Tutorial: MapReduce
Theory and Practice of Data-intensive Applications

Pietro Michiardi

Eurecom
Introduction
What is MapReduce

- **A programming model:**
  - Inspired by functional programming
  - Allows expressing distributed computations on massive amounts of data

- **An execution framework:**
  - Designed for large-scale data processing
  - Designed to run on clusters of commodity hardware
What is this Tutorial About

- **Design of scalable algorithms with MapReduce**
  - Applied algorithm design and case studies

- **In-depth description of MapReduce**
  - Principles of functional programming
  - The execution framework

- **In-depth description of Hadoop**
  - Architecture internals
  - Software components
  - Cluster deployments
Motivations
Big Data

- Vast repositories of data
  - Web-scale processing
  - Behavioral data
  - Physics
  - Astronomy
  - Finance

- “The fourth paradigm” of science [6]
  - Data-intensive processing is fast becoming a necessity
  - Design algorithms capable of scaling to real-world datasets

- It’s not the algorithm, it’s the data! [2]
  - More data leads to better accuracy
  - With more data, accuracy of different algorithms converges
Key Ideas Behind MapReduce
For data-intensive workloads, a large number of commodity servers is preferred over a small number of high-end servers

- Cost of super-computers is not linear
- But datacenter efficiency is a difficult problem to solve [3, 5]

Some numbers (∼ 2010):

- Data processed by Google every day: 20 PB
- Data processed by Facebook every day: 15 TB
Implications of Scaling Out

- **Processing data is quick, I/O is very slow**
  - 1 HDD = 75 MB/sec
  - 1000 HDDs = 75 GB/sec

- **Sharing vs. Shared nothing:**
  - Sharing: manage a common/global state
  - Shared nothing: *independent* entities, no common state

- **Sharing is difficult:**
  - Synchronization, deadlocks
  - Finite bandwidth to access data from SAN
  - Temporal dependencies are complicated (restarts)
Failures are the norm, not the exception

- **LALN data [DSN 2006]**
  - Data for 5000 machines, for 9 years
  - Hardware: 60%, Software: 20%, Network 5%

- **DRAM error analysis [Sigmetrics 2009]**
  - Data for 2.5 years
  - 8% of DIMMs affected by errors

- **Disk drive failure analysis [FAST 2007]**
  - Utilization and temperature major causes of failures

- **Amazon Web Service failure [April 2011]**
  - Cascading effect
Implications of Failures

- **Failures are part of everyday life**
  - Mostly due to the scale and shared environment

- **Sources of Failures**
  - Hardware / Software
  - Electrical, Cooling, ...
  - Unavailability of a resource due to overload

- **Failure Types**
  - Permanent
  - Transient
Move Processing to the Data

- **Drastic departure from high-performance computing model**
  - HPC: distinction between processing nodes and storage nodes
  - HPC: CPU intensive tasks

- **Data intensive workloads**
  - Generally not processor demanding
  - The network becomes the bottleneck
  - MapReduce assumes processing and storage nodes to be colocated: *Data Locality*

- **Distributed filesystems are necessary**
Introduction

Big Ideas

Process Data Sequentially and Avoid Random Access

- **Data intensive workloads**
  - Relevant datasets are too large to fit in memory
  - Such data resides on disks

- **Disk performance is a bottleneck**
  - Seek times for random disk access are *the* problem
    - Example: 1 TB DB with $10^{10}$ 100-byte records. Updates on 1% requires 1 month, reading and rewriting the whole DB would take 1 day
  - Organize computation for sequential reads

---

1 From a post by Ted Dunning on the Hadoop mailing list
Implications of Data Access Patterns

- MapReduce is designed for
  - *batch processing*
  - involving (mostly) *full scans* of the dataset

- Typically, data is collected “elsewhere” and copied to the distributed filesystem

- Data-intensive applications
  - Read and process the whole Internet dataset from a crawler
  - Read and process the whole Social Graph
Hide System-level Details

- **Separate the *what* from the *how***
  - MapReduce abstracts away the “distributed” part of the system
  - Such details are handled by the framework

