Hadoop

Chapter-content:

2020-2021

- MapReduce
- Hadoop's MapReduce
- HDFS

Table of content

2020-2021

Hadoop

- MapReduce
- Hadoop's MapReduce
- HDFS

A Silicon Valley star...

Google's CTO
"the" star on massively distributed systems
that allowed data processing/ML to scale.

Jeff dean facts:

- *Jeff Dean wrote an $O(n^2)$ algorithm once. It was for the Traveling Salesman Problem.*
- *You don't explain your work to Jeff Dean. Jeff Dean explains your work to you.*
- *Jeff Dean compiles and runs his code before submitting, but only to check for compiler and CPU bugs.*
- *gcc -O4 sends your code to Jeff Dean for a complete rewrite.*
- *Jeff Dean invented Bigtable so that he would have a place to send his weekly snippets*
- *When Jeff gives a seminar at Stanford, it's so crowded Don Knuth has to sit on the floor. (True)*
- *Jeff Dean got promoted to level 11 in a system where max level is 10. (True.)*

Latency numbers every programmer should know (J.Dean)

Latency numbers (2012)

```
L1 cache reference ......................... 0.5 ns
Branch mispredict ............................ 5 ns
L2 cache reference ........................... 7 ns
Mutex lock/unlock ........................... 25 ns
Main memory reference ...................... 100 ns
Compress 1K bytes with Zippy ............. 3,000 ns  =   3 µs
Send 2K bytes over 1 Gbps network ....... 20,000 ns  =  20 µs
SSD random read ........................ 150,000 ns  = 150 µs
Read 1 MB sequentially from memory ..... 250,000 ns  = 250 µs
Round trip within same datacenter ...... 500,000 ns  = 0.5 ms
Read 1 MB sequentially from SSD* ..... 1,000,000 ns  =   1 ms
Disk seek ........................... 10,000,000 ns  =  10 ms
Read 1 MB sequentially from disk .... 20,000,000 ns  =  20 ms
Send packet CA->Netherlands->CA .... 150,000,000 ns  = 150 ms
```

2019: for a 4kB block:

Latency numbers (2019)

```
Fast NVMe (Optane) ........................... 7 µs
Fast NVMe (Z-SSD) ........................... 12 µs
Round trip TCP packet on 10Gb Ethernet ... 20-50 µs
NVMe Flash SSD .............................. 80 µs
```

Distributed analytical processing of data: MapReduce and Hadoop

Goal: process huge volumes (TB, PB) of data to extract information.

Mostly about scalability and fault tolerance in a distributed setting. One does not care about:

- fast access to data (such jobs take a lot of time anyway).
- updating data (no index)

Applications can for instance be "bash"-like treatments:

- distributing a grep
- computing occurrences of words in files
- sorting data
- analyzing logs
- building an index ...

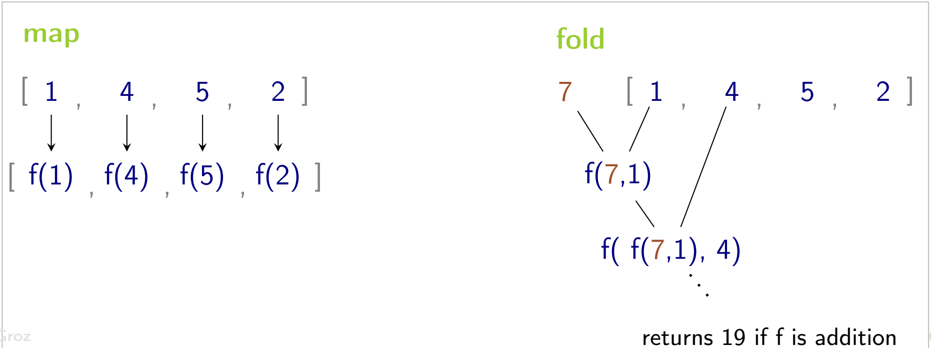
MapReduce: functional programming reminders

map : has type $(\alpha \to \beta) \to \alpha\ \mathit{list} \to \beta\ \mathit{list}$

takes as input a function and item list, and applies function to each item. Returns the list of transformed items (in input order).

fold : has type $(\alpha \to \beta \to \alpha) \to \alpha \to \beta\ \mathit{list} \to \alpha$

takes an agregate function f, an initial value for accumulator, and a list of input items, to which the function is iteratively applied together with the accumulator. The final value of accumulator is returned.



Properties of map and fold

- Input collections are not updated
- **map** transformations are independant (stateless): hence can be evaluated in parallel
- In **fold**, input order does not matter if agregation is associative and commutative

These are the undelying principles of *MapReduce*

map and fold in OCaml:

```
let words = [ "It'"; "s";  "a"; "beautifull"; "day" ]
(* We want to compute the total length of strings in words *)

let lengths = List.map (fun s -> String.length s) words
(*lengths=[3;1;1;10;3]*)

let total = List.fold_left (fun acc i -> i + acc) 0 lengths
(*total=18*)
```

MapReduce: a model for distributed computation

Model was promoted by Jeffrey Dean et Sanjay Ghemawat in article :
MapReduce: Simplified Data Processing on Large Clusters (OSDI, 2004)

Nothing really new (Distributed DB, functional programming) but the article:

- shows it's useful for everyday tasks at Google
- describes Google's implementation.

this turned MapReduce into a very popular framework, and triggered huge open-source project: Hadoop's MapReduce.

