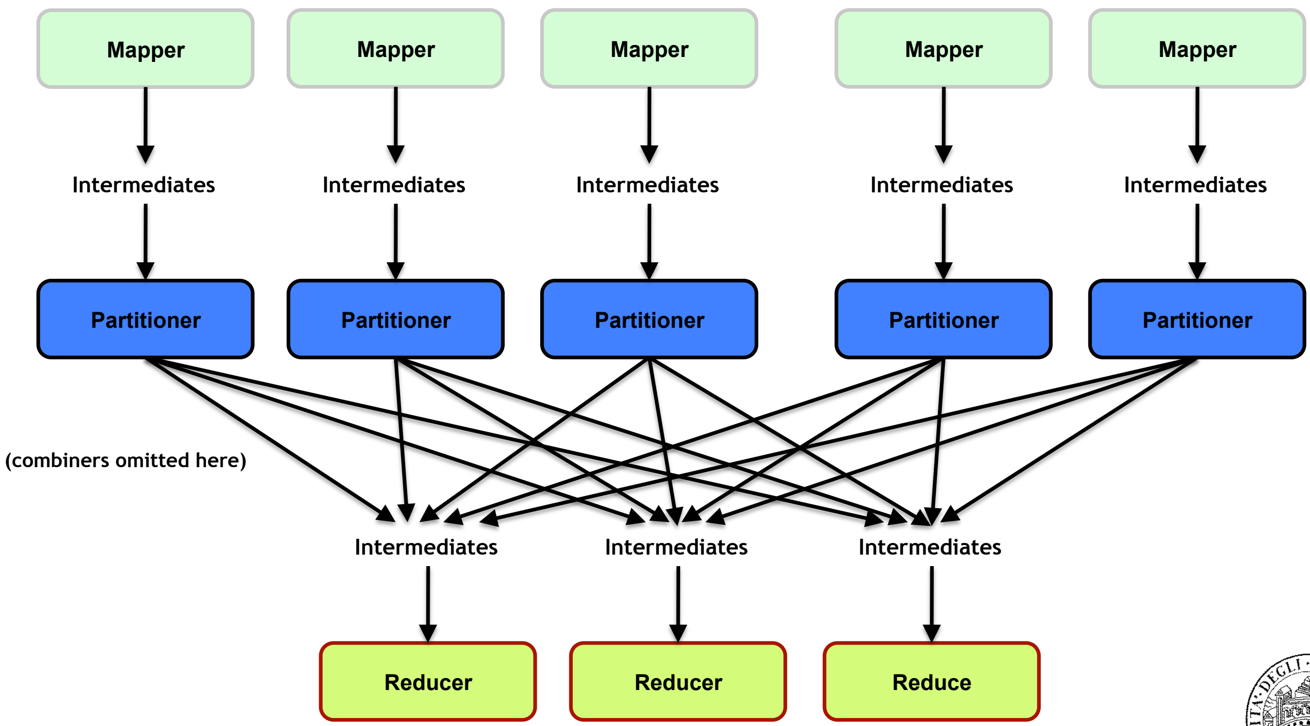
# Detour: how to divide the work among reducers?

- Solution: Partitioner
  - It is in charge of assigning intermediate keys to reducers
  - it can be customized

- Default: Hash-based partitioner
  - Computes the hash of the key modulo the number of reducers r
  - This ensures a roughly even partitioning of the key space
    - However, it ignores values: this can cause imbalance in the data processed by each reducer
  - When dealing with complex keys, even the base partitioner may need customization



# Partitioners



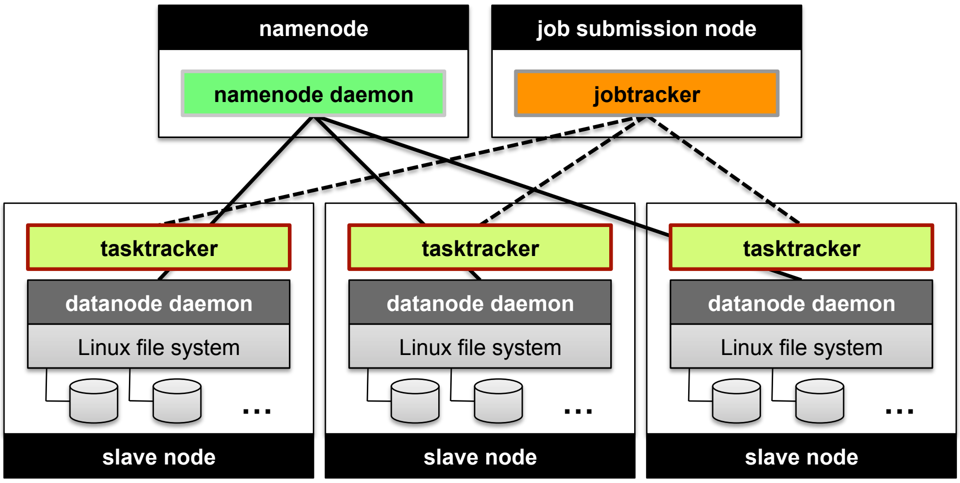
# Hadoop MapReduce: Summary



## Basic Cluster Components

- One of each:
  - Namenode (NN): master node for HDFS
  - Jobtracker (JT): master node for job submission
- Set of each per slave machine:
  - Tasktracker (TT): contains multiple task slots
  - Datanode (DN): serves HDFS data blocks

## Putting everything together…






## Basic Hadoop API

- Mapper
  - `void setup(Mapper.Context context)`
    Called once at the beginning of the task
  - `void map(K key, V value, Mapper.Context context)`
    Called once for each key/value pair in the input split
  - `void cleanup(Mapper.Context context)`
    Called once at the end of the task
- Reducer/Combiner
  - `void setup(Reducer.Context context)`
    Called once at the start of the task
  - `void reduce(K key, Iterable<V> values, Reducer.Context context)`
    Called once for each key
  - `void cleanup(Reducer.Context context)`
    Called once at the end of the task

## Basic Hadoop API

- Partitioner
  - `int getPartition(K key, V value, int numPartitions)`
    Get the partition number given total number of partitions
- Job
  - Represents a packaged Hadoop job for submission to cluster
  - Need to specify input and output paths
  - Need to specify input and output formats
  - Need to specify mapper, reducer, combiner, partitioner classes
  - Need to specify intermediate/final key/value classes
  - Need to specify number of reducers (WHY?)



## Three Gotchas

- Avoid object creation at all costs
  - Reuse Writable objects, change the payload
- Execution framework reuses value object in reducer
- Passing parameters via class statics