- **In-depth knowledge of the framework is key**
  - Custom data reader/writer
  - Custom *data partitioning*
  - Memory utilization

- **Auxiliary components**
  - Hadoop Pig
  - Hadoop Hive
  - Cascading/Scalding
  - ... and many many more!
We can define scalability along two dimensions

▶ In terms of data: given twice the amount of data, the same algorithm should take no more than twice as long to run
▶ In terms of resources: given a cluster twice the size, the same algorithm should take no more than half as long to run

Embarassingly parallel problems

▶ Simple definition: independent (shared nothing) computations on fragments of the dataset
▶ It’s not easy to decide whether a problem is embarrassingly parallel or not

MapReduce is a first attempt, not the final answer
Part One
The MapReduce Framework
Preliminaries
Divide and Conquer

- **A feasible approach to tackling large-data problems**
  - Partition a large problem into smaller sub-problems
  - *Independent* sub-problems executed in parallel
  - Combine intermediate results from each individual worker

- **The workers can be:**
  - Threads in a processor core
  - Cores in a multi-core processor
  - Multiple processors in a machine
  - Many machines in a cluster

- **Implementation details of divide and conquer are complex**
Divide and Conquer: How to?

- **Decompose** the original problem in smaller, parallel tasks

- Schedule tasks on workers distributed in a cluster
  - Data locality
  - Resource availability

- Ensure workers get the data they need

- Coordinate synchronization among workers

- **Share** partial results

- Handle failures
The MapReduce Approach

- **Shared memory approach** (OpenMP, MPI, ...)
  - Developer needs to take care of (almost) everything
  - Synchronization, Concurrency
  - Resource allocation

- **MapReduce: a shared nothing approach**
  - Most of the above issues are taken care of
  - Problem decomposition and sharing partial results need particular attention
  - Optimizations (memory and network consumption) are tricky
The MapReduce Programming model
Functional Programming Roots

- **Key feature: higher order functions**
  - Functions that accept other functions as arguments
  - **Map** and **Fold**

**Figure:** Illustration of *map* and *fold.*
Functional Programming Roots

**map phase:**
- Given a list, *map* takes as an argument a function $f$ (that takes a single argument) and applies it to all elements in the list.

**fold phase:**
- Given a list, *fold* takes as arguments a function $g$ (that takes two arguments) and an initial value.
- $g$ is first applied to the initial value and the first item in the list.
- The result is stored in an intermediate variable, which is used as an input together with the next item to a second application of $g$.
- The process is repeated until all items in the list have been consumed.
Functional Programming Roots

We can view map as a transformation over a dataset
- This transformation is specified by the function $f$
- Each functional application happens in isolation
- The application of $f$ to each element of a dataset can be parallelized in a straightforward manner

We can view fold as an aggregation operation
- The aggregation is defined by the function $g$
- Data locality: elements in the list must be “brought together”
- If we can group element of the list, also the fold phase can proceed in parallel

Associative and commutative operations
- Allow performance gains through local aggregation and reordering
**Functional Programming and MapReduce**

- **Equivalence of MapReduce and Functional Programming:**
  - The map of MapReduce corresponds to the map operation
  - The reduce of MapReduce corresponds to the fold operation

- **The framework coordinates the map and reduce phases:**
  - Grouping intermediate results happens in parallel

- **In practice:**
  - User-specified computation is applied (in parallel) to all input records of a dataset
  - Intermediate results are aggregated by another user-specified computation
What can we do with MapReduce?

- **MapReduce “implements” a subset of functional programming**
  - The programming model appears quite limited

- **There are several important problems that can be adapted to MapReduce**
  - In this tutorial we will focus on illustrative cases
  - We will see in detail “design patterns”
    ★ How to transform a problem and its input
    ★ How to save memory and bandwidth in the system
Mappers and Reducers
**Data Structures**

- **Key-value pairs are the basic data structure in MapReduce**
  - Keys and values can be: integers, float, strings, raw bytes
  - They can also be arbitrary data structures

- **The design of MapReduce algorithms involves:**
  - Imposing the key-value structure on arbitrary datasets
    - E.g.: for a collection of Web pages, input keys may be URLs and values may be the HTML content
  - In some algorithms, input keys are not used, in others they uniquely identify a record
  - Keys can be combined in complex ways to design various algorithms
A MapReduce job

