Data-intensive Scalable Computing

Introduction

Pietro Michiardi

Eurecom

Introduction and Recap

From Theory to Practice

- **The story so far**
  - Principles behind the MapReduce Framework
  - Programming model
  - Algorithm design and patterns

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

- **Hadoop Deployments**
  - The BigFoot platform

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 (a.k.a. speculative execution)
  - Task ID from TaskInProgress is not a unique identifier

Hadoop Distributed File-System

Collocate 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

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[1, 2]**
  - Very large files
  - Streaming data access
  - Commodity hardware

HDFS Blocks

- **(Big) files are broken into block-sized chunks**
  - Blocks are big! [64, 128] MB
  - Avoids problems related to metadata management
  - `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
  - Maintains overall health of the file system

- **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 Illustration

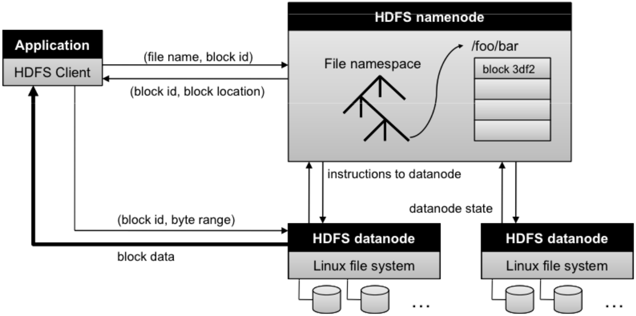


Figure: The architecture of HDFS.

Anatomy of a File Read

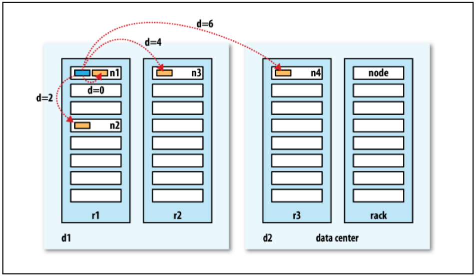
- **`NameNode` is only used to get block location**
  - Unresponsive `DataNode` are discarded by clients
  - Batch reading of blocks is allowed
- **"External" clients**
  - For each block, the `NameNode` returns a set of `DataNodes` holding a copy thereof
  - `DataNodes` are sorted according to their proximity to the client
- **"MapReduce" clients**
  - `TaskTracker` and `DataNodes` are collocated
  - For each block, the `NameNode` usually[1] returns the local `DataNode`

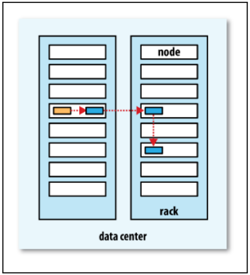
[1] Exceptions exist due to stragglers.

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





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)

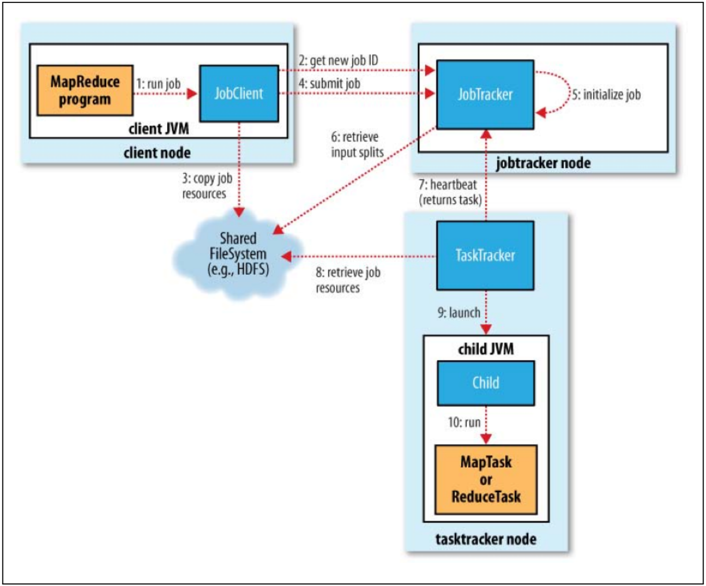
Hadoop MapReduce

Disclaimer

- **MapReduce APIs**
  - Fast evolving
  - Sometimes confusing

- **Do NOT rely on this slide deck as a reference**
  - Use appropriate API docs
  - Use Eclipse

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
  - Queuing 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

- **Heartbeat-based mechanism**
  - `TaskTrackers` periodically send heartbeats 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.* 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 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
  - NOTE: this feature is heavily used in industry, with some tricky downsides

Scheduling in detail

- **FIFO Scheduler (default behavior)**
  - Each job uses the whole cluster[2]
  - 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 into pools, one for each user
    - Users that submit more jobs have no more resources than oterhs
    - Can guarantee minimum capacity per pool
  - Supports preemption
- **Capacity Scheduler**
  - Hierarchical queues (mimic an organization)
  - FIFO scheduling in each queue
  - Supports priority

[2] To be precise, all required slots are assigned to highest priority job.

