Scalable parallel processing with mapreduce

Understand the challenge of parallel processing

MapR using MongoDB

Manipulating large data

↳ few possible points of interaction

↳ few conflicts & less management

I/O bottle neck ——prevent——→ fast & efficient processing

1) One solution build super computers

↳ Not practical

↳ 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 cycles 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.

MapReduce

- little interdependence among parallel processes & keeping data and computation together.
- Developed for distributed computing & Patented by Google.

Map Reduce on MongoDB clusters

Aggregate functions like → sum, count, max, average

Mapreduce
↳ MongoDB
↳ Hadoop
↳ HBase

* Import csv file in mongoDB

1)
```
mongoimport --type csv --db mydb --collection
myse --headerline Name-of-file.csv
```

formats) db.myse.findOne()
↳ will get one document.

MapReduce has following 2 parts
1) map function
2) reduce function

The two functions are applied to data sequentially though the underlying system frequently runs computation 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 same operation of another node.

MapReduce framework 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_high});

    }


var reduce = function (key, values)
{
    var high_price = 0.0;
for (var i=0 ; i < values.length ; i++)
    {
        if (values[i].stock_price_high > high)
            { high = }
    }
```

return { highest-stock-price: high };

};

finalize function → to do additionally 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

→ goto ../sbin/

./start-all.sh → start hadoop

cd ../bin/

open http://localhost:50070

cd ../sbin/

./stop-all.sh → stop hadoop

MongoDB

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 → increas locality and availability.

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

Primary receives all write operations

Rest 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 the operations to their data set.

↓

Secondary data set reflect the primary data set

4) If primary is unavailable

↳ replica will elect a secondary to be primary.

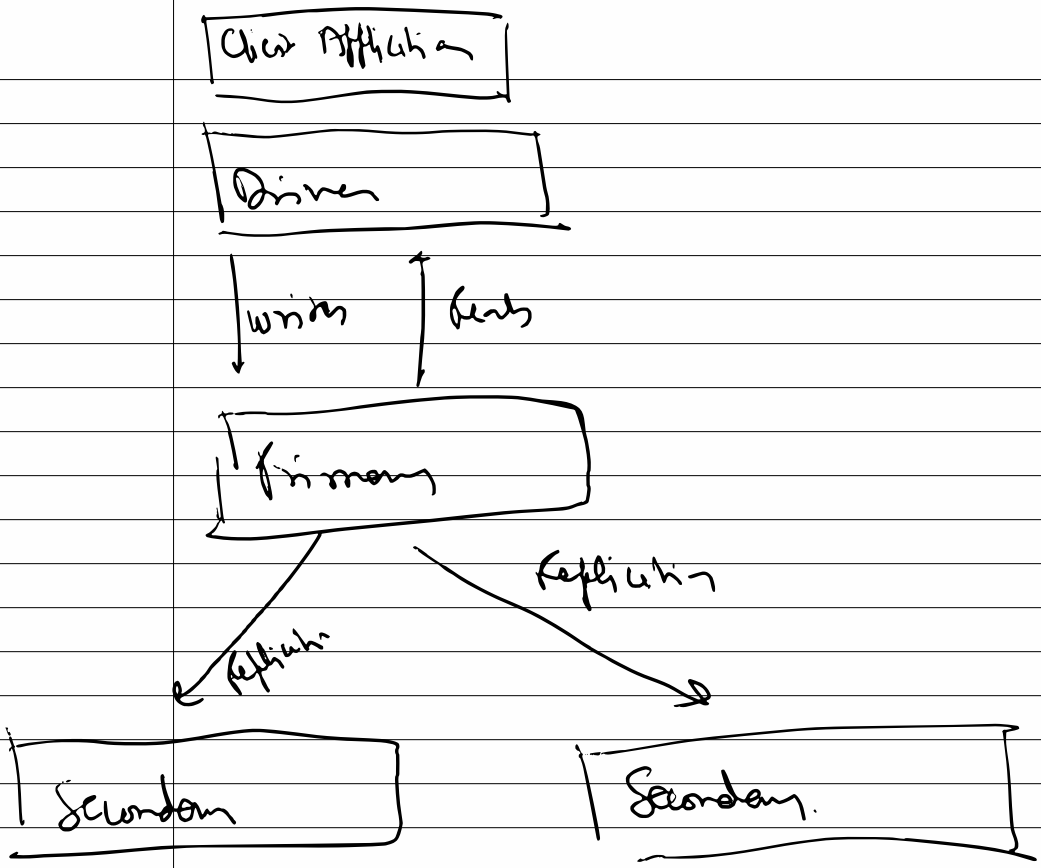
By default client reads from primary, but client can specify to read from secondaries.

Arbiter :-

