HADOOP Big Data

Size

Variety — Volume — Velocity — Veracity

Variety
- Structured (RDBMS)
- Unstructured (pdf, doc)
- Semi Structured (xml)

Volume

Velocity — Speed at which data arrives

Veracity — correctness of data

Traditional Solutions

Cost — RDBMS — Scalability?

RDBMS — Format

Storage — Computation

Not Scalable horizontally.

Requires low-level knowledge

Not for mainstream

High Performance computing for

Grid computing — Low Volume

Vertical Scalability - more resources (CPU, RAM, High speed SSD on one machine)

Horizontal Scalability - Cluster of machines which provide distributed storage & computation.

HADOOP -

- Supports huge volume
- Storage efficiency. (Rack Awareness)
- Good data Recovery solution (Replication)
- Horizontal scaling
- Cost effective (commodity hardware)
- Easy for Programmers.

Hadoop achieves speed in processing & storage by distributed storage & computation.
When a file is loaded in HDFS, it is broken into blocks which are stored on different nodes.
When the data needs to be processed the job that processes the data is sent to different nodes on the cluster where it operates on distinct chunks of data parallelly. The results of each parallel computation are then combined to give a final result.

- Hadoop distributes the data evenly across the cluster and then brings the computation to the data.

Hadoop { HDFS – Storage
MapReduce – Programming model for distributed processing of large datasets

Hadoop is framework for distributed processing of large data sets across cluster of commodity computers

Note: Hadoop is good (performant) when you have ~~large~~ small number of large files but if you have large number of small files Hadoop will drop in performance.

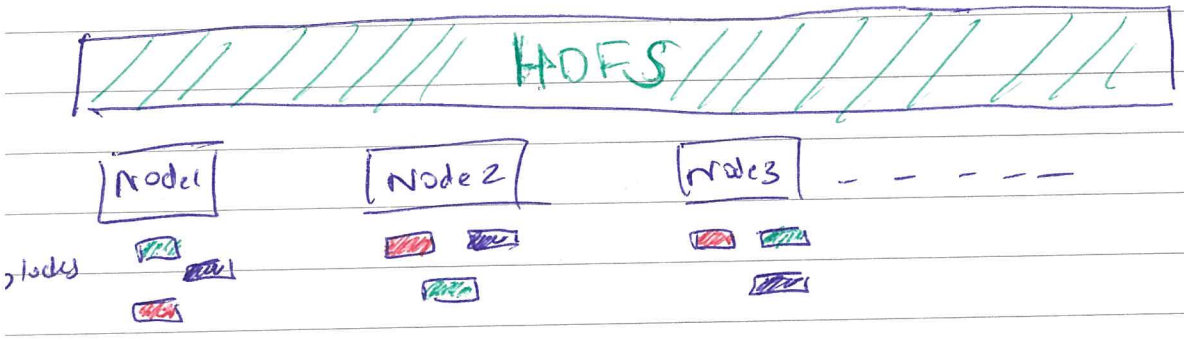
Rack awareness - Hadoop tries to keep data as local to the rack as possible to preserve network bandwidth across Racks when doing shuffle phase. It also tries to assign ~~jobs~~ mapper to task which are shortest path to the data present in the input split assigned to that the mapper.

HDFS:

Microsoft - FAT32 - 4GB File limit 32GB Volume limit
NTFS - 16EB File limit 16EB Volume limit

Apple - HFS - 2GB File limit 2TB Volume limit
HFS+ 8EB File limit 8EB Volume limit

Linux - ext3 - 2TB File limit ~~8F~~ 32TB Volume limit
ext4 - 16TB File limit 1EB Volume limit
XFS - 8EB File limit 8EB Volume limit



Node1: does not know about data on node 2
However HDFS file knows the different blocks on different nodes that make up the file. The blocks are of size 128MB. By default the HDFS will store 3 (default) copies of each block that make up the file on different nodes (possibly on different racks) to recover from a hardware failure.

dfsadmin - refreshNodes

Benefits of HDFS

- Supports distributed processing
- Replication
- Scalability (Easily support expansion by simply adding more nodes)
- Cost effective.

Commands

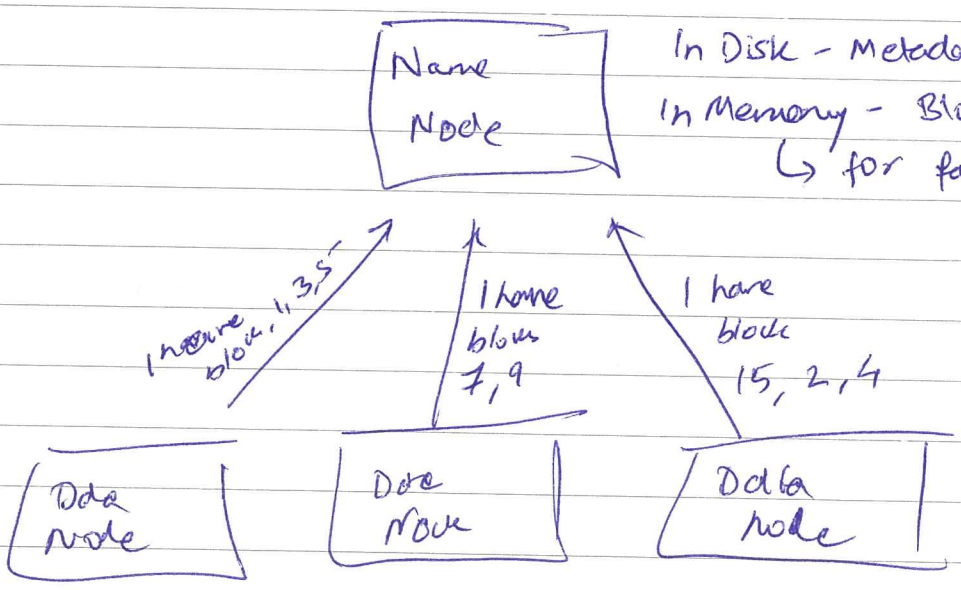
fsck - File System check

Master — Name Node
- keeps track of all files in HDFS.
- keeps information on list of blocks that belong to same dataset.
- Location of blocks (which datanode has which block) ~~(in memory)~~
- metadata (owner, permissions, etc....)

Slaves — Data Node 1, DataNode 2

Each datanode knows ONLY the block information that it stores.

Name node persists all the metadata for a file ~~except~~ on the disk except for block locations. These are kept in memory of the name node. This information is stored on hard disk of data node.



Note: Explanation of all properties

https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml

HDFS configuration

core-site.xml - made available to all nodes of the cluster

fs.defaultFS → location of name node

hdfs-site.xml

dfs.namenode.name.dir - location in local file system where name node can store its files (metadata)

dfs.datanode.data.dir - location in local file system of data node where it stores the blocks.

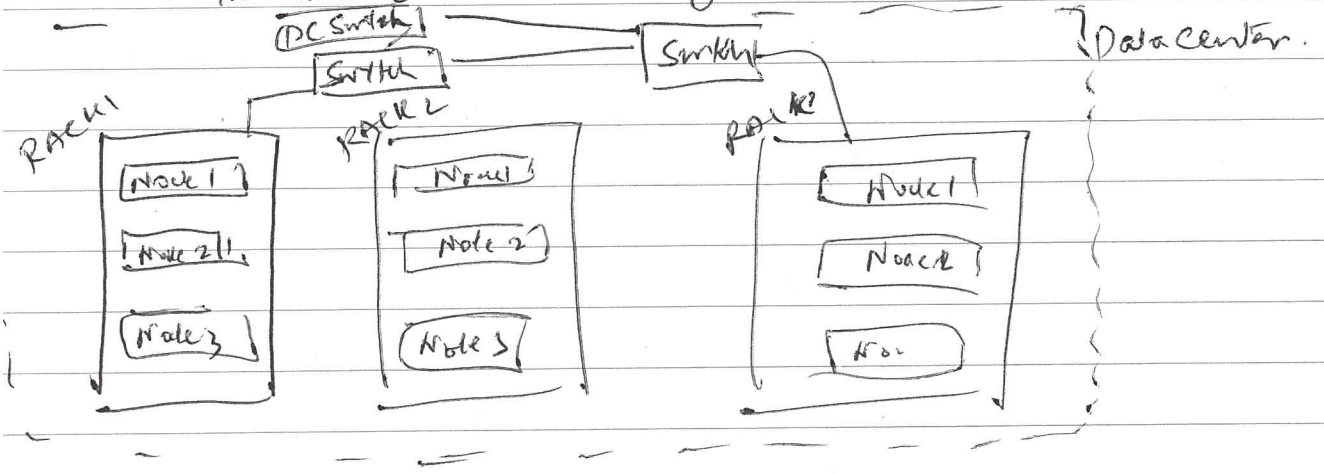
dfs.namenode.http-address - HTTP URL of name node (port 50070)

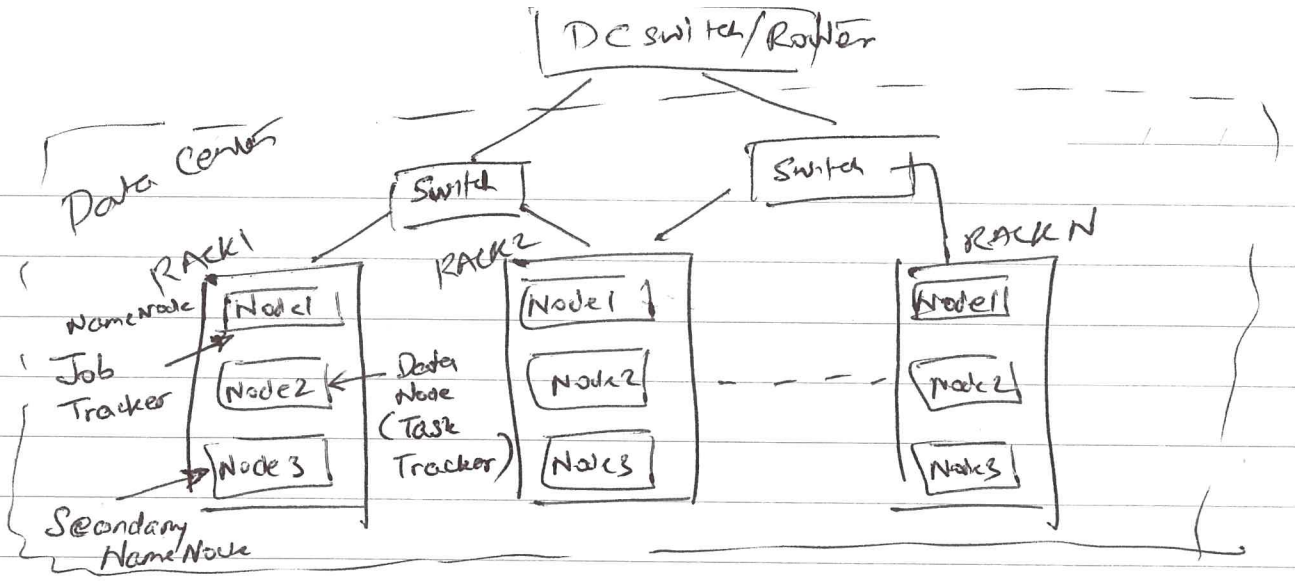
dfs.replication - replication factor for files in HDFS.

Masters file - name node (hostname or IP)

Slaves file - list of data nodes (host or IP)

NOTE: Name node is normally the most powerful Node in the cluster with high RAM.





Secondary NameNode (Checkpoint Server)

→ Name Node holds a snapshot of the entire HDFS in memory.

→ Client always communicates to NameNode after asking for certain data but the NameNode knows which DataNode has the data which the client requested so the NameNode gives a list of block locations of the data & then the client communications directly with dataNodes to serve the actual blocks to client.

→ Secondary Name Node is not a high availability node. It does not automatically take over if Name Node fails.