Setting (GFS, HDFS):

- large number of (commodity hardware, as opposed to special-purpose high end machines) in a datacenter.
- each has chacune dispose localement d'une partie des données
- 1 nœud joue le rôle d'orchestrateur (master) les autres traitent les données (workers).

Programmer specifies MapReduce transformation through 2 functions:

`map :  (K1, V1)` → `list (K2, V2)`

`reduce :  (K2, list(V2))` → `list (K3, V3)` (like fold from func. pr.)

MapReduce: phases

```
map : (K1, V1) → list (K2, V2)
reduce : (K2, list(V2)) → list (K3, V3)
```

(la fonction fold)

A MapReduce transformation is split into 2-3 successive steps:

1. **Map:** each worker (Mapper) computes `map` transformation on its data
2. **Shuffle:** redistributes results from Mappers to Reducer (nodes)
3. **Reduce:** each worker (Reducer) aggregates data it received using `reduce`

Input and Output of each step are (key, value) pairs in files. Map and Reduce are *local* tasks. Only Shuffle step transfers data through network.

"Successive" ... not quite: the Shuffle part of Reduce can start as soon as one Mapper is done (set value of `mapreduce.job.reduce.slowstart.completedmaps` in Hadoop); no need to wait for all Mappers. But *reduce* function only starts once all Map tasks are over.

Terminology:

Job a (global) MapReduce step

Task (local) computation map (map task) or reduce (reduce task) at a node onn one chunk of data.

MapReduce: Map phase

(Before the job: `InputFormat` interface splits input file(s), assign each split to Mapper. Provides `RecordReader`)

Map: launches one mapper task per split. Each worker applies`map` transformation independently to each (ley, value) pair he was assigned by the framework. Intermediate (key, value) pairs thus produced are writtent in local file. (Map-only jobs produce 1 file per mapper, but HDFS).

Sample input (key, value): (offset, ligne)

Rk: for each input, Map can produce one or several pairs in output. Key or Value is not always used.

Partition: by default, hashcode modulo number of Reducers
Sorts data (SortComparator) before writing to disk.

MapReduce: Reduce phase

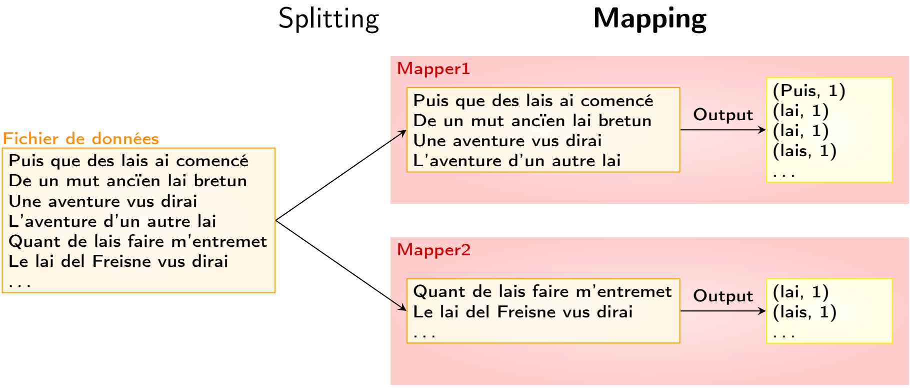
Shuffle: and groups pairs produced by Map *according to key*: sends to the same node (Reducer) all pairs having same key. The Reducer deduces (key, list of values).
Shuffle performed at the end of Map. Reads temporary pairs from disk ⇒ generates a lot of network traffic.

Sort: groups (sorts) by the key all the pairs he was assigned by Partitioner during shuffle. If we want to refine sort order, can specify secondary sort with Shuffle performed at the end of Map. Reads temporary pairs from disk ⇒ generates a lot of network traffic.

Reduce: Each worker computes the `reduce` function independently for each of its (key, value list) pair. Successive pairs retrieved through iterator. Each reducer processes keys in (increasing) order, which is how we can sort result. Result generally smaller (aggregate). Result is one file per Reducer (named part-x-yyyy where x is m (map) or r (reduce), and xxxx is task number). Result stored in HDFS. Can be used by user or by another Map phase.

The standard exemple: word count (Map)

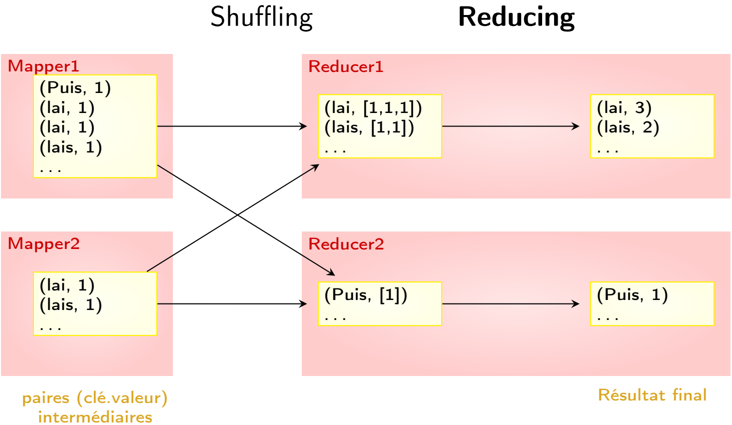
```
# pseudocode
map(InputKey file, InputValue content) {
  for each word in content {
    Output(word, 1);
  }
}
```