Extra mongod instance as an arbiter.
Arbiter 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 and 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 the member to become primary.
- The secondary which will get maximum vote in an election will become the 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 Mongo instances that provide redundancy and high availability

The members are
- ↳ primary → all write operations
- ↳ Secondary → replicates from primary to maintain an identical data set
  - ↳ 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 set can have upto 12 members. Only 7 at a time can take part in election.

Minimum requirement
- ↳ primary, secondary, arbiter.

Must however keep 3 member
- 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 reside in secondary data centers or to serve as a cold standby.
     Also can vote.
2) Hidden replica set member
   - Prevents application from reading from it. allows it to run apps that require isolation 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 can not trigger elections

Maintains copy, accepts read operations and votes in election.

Hidden Replica Set Members

Maintains 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 that replication lag is minimal or non-existent.

Delayed Replica Set members

makes copy
Reflects a delayed state of set.
So helpful in accidental recovery of data.

- Must be priority 0 → can't become primary
- should be hidden → no read operation

Vote in election

Delay → equal to or greater than maintenance window
smaller than capacity of oplogs.

Arbiter

Does not have copy & can't become primary
↳ used to add vote in election
↳ allows for uneven number

Important—

↳ Don't host arbiter on system that also host primary and secondary

↳ only add arbiter to even number of members
else
may suffer from tied elections.

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 throughput.

sharding solves the problem with horizontal scaling

Purpose of sharding

↳ high data & 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.

Reduces the amount of data each server needs to store.

1 TB → 40 shards → 25 GB per shard.

MongoDB supports sharding

↳ configuration of sharded cluster.

Shard cluster has the following components:-

1) Shards
