

Scalable parallel processing with MapReduce
Understand the challenge of parallel processing

Map & using MongoDB

Manipulating large data

- ↳ few possible points of interaction
- ↳ few conflicts & less management

I/O bottleneck $\xrightarrow{\text{forward}}$ fast & efficient processing

1) One solution build super computer

- ↳ Not possible

↳ Expensive to build & maintain

2) Grid computing

↳ Distribute work among set of nodes

↳ reduce computational time that a single large machine takes to complete the same task.

↳ Data passing between nodes is managed by Message Passing Interface (MPI)

this topology works well with extra CPU cycle
but not so much when large data needs
to be transferred.

I/O & bandwidth limitation

public computing project like

SETI @ Home

Folding @ Home

None linked keep data locally in a compute
grid to minimize bandwidth blockages.

Map Reduce

→ little interdependence among parallel processes
& keeping data and computation together.

→ Developed for distributed computing & patented by
Google.

Map Reduce on Mongo DB clusters

Aggregate functions like → sum, count, max, analyse

Mapreduce

- ↳ MongoDB
- ↳ Hadoop
- ↳ HBase

→ Import CSV file in mongoDB

mongorestore --type CSV --db mydb --collection myc --headerline Name-of-file.csv

From → db.myc.findOne()

↳ will get one document.

MapReduce has following 2 parts

- ↳ map function
- ↳ reduce function

The two functions are applied to data sequentially
though the underlying system frequently runs
combination in parallel.

Map takes key/value pair & emits another key/value
pair.

Reduce takes output of map phase and manipulates
the key/value pairs to derive the final results.

Map function
is applied to each document of collection

completely independent of some operation of another node.

Map reduce phase most takes care of collecting and sorting the output from multiple nodes.

Reduce function

Takes key / value pairs that come out of a map phase and manipulate it further to come to final result.

Aggregating values on the basis of common key.

var map = function () {

emit (this.stock_symbol, { stock_price_high :
this.stock_price_hi});
}

}

var reduce = function (key, values) {

var high_price = 0.0;
for (int i = 0; i < values.length; i++) {

} { values[i] > (key.price_hi) ? key.price_hi : key.price_hi };

```
return {highStockRule: high});  
};
```

finalize function → to do additional
operation of sum or count function.

Hadoop Install

echo JAVA_HOME

↳ get path of java

install ssh-server

↳ sudo apt-get install openssh-server

ssh localhost

→ go to .. /sbin/

.. /start-all.sh → start hadoop

cd .. /bin/

open http://localhost:50030

cd .. /sbin/

.. /stop-all.sh → stop hadoop

Mongo DB

1) Replication

↳ Redundancy & Data availability

Multiple copies on different servers

↳ R protects database from loss of single server.

Allows to recover from hardware failure and service interruptions

Increase read capacity

Maintain copies in different data centres → increases locality and availability.

Replica Set → group of mongod instances that hosts some dataset.

Primary receives all write operations

For all → secondaries → apply operations from primary so that they have same dataset.

1) Primary accepts write operation from client
(Replica set can have only one primary)

so since only one member can accept write operation → replica set provide strict consistency

2) primary logs all changes in data set in oplog

3) secondaries replicate the primary's oplog & apply the operation to their data set.



Secondary data set reflect the primary data set

4) if primary is unavailable

↳ replicas will elect a secondary to be primary.

By default client reads from primary, but client can specify to read from secondary.

Arbitrator :-

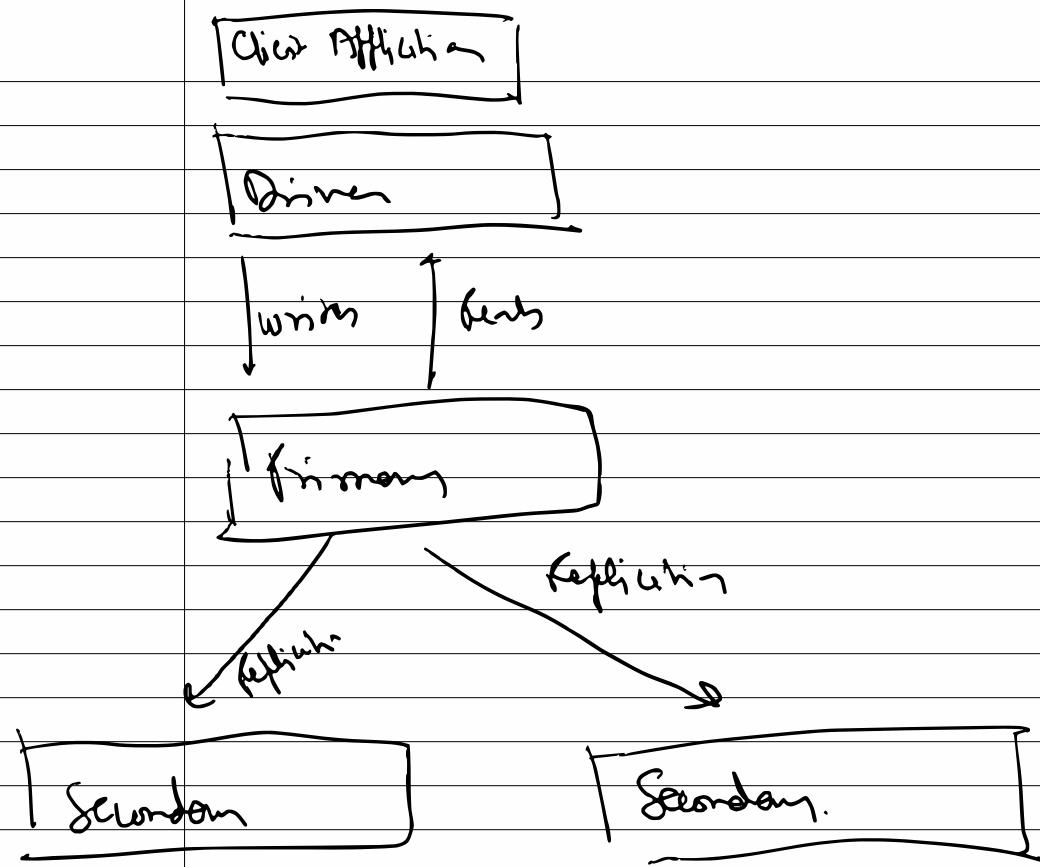
Extra mongod instance as an arbiter.

Arbitrators do not maintain a data set.

↳ only exist to vote in election.

They do not require dedicated hardware

if the replicas are even number then add one arbiter to get majority of votes in an election.



Secondaries apply operations to primary asynchronously.

