Contents

Chapter 1

What the crap is this?

We will come back to this later. Bcoz we don't know what to write?

1.1 YYY??

1.1.1 Compare: one of the first mistaken attitudes!

A typical transaction system lives inside the range of some gigabytes data, in random access. It runs inside an application server, acceses and takes control over a relational database allocated in a different server, transporting data in and out of it. Interacts with online clients, keeping a reduced size of data in transport, with a shared and limited bandwith, the operations are most of all continuous reading over small sets of data, combined with some maintenance & CRUD operations. Bigger data processing is done in batch, but the architecture is the same.

1.1.2 But a different scenario ?

What happens if we need to process 1 petabyte/week ?, let's say a much lesser size, 1 terabyte/day, less, 100 gigabytes/day Under the traditional schema, it would be like moving an elephant over a string thousands of times, the time-windows would not allow to work with realtime information, but always trying to keep up and losing the race.

1.1.3 Want problems ?

The network would saturate, batch processes would take days to finish information of hours, the harddisk latency access it will transform into an incremental overhead, having a painful impact on overall costs. The traditional approach would mantain the architecture, but changing hardware, sophisticated requirements, each time more expensive, bigger, but over a limited growth architecture. How many times will you change your system's architecture to non-linear scalar solutions to keep up with the race of growing data, ?

1.1.4 The solution is in the focus !

Problem 1: Size of data
Systems handling public data, can have huge processing flows with hundreds of terabytes, public websites have had increased their data up to petabytes !
Strategy 1: Distribute and paralellize the processing
If chewing 200 Tb of data takes 11 days in 1 computer, let divide the whole process into as many machines needed to finish the task in minutes or hours ! If we'll have lot of computers, so be they cheap and easily mantained !, into a simple add & replace node architecture.

Problem 2: Continuous failure and cluster growth
In a big cluster, lot of machines fail everyday, besides the cluster size cannot be fixed and sometimes cannot be planned, it should grow easily at the speed of data.
Strategy 2: High fault tolerance and high scalability
Any design must have an excelent fault-tolerance mechanism, a very flexible maintenance, and its service availability should keep up naturally to its cluster size. So we need a distributed storage, atomic, transactional, with perfect linear scalability.

Problem 3: Exponential overhead in data transport
From data source to its processing location, bad harddisk latency and saturated networks will end up exploting
Strategy 3: Move application logic to data storage It must allow to handle data and processes separately and in a transparent way, applications should take care of their own business application code, and the platform

should automatically deploy it into the datastorage, execute and handle their results,

So... Distribute and Paralellize processing High fault-tolerance mechanism, High scalability Take application processes to data storage

One Solution: Hadoop Distributed Filesystem + Hadoop MapReduce

Anyway, Hadoop is not bullet-proof, actually, it's a very specific solution for some big-data scenarios, with needs of realtime processing, and technology choices for random access to data. (See HBase, Hive)

Hadoop is not designed to replace RDBMS, but instead, has been proven to handle -in a much performant way- huge amounts of data, whereas traditional enterprise database clusters, wouldn't work not even close at the same overall costs !

Chapter 2

Architecture

2.1 FileSystem

- Scales linearly over a set of low-budget comodity machines, doubled the amount of machines = reduced the processing time to half
- Tolerates faults at different levels, from network, switches, disks, nodes, readdressing data traffic to other nodes, accomplishing a replica factor
- Flexible scalability, for maintenance tasks, you only have to dis/connect computers into the rack
- Lets you allocate any kind of files and formats, although it has better performance on files bigger than 128mb
- The results of MapReduce jobs are stored in the filesystem
- There's a unique namespace, with an automatic replication schema administered by the master, it's not possible to impact on a certain node in the cluster, whether to allocate files or execute jobs
- The master itself balances the workload over execution plans and status reports from the slave nodes
- Has a mechanism of continuous replication per file, rack-aware, to extend data reliability and data availability warranties
- Has an automatic file checksum with inmediate correction
- Works in a master-slave schema where slaves share nothing between them, they only respond to master requests
- The master coordinates all types of transactions, read/write, replication, restore, manages the tx log and the filesystem namespace
- The slaves only take charge on low level operations over data, read, write,