2) Query Routers
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 Routers

↓

Mongos instances

↳ interface with client application

↳ direct operation to appropriate shards

→ preprocess → targets operation to shards

↓

return result to user.

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

3) Config Servers

↳ store clusters metadata

↙

mapping of data to the shards

Query routers 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.

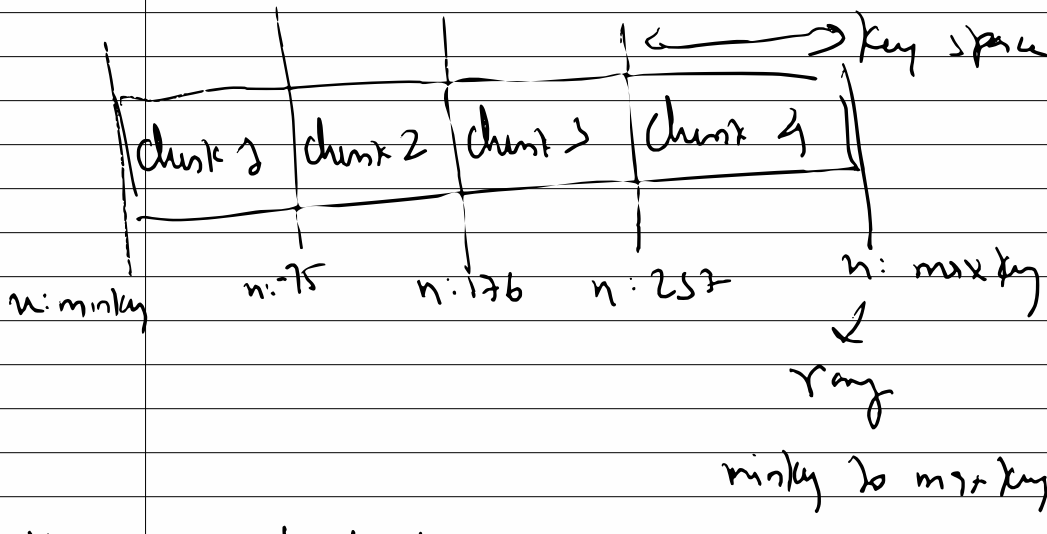
To distribute the shard key values into chunks, MongoDB uses —

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 → devide into non overlapping ranges → chunks



Hash Based Sharding

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

In hash based sharding close values may have different hash codes and may fall into different chunks → more random partitioning of values

Maintaining a Balanced Data Distribution

↳ Addition of new data or addition of new servers can result in data distribution imbalances 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.

Insert & update triggers split.

↓

efficient meta-data change

To create splits mongoDB doesn't migrate any data or affect any shards.

Balancing

If the distribution of sharded collection is uneven the balancer process migrates chunk from shard that most data to shard that has least data.

This is a background operation
↳ During this all request to chunk data owns 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:—

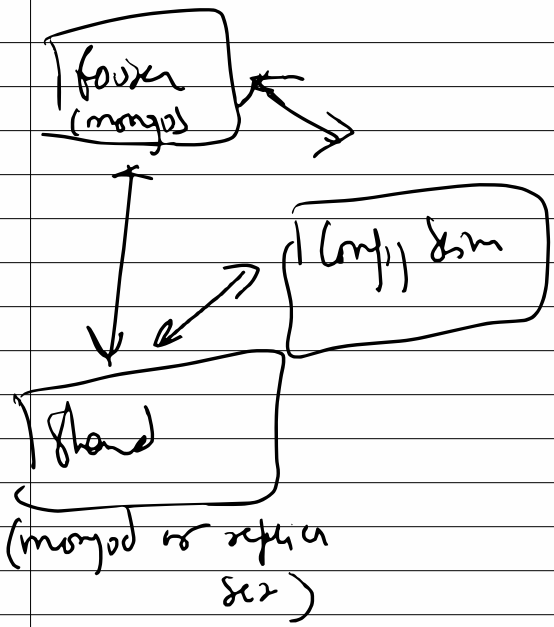
- 3 config servers.
  - ↳ Each of different machine
  - ↳ A single sharded cluster must have exclusive use of its config servers.
    - ↳ If you have multiple sharded clusters ↳ need group of config servers for each cluster.

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

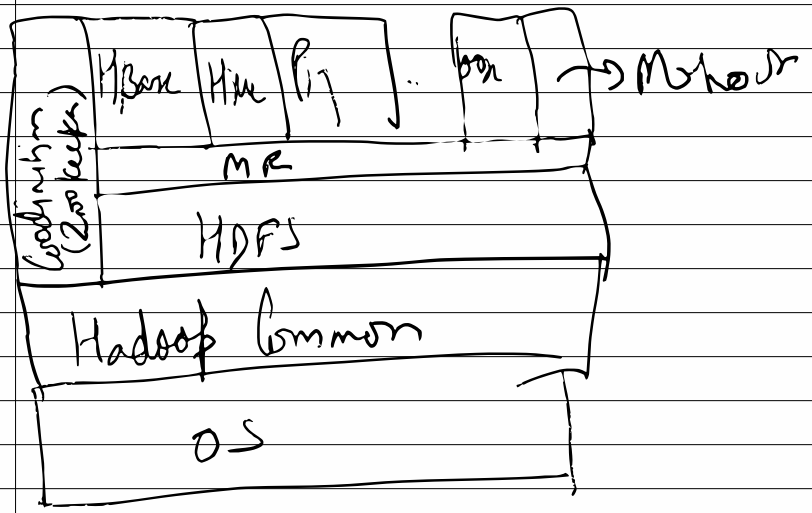
→ One or more mongos instances.

Sharded cluster Test Architecture

1 config server
1 shard (or more)
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 → MapReduce
- Common utilities
- A column oriented data storage table (HBase)
- 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.

MapReduce

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 & combining the key value pairs by keys.