Sets can continue to function without some members

- * Secondaries may not return the most current data

Automatic Failover

- When a primary does not respond to client for more than 10 seconds.
- Replica set will select one member to become primary.
- The secondary which will get maximum vote in an election will become primary.

Additional Features

You may deploy replica sets with members in multiple data centers

Or control the outcome of elections by adjusting the priority of some members.

Replica set also support dedicated member for reporting, disaster recovery, or backup functions.

Replica Set Members

Replica Set

↳ group of MongoDB instances that provide redundancy and high availability

The members are

- ↳ primary → all write operations
- ↳ secondary → replicates from primary to maintain an identical database
- ↳ may have additional configurations for special usage
- ↳ priority may be set to 0 → no voting

- Also, can have arbiter

- ↳ do not keep copy of data
- ↳ only used in voting.

Replica sets can have up to 12 members. Only 7 at a time can take part in election.

Minimum requirements

- ↳ primary, secondary, arbiter.

Must however keep 3 members

1 - primary

two secondary.

Client can not write data on secondaries but client can read from secondaries.

Configuration of secondary members

1) Priority 0 replica set members

- ↳ prevents it from becoming primary in an election, which allows it to remain secondary data centers or to serve as a cold standby.
Also casts vote.

2) Hidden replica set member

- ↳ Prevents application from reading from it. allows it to run apps that require separation from traffic.

3) Delayed replica set members

- ↳ keep a running "historical" snapshot for use in recovery from certain errors, such as unintentionally deleted databases.

Priority 0 Replica Set members

Can not become primary and cannot trigger elections

Makes copy, accepts read operation and votes in election.

Hidden Replica Set Members

Maintaining copy but is invisible to client.

They are also priority 0 and can not become primaries

db.isMaster() method does not display hidden members.

They vote in election

Ensure low network latency to network to primary or likely primary

Ensure their replication lag is minimal or non-existent.

Delayed replica set members

makes copy

Refers to a delayed state of a 2.

So helpful in accidental recovery of data.

- Must be primary \rightarrow can't become primary
- Should be hidden \rightarrow no read operation

Vote in elections

Delay \rightarrow equal to or greater than maintenance window
smaller than capacity of logs.

Arbiters

- Does not have copy & can't become primary
- \hookrightarrow used to add vote in elections
- \hookrightarrow allows for uneven members

Important

\hookrightarrow Don't have arbiters on system that also has primary and secondary

\hookrightarrow only add arbiters to even numbers of members
else

may suffer from tied decisions.

Sharding

- ↳ storing data records across multiple machines
- ↳ meeting demand of data growth

As data increase, single machine may not be sufficient to store data nor provide an acceptable read and write through put.

Sharding solves the problem with horizontal scaling

Purpose of Sharding

- ↳ high data I/O throughput → challenge single server
- ↳ high query rates can exhaust the CPU capacity of server
- ↳ large data set exceeds capacity of a single machine.
- ↳ working sets larger than CPU RAM stress the I/O capacity of disk drives

To address these DB has two approaches

- Vertical Scaling
- Sharding

1) Vertical Scaling

↳ Adds more CPU and storage resources to increase capacity

Limitation

- High performance CPU with large RAM are more expensive than smaller systems.
- cloud may allow users to provision only smaller instances.

There is practical maximum capability for vertical scaling.

2) Sharding or horizontal scaling divides the data sets and distributes the data over multiple servers (shards)

Each shard is an independent database and collectively the shards makeup a single logical database.

Sharding addresses the challenge of scaling to support high throughput and large data sets.

Sharding reduces the number of operations each shard handles.

As cluster grows → each shard process fewer operations
shared cluster can increase capacity and throughput horizontally.

Revolves the amount of data each server needs to store.

$$1\text{ TB} \rightarrow 40\text{ shards} \rightarrow 25\text{ GB per shard.}$$

Mongo DB supports sharding

↳ configuration of sharded cluster.

Shard cluster has the following components:-

- 1) Shards
- 2) Query Router
- 3) Config Servers

* For development & testing purpose only, each shard can be a single mongod instead of a replica set.
Do not deploy production cluster without 3 config servers.

- 1) Shards

Store the data

To provide high availability and data consistency in production sharded cluster, each shard is a replica set.

2) Query Router

↳

Mongos instances

↳ interface with client application

↳

direct operation to appropriate shards

→ preprocesses → targets operation to shard

↳

return result to user.

→ may have more than one query router
to divide the client request load.

3) Config Servers

↳

store cluster metadata

↖

mapping of data to the shards

Query Router uses this metadata to target operations to specific shards.

Production sharded clusters have exactly 3 config servers.

Data Partitioning

MongoDB distributes data, or shards, at the collection level



Sharding partitions the data by the shard key.

Shard Key

↳ To shard need to select a shard key

↳ either an indexed field or indexed compound field that exists in every document

↳ MongoDB divides the shard key values into chunks and distributes the chunks evenly across the shards.

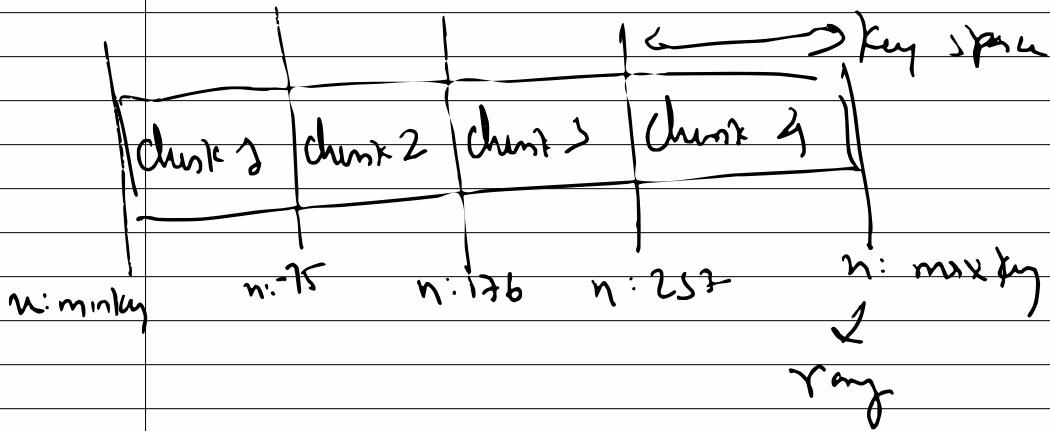
To distribute the shard key values into chunks, MongoDB uses —

range based partitioning
hash based partitioning

Range Based Sharding

Divides the data set into some ranges determined by shard key values.