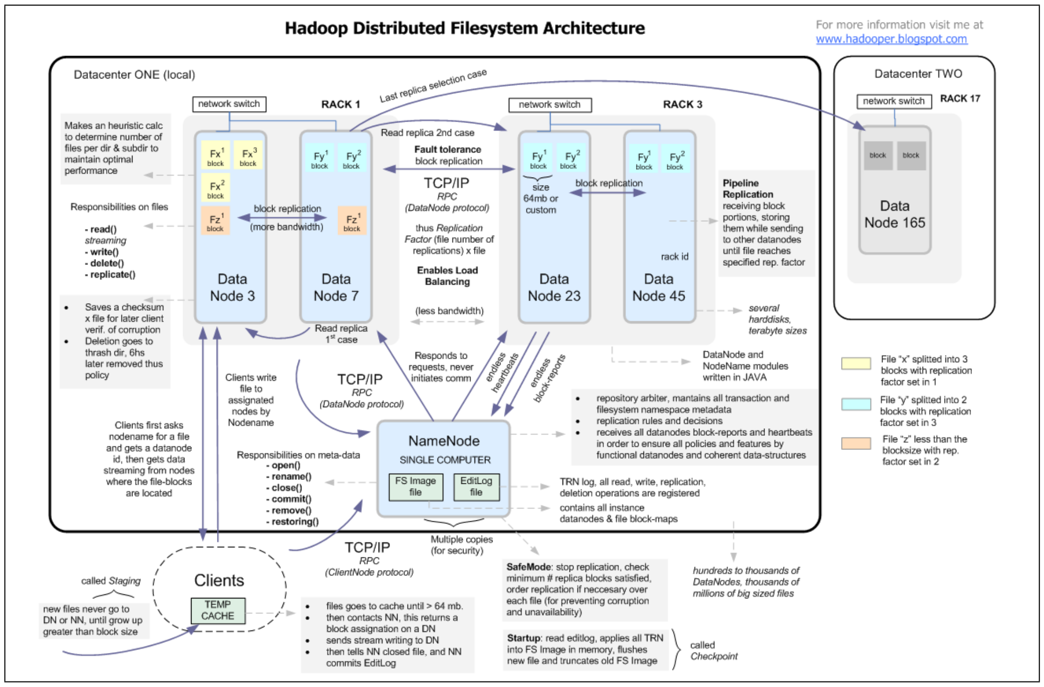

Hadoop Distributed Filesystem Architecture
For more information visit me at www.hadooper.blogspot.com
Datacenter ONE (local)
network switch
RACK 1
Last replica selection case
network switch
RACK 3
Datacenter TWO
network switch
RACK 17
Makes an heuristic calc to determine number of files per dir & subdir to maintain optimal performance
Responsibilities on files
- read() streaming
- write()
- delete()
- replicate()
Saves a checksum x file for later client verif. of corruption
Deletion goes to thrash dir, 6hs later removed thus policy
Fx1 block
Fx3 block
Fx2 block
Fz1 block
Fy1 block
Fy2 block
Fz1 block
block replication
(more bandwidth)
Data Node 3
Data Node 7
Read replica 1st case
Read replica 2nd case
Fault tolerance block replication
TCP/IP
RPC
(DataNode protocol)
thus Replication Factor (file number of replications) x file
Enables Load Balancing
(less bandwidth)
size 64mb or custom
block replication
rack id
Data Node 23
Data Node 45
Pipeline Replication receiving block portions, storing them while sending to other datanodes until file reaches specified rep. factor
several harddisks, terabyte sizes
block
block
Data Node 165
DataNode and NodeName modules written in JAVA
Clients write file to assigned nodes by Nodename
TCP/IP
RPC
(DataNode protocol)
Responds to requests, never initiates comm
endless heartbeats
endless block-reports
Clients first asks nodename for a file and gets a datanode id, then gets data streaming from nodes where the file-blocks are located
Responsibilities on meta-data
- open()
- rename()
- close()
- commit()
- remove()
- restoring()
NameNode
SINGLE COMPUTER
FS Image file
EditLog file
repository arbiter, mantains all transaction and filesystem namespace metadata
replication rules and decisions
receives all datanodes block-reports and heartbeats in order to ensure all policies and features by functional datanodes and coherent data-structures
TRN log, all read, write, replication, deletion operations are registered
contains all instance datanodes & file block-maps
Multiple copies (for security)
TCP/IP
RPC
(ClientNode protocol)
called Staging
new files never go to DN or NN, until grow up greater than block size
Clients
TEMP CACHE
files goes to cache until > 64 mb.
then contacts NN, this returns a block assignation on a DN
sends stream writing to DN
then tells NN closed file, and NN commits EditLog
SafeMode: stop replication, check minimum # replica blocks satisfied, order replication if neccesary over each file (for preventing corruption and unavailability)
Startup: read editlog, applies all TRN into FS Image in memory, flushes new file and truncates old FS Image
called Checkpoint
hundreds to thousands of DataNodes, thousands of millions of big sized files
File "x" splitted into 3 blocks with replication factor set in 1
File "y" splitted into 2 blocks with replication factor set in 3
File "z" less than the blocksize with rep. factor set in 2

Figure 2.1: The picture shows a more practical brief over HDFS responsibilities & execution flows

deletion, transport to-from client

2.2 Processing Model

Main characteristics of Map Reduce

- It's a processing model about dividing and distributing information, in two chained phases: map first, then reduce
- Both phases have as input and output, a key-value pair list,
- The schema allows to define method parameters and own logic for both phases, as well as their own partitioning system and intermediate storage between phases.
- The transactions are handled by a JobTracker daemon, that runs the initial data partitioning and the intermediate data combination, by posting tasks of type Map and type Reduce over the TaskTracker daemons (1-n x computer) of the nodes involved in the cluster, according the data being