The programmer defines a mapper and a reducer as follows\(^2\):

- **map**: \((k_1, v_1) \rightarrow [(k_2, v_2)]\)
- **reduce**: \((k_2, [v_2]) \rightarrow [(k_3, v_3)]\)

A MapReduce job consists in:

- A dataset stored on the underlying distributed filesystem, which is split in a number of files across machines
- The mapper is applied to every input key-value pair to generate intermediate key-value pairs
- The reducer is applied to all values associated with the same intermediate key to generate output key-value pairs

\(^2\)We use the convention \([\cdots]\) to denote a list.
Where the magic happens

- Implicit between the map and reduce phases is a distributed “group by” operation on intermediate keys
  - Intermediate data arrive at each reducer in order, sorted by the key
  - No ordering is guaranteed across reducers

- Output keys from reducers are written back to the distributed filesystem
  - The output may consist of \( r \) distinct files, where \( r \) is the number of reducers
  - Such output may be the input to a subsequent MapReduce phase

- Intermediate keys are transient:
  - They are not stored on the distributed filesystem
  - They are “spilled” to the local disk of each machine in the cluster
A Simplified view of MapReduce

**Figure:** Mappers are applied to all input key-value pairs, to generate an arbitrary number of intermediate pairs. Reducers are applied to all intermediate values associated with the same intermediate key. Between the map and reduce phase lies a barrier that involves a large distributed sort and group by.
"Hello World" in MapReduce

Figure: Pseudo-code for the word count algorithm.
“Hello World” in MapReduce

**Input:**
- Key-value pairs: (docid, doc) stored on the distributed filesystem
  - docid: unique identifier of a document
  - doc: is the text of the document itself

**Mapper:**
- Takes an input key-value pair, tokenize the document
- Emits intermediate key-value pairs: the word is the key and the integer is the value

**The framework:**
- Guarantees all values associated with the same key (the word) are brought to the same reducer

**The reducer:**
- Receives all values associated to some keys
- Sums the values and writes output key-value pairs: the key is the word and the value is the number of occurrences
Implementation and Execution Details

- The **partitioner** is in charge of assigning intermediate keys (words) to reducers
  - Note that the partitioner can be customized

- **How many map and reduce tasks?**
  - The framework essentially takes care of map tasks
  - The designer/developer takes care of reduce tasks

- **In this tutorial we will focus on Hadoop**
  - Other implementations of the framework exist: Google, Disco, ...
Handle with care!

- **Using external resources**
  - E.g.: Other data stores than the distributed file system
  - Concurrent access by many map/reduce tasks

- **Side effects**
  - Not allowed in functional programming
  - E.g.: preserving state across multiple inputs
  - State is kept internal

- **I/O and execution**
  - **External** side effects using distributed data stores (e.g. BigTable)
  - No input (e.g. computing $\pi$), no reducers, never no mappers
The Execution Framework
The Execution Framework

- **MapReduce program, a.k.a. a job:**
  - Code of mappers and reducers
  - Code for combiners and partitioners (optional)
  - Configuration parameters
  - All packaged together

- **A MapReduce job is submitted to the cluster**
  - The framework takes care of everything else
  - Next, we will delve into the details
Scheduling

- Each Job is broken into tasks
  - Map tasks work on fractions of the input dataset, as defined by the underlying distributed filesystem
  - Reduce tasks work on intermediate inputs and write back to the distributed filesystem

- The number of tasks may exceed the number of available machines in a cluster
  - The scheduler takes care of maintaining something similar to a queue of pending tasks to be assigned to machines with available resources

- Jobs to be executed in a cluster requires scheduling as well
  - Different users may submit jobs
  - Jobs may be of various complexity
  - Fairness is generally a requirement
Scheduling

- **The scheduler component can be customized**
  - As of today, for Hadoop, there are various schedulers

- **Dealing with stragglers**
  - Job execution time depends on the slowest map and reduce tasks
  - *Speculative* execution can help with slow machines
    - But data locality may be at stake