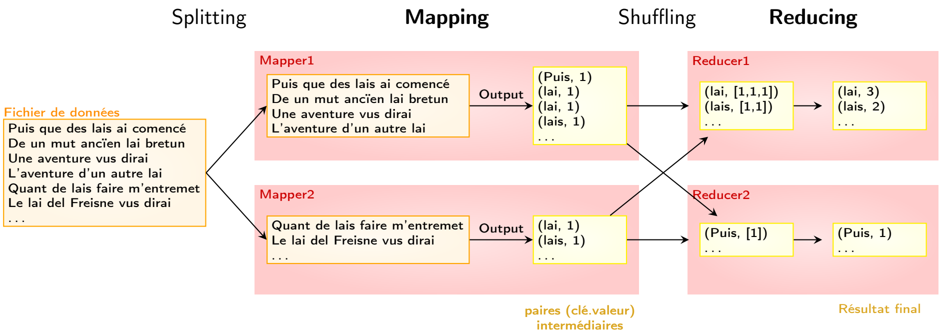
paires (clé.valeur) intermédiaires

The standard exemple: word count (Reduce)

```
reduce(OutputKey word, InputermediateValue list ones) {
  int total = 0;
  for i in ones {
    total += i;
  }
  return total;
}
```



The standard exemple: word count (bird's view)



Parallelism, computing costs

Each node works in parallel during Map and Reduce phases.

Stragglers can slow all tasks. Partly tackled through speculative execution .

Shuffle is costly. Cost model tries to minimize network traffic, hence shuffle (so cost model diverges from RDBMS).

One rather obvious Optimization : see *Combiner* below. Harder: many parameters to tune.

Relational DBMS vs MapReduce

	SGBD	MapReduce
Taille des données	GB	PB
Acccès	Interactif et batch	batch
Transaction	ACID	Non
Structure	Schema-on-write	Schema-on-read
Integrity	élevée	Faible
Passage à l'échelle	Nonlinéaire	Linéaire

[*Hadoop, the definitive guide*, Tom White]

Parallelism to be used when a single machine cannot efficiently process the data ($> 1TB$).

Computation must be formulated as sequence of Map and Reduce stages.

MapReduce inadequate in many cases:

- joins (too many data transfers ⇒ denormalize),
- multiple successive iterations (data stored on disk, at least for early versions),
- updating data (immutable)
- real-time analysis (startup, slow)
- multiple small files (metadata)

Fault Tolerance

- Data is **replicated** (default: 3 copies of same data)
- Task failure or lag. For failed task restarts task on some available node. Similarly, if worker lags (straggler, defined by threshold on progress score), Hadoop starts a "backup" computation on another node with empty slot. This is *speculative execution*. Priority to keep empty slot busy: failed task, then stragglers (for maps, preferably those whose data is already on the node.
- Redundancy also for $\simeq$orchestrators (Hadoop≥ 2):

  1. (HDFS) namenode: a standby becomes active
  2. (YARN) If Ressource manager fails: a standby becomes active.
  3. (YARN) If application master fails: a new one is created by ressource manager, and recovers completed jobs.

https://www.usenix.org/legacy/event/osdi08/tech/full_papers/zaharia/zaharia_html/index.html

Hadoop's MapReduce: other operations...

- `Combiner` : to reduce data transfer during shuffle, we can use a `Combiner` to apply some (associative and commutative) `reduce` function locally on each Mapper before the shuffle. Often we reuse the `Reducer` as a Combiner. Cannot control how many times Combiner is executed: possibly 0,1 or several times.
- `Partitioner` : by default, `HashPartitioner` is hashcode of key modulo number of Reducers. Can be replaced with a custom Partitioner to control how keys are assigned to reducers (for instance, grouping based on part of a composite key). `TotalOrderPartitioner` allows to use range partitioning. Ranges are defined automatically to balance load, based on sampling.
- `Sorting` : MapReduce sorts keys both when grouping output at Mappers, and on Reducer to gather keys received. `SortComparator` (on both mapper and reducer) controls how the list of values are sorted for each key. `GroupComparator` defines which keys on the reducer are merged in the same list.

Table of content

2020-2021

Hadoop

- MapReduce
- Hadoop's MapReduce
- HDFS

hadoop

Original objective: make *Lucene* scalable (index the web).

Distributed framework for data storage and processing. Includes modules:

- Hadoop Common: common utilities that support the other Hadoop modules.
- *HDFS*: distributed file system (high-throughput, reliable access to data)
  (open-source implementation of *GFS*)
- *MapReduce* (parallel processing)
- from Hadoop 2 (2012), *YARN*: framework for job scheduling and cluster resource management.
  (*Hadoop MapReduce* now based on *YARN*)
- from Hadoop 3 (2020), *Ozone*: object store for Hadoop (billions small files).