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[3]
  - `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[4]
    - Default behavior: if any task fails 4 times, the job fails

[3] With streaming, you need to take care of the orphaned process.

[4] Exception is made for speculative execution

Handling Failures

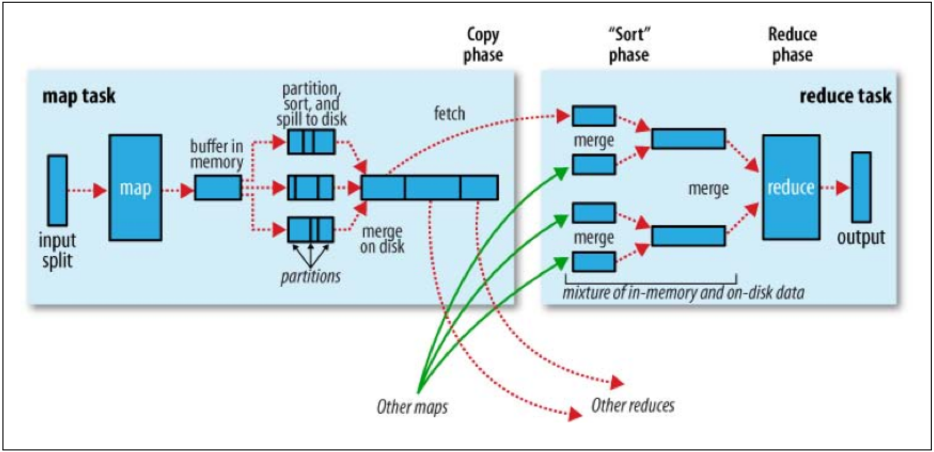
- **`TaskTracker` Failure**
  - Types: crash, running very slowly
  - Heartbeats will not be sent to `JobTracker`
  - `JobTracker` waits for a timeout (10 minutes), then it removes the `TaskTracker` from its scheduling pool
  - `JobTracker` needs to reschedule even *completed* tasks (WHY?)
  - `JobTracker` 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 (and commercial) releases:
    - ⋆ Multiple `JobTrackers`
    - ⋆ Use ZooKeeper as a coordination mechanisms
    - → High Availability

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



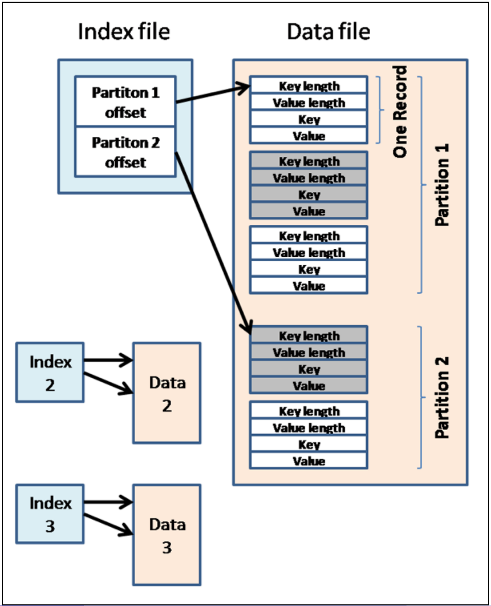
Shuffle and Sort: the Map Side

- **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 (WHY?)

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



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 output 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
  - All map outputs are sorted maintaining their sort ordering, in rounds

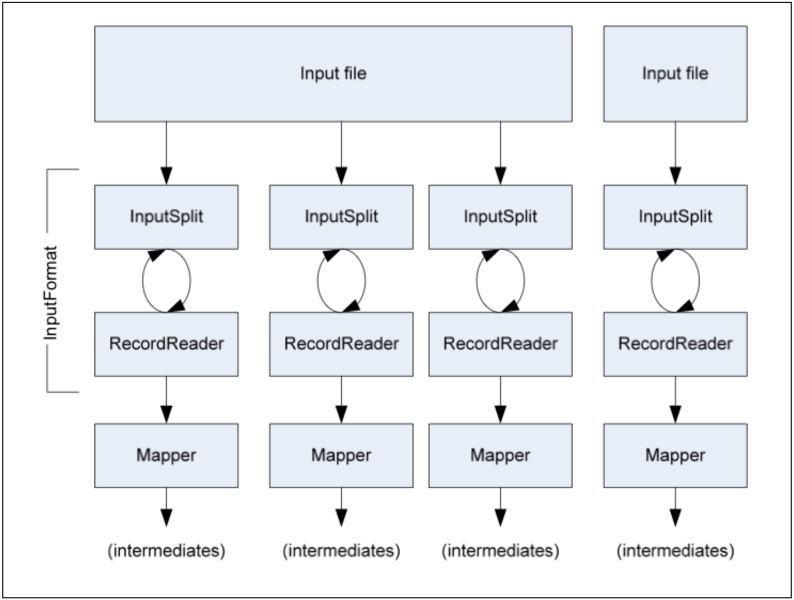
MapReduce Types

- **Recall: Input / output to mappers and reducers**
  - map: $(k1, v1) \to [(k2, v2)]$
  - reduce: $(k2, [v2]) \to [(k3, v3)]$
- **In Hadoop, a mapper is created as follows:**
  - `void map(K1 key, V1 value, Context context)`
- **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