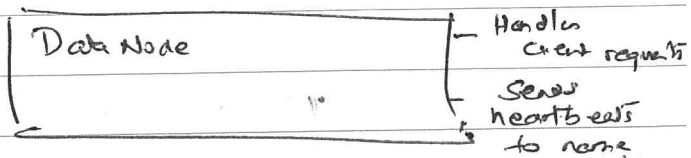
→ It is more like a backup of the Name Node which is done periodically. In case of Name Node failure the Sec-Name Node has a complete snapshot image of entire HDFS in order to restore.

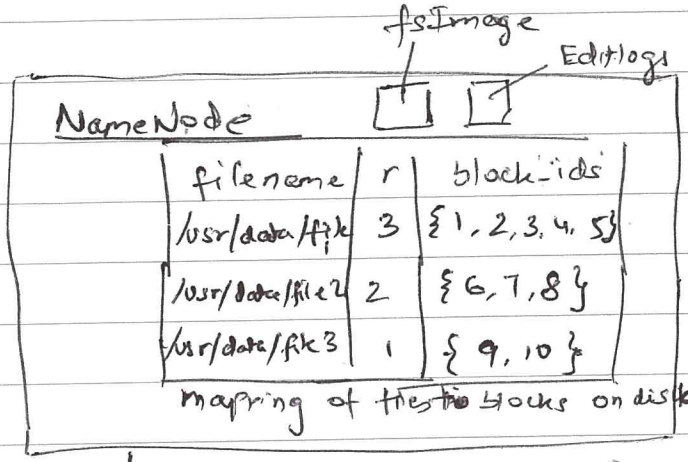
→ By supplying HDFS with knowledge of our network topology we can make HDFS RACK aware means the blocks of data will be replicated on DataNodes sitting in different RACKS. This protects us from a RACK-failure.

Name Node RPC : 8020
Name Node HTTP : 50070
Data Node HTTP : 50075
Secondary name node : 50090

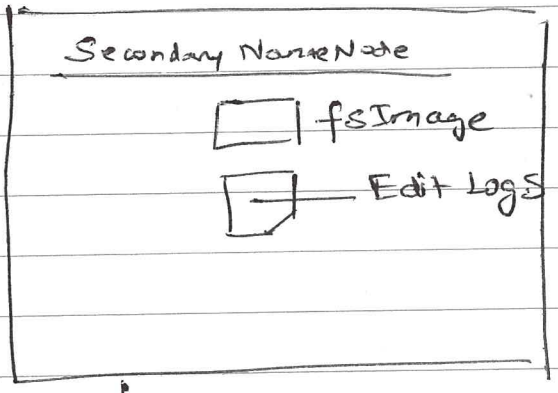
→ RACK Awareness also allows Hadoop to keep blocks local to a rack in order to minimize cross rack communication & preserve network bandwidth

HDFS Internals





- Contains entire filesystem metadata (fsImage)
- Client communicate
- Monitor health of DataNodes in the cluster.



- Housekeeping / Backup of name node
- Metadata (fsImage) backup & rebuild (update fsImage with Edit logs) and communicate updated fsImage to NameNode

→ Name node maintains a list of all changes to HDFS in an Edit log. (in memory)

→ Secondary Name node periodically (1 hour default) gets the Edit logs from the name node & updates its fsImage with the changes from the Edit logs. It then uploads the fsImage back to NameNode

→ In case of failure of Name Node, the fsImage on the Secondary Name Node can be used to rebuild the Name Node.

→ Users are created only in Linux (We cannot create users in hdfs)

→ Ultimate Owner of HDFS is a user named hdfs

→ Hadoop tries to locate user space in /user/<username>

eg: for logged in linux user root Hadoop assume user space /user/root

for logged in user ec2-user hadoop will take in /user/ec2user

→ HDFS will try to place blocks belonging to a file in data nodes close to each other (shortest path) in order to improve performance.

Rebalance: If all datanodes start getting full, we can add more datanodes to a cluster but this won't still improve performance since the existing data nodes are still full.
- Balancer Tool: When runs it will evenly distribute the data on existing data Nodes across the cluster including the newly added nodes.

Replication Management: Every 3 seconds the data nodes send a heartbeat to Name Node.
- Every 10th heartbeat the DataNode sends a block report to Name Node. The Name Node uses the block report to see if a block is under/over replicated.
- If over replicated the Name Node removes the additional copies. (mark to remove)
- If Under replicated the Name Node adds the blocks to a priority queue. (Files with less number of blocks are higher in priority).
- Also if a Name Node determine from Block report that a block is corrupted it will mark the block for removal & replicate the good copy

Tools:
DFSAdmin

Note : 9page search "Hadoop Command Guide"

HDFS paths start with hdfs://server:port/<path>

NN : 50070
DN : 50075

HDFS Tools

Dfs Admin :

- Safemode - Read only mode (during upgrade)
- refreshNode - During decommissioning of DataNodes.

Rack Awareness

- 打

Configuring Rack Awareness

Core-site.xml

topology.script.file.name → point to script.sh

```
if $1 = "192.168.0.3"
    echo -n "/rack1"
else
    echo -n "default rack"
```

OR

have .txt file

read this file to return rack based on IP address passed to the script

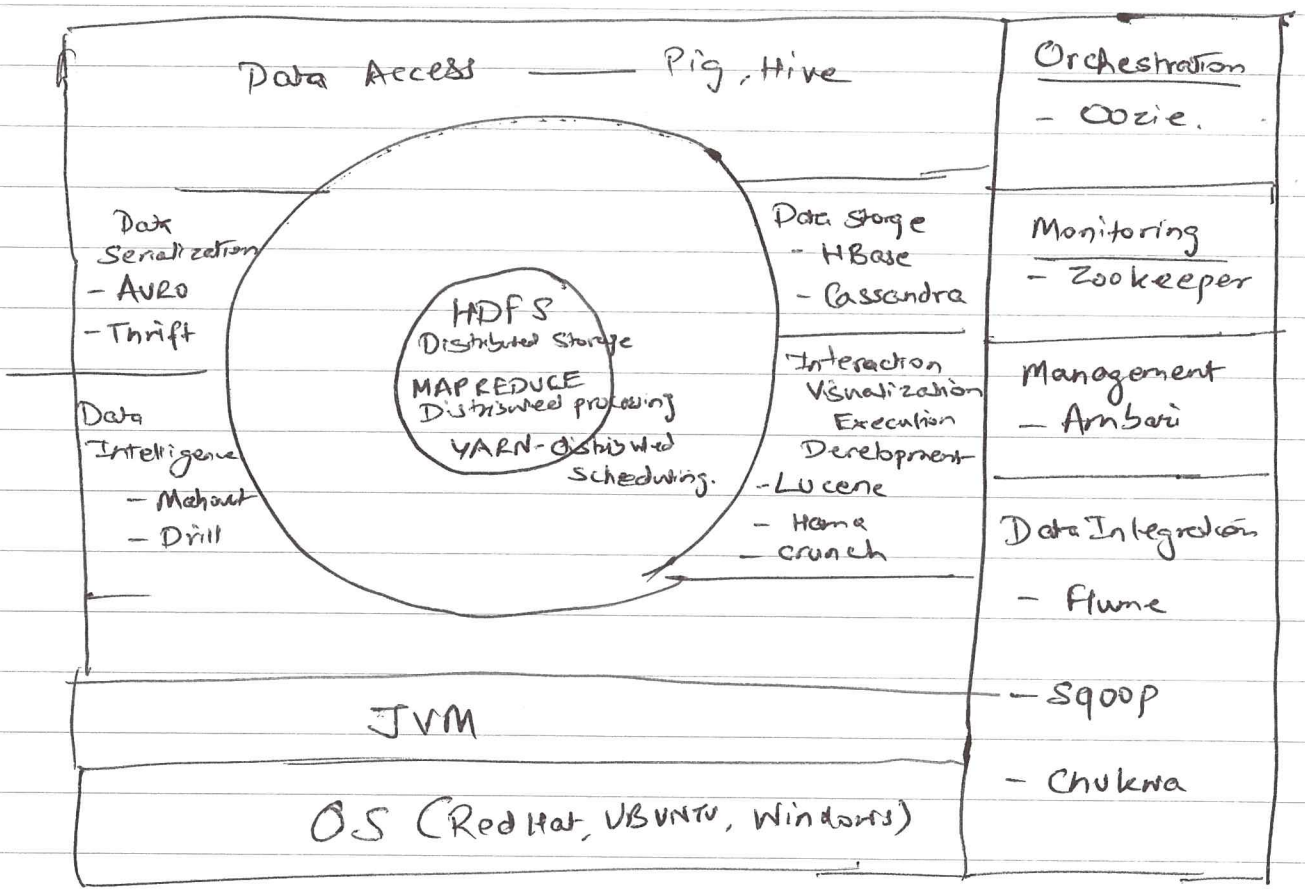
10.1.1.1	/dc1/rack1/n1
10.1.1.2	/dc1/rack1/n2
10.1.2.1	/dc1/rack2/n1
10.1.2.2	/dc1/rack2/n2

Decommission DataNodes :

→ /etc/hadoop/<ver>/conf/exclude Edit this file

→ Add IP address of node to decommission to this file

HADOOP TECH STACK



HADOOP Core - HDFS, Map Reduce, YARN (2.0)

HADOOP ESSENTIALS - Pig, Hive, HBase, Avro, Zookeeper, Mahout, Oozie, Sqoop, Ambari

HADOOP INCUBATOR - HCatalog, Spark.

HDFS - Distributed Storage

MapReduce - Distributed Processing

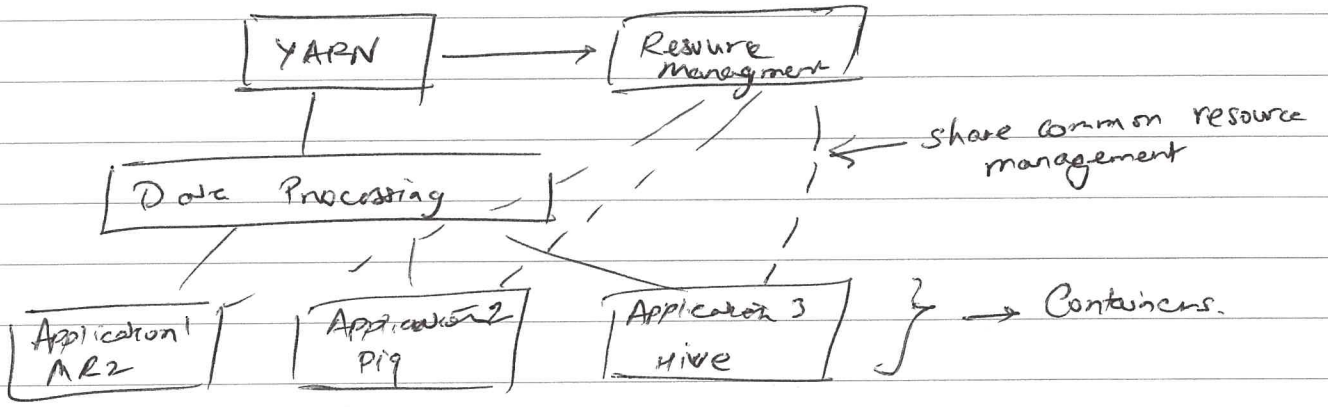
YARN - (Yet Another Resource Negotiator) (Also called MR2)

- rewrite of MapReduce 1
  - → Breaks down the Job Tracker in MR1 into seperate daemons for managing resources & managing applications (which are containers for jobs).

In Hadoop 2.0
YARN now performs cluster resource management.

- MapReduce 2 solely focuses on data processing via YARN
- Other tools (Pig, Hive) that perform map reduce also perform data processing via YARN

- YARN provides a generic processing platform not constrained to mapreduce.