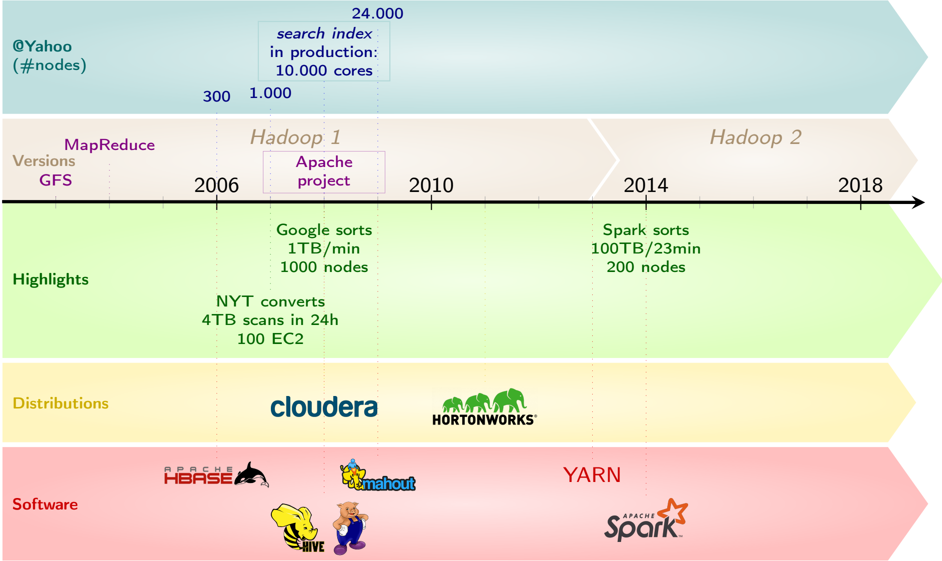
Driving ideas:

- " distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines"
- Manage failures in the framework instead of at hardware level.

Huge software stack built on top of Hadoop:
Hive, HBase, Mahout, Pig, Spark, Tez, Impala...
Like Hadoop, most developed under the *ASF (Apache)*.

https://hadoop.apache.org/

Hadoop: timeline



Values are rounded. Dates debatable (multiple versions : alpha, beta. . .)

https://en.wikipedia.org/wiki/Apache_Hadoop#Timeline

Hadoop's MapReduce Java API

librairie org.apache.hadoop.mapreduce

- Class `Mapper<IK, IV, OK, OV>`. Extend by defining

  `public void map(IK ik, IV iv, Context ctx)`.

- Class `Reducer<IK, IV, OK, OV>`. Extend by defining

  `public void reduce(IK ik, Iterable<IV> iv, Context ctx)`.

- Combiner also extends

  `org.apache.hadoop.mapreduce.Reducer<IK,IV,OK, OV>`

  (hence also implements `reduce` method)

- Object `Context`. Allows input and output from the tasks (supplied to mapper and reducer). Contains some configuration options... Mappers and Reducers produce key/value pairs with its method:

  `write(OK ok, OV ov)`.

- `Job` object contains instructions (configuration and launching the job).

Hadoop written in Java, but with `Hadoop Streaming` API, Mapper and Reducer can be any executable or script, so arbitrary languages can be used.

Hadoop's MapReduce Java API (2)

- Actually, Hadoop defines several Context classes:
  - MapContext<IK,IV,OK,OV> has `getInputSplit()`,
  - ReduceContext<IK,IV,OK,OV> has `nextKey()`, `getValues()`,
  - ⇒ both inherit from TaskInputOutputContext: `getCurrentKey`, `getCurrentValue`, `nextKeyValue`, `write`...
  - ⇒ both also inherit from JobContext ...
  - ⇒ both also inherit progress
- Classes for IK, OK, IV, OV : `IntWritable`, `DoubleWritable`,`Text`...
  Those are *efficiently* serializables (unlike `java.lang.Serializable`).
  To save space, datatype not stored in the serialization (it's already known).

  https://hadoop.apache.org/docs/r2.7.5/api/src-html/org/apache/hadoop/io/IntWritable.html

  User can define own types. Constraints are :
  - `Writable` objects implement : `readFields(Datainput in)`, `write(Dataoutput out)`
  - Keys must extend interface WritableComparable:

    https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/io/WritableComparable.html

Hadoop's MapReduce Java API (3)

```java
public static class TokenizerMapper
  extends Mapper<Object, Text, Text, IntWritable>{
  ...
  public void map(Object key, Text value, Context context) {
    ... context.write(cle, valeur);
  }
}

public static class IntSumReducer
  extends Reducer<Text,IntWritable,Text,IntWritable> {
  ...
  public void reduce(Text key, Iterable<IntWritable> values, Context context) {
     ...context.write(cle, resultat);
  }
}

...
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
...
```

from hadoop.apache.org MapReduce Tutorial

Global variables in Hadoop's MapReduce

- Counters. Both built-in (bytes read/written, number of records processed by task) and user-defined counters. Counter is named with ENUM type (groups together counters). Mapper/Reducer update counter with :
  `context.getCounter(MY_COUNTER).increment(my_step);`
  Job can read counters:
  `job.getCounters().findCounter(MY_COUNTER).getValue()`
- Add a file through the Job.
- Add an environment variable through the Job configuration.

```
// In driver program, after :
/* Configuration conf = getConf();
Job job = new Job(conf,"job name") */

conf.set("param","value")
```

```
// In Mapper/Reducer:
context.getConfiguration().get("param")
```