Eg → numeric shard key → divide into non overlapping ranges → chunks



minkey \rightarrow maxkey

Hash Based Sharding

Compute hash of field values and depending on that divides mem into chunks

In hash base case closer values may have different hash codes and may fall into different chunks \rightarrow more random partitioning of values

Maintaining a Balanced Data Distribution

↳ Addition of new data or addition of new servers can result in data distribution imbalance within the cluster.

MongoDB ensures balanced data distribution using following two background processes.

- 1) Splitting
- 2) Balancer.

1) Splitting

When a chunk grows from specified chunk size MongoDB splits the chunk in half.

Major update triggers split.



efficient meta-data change

To cause split MongoDB doesn't migrate any data or affect any sharding.

Balancing

of the distribution of sharded collection is whenever the balancer knows migrates chunk from shard that most data to shard that has less data.

This is a background operation

↳ During this all request to chunk data occurs happens on origin shard.

Migration

- 1) Destination shard receives all data
- 2) Destination shard captures & applies changes to data
- 3) Destination shard changes the metadata regarding the location of config server
 - ↳ If any error MongoDB aborts the process leaving the chunk on origin side
- 4) After the migration is complete successfully the MongoDB removes the data from origin.

Adding & Removing Shards from the cluster

- Adding a shard to a cluster creates imbalance since the new shard has no chunks
- While MongoDB begins migrating data to the new shard immediately, it can take some time before the cluster balances
- When removing a shard, the balancer migrates all chunks from to other shards
- After migrating all data and updating the meta data, you can safely remove the shard.

Sharded Cluster Requirements

It takes time to shard

Think ahead and while designing take into consideration for need of sharding for your data model.

Don't wait until your system reaches overcapacity for sharding.

Production Cluster Architecture

Ensure → data is redundant ⇒ systems are highly available.

Must have following:-

- 3 config servers.

↳ Each of different machine

↳ A single sharded cluster must have exclusive use of its config servers.

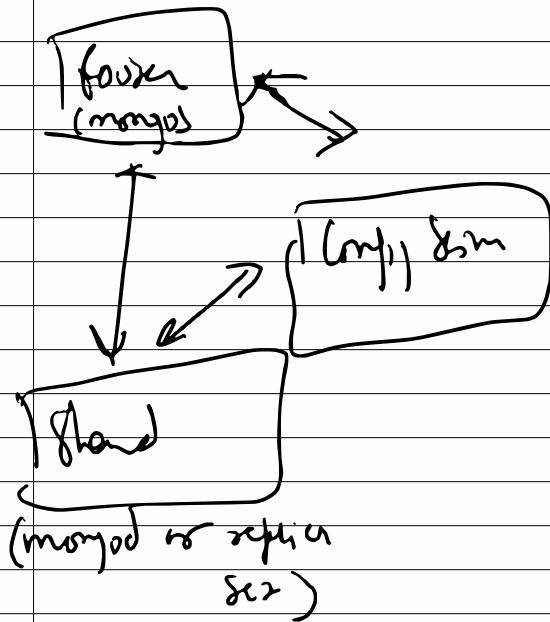
↳ If you have multiple sharded clusters → need group of config servers for each cluster.

- 2 or more replica sets → These are the shards.

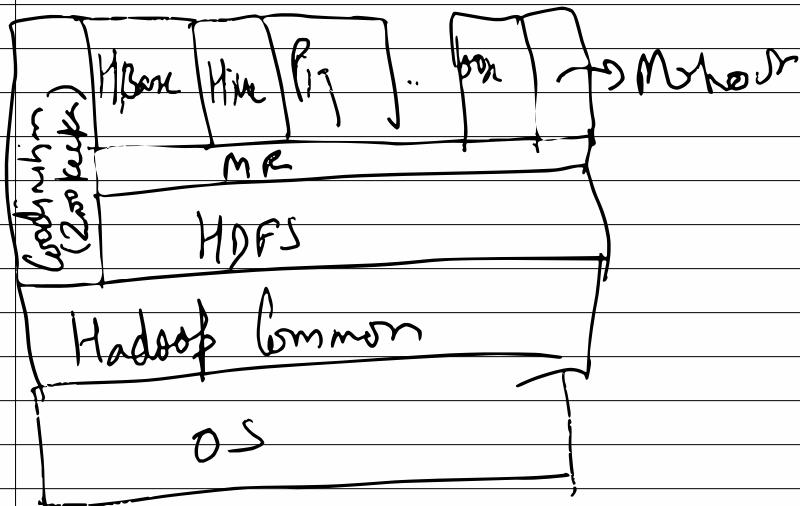
→ One or more mongos instances.

Sharded cluster Test Architecture

- 1 config server
- 1 shard (12 shards)
- 1 mongos instance



4) Hadoop Architecture



Although hadoop borrows its idea from Google's mapreduce it is more than mapreduce.

A typical hadoop based big data platform includes

↳ HDFS

↳ Parallel computing framework → Map Reduce

↳ Common utilities

↳ A column oriented data storage table (→ Base)

↳ High level data management system (pig & hive)

↳ Big Data analytics library (Mahout)

↳ A distributed coordination system (Zookeeper)

HDFS

It was designed as a distributed filesystem that provides high throughput access to application data.

Data in HDFS is stored as data blocks

↗
replicated on several computing nodes
↗ their checksums are computed

In case of a checksum error or system failure, erroneous or lost data blocks can be recovered from backup blocks located on other nodes.

Map Reduce

computation over key value pairs

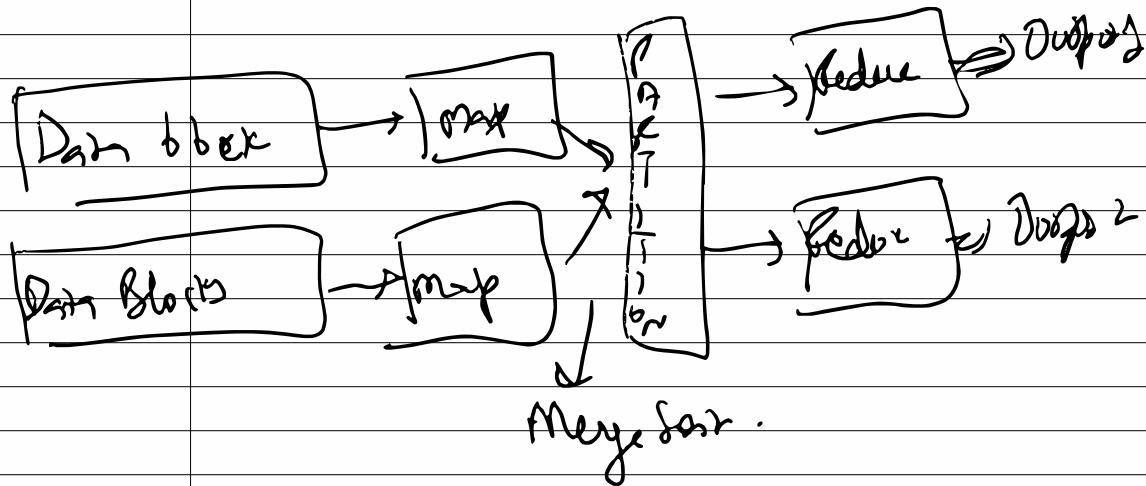
map tasks on slave nodes are executed in parallel

↳ Once some or all the map tasks have finished, the shuffle process begins which aggregates the map task outputs by sorting and combining the key value pairs by keys.