Pig :- High level data flow scripting language
- developed by Yahoo.
- getting data, loading, transformation & storing or returning results
- 2 components (Pig Latin - Programming language)
Pig Runtime - compiles Pig latin and converts it to MapReduce job to be submitted)

HIVE :- Database or Datawarehouse built on top of HDFS.
- Hive QL allows querying data similar to SQL.
- converts queries written in HQL to MapReduce jobs

HBase - columnar database on top of HDFS to allow interactive processing & updates & seek particular data in the datasets
- based on Google ~~File~~ Big Table.
- NoSQL
- ~~Can~~ can be queried from Pig, Hive

Cassandra - based on Amazon Dynamo DB.
- Real time, interactive transaction processing

HCatalog : - Shared schemas. Allows Pig, Hive Sqoop to interoperate
- allows to have consistent views of data across different tools like Pig Hive Sqoop.

Avro :- Data serialization standard
- Allows to get data from Application & package it so that other application can deserialize it & use it.

Thrift - Language independent serialization framework.
- Allows data serialized from objects in one programming language to be deserialized and used in another.

Drill :- Interactive analysis of nested data.

Mahout - machine learning
- recommendation (collaborative filtering), clustering (grouping related text/document ~~data~~ together), categorize related text/documents

Sqoop - Allows data exchange between RDBMS & Hadoop.
- Pushing data from RDBMS to hadoop for Archiving
- Pushing result from jobs in hadoop into RDBMS.

Flume - Streaming log data or any data into real time in hadoop.

Oozie - workflow library
- allows running Pig script, Hive query & store result in Sqoop

Zookeeper - distributed service coordinator
- allows synchronization between services running on different nodes across the cluster.
-

Ambari - Centralized cluster management
- Installing, Starting, Stopping, configuring services
- monitoring.

hadoop-env.sh : sets up Hadoop Env settings such as Java path and security

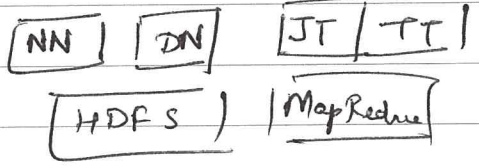
core-site.xml : Define NN & DFS temp dir

mapred-site.xml : No of mapper, reducers other MR settings

YARN

Hadoop 1.0

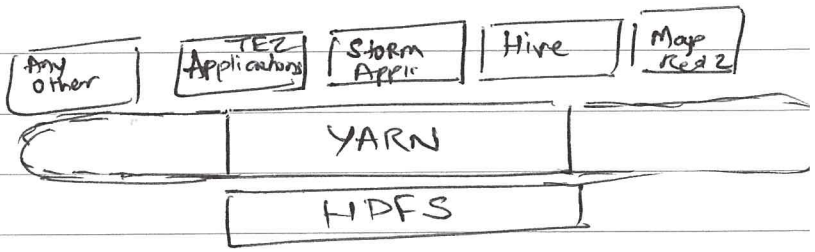
- Batch
- Only MapReduce
- Resource & Job execution tightly coupled



- NN is the only place where Jobs are queued, scheduled for execution which results in single point of failure
- Hard partition of resources on DN into Map & Reduce slots.
- Failure of NameNode kills all queued & running jobs.
- Also since single Job tracker which run on single node is burdened with lot of responsibility to keep track of Task Trackers, queueing, scheduling of jobs and monitor map/reduce job execution. This results in limited scalability. Thats why HADOOP 1.0 can support 4000 nodes only & 40000 simultaneous jobs.
- Client & Cluster must be on same version of hadoop.

Hadoop 2.0

- Interactive, Real time, Batch
- Resource management seperate from processing
- Application on top of YARN need not be MR.



Application - job submitted to the framework. (need not be MR.)

Container - Basic unit of Allocation of resources (CPU, RAM, Disk)
- Replaces fixed Map/Reduce sl

Resource Manager
- Cluster level resource scheduler

Node Manager -
- Per machine agent.
- Manage life cycle of container
- Container resource monitoring.

Application Master
- Per Application
- manages application scheduling & task execution.

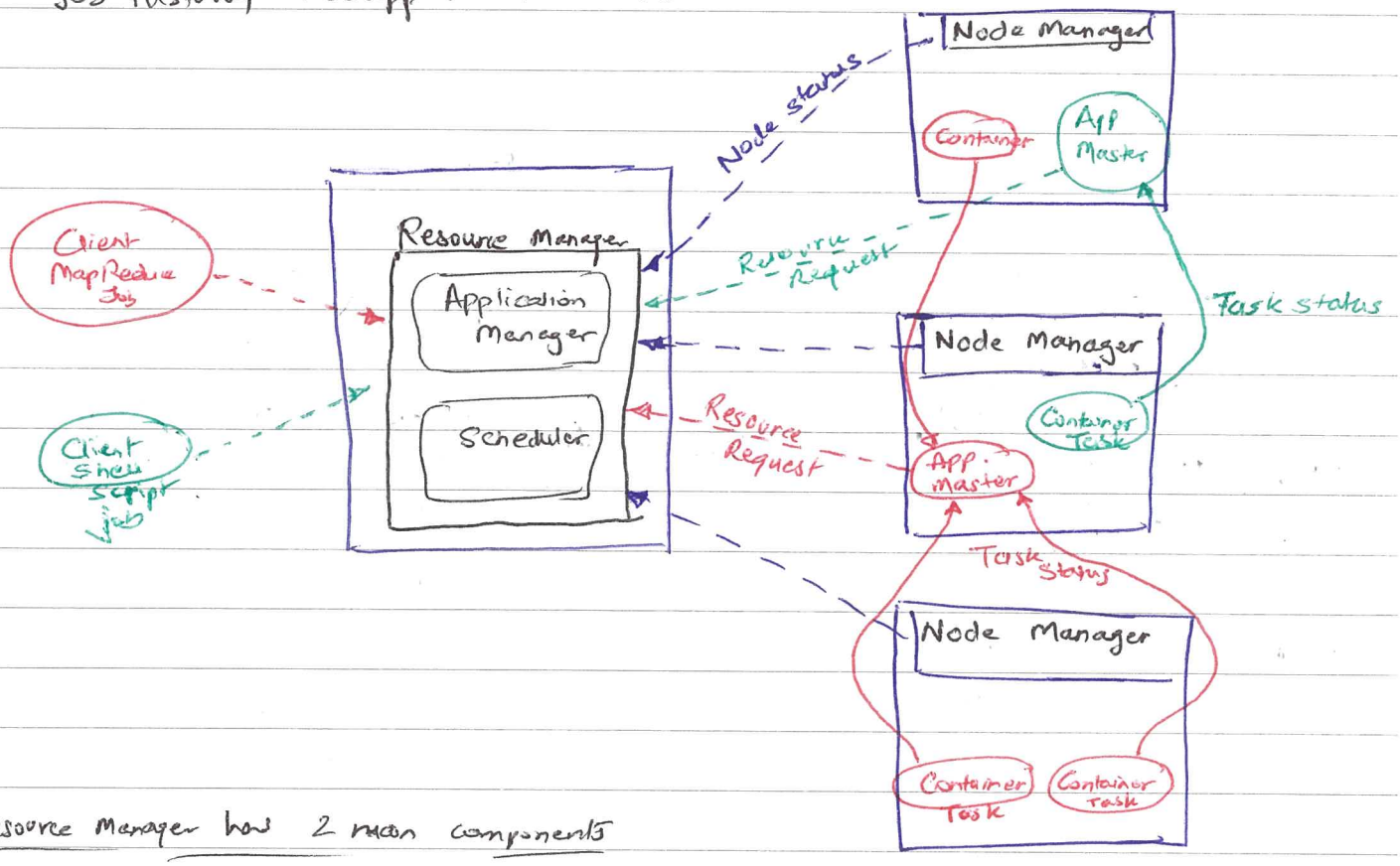
eg: Map Reduce Application Master

Resource manager address : IP : 8050

Resource Manager ~~HTTP~~ web address : 8088

App Timeline Servic : 8188
(related for non-M
& MR2)

Job History Web app : 19888 related to mapreduce 2



Resource Manager has 2 main components

Scheduler - responsible for allocating resources to various running applications subject to the constraints of resources.

- pure scheduler. Does not perform monitoring of Applications or tracking their status.
- Scheduling is performed based on resource requirement of the application expressed in required Resource containers (CPU, RAM, Disk)

Application Manager

- responsible for accepting jobs submissions.
- negotiating the first container for running application specific application master.
- provides services to restart application master container on failures.
- does monitoring, handles failures of Application as a whole

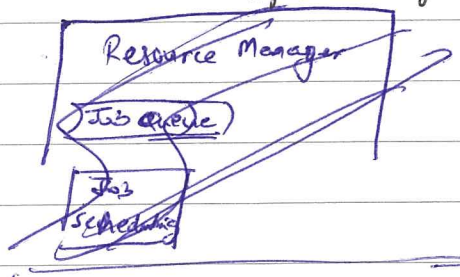
Node Manager

- per machine framework agent responsible for containers, monitoring container resource usage & reporting this to Resource manager/ Scheduler.
- does monitoring & reporting of health of a node.
- does not monitor Application master or tasks within a container

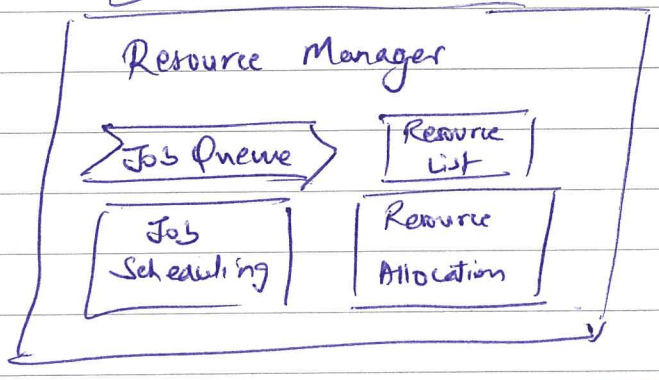
* Application manager – responsible for managing application (created)
* Application Master – responsible for managing tasks created by it
* Resource Manager – responsible to monitor cluster wide resources based on status from Node managers
* Node Manager – monitors containers & resources of a single node & report status to RM.

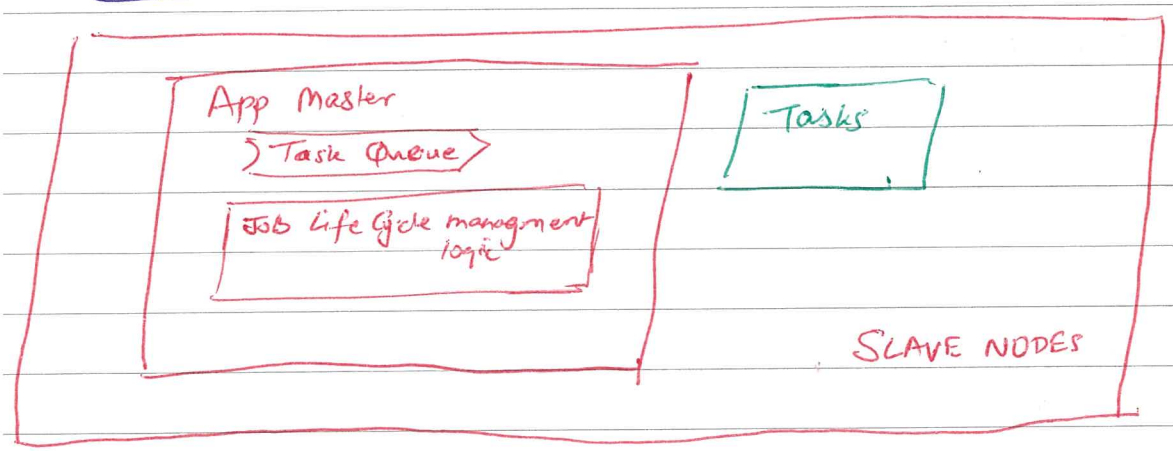
Application – Master

- per Application (eg: Map Reduce, Shell script etc)
- responsible for negotiating appropriate resource containers from Resource manager, tracking their status & monitor progress.
- Application – Master handles ~~the~~ monitoring and restarting of failed tasks irrespective of the nodes they run. It monitor tasks which are created only by itself