Hadoop Streaming

```
mapred streaming \
  -files mapper.py,reducer.py \
  -input inputdir_or_file \
  -output outputdir \
  -inputformat JavaClassName \
  -mapper mapper.py \
  -reducer reducer.py
```

to load files on the nodemanagers

executable, of JavaClassName (identify by default)

Also:

```
  -combiner combiner.py
  -partitioner JavaClassName
  -outputformat JavaClassName
  -numReduceTasks
  -cmdenv MY_Param=param_value

  -D <property>=<value>
```

global variables

job configuration parameters

For instance:

```
  -D mapreduce.job.reduces=0
  -D stream.map.input.field.separator=,
  -D stream.map.output.field.separator=.
  -D stream.num.map.output.key.fields=4
```

defines the separator to be used

prefix up to the 4^{th} "." in a line w
be the key

https://hadoop.apache.org/docs/stable/hadoop-streaming/HadoopStreaming.html#How_Streaming_Works

In-Mapper combiner

In Mapper and Reducer class, user can define methods

- `setup()`, called once at the beginning of the task.
- `cleanup()` called once at the end of the task

This is as opposed to the map function which is called for every line in input split, and reduce which is run once for each key.

This can be used to aggregate pairs possibly more efficiently than with a traditional combiner:

- Create a hash-table when initializing the Mapper .
- Insert (key, values) produced by map into this associative array instead of producing them on-the-fly.
- Process the associative array to produce the (key, values) output we want the combiner to output.

But be careful that it fits in main memory ! In-mapper variables should be small.

Sorting

To sort, we exploit the sorting steps from Shuffle.

We are sorting `WritableComparable` Keys:

```
// Define my_class.class that extends WritableComparator
// Then:
job.setSortComparatorClass(my_class.class)
```

Identity reducer (Reducer) can be used; the reduce function has no effect, but data is sorted through the shuffle.

Sort comparator is used:

- once on Mappers
  (to prepare partitions from current mapper for all reducers).
- once on Reducers
  (to merge partitions from all mappers for current reducer).

Secondary sort

```
// Define my_class.class that extends WritableComparator
// Then:
job.setGroupingComparatorClass(my_class2.class)
```

Group comparator defines which keys will be sent to the same reduce call.

Therefore we can define a composite key that contains not only the key intended for grouping but also the data required for sorting in order to exploit the sorting, then rely on group comparator to group together keys.

https://hadoop.apache.org/docs/r3.2.2/api/org/apache/hadoop/mapreduce/Reducer.html

http://blog.ditullio.fr/2015/12/28/hadoop-basics-secondary-sort-in-mapreduce/

Sorting across Reducers

Define a custom partitioner. Best solution is a `TotalOrderPartitioner` which samples the input splits to balance load correctly.

https://hadoop.apache.org/docs/r3.0.3/api/org/apache/hadoop/mapreduce/lib/partition/TotalOrderPartitioner.html

Table of content

2020-2021

Hadoop

- MapReduce
- Hadoop's MapReduce
- HDFS

HDFS: Hadoop Distributed FileSystem

a distributed file system:

- designed to manage huge data (TB, PB)
- while optimizing sequential reads
  No random reads within a block: designed to optimize *throughput* for sequential reads, not *latency*.
- immutable files (support append, truncate): *Write-once Read-many*
- distributed on "commodity hardware" clusters (opposed to higher-end servers with built-in parallelization)
  (need for fault tolerance $\Rightarrow$ replication)

Hadoop MapReduce relies on HDFS to store/exchange data.

Not fully POSIX compliant (append-only, buffering writes, permissions. . .)

Hadoop (hdfs, mapreduce. . .) written in Java, so requires a JVM:

- Java 8 (11 runtime only) for Hadoop3
- Java 7 or 8 for Hadoop 2 ($\geq$ 2.7)

Interacting with HDFS

Access HDFS through

- `FS shell` CLI
- `FileSystem` Java API (also C wrapper)
- NFS gateway to mount as part of local filesystem...

```
bin/hadoop fs -appendToFile  localfile1  localfile2  /user/hadoop/hadoopfile
bin/hadoop fs -cat file:///file3    hdfs://nn1.example.com/file1

# if data on hdfs, can use hdfs dfs
hdfs dfs -cat /user/hadoop/file4
```

https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html#Accessibility

https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html

HDFS: Hadoop Distributed FileSystem

Master-slave architecture:

- NameNode: (1) manages filesystem (tree+metadata) (2) manages block placement, creation, replication.

  1: on disk+RAM, replicated. 2: in memory. Rebuilt if needed.

  User accesses data transparently through POSIX-style interface.

  From Hadoop2 we can federate NameNodes. Each Namenode is responsible (independently from others) for its namespace and the corresponding block pool. But each NameNode communicates with all DataNodes (Datanodes are common storage for all NameNodes they may contain blocks from several namespace).