- **Dealing with skew in the distribution of values**
  - E.g.: temperature readings from sensors
  - In this case, scheduling cannot help
  - It is possible to work on customized partitioning and sampling to solve such issues [Advanced Topic]
Data/code co-location

- **How to feed data to the code**
  - In MapReduce, this issue is intertwined with scheduling and the underlying distributed filesystem

- **How data locality is achieved**
  - The scheduler starts the task on the node that holds a particular block of data required by the task
  - If this is not possible, tasks are started elsewhere, and data will cross the network
    - Note that usually input data is replicated
  - Distance rules [11] help dealing with bandwidth consumption
    - Same rack scheduling
Synchronization

- In MapReduce, synchronization is achieved by the “shuffle and sort” barrier
  - Intermediate key-value pairs are grouped by key
  - This requires a distributed sort involving all mappers, and taking into account all reducers
  - If you have $m$ mappers and $r$ reducers this phase involves up to $m \times r$ copying operations

- **IMPORTANT**: the reduce operation cannot start until all mappers have finished
  - This is different from functional programming that allows “lazy” aggregation
  - In practice, a common optimization is for reducers to **pull** data from mappers as soon as they finish
Errors and faults

Using quite simple mechanisms, the MapReduce framework deals with:

- **Hardware failures**
  - Individual machines: disks, RAM
  - Networking equipment
  - Power / cooling

- **Software failures**
  - Exceptions, bugs

- **Corrupt and/or invalid input data**
Partitioners and Combiners
Partitioners

Partitioners are responsible for:

- Dividing up the intermediate key space
- Assigning intermediate key-value pairs to reducers
- Specify the task to which an intermediate key-value pair must be copied

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
Combiners

- **Combiners are an (optional) optimization:**
  - Allow local aggregation before the “shuffle and sort” phase
  - Each combiner operates in isolation

- **Essentially, combiners are used to save bandwidth**
  - E.g.: word count program

- **Combiners can be implemented using local data-structures**
  - E.g., an associative array keeps intermediate computations and aggregation thereof
  - The map function only emits once all input records (even all input splits) are processed
Partitioners and Combiners, an Illustration

**Figure:** Complete view of MapReduce illustrating combiners and partitioners.

**Note:** in Hadoop, partitioners are executed before combiners.
The Distributed Filesystem
Colocate data and computation!

- 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 eventually 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.
In this tutorial we will focus on HDFS, the Hadoop implementation of the Google distributed filesystem (GFS)

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
HDFS

- **Divide user data into blocks**
  - Blocks are big! [64, 128] MB
  - Avoids problems related to metadata management

- **Replicate blocks across the local disks of nodes in the cluster**
  - Replication is handled by storage nodes themselves (similar to chain replication) and follows distance rules

- **Master-slave architecture**
  - NameNode: master maintains the namespace (metadata, file to block mapping, location of blocks) and maintains overall health of the file system
  - DataNode: slaves manage the data blocks
HDFS, an Illustration

Figure: The architecture of HDFS.
HDFS I/O

A typical read from a client involves:
1. Contact the NameNode to determine where the actual data is stored
2. NameNode replies with block identifiers and locations (i.e., which DataNode)
3. Contact the DataNode to fetch data

A typical write from a client involves:
1. Contact the NameNode to update the namespace and verify permissions
2. NameNode allocates a new block on a suitable DataNode
3. The client directly streams to the selected DataNode
4. Currently, HDFS files are immutable

- Data is never moved through the NameNode
  - Hence, there is no bottleneck
HDFS Replication

- By default, HDFS stores 3 separate copies of each block
  - This ensures reliability, availability and performance

- Replication policy
  - Spread replicas across different racks
  - Robust against cluster node failures
  - Robust against rack failures

- Block replication benefits MapReduce
  - Scheduling decisions can take replicas into account
  - Exploit better data locality
HDFS: more on operational assumptions

- A small number of large files is preferred over a large number of small files
  - Metadata may explode
  - Input splits for MapReduce based on individual files
    - Mappers are launched for every file
    - High startup costs
    - Inefficient “shuffle and sort”