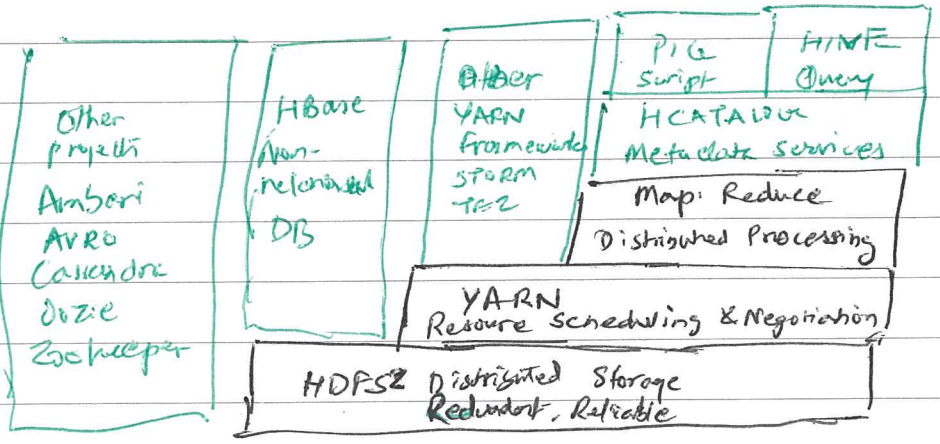


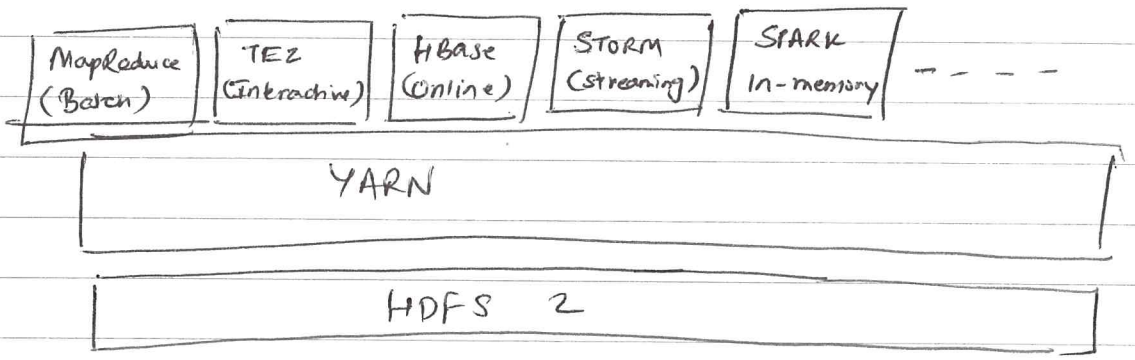
~~MapReduce 2.0~~





HADOOP 2.0 Ecosystem





TEZ - provides pre-warmed containers & low latency dispatch, avoids disk writes and faster query execution
- will be used by Hive & Pig.

STORM - Streaming & large scale live event processing implementa

SPARK - In Memory computing.
- All the Queries & instructions are hitting data in memory (so really fast)

Distributed cache : used to add files/archives to a cache that gets copied across nodes as part of running the MapReduce or other application so that these files are accessible by the tasks running on these nodes.
- Also used to add libraries (jars) to the classpath

Configuration : yarn-site.xml → property yarn.nodemanager.local-di

API	Command Line	Desc
addCacheFiles(URI)	-files file1, file2 ...	Add files to dist
addCacheArchive(URI)	-archives arch1, arch2	Adds archives to cache. Its copied & unarchives on task n
addFile to ClassPath (Path file)	-libjars jar1, jar2	Add files to dist cache to be added to MR classpath. These are not unarchived on task nodes

← To pass configuration parameters ... -D {configName = value}

MAP REDUCE

- is a programming model to perform distributed processing of large datasets.
- Hadoop implements MapReduce. It manages communication, data transfers, parallel execution across distributed servers
- is useful only for nature of data is WORM (Write Once Read Many)

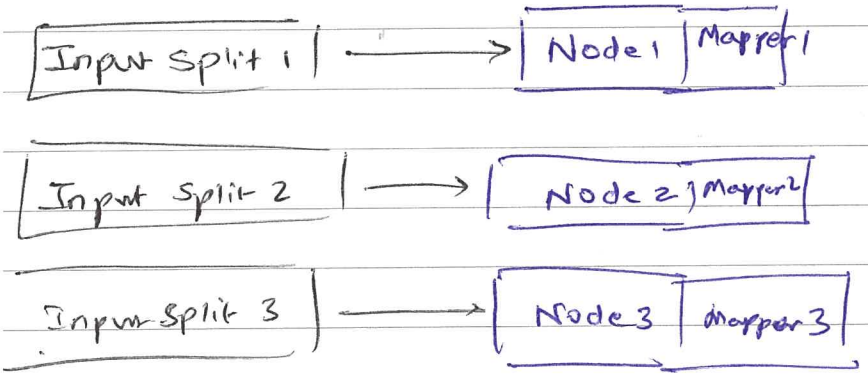
Example

Stocks Dataset — Stock Exchange, Symbol, open price, close, day high, day low

ABCSE, TWI, 1.58, 1.76, 1.79, 1.3, 22751

Problem : Find max close price for each symbol.

Distributed



Information required for mapreduce job

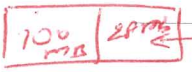
* Location of input
* Location of output
* Input Format
* Output Format
* Mapper class
* Reducer class (optional)
* Combiner class (optional)
* Partitioner class (optional)

- Dataset is splitted into logical chunks called input split.
- A mapper works on a single Input split. So the number of mappers is equal to the number of input splits.

→ A mapper will process one record at a time from the associated Input split.

→ Output of a mapper is key-value pair.

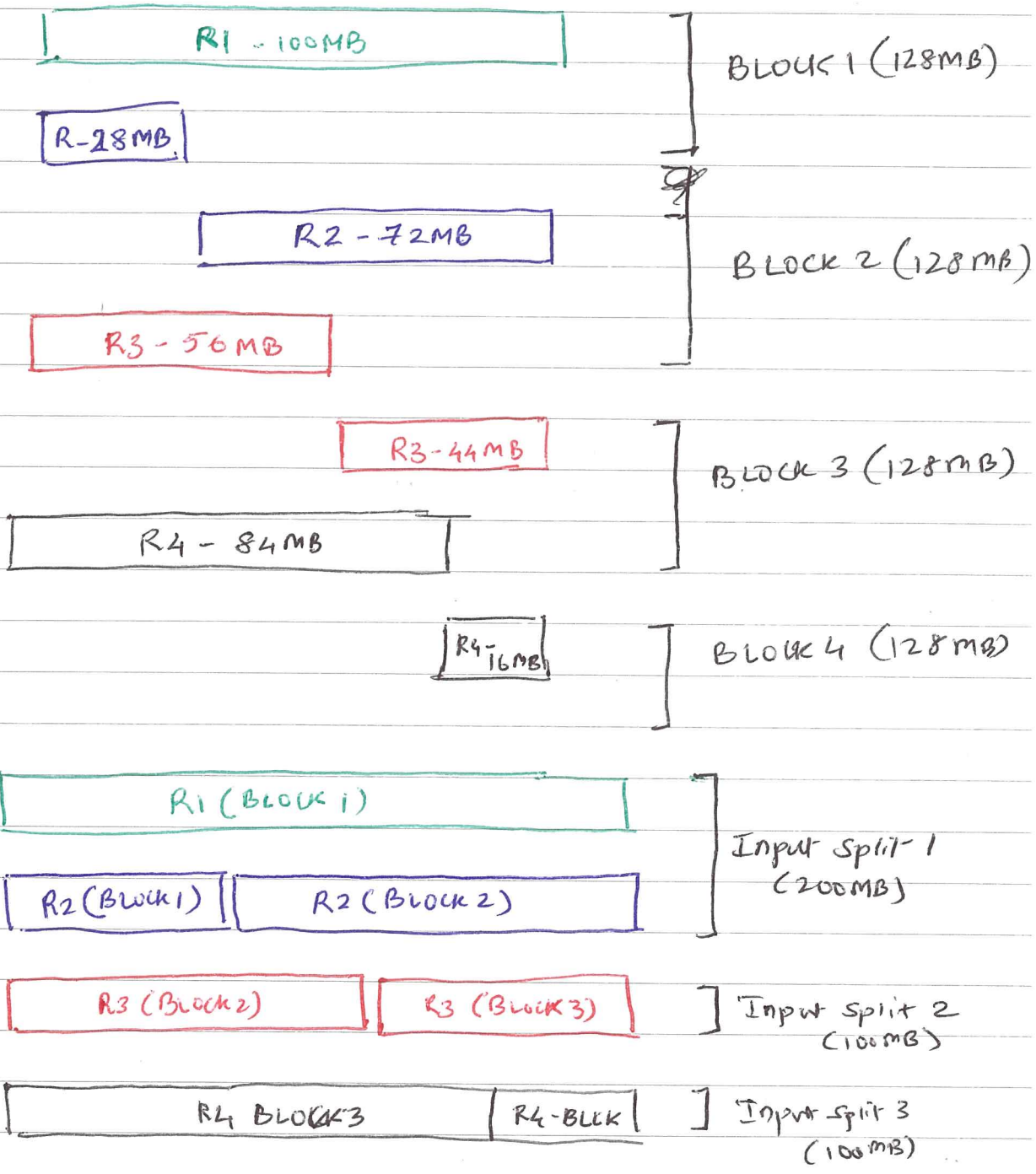
* If the file or last part of the file is less than blocksize the block will be only of that size
Example filesize 100MB.



not associated with any f

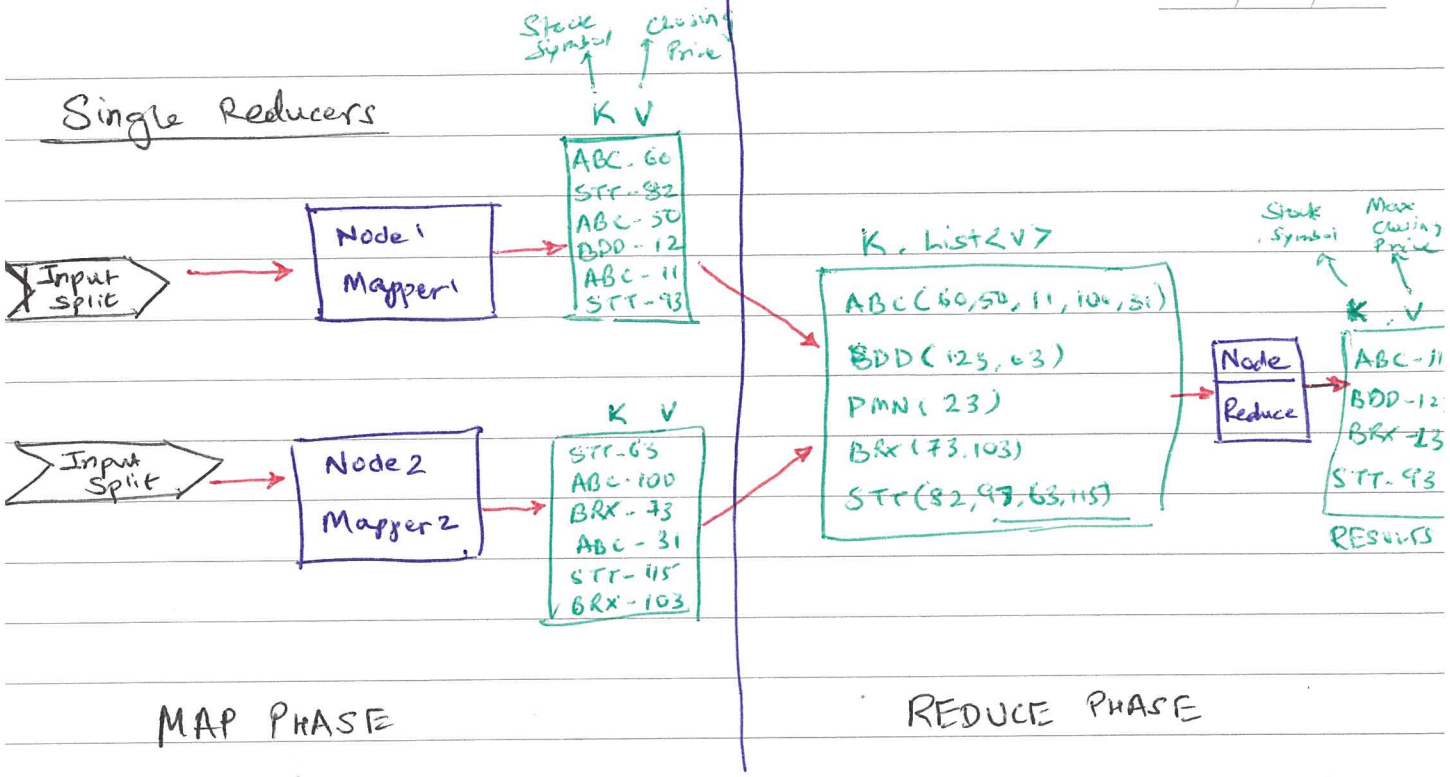
→ Input split is a logical boundary recorded whereas a block is a hard limit (ex: 128MB).