→ Shuffled data partitions are copied to reducer machines.

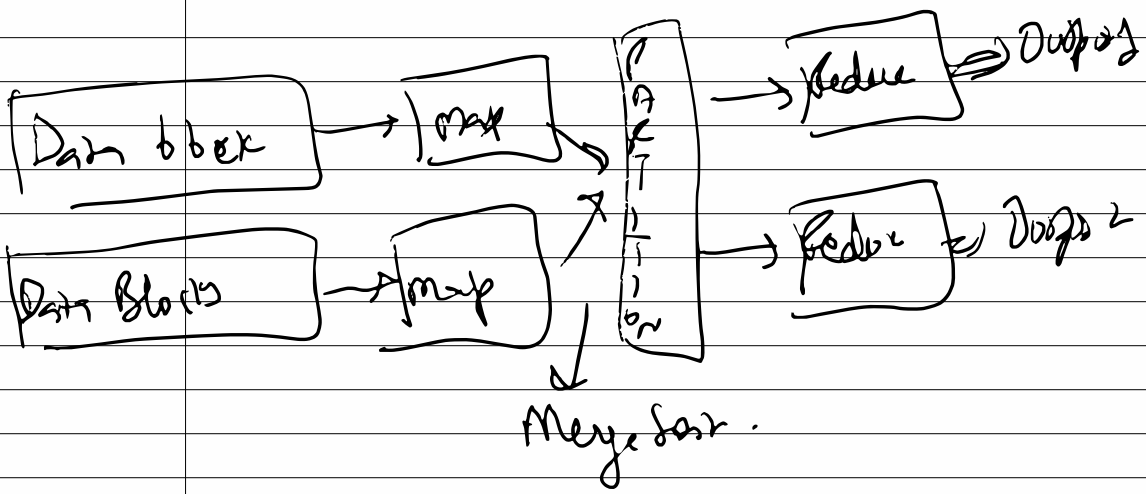
Then the reducer task will run on shuffled data and generate final results.

↓

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 file system 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 jobs into smaller tasks
↓
which is then assigned to Task trackers 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 million of 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 & 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

↓

It requires application to explicitly built for this.

HDFS is a distributed file system

↓

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

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

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

Normal file system → 4Kb / 8Kb block

HDFS → 64 MB by default.

↓

cluster admin raise it to 128 or 256 MB

↓

or even

1 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 drops below

↓

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

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

Daemons

Namenode	1 per cluster	stores file metadata
secondary Namenode	1 per cluster	performs internal namenode transaction 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 datanodes with additional disk capacity

⇓

or by adding disks to existing datanodes.

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 → cost → performance

multiply copies → protect → data loss

↑

can access from anywhere

↑

improves performance.

Name Node

↳ stores filesystem metadata

↳ maintains complete picture of filesystem

client connect to the namenode → filesystem operation

although ↳ data is streamed to/from datanodes directly.

Datanodes report their status to namenodes in a heartbeat.

↳ means → at any given time → namenode 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 changes.

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

---

Datanode failure

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

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

Downside ⇒ while starting → namenode waits to recive BR

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 applications

- Hadoop is not a good fit for large graph data sets.