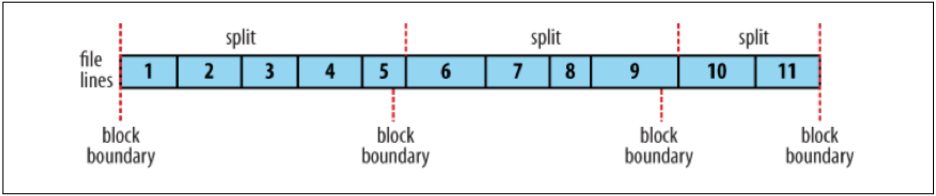


Reading Data

- **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 `Context` object**
  - `Context.write()` 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:**
  `JobConf.setOutputValueGroupingComparator()`

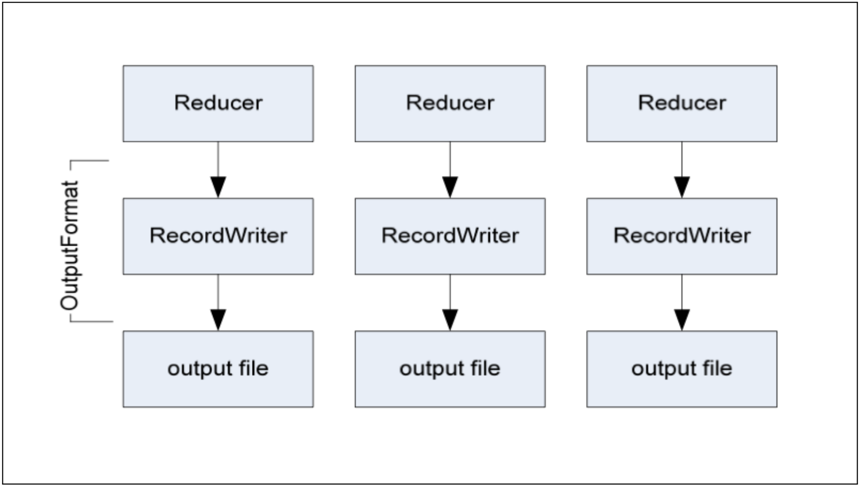
Partitioner

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



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 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 [2]

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 a simple **checksum** mechanism

- **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 in addition to detection

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

- **Unsplittable 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 [2]

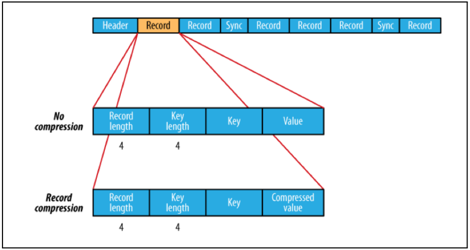
Serialization

- **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



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 2012 specification**:
  - Dual socket, Two exacore
  - 128 GB ECC RAM
  - 8×1 TB disks[5]
  - {1,10} Gigabit Ethernet

[5] 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 collocated
  - Large cluster: requires more RAM at the NameNode

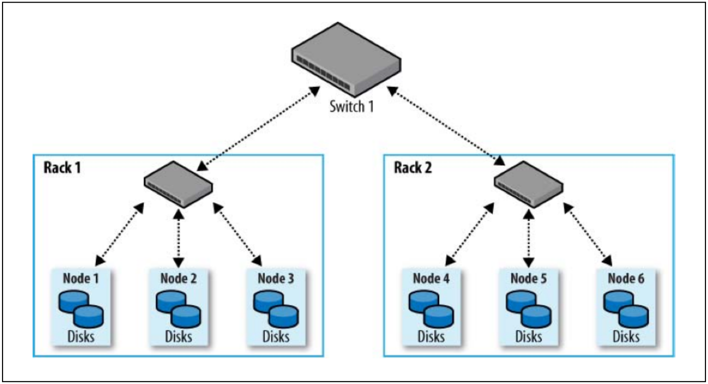
Cluster Specification

- **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

- **What's the role of Java?**
  - Recent releases (Java6) implement some optimization to eliminate large pointer overhead

$\rightarrow$ A cluster of 64-bit machines has no downside

Network Topology



Network Topology

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

- **Typical configuration**
  - 30-40 servers per rack
  - 10 GB switch TOR
  - Core switch or router with 10GB 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**
  - Hundreds of parameters!!
  - 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

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

Table: Hadoop Configuration Files

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 $\times$ 200MB
    - Child JVM reduce task: 2 $\times$ 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 ($\rightarrow$ 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, boto, ...}
  - 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 Chapter 9 [2]

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

References I

[1] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler.
The hadoop distributed file system.
In *Proc. of the 26th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*. IEEE, 2010.

[2] Tom White.
Hadoop, The Definitive Guide.
O'Reilly, Yahoo, 2010.