↳ shuffled data partitions are copied to reduce machines.

Then the reducer task will run on shuffled data and generate final results.

↓
it will reside in multiple files
depending on number of reducers used in the job.



HDFS has two types of nodes

- Name node
- Data node

Name node keeps track of the filesystem metadata
(the location of data blocks)

for efficiency

metadata is kept in the main memory of a master machine

Data node holds the physical data blocks

and communicates with client for data reading and writing.

In addition, it periodically reports a list of its hosting blocks to the NameNode in the cluster for verification and validation purposes.

The mapReduce framework has two types of nodes :

- 1) Master node
- 2) Slave node

Job Tracker is the daemon of Master Node

Task Tracker is the daemon of Slave Node

The master node is manager of MapReduce jobs



It splits a job into smaller tasks



which is then assigned by Task tracker
of slave nodes by Job Trackers of
Master node

When a slave node receives a task, its Task
Tracker will fork a Java process to run it.

Task Tracker is also responsible for tracking and
reporting the progress of individual tasks.

Hadoop Distributed file system

HDFS was built to support high throughput streaming reads and writes of extremely large files.

Traditional large storage

This is where traditional storage doesn't necessary scale.

Goals of HDFS

store millions of large files
use scale out model based on JBOD

↓
just bunch of disks.

optimize for large streaming reads and writes rather than low-latency access to many small files.

gracefully deal with system failure of machines and disks

support the functionality and scale requirements of MapReduce processing.

HDFS

files are stored as opaque blocks of metadata exists



common to linux file systems.

HDFS → userspace filesystem.

code runs outside of kernel as
OS processes

We don't mount HDFS as ext3



∴ requires application to explicitly build
for this.

HDFS is a distributed file system



Each machine in cluster stores a subset
of data → more data → just add more
machines

File system metadata is stored on a centralized
server, acting as a directory of block
data → providing picture of file system's state.

Major difference between HDFS and other file system is its block size

Normal file system \rightarrow 4kb / 8kb block

HDFS \longrightarrow 64 MB by default.



cluster admin raise it to 128 or 256 MB



or user

3 GB

More data will be written in larger contiguous chunk on disk



This minimize drive seek operations

HDFS Replication

Rather than relying on specialized storage subsystem data protection, HDFS replicates each block to multiple machines in cluster.

by default, each block is replicated 3 times.

In HDFS files are written once & once the replica is written, it is not possible for it to change.



This avoids need of complex reasoning about consistency of replicas



file can be accessed from any replica.



Multiple replicas mean multiple machine failures are easily tolerated



HDFS tracks and manages the number of replicas of a block as well

If number of copies of a block drop below



file-system automatically makes a new copy from one of remaining replicas.

HDFS presents the filesystem to developer as a high level, POSIX like API with familiar operations and concepts.

Daemons

Namenode 1 per cluster stores file metadata

Secondary Name node 1 per cluster performs internal name node handoff and log checkpointing.

Datanode many Stores block data (file contents)

Blocks

Blocks are nothing more than chunks of a file binary blobs of data

↳ daemon responsible for storing and retrieving block data → datanode

data node has access to one or more disks —
data disks → for which it is allowed
to store the data.



In production these disks are usually reserved
for Hadoop.

Storage can be added by adding more data nodes
with additional disk capacity

↓
or by adding disks to existing
data nodes.

HDFS does not require RAID storage



for storing block of data

↳ commodity hardware design goal → reduce
cost as cluster size grows

No RAID safety → block data written to multiple
machines.

safety \rightarrow cost \rightarrow performance

multiplying copies \rightarrow power \rightarrow data loss



can access from anywhere



improves performance.

Name Node

\hookrightarrow stores filesystem metadata

\hookrightarrow maintains complete picture of filesystem

client connects to the name node \rightarrow filesystem operation

although \rightarrow data is stored to/from datanodes directly.

Datanodes report their status to name nodes in a heartbeat.

\hookrightarrow means \rightarrow at any given time \rightarrow

name node has complete view of datanode in a cluster, health, what blocks they have available

Block Report

when datanode starts up, as well as every hour thereafter, it sends Block Report to the namenode.

→ list of all blocks the datanode has on its disk → allows namenode to keep track of change.

file to block mapping on the namenode is stored on disk → location of the blocks are not written on disk

Datanode failure

Administrator simply removes its hard drives, place them in new chassis & starting the new machine

As far namenode is concerned the blocks have simply moved to new datanode

Downside → while starting → namenode waits to receive ~~BC~~

Secondary Name node

- ↳ Performs internal housekeeping for the name node
- ↳ Not backup for namenode

Drawbacks

- ↳ Difficult to configure & manage
 - Not optimal choice for real-time ,
 - Responsive big Data application
 - Hadoop is not a good fit for large graph datasets.
- Not good for data that is not recognized as big data .
- ↳ small datasets
 - ↳ datasets with processing that requires transaction and synchronization.