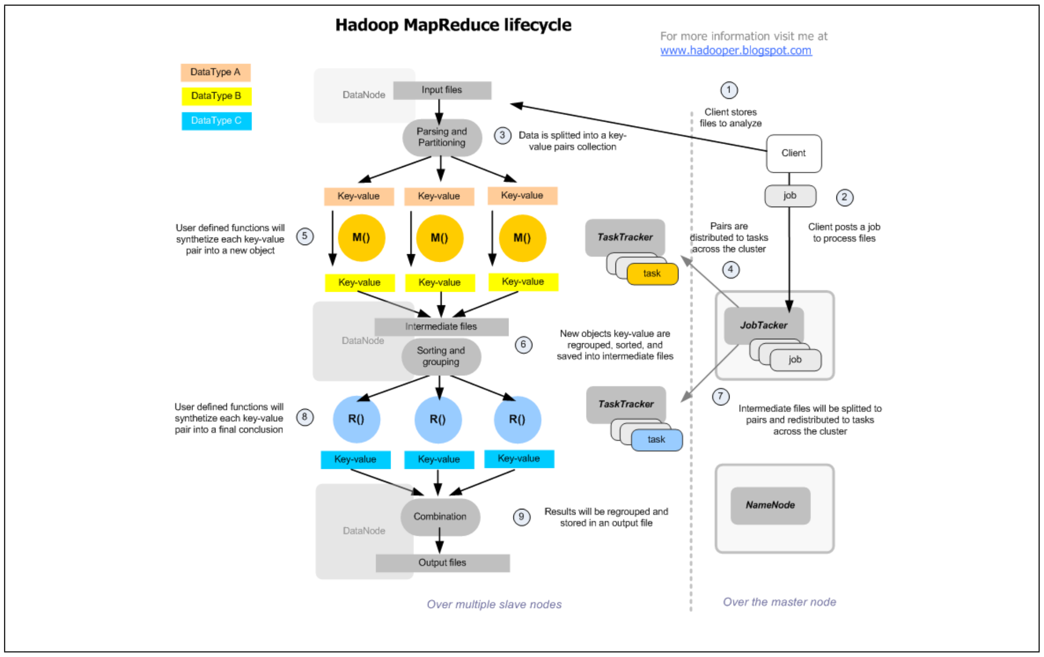
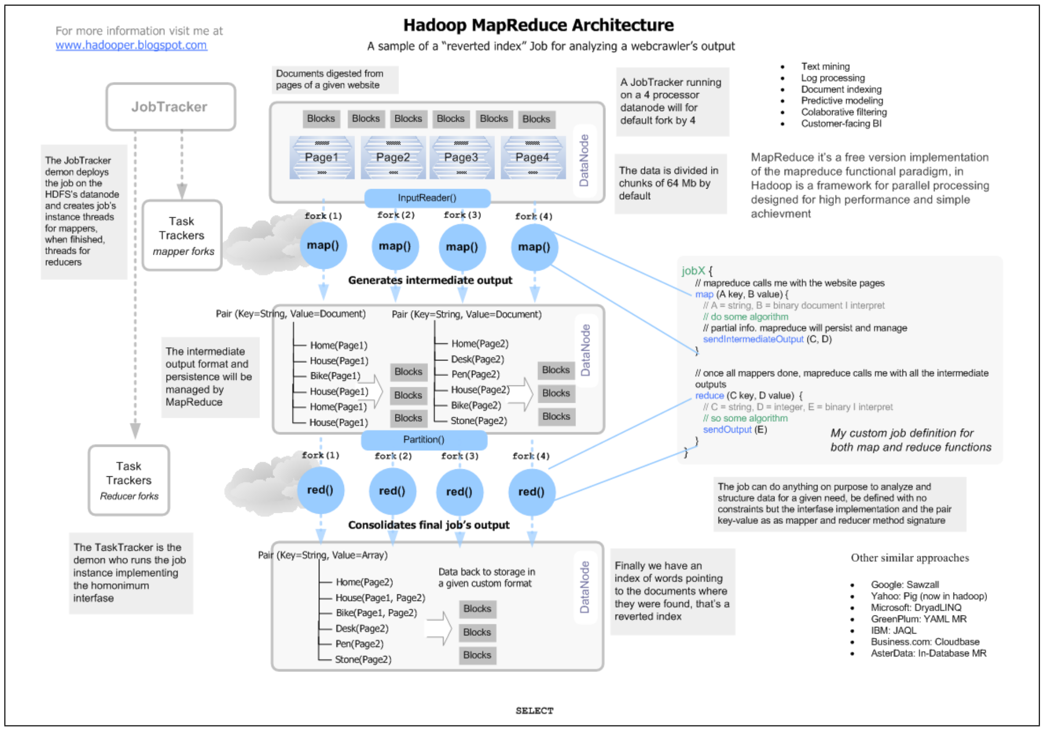

Hadoop MapReduce lifecycle
For more information visit me at
www.hadooper.blogspot.com
DataType A
DataType B
DataType C
DataNode
Input files
1
Client stores files to analyze
Parsing and Partitioning
3
Data is splitted into a key-value pairs collection
Client
job
2
Client posts a job to process files
Key-value
Key-value
Key-value
User defined functions will synthetize each key-value pair into a new object
5
M()
M()
M()
TaskTracker
task
Pairs are distributed to tasks across the cluster
4
Key-value
Key-value
Key-value
JobTacker
job
DataNode
Intermediate files
Sorting and grouping
6
New objects key-value are regrouped, sorted, and saved into intermediate files
User defined functions will synthetize each key-value pair into a final conclusion
8
R()
R()
R()
TaskTracker
task
7
Intermediate files will be splitted to pairs and redistributed to tasks across the cluster
Key-value
Key-value
Key-value
DataNode
Combination
9
Results will be regrouped and stored in an output file
Output files
NameNode
Over multiple slave nodes
Over the master node

Figure 2.2: The picture shows how these methods will interact in phases.

processed

- The Reduce phase only starts when finished the Map, cause after the Map the resulting keys are combined, to distribute a sorted list of key-value pairs between the Reducers, that can be matched at the end of them.
- The process is transactional, those map or reduce tasks not executed, (for data availability issues) will be reattempted a number of times, and then redistributed to other nodes.