Not good for data that is not recognized as big data.

↳ small datasets

↳ datasets with processing that requires transaction and synchronization.

Hadoop Architecture

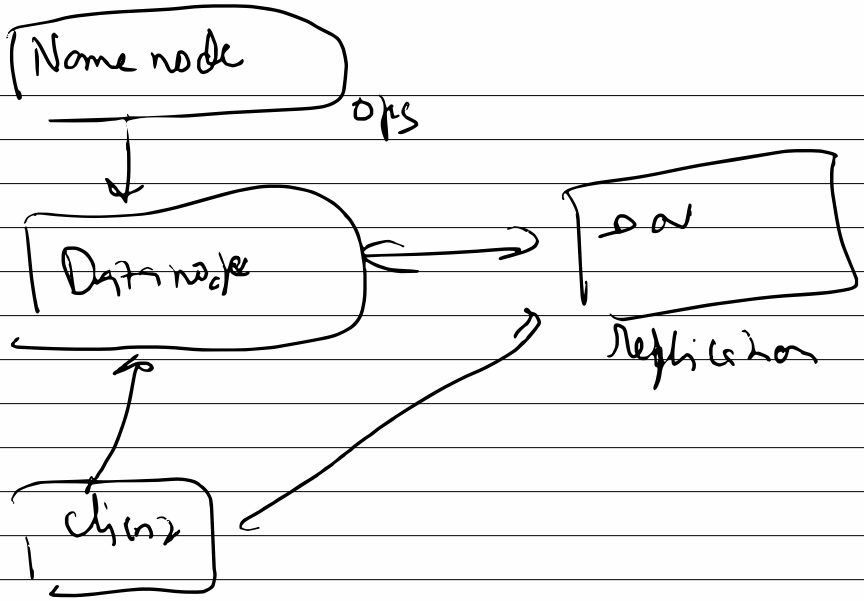
Running Hadoop on a Single-Node Cluster

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
Is designed to be deployed on low cost hardware.

It provides high throughput access to application data and is suitable for applications that have 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.

check -ssh
sudo apt-get install ssh

ssh localhost.

formatting name node.
name node format

jps tool to check processes running

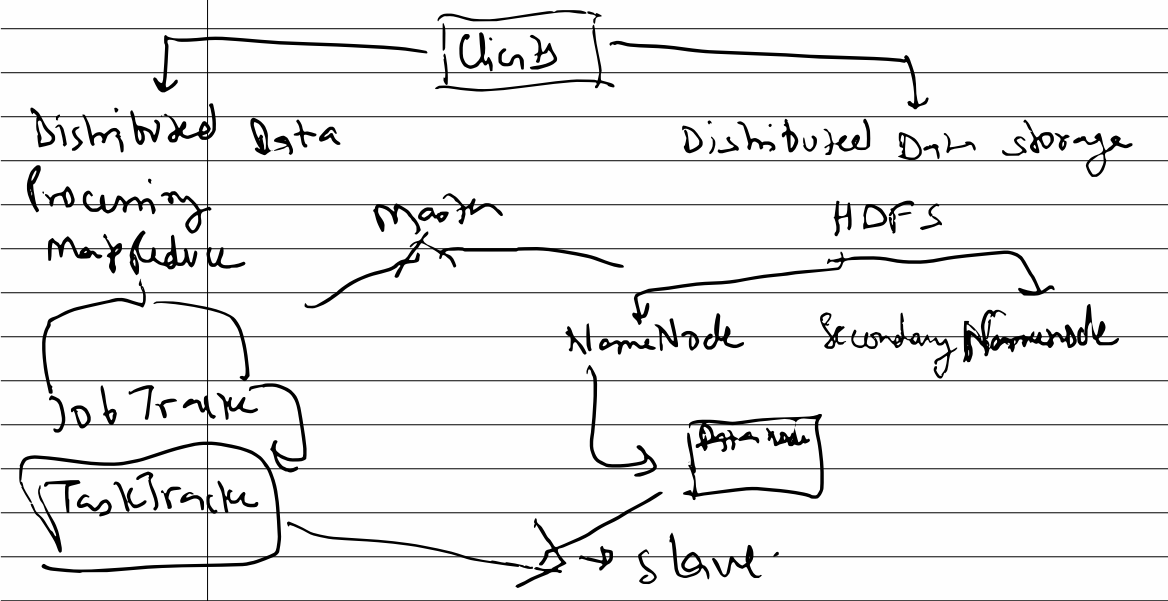
NameNode
Secondary NameNode
DataNode
Resource manager
jps
Node Manager

stop → stop-all.sh

hen word count

hadoop jar mr.jar wordcount /down/books
/output

Hadoop Server Roles



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 JobTracker oversees and coordinates the parallel processing of data using MapReduce

2) Slave Nodes

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

Each slave node runs both DataNode & TaskTracker

TaskTracker slave of JobTracker
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, JobTracker & NameNode both.

With medium to large clusters → 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 those 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 → dfs.replication in file hdfs-site.xml

Name Node

1. The client breaks files into 3 blocks

- client consults NameNode (TCP Port 9000) and receives list of 3 data nodes to copy
- client writes directly to data nodes (TCP 50010)
- The data nodes copy & replicates the block

- 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 bulky flow in-rack whenever possible

↙

higher bandwidth
& lower latency

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

This somebody is Name node.

HDFS writes

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

Rule → 2 copies on same rack
another copy on different rack.

Client sends to DataNode 1 to check for 5/6 → then 5 sends to 6 → after acknowledgement is received → client writes

Data node passes data along as they receives

TCP – 50010

Pipeline writes

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

Data node receives the block → same time pushes a copy to the next node in the pipeline