Hadoop Architecture

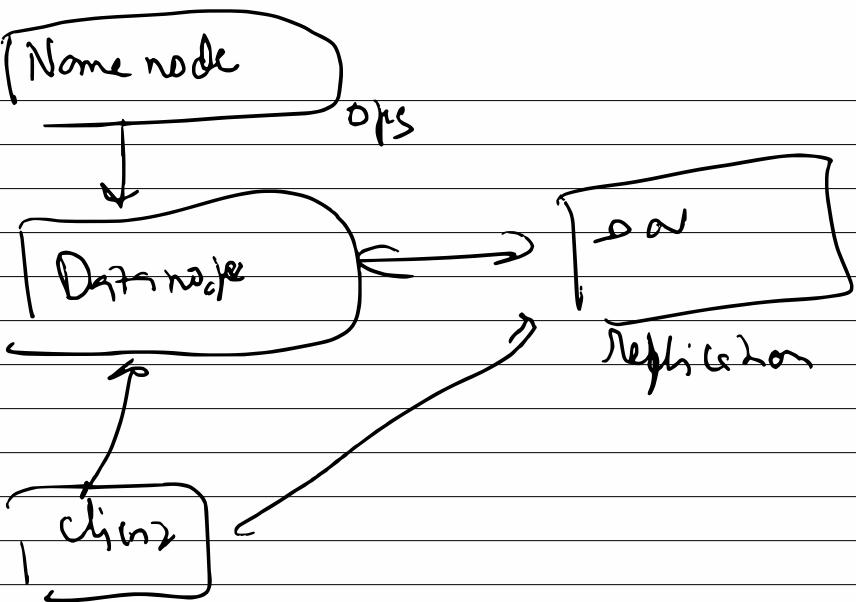
Running Hadoop on a Single-Node Cluster

Hadoop is a framework written in Java for running applications on large clusters of commodity hardware and incorporates features similar to those of the Google file system (GFS) and of the MapReduce computing paradigm.

Hadoop's HDFS is a highly fault tolerant distributed file system designed to be deployed on low cost hardware.

It provides high throughput access to application data and is suitable for applications that handle large datasets.

Hadoop → Apache Hadoop project → Apache Lucene project.



Installation

Required Software

Java 1.6 or later
 ssh must be installed and sshd must be running to use the Hadoop scripts.

cheat - ger

sudo apt-get install ssh

ssh localhost

Formatting name node.

name node format

⑥) PS 100] to check processes running

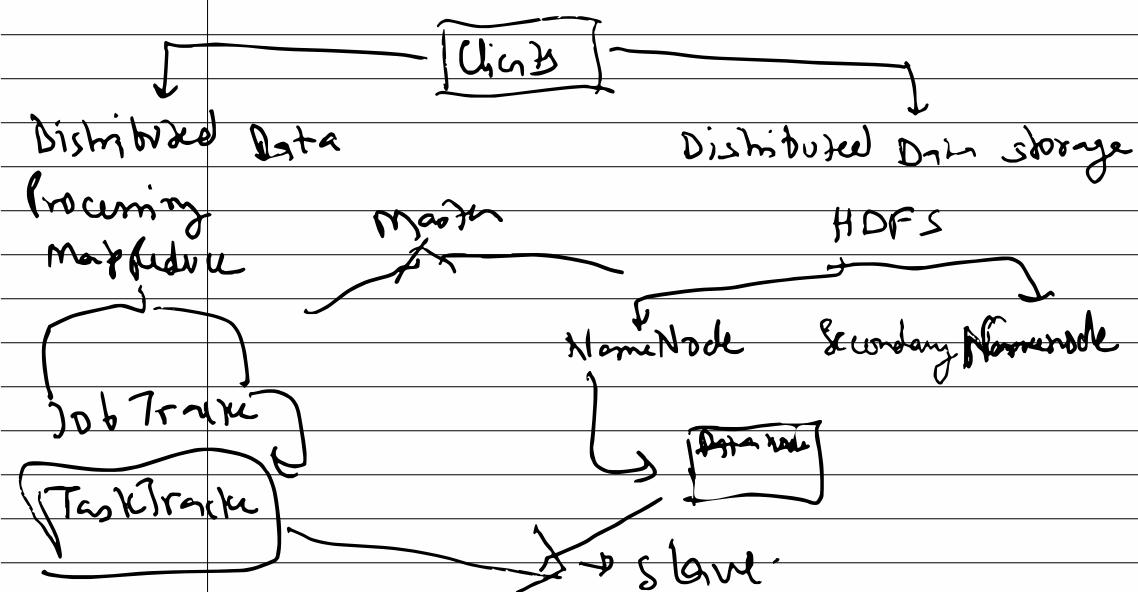
Name Node
Secondary Name Node
Data Node
Resource Manager
Map Reduce
Node Manager

Stop → stop-all.sh

(in word count)

Hadoop jar mr.jar wordcount /down/boot
/output

Hadoop Seven parts



Categories of machine roles in Hadoop

- 1) Client machines
 - 2) Master nodes
 - 3) Slave nodes.
-

1) Master Nodes

Oversees 2 key functions

- storing lots of data (HDFS)
- running parallel computation of all that data (mapreduce)

NameNode oversees & coordinates the data storage function (HDFS)

The Job Tracker oversees and coordinates the parallel processing of data using mapreduce

2) Slave Nodes

Make up vast majority of machines to do all the dirty work of storing data & doing computation

Each slave node run two DataNode & Task Tracker

Task Tracker slave of Job Tracker

DataNode slave to NameNode

3) Client Machines

These have hadoop installed with all the cluster settings but are neither a master nor a slave.

Role :-

- Load data in cluster
- submit map reduce jobs
- Retrieve / view the results

In small cluster (~ 40 nodes) single server playing multiple roles, Job Tracker & Name node b/w.

With medium to large cluster \rightarrow each role operating on a single server machine.

Typical Workflow

- Load data into cluster (HDFS writes)
- Analyze the data (map reduce)
- Store the results in cluster (HDFS writes)
- Read results from the cluster (HDFS reads)

Why hadoop?

Companies have lots of data

- we break into smaller chunks and parallel process each chunk we will get our answers much faster.

Writing Files to HDFS

- client consults Name Node
- client writes block directly to one Data Node
- Data Node replicates block
- cycle repeats for next block.

Loading files into Hadoop cluster

- Break data into smaller blocks & place these on different machines in cluster
- more block → more machines → more parallel processing
- every block on multiple machine → machine failure

Replication

- ↳ Each block is replicated as it's loaded
- Standard 3 copies
- configured with → replication in file `hdfs-site.xml`

Name Node

↳ The client breaks files into 3 blocks

- Client consults NameNode (TCP port 9000)
and receives list of 3 data nodes
↳ copy
- Client writes directly to data nodes (TCP 50010)
- The data nodes copy & replicates the blocks
- Namenode is not in the data path

The NameNode only provides the map of where data is and where data should go in cluster (file system metadata)

Hadoop Rack Awareness

Never loose all the data if entire rack fails.
Keep bulk flow in-rack whenever possible

↙
higher bandwidth
→ lower latency.

Somebody should know where DataNodes are located in network topology & make informed decision while replicating data nodes

This somebody is NameNode.

HDFS units

↳ Client ask to write
Name node provide list of data nodes

Rule \rightarrow 2 copies on same rack
another copy on different rack.