- Workloads are **batch oriented**

- Not full POSIX

- Cooperative scenario
Hadoop implementation of MapReduce
From Theory to Practice

- **The story so far**
  - Concepts behind the MapReduce Framework
  - Overview of the programming model

- **Hadoop implementation of MapReduce**
  - HDFS in details
  - Hadoop I/O
  - Hadoop MapReduce
    - Implementation details
    - Types and Formats
    - Features in Hadoop

- **Hadoop Deployments**
  - The BigFoot platform (if time allows)
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)
- Task ID from TaskInProgress is not a unique identifier
HDFS in details
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

- Hadoop Distributed Filesystem[10, 11]
  - Very large files
  - Streaming data access
  - Commodity hardware
HDFS Blocks

- **(Big) files are broken into block-sized chunks**
  - **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**
  - Reliability and parallel access

- **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 namespce 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
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\(^3\) returns the local **DataNode**

\(^3\)Exceptions exist due to stragglers.
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
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 \texttt{sync()} to force synchronization
- \texttt{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 I/O
I/O operations in Hadoop

- **Reading and writing data**
  - From/to HDFS
  - From/to local disk drives
  - Across machines (inter-process communication)

- **Customized tools for large amounts of data**
  - Hadoop does not use Java native classes
  - Allows flexibility for dealing with custom data (e.g. binary)

- **What’s next**
  - Overview of what Hadoop offers
  - For an in depth knowledge, use [11]
Data Integrity

- Every I/O operation on disks or the network may corrupt data
  - Users expect data not to be corrupted during storage or processing
  - Data integrity usually achieved with checksums

- HDFS transparently checksums all data during I/O
  - HDFS makes sure that storage overhead is roughly 1%
  - DataNodes are in charge of checksumming
    - With replication, the last replica performs the check
    - Checksums are timestamped and logged for statistics on disks
  - Checksumming is also run periodically in a separate thread
    - Note that thanks to replication, error correction is possible
Compression

- **Why using compression**
  - Reduce storage requirements
  - Speed up data transfers (across the network or from disks)

- **Compression and Input Splits**
  - IMPORTANT: use compression that supports splitting (e.g. bzip2)

- **Splittable files, Example 1**
  - Consider an uncompressed file of 1GB
  - HDFS will split it in 16 blocks, 64MB each, to be processed by separate Mappers
Compression

- **Splittable files, Example 2 (gzip)**
  - Consider a compressed file of 1GB
  - HDFS will split it in 16 blocks of 64MB each
  - Creating an `InputSplit` for each block will not work, since it is not possible to read at an arbitrary point

- **What’s the problem?**
  - This forces MapReduce to treat the file as a single split
  - Then, a single Mapper is fired by the framework
  - For this Mapper, only 1/16-th is local, the rest comes from the network

- **Which compression format to use?**
  - Use bzip2
  - Otherwise, use `SequenceFiles`
  - See Chapter 4 (page 84) [11]
Serialization

- Transforms structured objects into a byte stream
  - For transmission over the network: Hadoop uses RPC
  - For persistent storage on disks

- Hadoop uses its own serialization format, Writable
  - Comparison of types is crucial (Shuffle and Sort phase): Hadoop provides a custom RawComparator, which avoids deserialization
  - Custom Writable for having full control on the binary representation of data
  - Also “external” frameworks are allowed: enter Avro

- Fixed-length or variable-length encoding?
  - Fixed-length: when the distribution of values is uniform
  - Variable-length: when the distribution of values is not uniform
Sequence Files

- **Specialized data structure to hold custom input data**
  - Using blobs of binaries is not efficient

- **SequenceFiles**
  - Provide a persistent data structure for binary key-value pairs
  - Also work well as containers for smaller files so that the framework is more happy (remember, better few large files than lots of small files)
  - They come with the `sync()` method to introduce sync points to help managing `InputSplits` for MapReduce
How Hadoop MapReduce Works
Anatomy of a MapReduce Job Run

1: run job
2: get new job ID
3: copy job resources
4: submit job
5: initialize job
6: retrieve input splits
7: heartbeat (returns task)
8: retrieve job resources
9: launch
10: run