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

Now, name Node updates the metadata of that 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 ~~of~~ of machines → increase density of machine

Name Node

- Data node sends heart beats
- Every 10th heart beat is a block report
- Name node builds metadata from Block reports
- If Name node is down → HDFS is down

All system metadata → Knows health of Data nodes → coordinates access to data.

↓

central controller of HDFS

Don't hold data

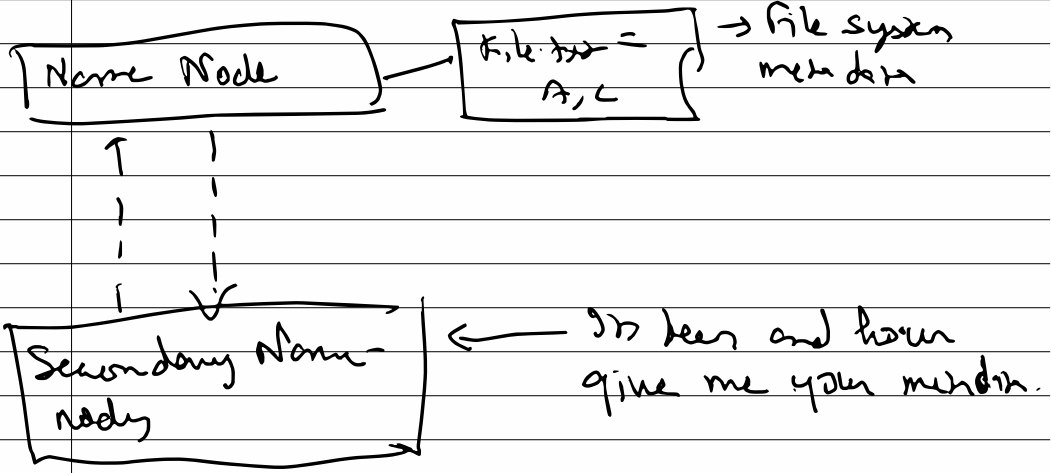
Knows
- which blocks makes up file
- where these blocks are in Datanode
- points client to datanode to talk.

Data nodes sends heartbeat to name node every 3 seconds → TCP handshake → 9000
10th is Block report.

Replicating missing replicas

- Missing heartbeat signifies loss
- Name node consults metadata
  - finds affected data
  - consults rack awareness script
  - initiates replication of 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)

↓

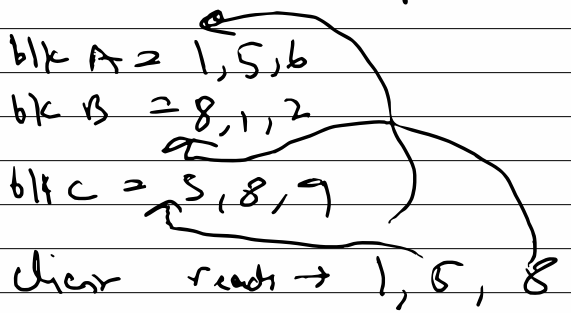
combines this info in fresh set of files sending back to namenode ⇒ keeping a copy for itself. ∞

Can be used in case of namenode failure.

In case of busy clusters 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 # TCP - 50010

↑

default port number for datanode daemon

Data node reading like from HDFS

- Name node provide rack 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 heart beat and task status back to the job tracker

↓

As map completes

↓

Each node store the result of its local computation in temporary local storage → intermediate data.

↓

Next step

↓

Send this data to reduce phase.

4) Data is not local

↳ If data node is busy → Name node will assign the map job to a diff. node in same 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 MapReduce of HDFS → rate 1 MB/sec.

The amount of traffic balancer uses is pretty low → 1 MB/sec.

↓

can be changed in

dfs.balance.bandwidthPerSec

in

hdfs-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 timestamp.

So we emit

(K2, V2) = ({symbol, timestamp}, 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();
    if(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 K2 & V2 may go to different reducer