Client sends to DataNode 7 to check for
S, b \rightarrow then S sends to 6 \rightarrow after
acknowledgement is received \rightarrow client writes

Data node passes data along as they receive

TCP - 50010

Pipeline units

As data is being written a replication
pipeline is being created between the
3 data nodes

Data node receiving the block \rightarrow same
time pushes a copy to the next node
in the pipeline

When all blocks successfully written name node gets servers notification from client who gets it from data nodes

Now, name Node updates the metadata of the file with data node list.

More blocks → more machines → more CPU cores → more disk drives → more parallel processing power → faster results.

Another approach to scaling is go deep

↳ scale up the machines with more disk drives and more CPU cores.

↳ instead of increasing the number ~~size~~ of machines → increase density of machine

Name Node

- Data node sends heartbeats
- Every 10m heartbeat is a block report
- Name node builds metadata from block reports
- If Name node is down \rightarrow HDFS is down

All system metadata \rightarrow oversees health of
Data nodes \rightarrow coordinates access to data.

↓
central controller of HDFS

Don't hold data

knows

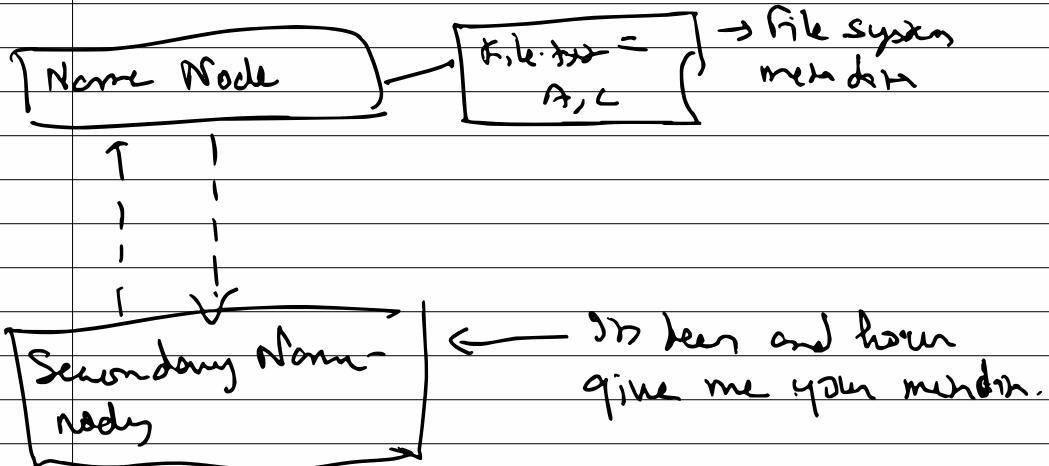
- \hookrightarrow which blocks makes up file
- \hookrightarrow where these blocks are in Data node
- \hookrightarrow points client to data node to talk.

Data nodes send heartbeats to name node every
3 seconds \rightarrow TCP port 50030 \rightarrow 9000
10m is Block reporter.

Replicating missing replicas

- Missing heartbeat signifies loss
- Name node consults meta data
 - ↳ finds affected dir
 - ↳ consult rack awareness script
 - ↳ weighs re missing data to new data nodes.

Secondary Name nodes



- Not a hot standby for the name node
- connects to name node every hour
- housekeeping, backup of Name node metadata
- saved metadata can help build failed name node.

No & high availability back up of namenode



occasionally connects (1 hr) → get metadata
→ files used to store metadata (both of
which may be out of sync)



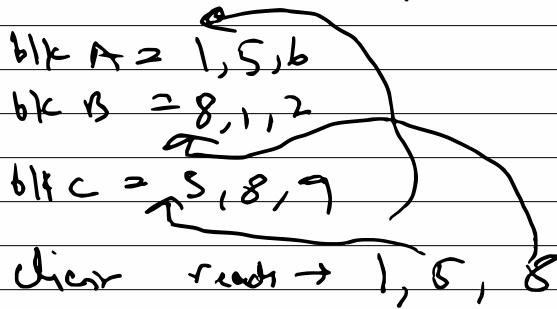
combines this info in fresh set of files
sending back to namenode → keeping a copy
for itself. e.g.

Can be used in case of namenode
failure.

In case of busy cluster admin can
increase the frequency from 1 hr to
say 1 minute.

client reading from HDFS

- receives data node list for each block
- picks first data node for each block
- reads blocks sequentially



with ~~not~~ TCP - 50010
↑

default port number
for datanode daemon

Datanode reading file from HDFS

- Name node provide stack local nodes first
- leverage in-rack bandwidth, single hop

Data Processing : Map

Job tracker delivers Java codes to nodes with local data

Job tracker takes node information from name node



Provides Task tracker on those nodes with Java code to do the computation locally



Task Tracker provides the heartbeat and task status back to the job tracker



As map completes



Each node stores the result of its local computation in temporary local storage

→ intermediate data.



Next step



Send this data to reduce phase.

If Data is not local

↳ If data node is busy \rightarrow Name node will assign the map job to a diff. node in some rack.



Rack Awareness.

2) Data Processing: Reduce

Map tasks sends output data to Reducer over the network.

↳ Reduce task data output written to and read from HDFS.

* Adding new server may not help as hadoop does not automatically transfer existing data to new server.

Cluster Balancing

- Balancer utility, if used, runs in background
- does not interfere with map/reduce of HDFS \rightarrow rate $\sim 1 \text{ MB/sec}$.

The amount of traffic balanced uses
is pretty low \rightarrow 1 MB/sec.



can be changed in

algs. balance . bandwidthPerSec
in file



helps - site - xml .

5) Secondary Sorting

Sometimes we need to sort values coming into the reducer of a Hadoop map/reduce jobs

We can sort the values by using following implementations

- 1) use a composite key
- 2) Extends org.apache.hadoop.mapreduce.Partitioner
- 3) extends org.apache.hadoop.io.WritableComparator

Solution for secondary sorting involves doing multiple things.

- 1) Use a composite key
 - ↳ A key that has many parts
↳ key will have the stock symbol and time stamp.

So we can't

$$(K2, V2) = (\{\text{symbol}, \text{timestamp}\}, \text{price});$$

Natural key → by which values are grouped by.