→ So it is possible that the input split may span data having in multiple physical blocks.



* HDFS stores blocks but ~~at~~ OS level blocks are nothing but ~~stores block~~ files of Local FS. For Linux the block size is 4kb. So a 100MB block will be broken by OS into 4kb blocks & stored hence HDFS does not waste remaining 28MB

Single Reducers



NOTE : No Shuffle phase.

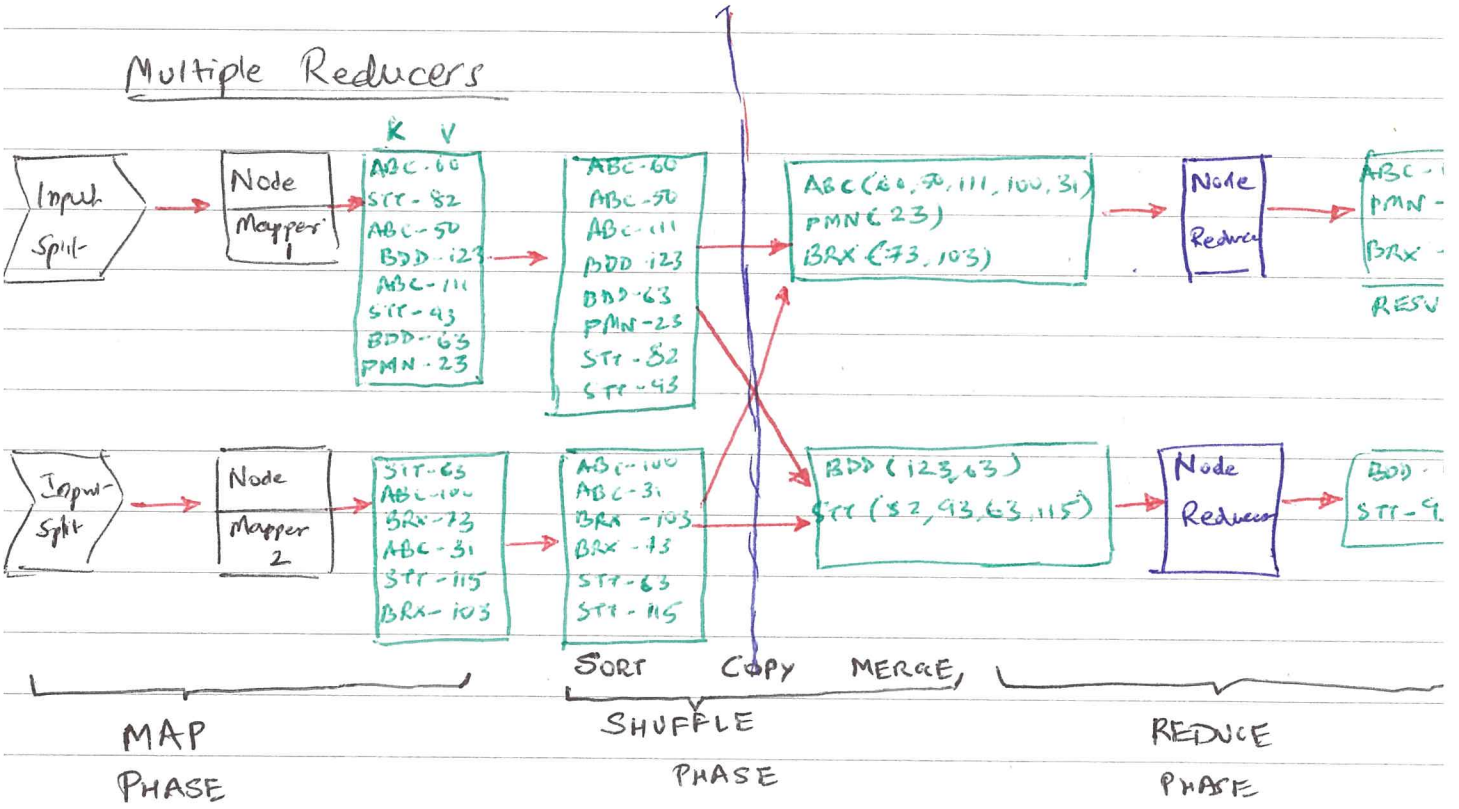
Multiple Reducers :

- The number of reducers can be configured.
- Having a single reducer may created performance bottlenecks at reduce phase.
- Depending on the size of the dataset we can configure more reducers.

API : fs (filesystem) Common
io

setPartitionerClass( )
setCombinerClass( ) MapReduce

Multiple Reducers

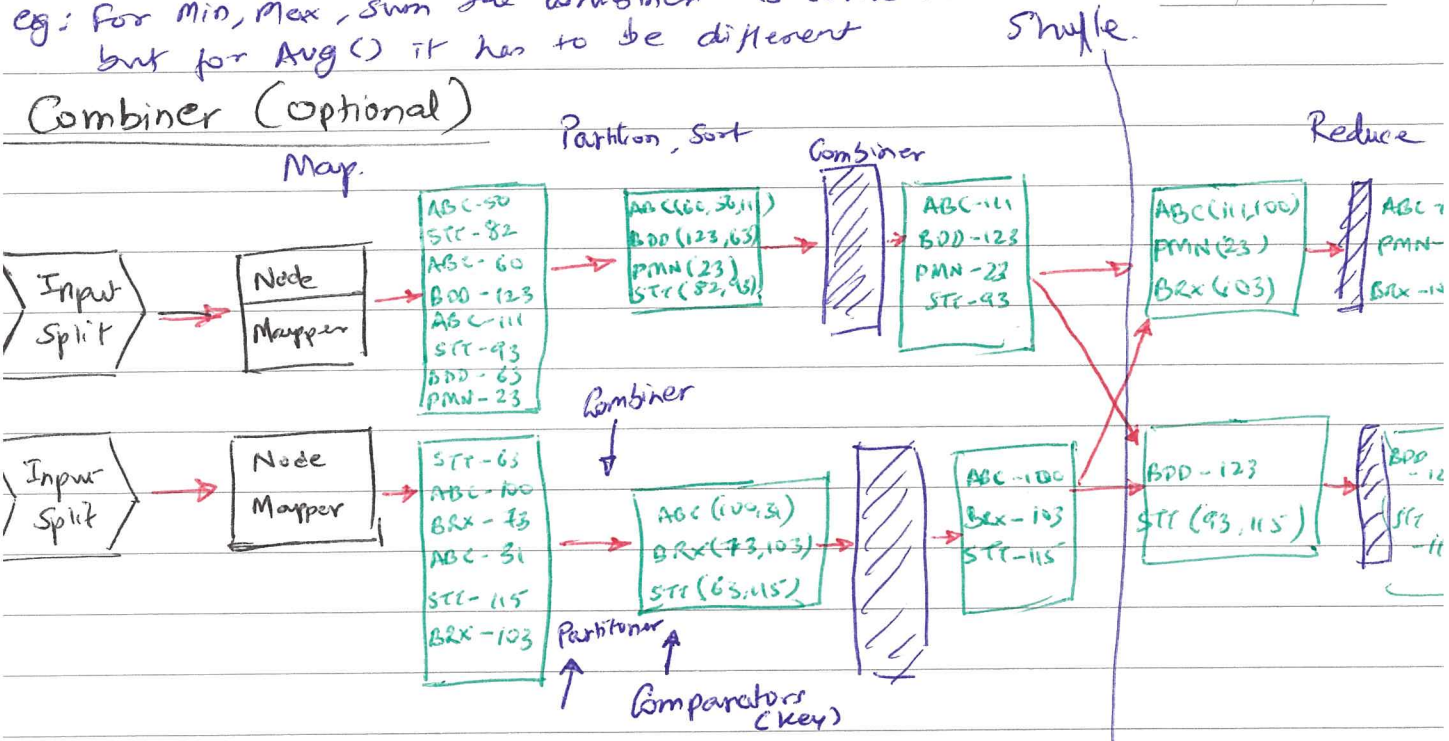


- Keys from Map Phase are assigned to a particular reducers.
- All the values (irrespective of the mapper) for a particular key should go to the same reducer.
- This is done in Shuffle phase where the keys, value from many mappers grouped & assigned to reducer and all values are moved across nodes to send all the values along with the key to the assigned reducer.
- Additionally, the shuffle phase also sorts the key, value pairs from Mapper by key

- The datatypes used as Values should implement Writable interface
- The datatypes to be used as keys should implement Writable Comparable interface.
- the Writable interface allows data of these types to be serialized to HDFS or over the network when transferring from Map to Reduce phase (since normally map & reduce task run on different nodes)

NOTE: Combiner may or may not be the same function as Reduce and it depends on the nature of the problem:
eg: For Min, Max, Sum the combiner is same as reduce but for Avg() it has to be different

Combiner (optional)



- The combiner acts like a local reducer & runs on the same node as the corresponding mapper. It takes the output of the mapper & applies reduce function locally to eliminate the values for a given key
- The advantage of the combiner is that it reduces the amount of values from the key/value pairs from map output & hence reduces the data that needs to be sent over the network to the appropriate reducers.

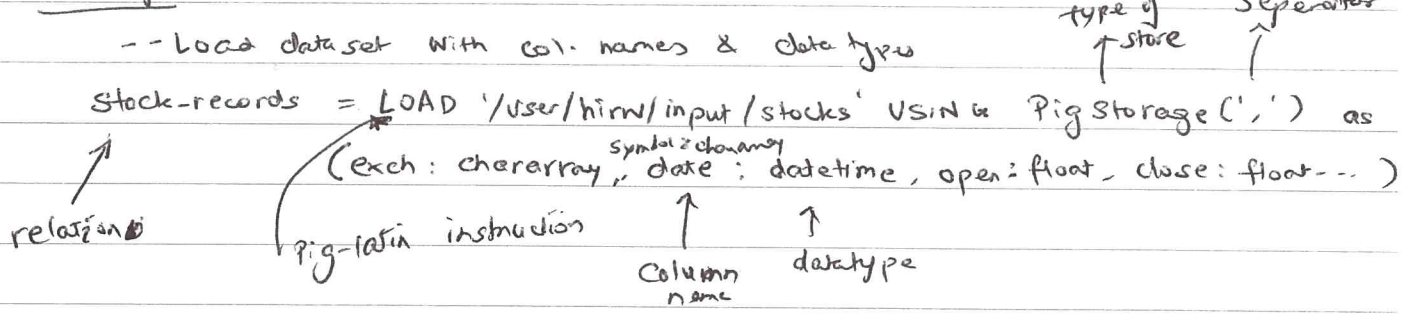
- ~~Sort~~ The partitioner will create partitions for storing outputs of mappers & assign data in partition to individual reducers
- The partition will split the keys to partition by using some hash function on keys and evaluate Hash(key)% numReducers.
- Within each partition keys will be sorted (hence the keys datatype must implement comparable) & values will be grouped together by key.
- Finally each key & associated values within the partition will be passed to the reduce function().