```
public class NaturalKeyGroupingComparator extends
                WritableComparator {
    protected NaturalKeyGroupingComparator() {
        super(StockKey.class, 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 partition data 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;
}
}
```

4) 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(SSJob.class)
job.setPartitionerClass(NKP.class);
job.setGroupingComparatorClass(NKGC.class)
job.setSortComparatorClass(CompositeKeyCom.class);

// Map Class

job.setMapOutputKeyClass(StockKey.class);
job.setMapOutputValueClass(DoubleWritable.class);

// reducer output

job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);

// input & output format class

job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);

// set mapper & reducer 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 Job conf object with config parameters of MR job and passes Job conf to JobClient.runJob()
↳ Getting MR drivers of one MR job after another.
↳ input path of other → output path of previous
↳ Delete intermediate data generated at each step.

2) If non linear & complex dependency

↳ Job and JobControl classes.

↓

MapReduce Job

Instantiate 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 → spawn 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 classes to simplify the composition of pre and post processing.

Chaining

↳ [MAP | REDUCE]+

- ChainMapper.addMapper()
- ChainReducer.setReducer()
- 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 results in heap space issue

MAP

key → hour of day
value → comment length

REDUCE

Adding all values to in memory list sort & calculate median and SD

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

key → hour of day
value → tuple ⇒ {median, SD}

Combiner

→ can not be used

2) Complex Solution

Keep count of each value

Combiner can be used

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

↳ maximum comment length

instead of

$T(n) = O(n)$

Mapper:-

key: hour of the day

value: sorted map with key

↳ comment length, count 1.

Reducer

in Value: Result Tuple { Median, SD }

Combiner

Aggregate the sorted map entries for each local mapper.

2) Inverted Index Summarization

↳ commonly used as an example for MapReduce analytics

Intent

Generate and index from 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: desired field
value: unique identifier

Combiner:

Optional. → same as reducer as its groups

Partitioner

: can be used for load balancing

Reducer

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

Performance → good parsing, removing 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.

Intent

↳ Retrieve count summarization of large data sets.

↓

Instead of sending to reducer to count

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

↓

No reduce phase, no summation!

No output ⇒ (Emp_id, 1)

||

create counter with emp-id & increment by 1

||

After the job save the output anywhere.

Caveats

- Number of counters should be dozens or hundred or more.
- or they will bog down the Job Trackers.

When

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

Output

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

Performance

- Very fast

depends on number of map tasks being executed.

mapper :-