2) Composite key comparator

↳ where the secondary sorting takes place

here we sort based on symbol and timestamp

All the components are considered

public class CompositeKeyComparator extends

WritableComparator {

protected CompositeKeyComparator () {

super (StockKey.class, true);

}

@Override

public int compare (WritableComparable w1,
 WritableComparable w2)

{

StockKey k1 = (StockKey) w1;

StockKey k2 = (StockKey) w2;

int result = k1.getSymbol().compareTo
k2.getSymbol();

} (result == 0)

```
result = -1 * k1.getTimestamp().compareTo(   
    k2.getTimestamp());  
}  
}
```

```
return result;  
}  
}
```

3) Natural Key Grouping comparator

↳ groups values together according to
natural key

↓
without this each $k_2 \rightarrow v_2$ may go to
different reducer

```
public class NaturalKeyGroupingComparator extends  
WritableComparator {  
protected NaturalKeyGroupingComparator() {  
super(ShardingClass, true);  
}  
}
```

@override

public int compare (WritableComparable w1,
WritableComparable w2)

{

StockKey k1 = (StockKey) w1;

StockKey k2 = (StockKey) w2;

return k1.getSymbol().compareTo(k2.getSymbol());

}

}

3) Natural Key Partitioner

↳ The natural key partitioner uses the natural key to partition the data to the reducers

Public class NaturalKeyPartitioner extends

Partitioner < StockKey, DoubleWritable > {

@override

public int getPartition (StockKey key, DoubleWritable
value, int numPartitions)

{

```
int hash = key.getStockSymbol().hashCode();
int partition = hash % numPartitions;
return partition;
}
```

↳ Now we may configure the job as follows :-

```
public class SSJob extends Configured implements Tool {
    public static void main(String[] args) throws
        Exception {
        ToolRunner.run(new Configuration(), new SSJob(),
            args);
    }
}
```

@Override

```
public int run(String[] args) throws Exception {
```

```
    Configuration conf = getConf();
```

```
    Job job = new Job(conf, "secondary sort");
```

job.setJarByClass(ss.job.class);

job.setPartitionerClass(NFKP.class);

job.setGroupingComparatorClass(NKGc.class);

job.setSortCombinerClass(CompositeKeyCombiner.class);

// Map Class

job.setMapperOutputKeyClass(StockKey.class);

job.setMapperOutputValueClass(DoubleWritable.class);

// reduce output

job.setReducerOutputKeyClass(Text.class);

job.setReducerOutputValueClass(Text.class);

// input & output format class

job.setInputFormatClass(TextInputFormat.class);

job.setOutputFormatClass(TextOutputFormat.class);

// set map & reduce class

job.setMapperClass(SsMapper.class);

job.setReducerClass(SsReducer.class);

job.waitForCompletion(true);

return 0;

}

Chaining MapReduce Jobs

↳ Getting inverted index

↳ getting top 10

Can be chained so that output of 1 MapReduce job is input of other.

MR-1 | MR-2 | . . .



like Unix pipes.

1) Normal chaining

↳ Driver sets up JobConf object with configuration of MR job and passes JobConf to JobRunner.runJob()

↳ Getting MR driver of one MR job after another.

↳ Input part of other → output part of previous

↳ Delete intermediate data generated in each Job.

2) non linear & complex dependency

↳ Job and JobControl classes.



MapReduce Job

Implement by passing JobConf object to its constructor.

Now, in addition to configuration information it can also have dependency information

specified through addDependency() method.
job x and y.

x.addDependency(y)

↓

x will not start until y has finished

JobControl object do the managing and monitoring
of job execution

↓

add jobs to jobcontrol object → addJob() method

After adding all the jobs and dependencies call
JobControl's run() method → spawns a thread
to submit and monitor jobs for execution.

It also has methods like allFinished() and
getFailedJobs() to track the execution of
various jobs within the batch.

Chaining Preprocessing & Post Processing Steps:-

Hadoop introduces ChainMapper & ChainReducer class to simplify the composition of pre and post processing.

Chaining



[map() reduce()]+



(ChainMapper.addMapper())

(ChainReducer.addReducer())

(ChainReducer.addMapper())

Hadoop Summarization Pattern

1) Median And Standard Deviation

Easy solution

→ copy the list in temporary list
and send to reducer to get median
and SD

May result in heap space issue

MAP

key → hour of day
value → comment length

REDUCE

Adding all values to in memory list
so we → calculate median and SD

$$SD^2 = \sqrt{\frac{\sum (n - \bar{n})^2}{n-1}}$$

key → hour of day
value → tuple $\Rightarrow \{ \text{median}, SD \}$

Combining

↳ can not be used

2) Complex Solution

keep count of each value

combiner can be used

$$T(n) \geq O(\max(m))$$

instead of
 $T(n) = O(n)$

Mapper:-

key: hour of the day

value: sorted map with

by common length, word ↗

Reducer

value : $\text{Row} \rightarrow \text{Tuple} \{ \text{Median}, \dots \}$

combiner

Aggregating the sorted map entries for
each local mapper.

2) Inverted Index Summarization

↳ commonly used as an example for MapReduce analytics.

Intuit:

generate and index given data set to allow for faster searches or data enrichment capabilities.

↳ Extra processing up front, so that later the search takes less time.

quick search query responses are required

↳ Result of such a query can be preprocessed and ingested into database.

Mapper

key: derived field

value: unique identifier

Combiner:

optional. → same as reducer as it groups

Partitioner

: can be used for load balancing

Reducer

- list of values concatenated → key, value or values added with key.

Performance → good parsing, summing unnecessary words like → a, an, the etc.

3) Counting with counters

The pattern utilizes MR framework's counters utility to calculate a global sum entirely on map side without producing any output.

Intuit

↳ retrieve count summarization of large data sets.



↳ instead of sending to reduce to count

↳ use MR counting mechanism to keep track of the number of input records.



No reduce phase, no summation !

No output $\Rightarrow \{ \text{emp-id}, 1 \}$

↳ create counter with emp-id dimension by 1

↳

After the job save the output anywhere.

Counters

- ↳ Number of counters should be tens or hundred at most.
- ↳ or they will bog down the Job Trackers.

when

- ↳ need counts over large data sets
- ↳ number of counters is small in tens.

Output

- ↳ grabbed directly from the job framework.
 - ↳ directory will have part file as per mapper
- will be deleted on job completion.

Performance

- ↳ Very fast

depends on number of map tasks being executed.

Mapper :-

comparator.getCounter (STATE-GROUPS-HC)
· increment (1);

Main ()

{

int code = job.waitForCompletion(true);

if (code == 0)

{

for (Counter counter : job.getContainers()) -

getGroup() | Map[Container -> String -> Group])

System.out.println(counter.getDisplayName() + " - " +

counter.getValue());

}

7) Bloom Filter

Conceived by Burton Howard Bloom in 1970



Probabilistic data structure used to test whether an element is a member of a given set.



space advantage over hash set as all member uses some amount of space, no matter its actual size.



Part of pattern → Bloom filtering

Memory advantage but not 100% accurate.

False positives are possible but false negatives are not.

So result :- Definitive No

or, may be

Never Definitive Yes.