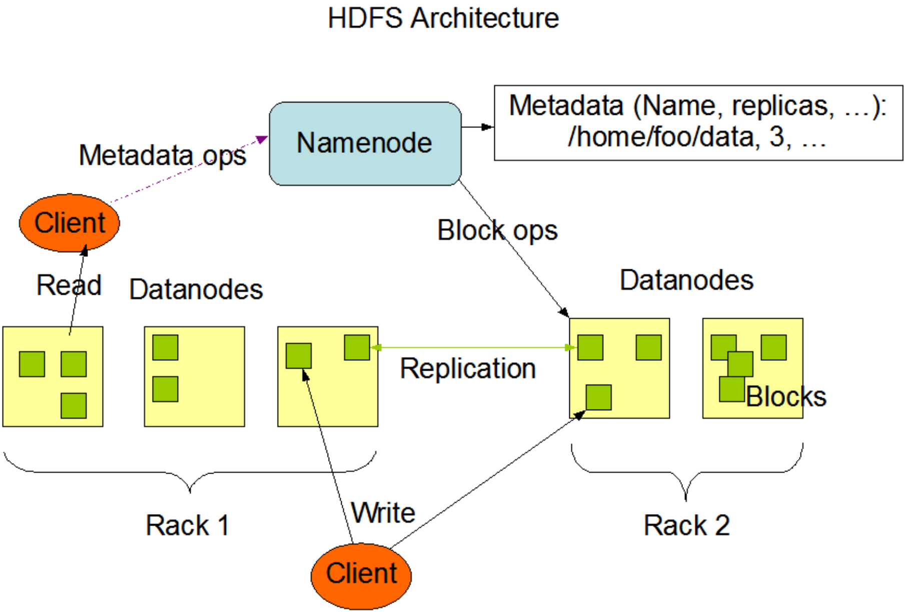
- DataNode: contains blocks of data.

  Read from disk in general, but can be cached in memory.

DataNodes register to the NameNode, send Heartbeats.

NameNode was *Single point of failure* till Hadoop 1 (and still default).
(solution : copy on a standby machine)

HDFS architecture



HDFS: Hadoop Distributed FileSystem

HDFS splits data into large blocks

(default: 128MB. Compare to: $\simeq$ 512B for disk, 2kB for OS, 8kB for DBMS)

When HDFS splits input file, each chunk= 1 HDFS block.

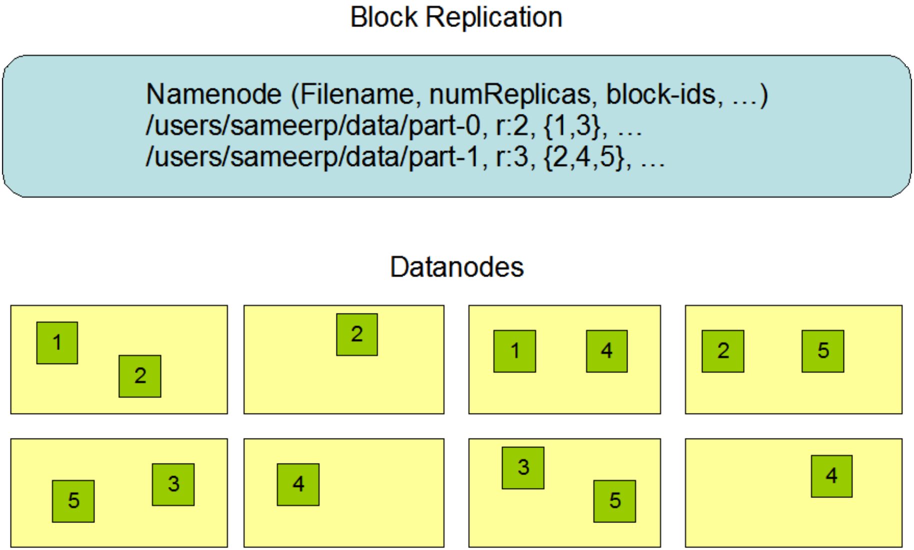
Blocks distributed among (data)nodes. (3 copies by default).

HDFS (and more generally Hadoop components) placement is rack-aware. To optimize placement of copies and queries and to optimize queries by using data locality (same node$<$rack$<$datacenter).

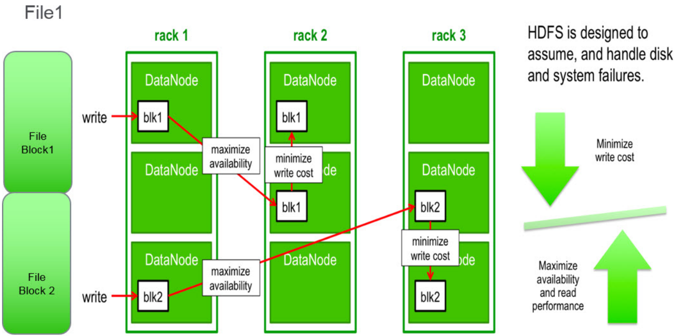
Map writes output on local disk, not HDFS (no replica), Reduce on HDFS.

http://hadoop.apache.org/docs/r3.0.1/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

HDFS replication



HDFS replication: block placement



NameNode Meta Info

Block	Node
File1_Block1	DN1, R1 DN2, R2 DN1, R2
File1_Block2	DN3, R1 DN2, R3 DN3, R3

- File size 200 MB
- HDFS block size 100 MB

https://community.cloudera.com/t5/Community-Articles/Understanding-basics-of-HDFS-and-YARN/ta-p/248860