MapReduce program
JobClient
JobTracker
TaskTracker
Child
MapTask or ReduceTask

client JVM
client node
jobtracker node
tasktracker node

Shared FileSystem (e.g., HDFS)
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 hearbeats to the JobTracker**
  - TaskTracker **is alive**
  - Heartbeat contains also 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. 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
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
  - We will cover Dumbo
Handling Failures

In the real world, code is buggy, processes crash and machine fails

**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

---

4 With streaming, you need to take care of the orphaned process.
5 Exception is made for speculative execution
Handling Failures

- **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 releases:
    - Multiple JobTrackers
    - Use ZooKeeper as a coordination mechanisms
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*
  - “Contrib” module, requires manual installation

- **Capacity Scheduler**
  - Hierarchical queues (mimic an organization)
  - FIFO scheduling in each queue
  - Supports priority
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 Map Side

Index file

Partition 1
offset

Partition 2
offset

Data file

Key length
Value length
Key
Value

One Record

Partition 1

Key length
Value length
Key
Value

Partition 2

Key length
Value length
Key
Value

Index 2

Data 2

Index 3

Data 3
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 tasktracker 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
- All map outputs are sorted maintaining their sort ordering, in rounds
Hadoop MapReduce Types and Formats
MapReduce Types

- **Input / output to mappers and reducers**
  - map: \((k_1, v_1) \rightarrow [(k_2, v_2)]\)
  - reduce: \((k_2, [v_2]) \rightarrow [(k_3, v_3)]\)

- **In Hadoop, a mapper is created as follows:**
  - `void map(K1 key, V1 value, OutputCollector<K2, V2> output, Reporter reporter)`

- **Types:**
  - \(K\) types implement `WritableComparable`
  - \(V\) types implement `Writable`
What is a **Writable**

- **Hadoop defines its own classes for strings (Text), integers (intWritable), etc...**

- **All keys are instances of WritableComparable**
  - Why comparable?

- **All values are instances of Writable**
Getting Data to the Mapper

Diagram:

- Input file
  - InputSplit
    - RecordReader
      - Mapper
        - (intermediates)
Reading Data

- **Datasets are specified by InputFormats**
  - InputFormats define input data (e.g. a file, a directory)
  - InputFormats is a factory for RecordReader objects to extract key-value records from the input source

- **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
    - Largest split is processed first
  - Each split is divided into records, and the map processes each record (a key-value pair) in turn
  - Splits and records are logical, they are not physically bound to a file
The relationship between `InputSplit` and HDFS blocks
FileInputFormat and Friends

- **TextInputFormat**
  - Treats each newline-terminated line of a file as a value

- **KeyValueTextInputFormat**
  - Maps newline-terminated text lines of “key” SEPARATOR “value”

- **SequenceFileInputFormat**
  - Binary file of key-value pairs with some additional metadata

- **SequenceFileAsTextInputFormat**
  - Same as before but, maps `(k.toString(), v.toString())`
Filtering File Inputs

- **FileInputFormat** reads all files out of a specified directory and send them to the mapper.

- Delegates filtering this file list to a method subclasses may override.
  - Example: create your own “xyzFileInputFormat” to read *.xyz from a directory list.
Record Readers

- **Each InputFormat provides its own RecordReader implementation**

- **LineRecordReader**
  - Reads a line from a text file

- **KeyValueRecordReader**
  - Used by `KeyValueTextInputFormat`
Input Split Size

- **FileInputFormat** divides large files into chunks
  - Exact size controlled by `mapred.min.split.size`

- Record readers receive file, offset, and length of chunk
  - Example

  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.)