- Partitioner decides the grouping whereas comparator decides the order of output records within the groups
- Partitioner additionally can be tweaked to do global sorting ~~with~~ even with using multiple reducers

* Pig is a client tool, i.e. It need not be installed on all nodes in the cluster. It should be installed on client node or node from where pig scripts are run & ask to submit job to the cluster.

PIG: - a high level data-flow language.
- ~~Write~~ script written using Pig latin instructions.
- generates map-reduce jobs ~~to~~
- is suited for batch jobs, not suited for real time or performance. Suited for ETL

Example:




```
-- GROUP records by symbol
grp_by_sym = GROUP stock_records BY symbol;

-- Calculate max closing price
max_closing = FOREACH grp_by_sym GENERATE group, MAX(stock_records.c
                                                AS max_close
```

group by key

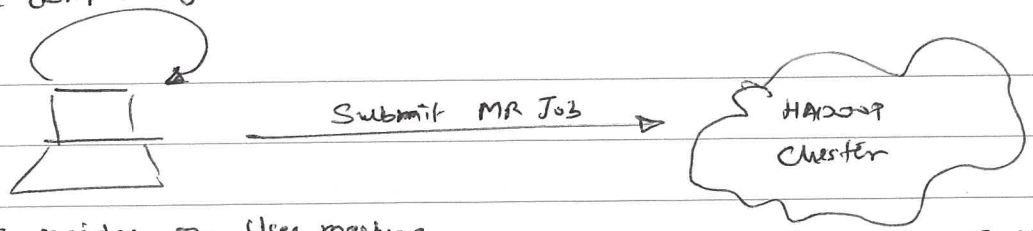
```
-- Store output
STORE max_closing INTO 'output/pig/stocks' USING PigStorage(',');
```

→ Pig can be run through Java using PigServer class

→ We can use GRUNT shell to execute PIG commands from command-line.

→ Pig can also be ~~load~~ run in local mode. In this mode Pig will use local file system instead of HDFS. Good for quick checks & debugging.

- compile Pig latin instructions



Pig resides on User machine

Execution of MR Job

Datatypes

Atom: Individual element - Simple element like int, float, chararray.
eg: 'Mike' '35'

Tuple: - An ordered set of fields. A field is a piece of data
Ex: ('Mike', 43)
↑ ATOM ↑ ATOM

Bag: collection of tuples, each individual tuple can have different structure. Nested tuples are allowed
Nested Bags are allowed
Ex: {('Mike'), ('Doug', (43, 45))}
↑ TUPLE ↑ TUPLE ↑ NESTED TUPLE

Map: - key/value
- key is chararray, value can be any datatype
Ex: [name # Mike, phone # 551212]

Relational Operators

Loading, Storing	LOAD	Loads data from filesystem or other storage into a relation
	STORE	Saves a relation to FS
	DUMP	Prints a relation to Console
Filtering	FILTER	Removes unwanted rows from relation
	DISTINCT	Removes duplicate rows from relation
	FOREACH... GENERATE	Adds or removes "fields" from relations
	STREAM	Transforms a relation using external program
Grouping and Joining	JOIN	Joins two or more relations
	COGROUP (Join & Group together)	Groups the data in two or more relations
	GROUP	Groups the data in a single relation
	CROSS	Creates a cross product of two or more relations
Sorting	ORDER	Sorts the relation by one or more fields
	LIMIT	Limits the size of relation to max number of tuples
Combining	UNION	Combines two or more relations into one
	SPLIT	Splits a relation into 2 or more relations

- Data for any data type can be NULL.
- By default, Pig treats undeclared fields as Byte Array.

Local mode : pig -x local.

Tez Local mode: similar to local but uses Tez runtime engine.
pig -x tez_local

MapReduce mode : This is the default mode.

Tez mode : Pig -x tez
: need access to Hadoop cluster.

Interactive mode : Pig (from command line) opens up the grunt shell.

Batch mode : create a pig script file & call the pig command with script file as argument.

Configuration :
{pig.temp.dir} - directory to store intermediate results
⇒ configured using
either
↳ Pig.properties file.
↳ -D and pig property in PIG_OPTS environment variable
↳ -P command line options
↳ set command

→ Pig relation is a bag of tuples. Its similar to a table in RDBMS. where tuples are like rows.

→ Field are referenced by position or by name (alias)
- Position is indicate by $0, $1 etc.

→ Complex types
- A field inside a tuple can be complex types like bag, tuples, map.
- Use dot operator to drill down into the heirarchy.

Outer Bag	Inner Bag (int, tuple)
(1, 2, 3)	(1, {(1, 2, 3)})
(4, 2, 1)	(4, {(4, 2, 1), (4, 3, 3)})
(8, 3, 4)	(8, {(8, 3, 4)})
(4, 3, 3)	

ex:
```
A = LOAD 'student' AS (name: chararray, age: int, gpa: float);
B = LOAD 'voterbike' AS (name: chararray, age: int, reg: chararray,
                         donation: float);
C = COGROUP A BY name, B by name;
D = FOREACH C GENERATE FLATTEN (ISEmpty(A) ? null: A),
                       FLATTEN ((ISEmpty(B) ? null: B));
```

DESCRIBE – shows the schema of a relation.

ILLUSTRATE – used for debugging. Prints out a data flow for a pig latin instructions or script

UNION – combine data from relation. Pile rows. Ho

JOIN – combine dat from relation. Pile columns

REGISTER - used reference external JAR containing UDF

ex: REGISTER /home/root/pigder/pigudfs.jar

DEFINE : To alias a UDF (including namespace) (Similar to using in

example:

DEFINE com.hirw.maxClose

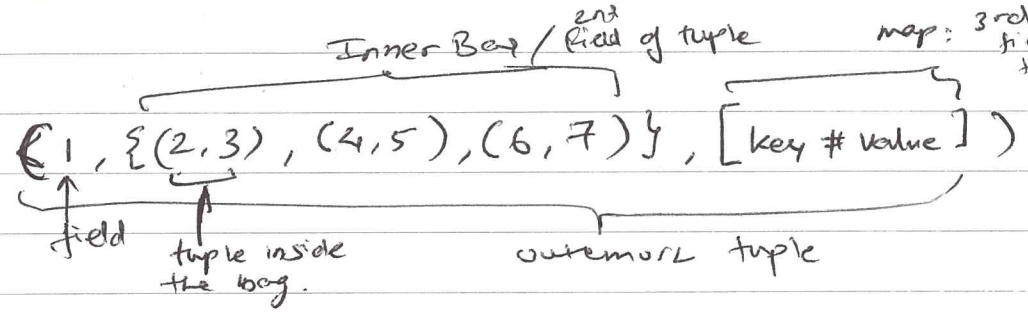
EXPLAIN : - Prints out the Pig script execution plan

- indicates where the bottlenecks

SPLIT : Used to split data in a relation into multiple relation based on some criteria

ex:

Examples

Data : 

Method	Example	Result
Position	$0	1
Name	field2	Bag {(2,3),(4,5),(6,
Projection	field2.$1	Bag {(3),(5),(7)}
Function	AVG(field2.$0)	(2+4+6)/3 = 4
Conditional	field1 == 1 ? 'yes' : 'no'	yes
Lookup	field3#key	Value

FLATTEN

Ex: tuples (a, (b,c))

GENERATE $0, flatten($1) ⇒ (a,b,c)

* Pig tries to implicitly convert unspecified scheme (bytearray) columns to other datatypes depending on the operation performed. So if a substring is performed on $0 that column will be converted to chararray.

FLATTEN for Bags

① ~~(a,~~ ({(b, c), (d, e)})

GENERATE flatten ($0) ⇒ (b, c) (d, e)

② (a, {(b,c), (d,e)})

GENERATE $0, flatten($1) ⇒ (a, b, c)
(a, d, e)

- pig -useHCatalog for running pig script which accesses Hive tables. for Load/Store

→ LOAD "<database name>.table_name>" ~~from~~ using
org.apache.hive.hcatalog.pig.HCatLoader();

* Join can be done on more than 1 columns
* We can use JOIN operator with more than two datasets (However, only INNER JOIN) (LEFT outer, Right outer, ~~Full~~ Not)
* COGROUP is more like full Outer join with grouping based on join columns.
* Cogroup can be performed on multiple columns

* To simulate an inner join using Cogroup we filter the COGROUP ~~result~~ result by filtering records where either left or right Record is Empty

Input Format	Record Reader	LoadFunc
TextInputFormat	LineRecordReader	PigStorage(,) →
XmlInputFormat	XmlRecordReader	PigRankLoad()

* Hive is schema on Read. means the data is not checked against table schema when loaded but when query is executed against the table. (if column type does not match data it returns NULL value for that column.

HIVE

- Database or Datawarehouse on top of HDFS.
- SQL oriented Query Language.
- Can be run in local mode or mapred. Local mode can access local file system.
- 3 Steps
  - → Create Hive table (CREATE TABLE statement)
  - → Load (LOAD DATA <INPATH>)
  - → Query.

Architecture :

Metastore : * Hive keeps all the information about databases, tables in those databases, table schemes, HDFS Mappings (Table to HDFS files)

* **External Tables** → pointers to ~~table~~ data in HDFS
  - → Deleting external table does not delete the HDFS file data
* **Managed Tables** (Internal Table) : → Data is actually filled in the table from HDFS
  - → Deleting Table will also delete the associated HDFS files.
  - → Use only if temp data

Partitions : - Slice data in a table so that it gets stored on disk in seperate files.
- Improves query performance

NOTE :- Metastore is stored on Local FS. (Stores info about Tables, Database, HDFS location etc)
- Actual table data is stored in HDFS.

Buckets :

- Hash Partitions on ranges & speed up joins.

/user/hive/warehouse

↳ /table1/ — all data belongs to table in file (NOT PARTITIONED)

↳ /Table 2/ 20160301/

/20160302/

Partition Column ← TABLE PARTITIONED BY DATE. Subdirs created under Table 2 will Date & records on particular date stored under these subdirs.

RDBMS → Schema on Write (Slow loads, querying fast)
→ OLTP (Real time Read/Write)
→ OLAP. (Batch Reporting)

HIVE → OLAP. (Batch, Reporting)
↳ Schema on Read (Fast loads, querying slow)
↳ if we want interactive/real time we use HBase

HIVE SERVER (Thrift) → allows connections to HIVE via JDBC.
→ Access data using Thrift API

NOTE ① When we create table and specify a location we don't need to Load data into the table using LOAD DATA statement. We can simply drop files in that location and HIVE will automatically pickup.

② LOAD DATA moves data from source location into target i.e. the source file is gone

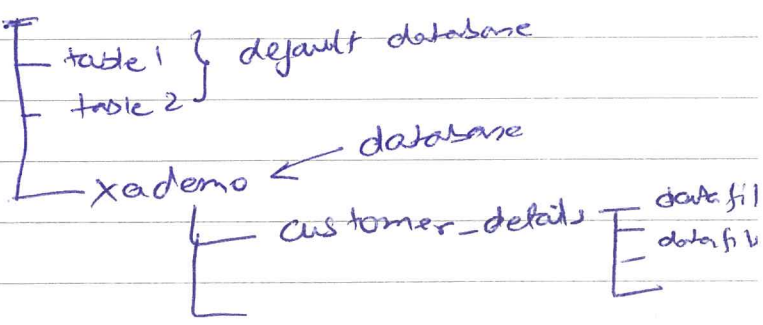
* To run Hive execution set it As
  SET hive.execution.engine = tez
  SET hive.execution.engine = mr

→ Hive metastore (where information about table schemas & the HDFS files that contains corresponding data) is stored in ~~/apps/hive/warehouse~~ MySQL or such other DB (In Hortonworks its mysql User : hive password : hive

→ Managed Tables are stored in the HDFS dir.
/apps/hive/warehouse

→ hive> DESCRIBE FORMATTED <dbName>.<TableName>;
This will provide all the information about the specified tablename.
eg: DESCRIBE FORMATTED xademo.customer_details.