HDFS architecture

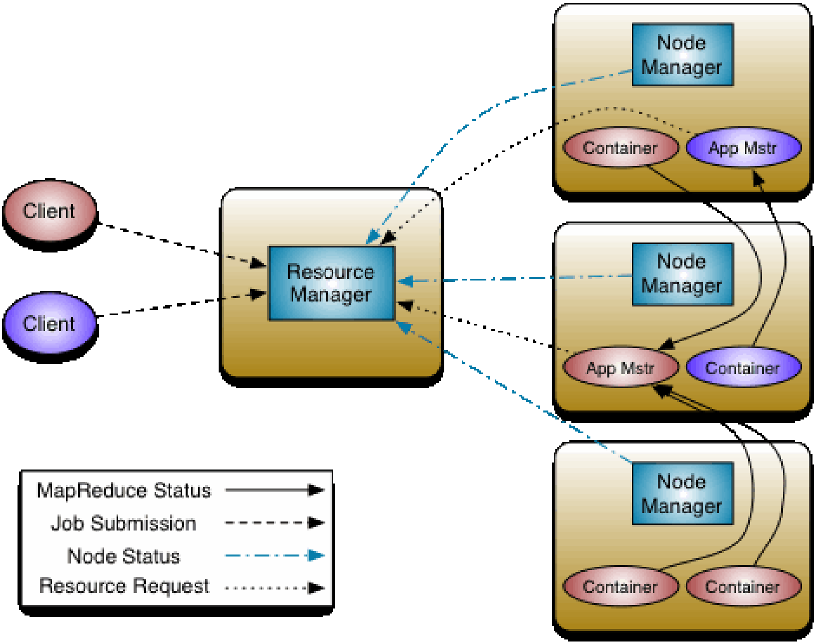
```java
package org.apache.hadoop.hdfs.server.blockmanagement;
[...]
/**
 * Manage datanodes, include decommission and other activities.
 */
@InterfaceAudience.Private
@InterfaceStability.Evolving
public class DatanodeManager {
  static final Logger LOG = LoggerFactory.getLogger(DatanodeManager.class);

  private final Namesystem namesystem;
  private final BlockManager blockManager;
  private final DatanodeAdminManager datanodeAdminManager;
  private final HeartbeatManager heartbeatManager;
  private final FSClusterStats fsClusterStats;

  private volatile long heartbeatIntervalSeconds;
  private volatile int heartbeatRecheckInterval;
  /** Stores the datanode -> block map.  ... **/
  private final Map<String, DatanodeDescriptor> datanodeMap
      = new HashMap<>();
```

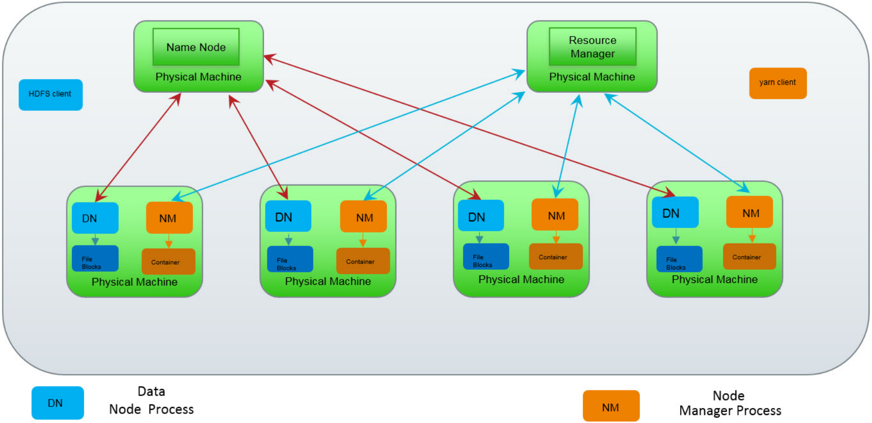
https://github.com/apache/hadoop/blob/trunk/hadoop-hdfs-project/hadoop-hdfs/src/main/java/org/apache/hadoop/hdfs/server/blockmanagement/DatanodeManager.java#L75

YARN



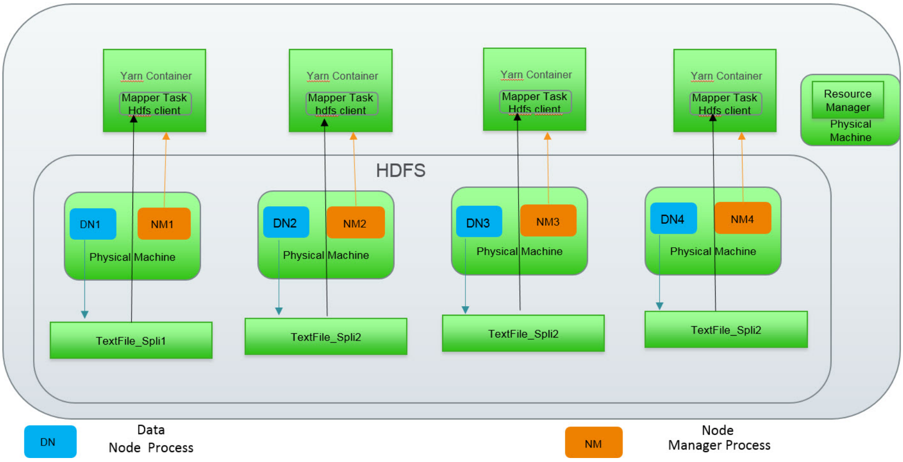
Client
Client
Resource Manager
Node Manager
Container
App Mstr
Node Manager
App Mstr
Container
Node Manager
Container
Container
MapReduce Status
Job Submission
Node Status
Resource Request

YARN on HDFS



https://community.cloudera.com/t5/Community-Articles/Understanding-basics-of-HDFS-and-YARN/ta-p/248860

MapReduce with YARN on HDFS



https://community.cloudera.com/t5/Community-Articles/Understanding-basics-of-HDFS-and-YARN/ta-p/248860

YARN

Optimization: executes a task on machine that has the data.
Push the query to the data. . . standard practice. A case of "data locality" : moving data is expensive, so minimize transfers.