```
context.getCounter(STATE_GROUP, State)
    .increment(1);
```

```
Main ()
{
  int code = job.waitForCompletion(true) ? 0 : 1;
  if (code == 0)
  {
    for (Counter counter : job.getCounters().
            getGroup(MapperClass.STATE_GROUP))
    {
      SOP(counter.getDisplayName() + "-" +
          counter.getValue()));
    }
```

7) Bloom Filters

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 Java 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, Maybe

Never Definitive Yes.

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

⇓

implementation to counter this limitation

⇓

Counting Bloom filter

↳ But, uses more memory.

As more data added ⇒ false positive increases.

Can not be resized like other data structure.

Following variables are used in more detailed explanation

m → The number of bits in the filter

n → The number of members in the set.

p → The desired false positive rate

k → The number of diff hash function used to map some element to one of the m bits with uniform random distribution.

Use Cases

1) Representing a data set
   - large data set
2) Reduce queries to External database
3) Google Bigtable

Down side
- Is false positive rate.

7> MapReduce 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 results of a deduplication

This differentiates filtering patter 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 words in text.

Sampling

↳ One common application of filtering

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

↓

Use for much smaller representative of
whole data ⇒ can be easily analysed.

Sub Sampling

is 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 Pattern

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

```
function()
{
    if (true)
        emit (key, value)
```

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

No reducer → only evaluate then keep or discard.

Same key value pair → record is unchanged.

Known User

1) Close view of data
2) Tracking a thread of events
3) Distributed grep
4) Data cleansing
5) Simple Random Sampling
6) Removing low quality 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 the map data.

Example:

1) Distributed grep :-

Global KE finds.

```
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)
  {
    if (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 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 reducers to 1 and don't specify reducer class

↓

It will make framework use an identity reducer.

3) Bloom filtering

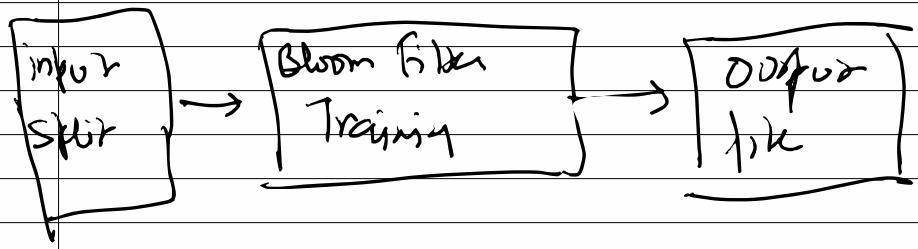
Use bloom filter to get the set

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

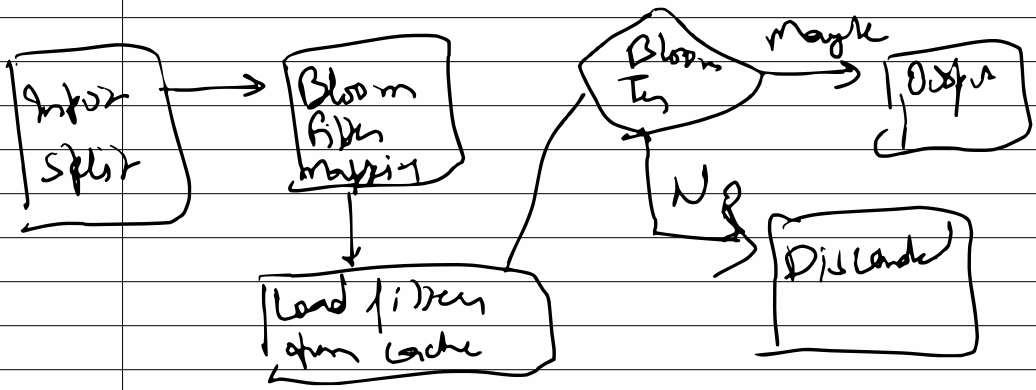
Bloom filter needs to be trained over a list of values.

↳ Resulting data object is stored in HDFS

Step 1



Step 2



Use cases

1) Removing most of non matched 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.

Use 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)
        sort records
        truncate record to top 10
        for record in records
            emit record
```

Distinct Pattern Description

Get only distinct values from the set.

Mapper required

```
Map (key, value)
context.write (value, null).

Reduce (key, List<value>
    {
    Emit (key, null)
    }
```