→ Hive tables are stored in HDFS under the folders which represents database name



→ Hive supports UDFs, various files & Formats

* HIVE follows schema on read. i.e it does not check if the data loaded into the HIVE table matches the table's schema.
* When the table is queried and if a column dtype does not match the corresponding data in the file the column value is simply written NULL.
  (eg: if table column expects Integer & data contains string)
* Also, if the dataset contains more columns than defined in create table, the additional columns are ignored when executing queries

hadoop fs -mkdir -p d1/d2/d3 - Automatically creates the entire directory heirarchy

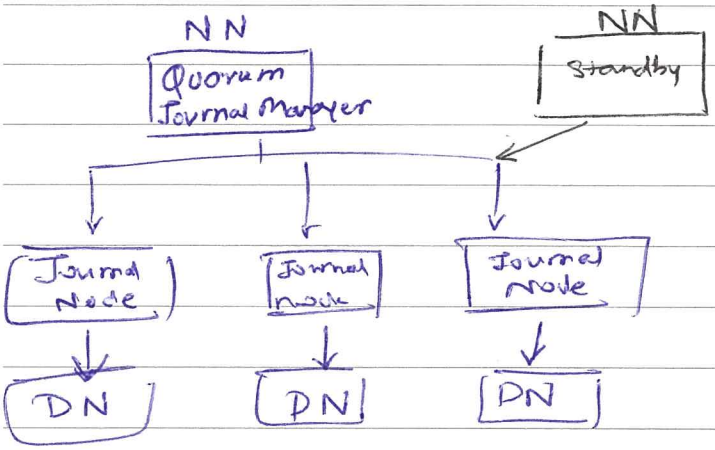
hadoop fs -mkdir test{1,2} - creates dir "test1", "test2"

-mkdir test1 test2 test3

-mkdir test1..10

HDFS Internals (Contd)

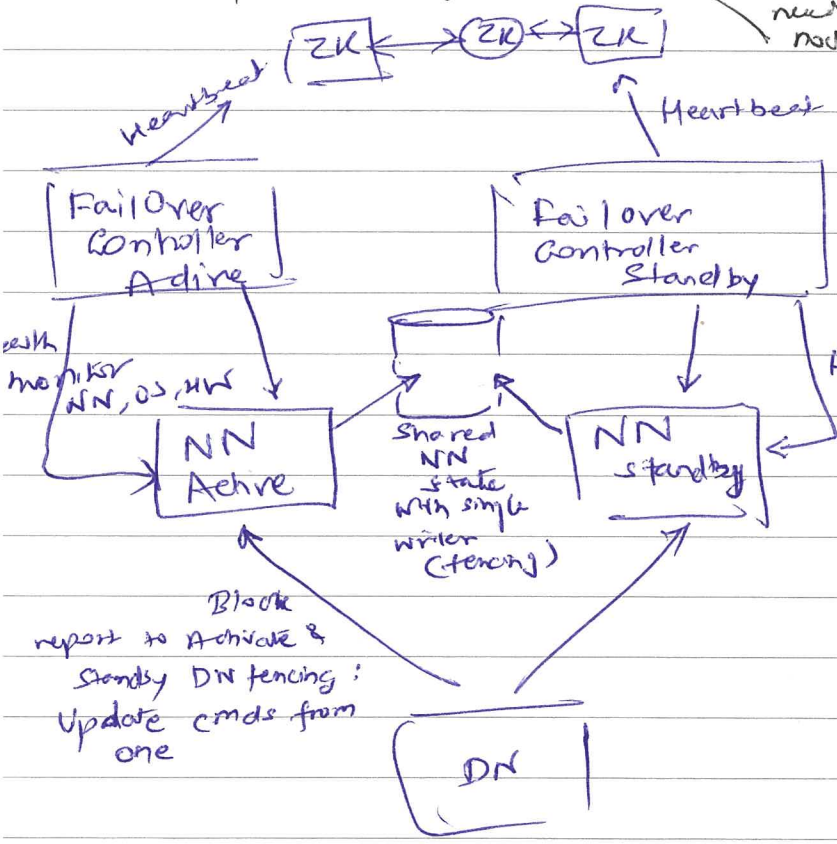
High Availability (HA) Implementation of Hadoop



This ensures that the HDFS is fully synchronized to the Standby before becoming Active

Quorum Based Storage

- Both Active & Standby NN communicate with daemons called Journal nodes
- Any modification done in HDFS Journal manager writes the modification to majority of Journal nodes
- Standby NN is capable of reading edits from Journal node. It then applies the mode to its own copy of HDFS metadata
- In Failover Standby ensures its has read all modifications from Journal nodes before promoting itself as Active



Shared Storage using NFS

- Both Active & Standby NN have access to directory on shared Network Storage (eg: NFS mount from NAS)

Spark → Iterative Processing
Storm → Stream Processing
Hadoop → Batch processing

* → HDFS guarantees - that blocks from the same file are never on the same data node.

* The client queries the Name node for block locations. The NN returns block locations to client. The client reads the data directly off the DN.

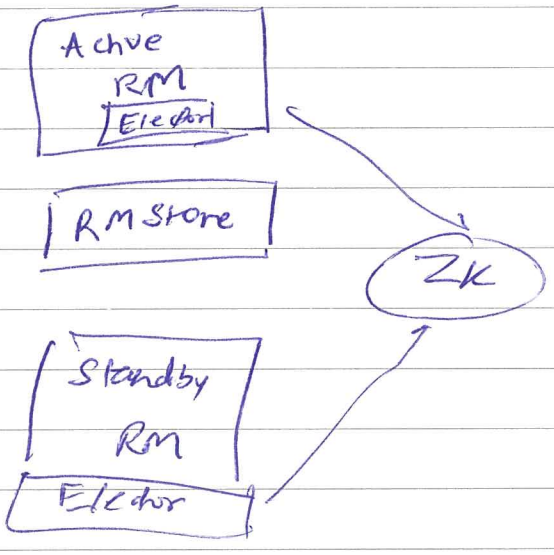
The DN respond to N
The NN redirects c
client to DN

Hadoop Configurations

1. Standalone mode : All Hadoop services run in a single JVM on a single machine
2. Pseudo - Distributed mode : Individual Hadoop services run in a individual JVM but on a single machine
3. Fully Distributed mode : Hadoop services run in individual JVMs but these JVMs reside in seperate machines in a single cluster.

* jps command to verify running daemons.

Resource Manager in HA mode (After 2.4 Version)



Critical Parameters for Hadoop

hadoop.tmp.dir - temp dir for HDFS & local FS

fs.default.name - host & port of NN

mapred.job.tracker : host & port of App master.

dfs.name.dir - location of NN metadata, comma separated list of directories. (Local FS

dfs.data.dir - storage location for data node blocks in the local FS. comma seperated list

mapred.local.dir - local dir on NN where intermediate output is stored. comma or space seperated list

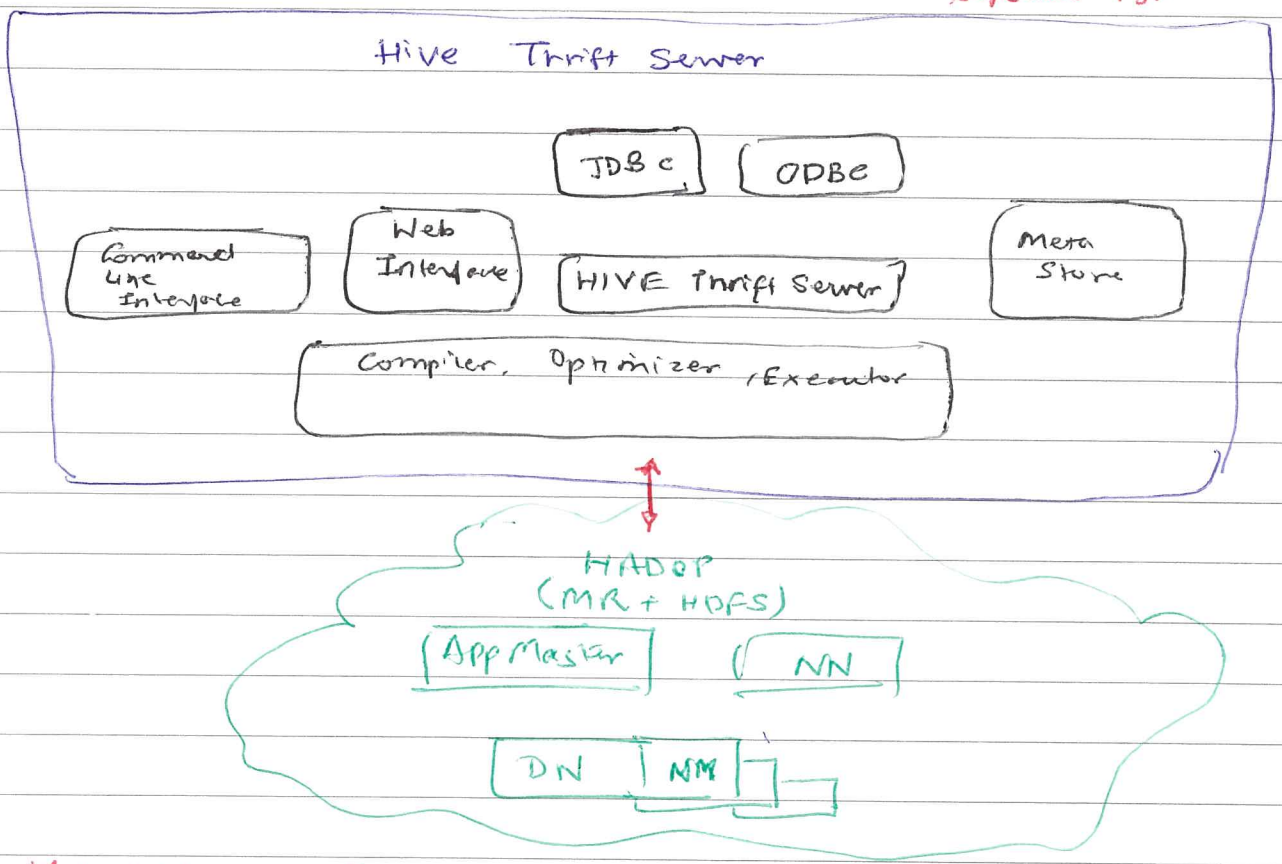
MapReduce Config

Input format

KeyValueTextInputFormat

JOINS

HIVE



Metastore config (properties)

javax.jdo.option.ConnectionURL	jdbc:mysql://dbhostname.com/hive_db
javax.jdo.option.ConnectionDriverName	com.mysql.jdbc.Driver
javax.jdo.option.ConnectionUserName	database_user
javax.jdo.option.ConnectionPassword	database_pass

* Hive is a client tool, need not be installed on all nodes. Only client node or where the Hive scripts are initiated & job submitted to cluster

1) Datatypes in Hive

Primitive : Boolean, int, float, String, Timestamp (yyyy-mm-dd hh:mm:ss), date (yyyy-MM-dd)

Complex : Struct : {a: int, b: int}
Maps : M['group']
Arrays : ['a', 'b', 'c'], A[] returns b

cast (expr as datatype)

UDT : Structs with Attributes.

Partitions - Contain nested sub-directories in HDFS for each combination of partition column values
- Allow efficient retrieval of rows

Ex : Partition columns : ds, country
HDFS for ds = 20120410, county = US
/wh/pvs/ds = 2012 0410/county = US

→ Creating Partitions

* CREATE TABLE test_part (ds string, hr int) Partitioned by (ds string, hr int);

* INSERT OVERWRITE TABLE test_part PARTITION (ds='2009-01-01', hr=12)

* ALTER TABLE test_part ADD_PARTITON (ds='2009-01-01', hr=11)

* SELECT * FROM test_part WHERE ds='2009-01-01';
This query will only scan all the files within /apps/hive/warehouse/test_part/ds=2009-01-01 directory.