- First the files are partitioned in parts that will be distributed to process across the cluster nodes
- Each part is parsed in pairs of Key(sorteable object) - Value(object), that will be the input parameters for the tasks implementing the Map function
- These user defined tasks (map), will read the value object, do something with it, and then build a new key-value list that will be stored by the framework, in intermediate files.
- Once all the map tasks are finished, it means that the whole data to process was completely read, and reordered into this mapreduce model of key-value paris.
- These intermediate key-value results are combined, resulting a new paris


Hadoop MapReduce Architecture
A sample of a "reverted index" Job for analyzing a webcrawler's output
For more information visit me at
www.hadooper.blogspot.com
JobTracker
Task Trackers
mapper forks
Task Trackers
Reducer forks
Documents digested from pages of a given website
Blocks
Page1
Page2
Page3
Page4
DataNode
InputReader()
fork (1)
fork (2)
fork (3)
fork (4)
map()
Generates intermediate output
Pair (Key=String, Value=Document)
Home(Page1)
House(Page1)
Bike(Page1)
House(Page1)
Home(Page1)
House(Page1)
Home(Page2)
Desk(Page2)
Pen(Page2)
House(Page2)
Bike(Page2)
Stone(Page2)
Partition()
red()
Consolidates final job's output
Pair (Key=String, Value=Array)
Home(Page2)
House(Page1, Page2)
Bike(Page1, Page2)
Desk(Page2)
Pen(Page2)
Stone(Page2)
Data back to storage in a given custom format
My custom job definition for both map and reduce functions
Other similar approaches
SELECT

The JobTracker demon deploys the job on the HDFS's datanode and creates job's instance threads for mappers, when fihished, threads for reducers

The intermediate output format and persistence will be managed by MapReduce

The TaskTracker is the demon who runs the job instance implementing the homonimum interfase

A JobTracker running on a 4 processor datanode will for default fork by 4

The data is divided in chunks of 64 Mb by default

- Text mining
- Log processing
- Document indexing
- Predictive modeling
- Colaborative filtering
- Customer-facing BI

MapReduce it's a free version implementation of the mapreduce functional paradigm, in Hadoop is a framework for parallel processing designed for high performance and simple achievment

```
jobX {
  // mapreduce calls me with the website pages
  map (A key, B value) {
    // A = string, B = binary document I interpret
    // do some algorithm
    // partial info. mapreduce will persist and manage
    sendIntermediateOutput (C, D)
  }

  // once all mappers done, mapreduce calls me with all the intermediate
  outputs
  reduce (C key, D value)  {
    // C = string, D = integer, E = binary I interpret
    // so some algorithm
    sendOutput (E)
  }
}
```

The job can do anything on purpose to analyze and structure data for a given need, be defined with no constraints but the interfase implementation and the pair key-value as as mapper and reducer method signature

Finally we have an index of words pointing to the documents where they were found, that's a reverted index

- Google: Sawzall
- Yahoo: Pig (now in hadoop)
- Microsoft: DryadLINQ
- GreenPlum: YAML MR
- IBM: JAQL
- Business.com: Cloudbase
- AsterData: In-Database MR

Figure 2.3: Let's see a sample job with a reverted-index function, for analyzing the webcrawler's output files (just for instance)

of key-value that will be the input for the next reduce tasks

- These user defined tasks (reduce), will read the value object, do something with it, and then produce the 3rd and last list of key-value pairs, that the framework will combine, and regroup into a final result.

MapReduce something, is about iterate a huge record collection, extract something good, mix and regroup intermediate results, that's all, it may look more complex than what it is.