Elements can be added to the set but
not removed. \Rightarrow Traditional



Implementation to counter this limitation
↓

Working Bloom filter

\hookrightarrow But, uses more memory.

As more data added \Rightarrow false positive
increases.

Can not be resized like other data structure.

Following variables are used in more
detailed explanation

$m \rightarrow$ The number of bits in the filter

$n \rightarrow$ The number of members in the set.

$p \rightarrow$ The desired false positive rate

$k \rightarrow$ The number of diff. hash function
used to map some elements to one
of the m bits with uniform random
distribution.

Use Case

- 1) Representing a dataset
 - ↳ large dataset
- 2) Reduce queries w/ extend database
- 3) Google Bigtable

Down sides

- ↳ false positive rate

7) Map Reduce Filtering Pattern

Filtering pattern don't change the actual records

These patterns find subset of data:

- It could be small
 - like top 10 listing

- large
 - like the result of a deduplication

This differentiates filtering pattern from those in previous lecture

↳ which was all about summarizing and grouping data by similar fields to get a top level view of the data.

Filtering is more about understanding

- ⇒ Smaller piece of your data
- ⇒ All records generated from a particular user
- ⇒ top 10 most used verbs in text

Sampling

↳ One common application of filtering

Pulling out sample data
like getting highest or taking
randomly some data



User for much smaller representation of
whole data \Rightarrow can be easily analysed.

Sub Sampling

↳ a well distributed sample over
whole data set.

Four patterns are listed

- filtering
- Bloom Filtering
- Top 10
- Distinct.

Filtering, Bloom filtering and simple

Random sampling is map only job.

We don't need reducers for them.

1) Filtering Patterns

Evaluate each record and based on some condition decides to include or not.

function ()

{
if (true)

emit (key, value)

Condition:-

Data can be parsed into "records" that can be categorised through some well specified criterion determining whether they are to be kept.

No reduce \rightarrow only evaluate then keep or discard.

Same key value pair \rightarrow record is unchanged.

Known Use

- 1) Close view of data
- 2) Tracking a thread of events
- 3) Distributed grep
- 4) Data cleaning
- 5) Simple Random Sampling
- 6) Removing low scoring data.

Performance Analysis

↳ very efficient as map only job-

but there can be many mapper files
↳ some dies by using identity reducer

does nothing but only emits one map data.

Example :-

- 1) Distributed grep :-

Global RF kind.

public static class GrepMapper
extends Mapper<Object, Text, NullWritable, Text> {

private String mapRegex = null;

public void setup (Context context) throws IOException,
InterruptedException {

mapRegex = context.getConfiguration().get("mapRegex");

}

public void map (Key key, Text value,
Context context)

{

i) (value.toString().matches (mapRegex))

{

context.write (NullWritable.get(),
value)

)

}

}

2) Simple random Sampling

To give each record equal probability of being selected. Take a random number generator and emit it if the value is in the threshold or rejected.

public static class SRSMapper

extends Mapper<Object, Text, NullWritable,
Text> {

private Random rand = new Random();

private Double percentage;

protected void setup(Context context)
{

String percentage = context.getConfiguration().
get("filter-percentage");

percentage = Double.parseDouble(percentage) / 100;
}

public void map (Object key, Text value,
Context context)

{

if (rand.nextDouble < percentage)

context.write (NullWritable.get(), value);

}

}

or set number of reducer to 1 and

don't specify reducer class

}

It will make framework
use an identity reducer.

3) Bloom filtering

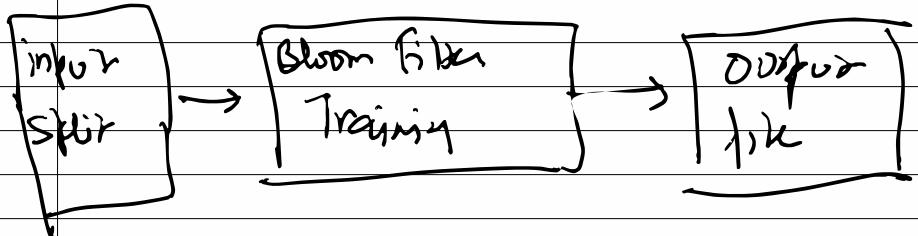
Use bloom filter to get the set

↳ kind of join operation → we don't care
about the values on right side.

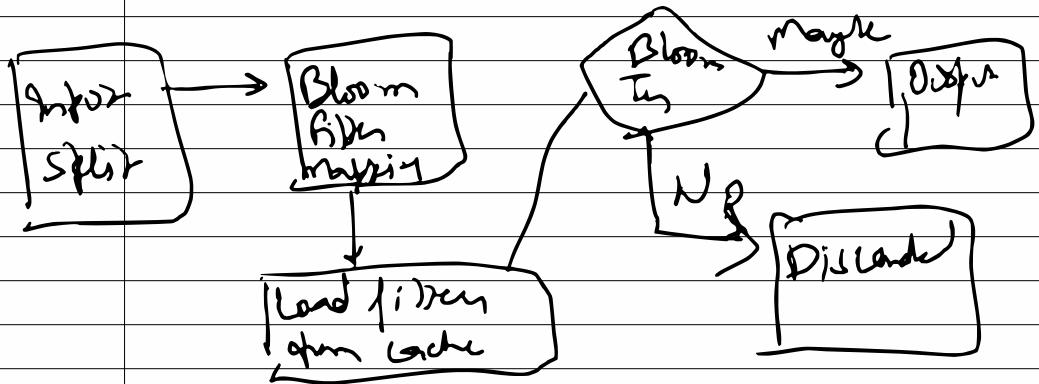
Bloom filter needs to be chained over a list
of values.

↳ (nothing data object is stored) in HDFS

Step 1



Step 2



Use cases

- 1) Removing most of non watched values
- 2) Prefiltering data set for an expensive set membership check.

Top Ten

Intent →

Get top k records, according to the ranking scheme

Motivation

↳ outliers
best records

Needs a comparator function which can compare the records.

No of output should be fewer than number of input records.

Uses both mapper & reducer.

Mapper will find local top k and all the local top k will compete for global Top k.

↳ since k is small → only need 1 Reducer.

class mapper :

setup () :

initialize top ten sorted list

map (key, record) :

insert record in top 10 sorted list if
length of array is greater than 10 then
truncate list to a length of 10

Cleanup ()

{

for record in top sorted 10 list :

emit (null, record)

class reducer

setup ()

initialize top 10 sorted list

reduce (key, values)

for record

truncate record to top 10

for record in records

emit record

Distinct Pattern Description

Get only distinct values from the set.

Mapper requires

Map (key, value)

context.write (value, null).

Reducer (key, list < value>)

{

Emit (key, null)

}