- Custom **InputFormat** implementations may override split size
Sending Data to Reducers

- Map function receives `OutputCollector` object
  - `OutputCollector.collect()` receives key-value elements

- Any `(WritableComparable, Writable)` can be used

- By default, mapper output type assumed to be the same as the reducer output type
WritableComparator

- **Compares** WritableComparable data
  - Will call the `WritableComparable.compare()` method
  - Can provide fast path for serialized data

- **Configured through:**
  ```java
  JobConf.setOutputValueGroupingComparator()
  ```
Partitioner

- `int getPartition(key, value, numPartitions)`
  - Outputs the partition number for a given key
  - One partition == all values sent to a single reduce task

- **HasPartitioner used by default**
  - Uses `key.hashCode()` to return partition number

- **JobConf used to set Partitioner implementation**
The Reducer

- `void reduce(k2 key, Iterator<v2> values, OutputCollector<k3, v3> output, Reporter reporter)`

- Keys and values sent to one partition all go to the same reduce task

- Calls are sorted by key
  - “Early” keys are reduced and output before “late” keys
Writing the Output

![Diagram showing the process of writing the output in Hadoop MapReduce. The diagram includes multiple reducers and record writers, leading to output files.](image-url)
Writing the Output

- Analogous to InputFormat

- TextOutputFormat writes “key value <newline>” strings to output file

- SequenceFileOutputFormat uses a binary format to pack key-value pairs

- NullOutputFormat discards output
Hadoop MapReduce Features
Developing a MapReduce Application
Preliminaries

Writing a program in MapReduce has a certain flow to it

- Start by writing the map and reduce functions
  - Write unit tests to make sure they do what they should
- Write a driver program to run a job
  - The job can be run from the IDE using a small subset of the data
  - The debugger of the IDE can be used
- Eventually, you can unleash the job on a cluster
  - Debugging a distributed program is challenging

Once the job is running properly

- Perform standard checks to improve performance
- Perform task profiling
Configuration

Before writing a MapReduce program, we need to set up and configure the development environment

- Components in Hadoop are configured with an ad hoc API
- Configuration class is a collection of properties and their values
- Resources can be combined into a configuration

Configuring the IDE

- In the IDE create a new project and add all the JAR files from the top level of the distribution and form the lib directory
- For Eclipse there are also available plugins
- Commercial IDE also exist (Karmasphere)

Alternatives

- Switch configurations (local, cluster)
- Alternatives (see Cloudera documentation for Ubuntu) is very effective
Local Execution

- **Use the GenericOptionsParser, Tool and ToolRunner**
  - These helper classes make it easy to intervene on job configurations
  - These are additional configurations to the core configuration

- **The run() method**
  - Constructs and configure a JobConf object and launch it

- **How many reducers?**
  - In a local execution, there is a single (eventually none) reducer
  - Even by setting a number of reducer larger than one, the option will be ignored
Cluster Execution

- Packaging
- Launching a Job
- The WebUI
- Hadoop Logs
- Running Dependent Jobs, and Oozie
Hadoop Deployments
Setting up a Hadoop Cluster

- **Cluster deployment**
  - Private cluster
  - Cloud-based cluster
  - AWS Elastic MapReduce

- **Outlook:**
  - Cluster specification
    - Hardware
    - Network Topology
  - Hadoop Configuration
    - Memory considerations
Cluster Specification

- **Commodity Hardware**
  - Commodity $\neq$ Low-end
    - False economy due to failure rate and maintenance costs
  - Commodity $\neq$ High-end
    - High-end machines perform better, which would imply a smaller cluster
    - A single machine failure would compromise a large fraction of the cluster

- **A 2010 specification:**
  - 2 quad-cores
  - 16-24 GB ECC RAM
  - $4 \times 1$ TB SATA disks
  - Gigabit Ethernet

---

6 Why not using RAID instead of JBOD?
Cluster Specification

Example:
- Assume your data grows by 1 TB per week
- Assume you have three-way replication in HDFS
  → You need additional 3TB of raw storage per week
- Allow for some overhead (temporary files, logs)
  → This is a new machine per week

How to dimension a cluster?
- Obviously, you won’t buy a machine per week!!
- The idea is that the above back-of-the-envelope calculation is that you can project over a 2 year life-time of your system
  → You would need a 100-machine cluster

Where should you put the various components?
- Small cluster: NameNode and JobTracker can be colocated
- Large cluster: requires more RAM at the NameNode
Cluster Specification

Should we use 64-bit or 32-bit machines?
- NameNode should run on a 64-bit machine: this avoids the 3GB Java heap size limit on 32-bit machines
- Other components should run on 32-bit machines to avoid the memory overhead of large pointers