Detail:

Input Split fragment corresponds to mapreduce task.

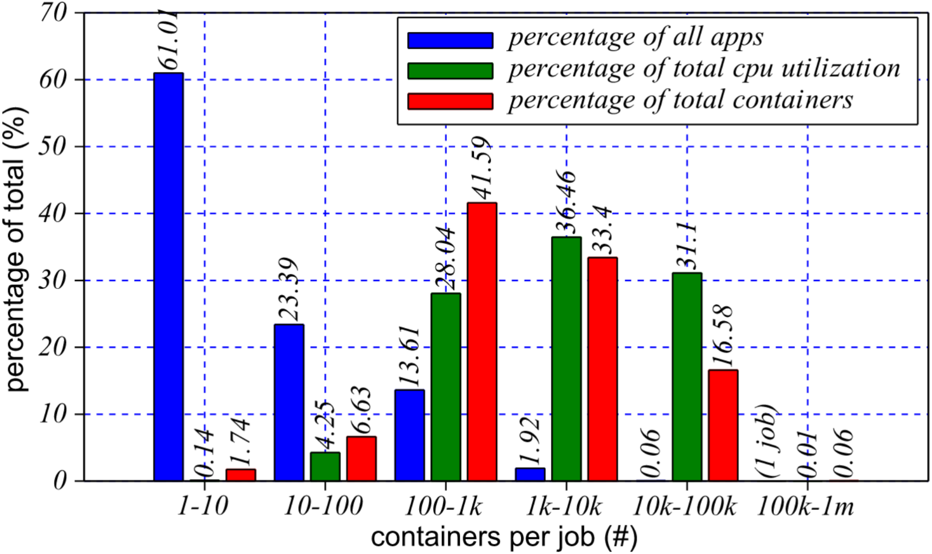
(logical partition of recors: 1 split$\leftrightarrow$1 map task)

Block HDFS. Default: 128MB data (max).

Ideally, both should match. In practice, record may be split over several blocks? Transparent for user: Map task performs remote read on other block.

Hadoop in the real world...

Yahoo, 2013



MapReduce vs SGBD

A polemic post by D.DeWitt and M.StoneBreaker (2008):

"MapReduce may be a good idea for writing certain types of general-purpose computations, but to the database community, it is:

1. *A giant step backward in the programming paradigm for large-scale data intensive applications*
2. *A sub-optimal implementation, in that it uses brute force instead of indexing*
3. *Not novel at all – it represents a specific implementation of well known techniques developed nearly 25 years ago*
4. *Missing most of the features that are routinely included in current DBMS*
5. *Incompatible with all of the tools DBMS users have come to depend on*

... "

https://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html

Then a research article (2009) comparing performance, less polemic:

http://www.science.smith.edu/dftwiki/images/6/6a/ComparisonOfApproachesToLargeScaleDataAnalysis.pdf

But many gaps have been bridged since.

In-memory systems in-memory, lazy query evaluation, vectorization and query optimization, transactions, interfaces (SQL support, operations, connectors)

MapReduce vs DBMS: 10+ years later

1. A giant step backward in the programming paradigm for large-scale data intensive applications

   Many NoSQL software have a SQL (or similar high-level) interface (Hive, Spark).

2. A sub-optimal implementation, in that it uses brute force instead of indexing
   Not meant for the same usage (that's counterpart to fault-tolerance). Some tools offer indexes (Hive. . .).

4. Missing most of the features that are routinely included in current DBMS:

Bulk-loader :	*have been implemented (Hive, Spark)*
Indexing :	*rather limited (Hive) but that's not the approach*
Transactions :	*have appeared in cloud/containerized nosql systems (generall*
Integrity :	*denormalization, schemaless (so limited)*

5. Incompatible with all of the tools DBMS users have come to depend on

   Not anymore (many connectors and new tools)

References. . .

- GFS, HDFS

  https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

  https://queue.acm.org/detail.cfm?id=1594206

  https://storageconference.us/2010/Papers/MSST/Shvachko.pdf

- MapReduce:

  https://static.googleusercontent.com/media/research.google.com/en//archive/papers/mapreduce-sigmetrics09-tutorial.pdf

  https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

- YARN

  http://web.eecs.umich.edu/~mosharaf/Readings/YARN.pdf

- Hadoop:

  https://hadoop.apache.org/docs/stable/index.html

  https://fr.hortonworks.com/tutorials/

  https://www.cloudera.com/more/training/library/tutorials.html

  http://blog.ditullio.fr/category/hadoop-basics/

  Hadoop The definitive guide, Tom White (B.U.) Ce cours traite chap1, chap2 et une partie de chap3.