What's the role of Java?
- Recent releases (Java6) implement some optimization to eliminate large pointer overhead
  → A cluster of 64-bit machines has no downside
Cluster Specification: Network Topology
Cluster Specification: Network Topology

- **Two-level network topology**
  - Switch redundancy is not shown in the figure

- **Typical configuration**
  - 30-40 servers per rack
  - 1 GB switch per rack
  - Core switch or router with 1GB or better

- **Features**
  - Aggregate bandwidth between nodes on the same rack is much larger than for nodes on different racks
  - **Rack awareness**
    - Hadoop should know the cluster topology
    - Benefits both HDFS (data placement) and MapReduce (locality)
Hadoop Configuration

- There are a handful of files for controlling the operation of an Hadoop Cluster
  - See next slide for a summary table

- Managing the configuration across several machines
  - All machines of an Hadoop cluster must be in sync!
  - What happens if you dispatch an update and some machines are down?
  - What happens when you add (new) machines to your cluster?
  - What if you need to patch MapReduce?

- Common practice: use configuration management tools
  - Chef, Puppet, ...
  - Declarative language to specify configurations
  - Allow also to install software
# Hadoop Configuration

## Table: Hadoop Configuration Files

<table>
<thead>
<tr>
<th>Filename</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hadoop-env.sh</td>
<td>Bash script</td>
<td>Environment variables that are used in the scripts to run Hadoop.</td>
</tr>
<tr>
<td>core-site.xml</td>
<td>Hadoop configuration XML</td>
<td>I/O settings that are common to HDFS and MapReduce.</td>
</tr>
<tr>
<td>hdfs-site.xml</td>
<td>Hadoop configuration XML</td>
<td>Namenode, the secondary namenode, and the datanodes.</td>
</tr>
<tr>
<td>mapred-site.xml</td>
<td>Hadoop configuration XML</td>
<td>Jobtracker, and the tasktrackers.</td>
</tr>
<tr>
<td>masters</td>
<td>Plain text</td>
<td>A list of machines that each run a secondary namenode.</td>
</tr>
<tr>
<td>slaves</td>
<td>Plain text</td>
<td>A list of machines that each run a datanode and a tasktracker.</td>
</tr>
</tbody>
</table>
Hadoop Configuration: memory utilization

- **Hadoop uses a lot of memory**
  - Default values, for a typical cluster configuration
    - DataNode: 1 GB
    - TaskTracker: 1 GB
    - Child JVM map task: 2 × 200MB
    - Child JVM reduce task: 2 × 200MB

- **All the moving parts of Hadoop (HDFS and MapReduce) can be individually configured**
  - This is true for cluster configuration but also for **job specific configurations**

- **Hadoop is fast when using RAM**
  - Generally, MapReduce Jobs are not CPU-bound
  - Avoid I/O on disk as much as you can
  - Minimize network traffic
    - Customize the partitioner
    - Use compression (→ decompression is in RAM)
Elephants in the cloud!

- May organization run Hadoop in private clusters
  - Pros and cons

- Cloud based Hadoop installations (Amazon biased)
  - Use Cloudera + Whirr
  - Use Elastic MapReduce
Hadoop on EC2

- **Launch instances of a cluster on demand, paying by hour**
  - CPU, in general bandwidth is used from within a datacenter, hence it’s free

- **Apache Whirr project**
  - Launch, terminate, modify a running cluster
  - Requires AWS credentials

- **Example**
  - Launch a cluster `test-hadoop-cluster`, with one master node (`JobTracker` and `NameNode`) and 5 worker nodes (`DataNodes` and `TaskTrackers`)
  - `hadoop-ec2 launch-cluster test-hadoop-cluster 5`
  - See project webpage and Chapter 9, page 290 [11]
AWS Elastic MapReduce

**Hadoop as a service**
- Amazon handles everything, which becomes transparent
- How this is done remains a mystery

**Focus on What not How**
- All you need to do is to package a MapReduce Job in a JAR and upload it using a Web Interface
- Other Jobs are available: python, pig, hive, ...
- Test your jobs locally!!!


References II


References III