* Serde helps to interpret nested & complex data and map it to columns
* Serde help to map just the needed columns instead of all columns in dataset

Buckets

- optimization technique similar to partitioning.
- distributes data load into user defined set of clusters by calculating hash code of key mentioned in query.
- Use 'bucketing' to run queries on column that has huge data, which makes it difficult to create partitions

Syntax: CREATE TABLE page_views (userid INT, session_id BIGINT, URL STRING)
PARTITIONED BY (day INT)
CLUSTERED BY (user_id) INTO 100;

→ data here will be classified based on hash of user_id column into 100 buckets.

→ when querying the processor will first calculate hash of the supplied user_id in query & only look into that bucket for fetching results.

HIVE File format - Different format, helps in performance

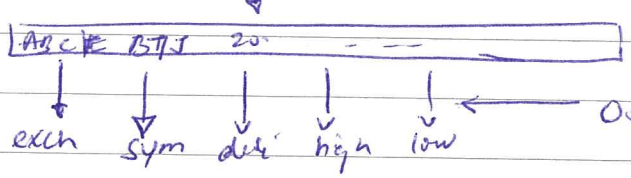
CREATE TABLE dest1 (key INT, value STRING) STORED AS
INPUTFORMAT 'org.apache.hadoop.mapred.SequenceInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.mapred.SequenceOutputFormat'

SERDE

ABCE, BTJ, 2009-03-18, 3.45, 5.73, 4.75 ...

↓ InputFormat - eg: TextInputFormat, LineRecordReader

Single Line



Serde (Deserializes the line of text as a whole record

ObjectInspector
- converts the record into each column
- it knows how construct column values from

Example : Database : website
Table : pageviews (userid, link, came-from)
└ Partition column datestamp
└ Bucketing column userid into 5 buckets

Rows inserted : ('jdoe', 'mail.com', 'sports.com') } 2014-09-23
('jlee', 'mail.com', 'null')

/apps/hive/warehouse → /website.db → database
└→ /pageviews → table
└→ /datestamp = 2014-09-23 → Partition

bucket files
- 00000-0
- 00000-1 (jlee)
- 00000-2
- 00000-3 (jdoe)
- 00000-4

The data for a record is stored in a particular bucket file based on the hash value of bucketing field in insert clause (ie. userid)

HIVE Datatypes

Primitive types - TINYINT, SMALLINT, INT, BIGINT, FLOAT, DOUBLE, STRING, VARCHAR
CHAR, DECIMAL(9,7) ← precision

Arrays : ARRAY<datatype>

Map : MAP<primitive type, primitive type>

* Partition columns cannot be same name as one of the table columns

HIVE DDLs

* CREATE DATABASE [IF NOT EXIST] database_name [COMMENT '
  [LOCATION hdfs_path] [WITH DBPROPERTIES (prop-name = propvalue'

* DROP DATABASE [IF EXISTS] db_name [CASCADE / RESTRICT]

* ALTER DATABASE db_name SET PROPERTIES (prop-name = prop_val)

* CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS]
  db_name . table_name (col_name col_type , col_name col_type)
  [PARTITIONED BY (col_name col_type , col_name col_type)]
  [CLUSTERED BY (colname , col_name) [SORTED BY (colname ASC|DESC
  hashcode generated for columns & stored in one of the — colname ASC|DESC)]
  INTO num_buckets BUCKETS] — within the buckets sorted by these columns
  [SKEWED BY (col-name , col_name) ON ((col-val , col_val) , (col_val , col_val)
  creates seperate files for column values specified
  [STORED AS DIRECTORIES]]
  instead of seperate files, stored as directories
  [
    [ROW FORMAT DELIMITED FIELDS TERMINATED BY 'char'
    *
    LINES TERMINATED BY 'char'
    for array fields → COLLECTION ITEMS TERMINATED BY 'char'
    for map fields → MAP KEYS TERMINATED BY 'char'
    Specify escape character → ESCAPED BY 'char'
    NULL DEFINED AS 'char'
    OR
    [SERDE serde-name [WITH SERDEPROPERTIES
    (propname = propval)]]]
  ]

PARTITIONED BY creates seperate directory for each distinct value combinations of columns specified here.
Further a TABLE/PARTITIONED can be bucketed by columns using CLUSTER BY and sorted within a bucket using SORTED BY columns.

[STORED AS (SEQUENCEFILE | TEXTFILE | RCFILE | ORC |
PARQUET | AVRO |
INPUTFORMAT 'classname' OUTPUTFORMAT 'classname'

OR

STORED BY 'storage.handler.class' WITH [SERDEPROPERTIES (...)]

]

[LOCATION hdfs_path]* * defines the HDFS path where the data for this table will be stored.
This applies to managed & external tables. * Any LOAD statement will actually move the data from specified location (HDFS) to the location specified here. If this location is not specified then by default it is /user/hive/warehouse/table/

[TBLPROPERTIES (prop_name = prop_val, prop_name ----)]

↳ tag table definition with own key/val pairs. Some predefined key val pairs some customized attached

[AS SELECT statement] ;

CTAS : CREATE TABLE table_name AS SELECT_statement;

The new table is created using the scheme of the query results alongwith the ~~data~~ source table properties like SERDE & Input/Output format

→ CTAS cannot be used for creating
- Partitioned table (No partitioned by)
- external table
- list bucketing table. (No clustered by)

CREATE TABLE table_name LIKE other_table;

This creates a table with the same scheme as source table but does not populate the target table with source table data.

Bucketed Sorted Table

```
CREATE TABLE page_view (viewtime int, userid int, page_url string
                        ref_url string, ip string)
    PARTITIONED BY (dt string, country string)
    CLUSTERED BY (userid) SORTED BY (viewtime) INTO 32 BUCKETS
    ROW FORMAT - - -
```

* The CLUSTERED BY AND SORTED BY does not affect insertion of data - only how it is read. In above example the data will be inserted into one of the 32 buckets (files) based on hash value of userid column value. Within this bucket the values (userid) are sorted in increasing order of viewtime column.

SKEWED TABLES

By specifying values that appear very often in certain columns using SKEWED BY CLAUSE. Hive will split those out into seperate files (or directories if STORED AS DIRECTORIES) option is specified.

example:

```
CREATE TABLE list_bucket_multiple (col1 string, col2 int, col3 string)
  SKEWED BY (col1, col2) ON (('s1', 1), ('s2', 2), ('s3', 3))
  STORED AS DIRECTORIES;
```

TEMPORARY TABLES: if TEMPORARY is specified Hive creates a temporary table which is visible only during current session. The temporary table is created in user's scratch directory. If a permanent table exists with the same name, then it will be accessible until the temporary table.

* Temporary table don't support.
  - Partition columns not supported
  - No support for indexes

DROP TABLE ~~table_name~~ [IF EXISTS] table_name [PURGE]

The data is moved to .Trash directory unless PURGE is specified

* DROPping EXTERNAL Table, data in the table is not deleted
* if a table is referenced by views, view is left dangling & must be dropped or recreated.

TRUNCATE TABLE table_name [PARTITION (part_col = partval, part_col = partval)]

Removes all rows from table or partition

ALTER TABLE:

ALTER TABLE table_name SET TBLPROPERTIES (prop_name = prop_val, 'comment' = 'val')

ALTER TABLE table_name [PARTITION (~~part~~)] SET SERDE serde_class [WITH SERDEPROPERTIES (...)]

ALTER TABLE table_name [PARTITION (..)] SET SERDEPROPERTIES (...)

ALTER TABLE table_name CLUSTERED BY (col, col) [SORTED BY (col, col)] INTO num_BUCKETS.

This change will only update the Hive table metadata & use it for new partitions & data. Existing partitions and/or data is not reformatted.

ALTER TABLE table_name SKEWED BY (cols) ON (col values) [STORED AS DIR]

ALTER TABLE table_name NOT SKEWED;

Makes the table not skewed & turns off bucketing feature

ALTER TABLE table_name NOT STORED AS DIRECTORIES

turns off list bucketing feature

* SET hive.exec.dynamic.partition = true;
* SET hive.exec.dynamic.partition.mode = nonstrict
* Hive will not allow querying a partitioned table without partition col in where clause

ALTER TABLE table-name SET SKEWED LOCATION (col1 = loc1, col2 = loc2)

ALTER PARTITION :

ALTER TABLE table-name ADD [IF NOT EXISTS] PARTITION (...)
LOCATION 'location1'
PARTITION (...) LOCATION 'location2'

Example :

ALTER TABLE page-view ADD PARTITION (dt = '2009-01-01', country = 'US')
location 'path to/part080808'
PARTITION (dt = '2008-01-09', country = 'US') location 'path to/part-009'

ALTER TABLE table-name PARTITION (...) RENAME TO PARTITION (...)

ALTER TABLE table-name1 EXCHANGE PARTITION (...) WITH TABLE table2
moves partitions between tables.

PARTITION Recovery

If we add new partitions to table using hadoop fs -mkdir command, then partition metadata won't be present in HIVE metastore.
However we can run metastore check command which will automatically add metadata about these partitions.

MSCK REPAIR TABLE table_name;

DROP PARTITION

ALTER TABLE table_name DROP [IF EXISTS] PARTITION (...)
PARTITION (...)
[IGNORE PROTECTION] [PURGE];

ALTER TABLE table_name [PARTITION (...)] SET FILEFORMAT ORC | RAR etc.

ALTER TABLE table_name [PARTITION (...)] SET LOCATION 'new loc';

ALTER TABLE table_name [PARTITION (...)] ENABLE | DISABLE NO DROP [CASCADE]

if no drop is enabled table cannot be dropped but with no drop cascade the partitions also cannot be dropped. Normally no drop will allow partitions to be dropped even if table cannot be dropped.

ALTER TABLE table_name [PARTITION (...)] ENABLE | DISABLE OFFLIN

ALTER TABLE table_name [PARTITION (partkey = value, partkey = value)] CONCATENATE;

- combines many small RCFILE or ORC file within a table or partitions into larger files.

ALTER COLUMN

ALTER TABLE table_name [PARTITION (...)]
CHANGE COLUMN col_name col_new_name col_type
[FIRST | AFTER column_name] [CASCADE | RESTRICT]

Dynamic Partition may create many small files under each partition.
To prevent this we can partition by few essential column & within that partition create buckets based on one of the other columns.
Bucketing calculates hash on the column value & based on this assigns a bucket number & stores it in corresponding file.

* select from_unixtime(cast(substr(order_date,1,10) AS bigint))
from table_name

Converts a unix timestamp to a date in yyyy-mm-dd hh:mm:ss format

VIEWS

CREATE VIEW [IF NOT EXISTS] db_name.view_name
[(col_name, col_name)]
[TBLPROPERTIES (...)]
AS SELECT statement;

* if no column names are specified the view columns are derived automatically from select clause.

* View is pure logical. with no associated storage
* Views schema is frozen when its created. Subsequent changes to underlying tables will not be reflected in view schema. If change to underlying table is a way incompatible with view (example changing column type) any further attempts to query the view will fail.

* Views are read-only.

DROP VIEW [IF EXISTS] db_name.view_name;

ALTER VIEW db_name.view_name AS select_statement;
(This changes the definition of the view)

INDEX

```
CREATE INDEX index_name ON TABLE tablename (col1, col2)
    AS index_type
    [WITH DEFERRED REBUILD]
    [IDXPROPERTIES (...)]
    [IN TABLE index_table_name] [PARTITIONED BY (cols)]
    [ [ROW FORMAT ...] STOREDAS ! STORED BY
    ]
    [LOCATION hdfs_path] → location where index files would be stored
    [TBLPROPERTIES (...)] ;
```

index_table_name - name of the table where the index information is stored. if not specified, hive creates a table automatically using certain name.

* By default index partitioning matches the partitioning of the base table but can be overridden using PARTITION BY clause
* WITH DEFERRED REBUILD then newly created index is initially empty we should use to generate index data

ALTER INDEX index_name ON table_name [PARTITION(...)] REBUILD

MACRO

CREATE TEMPORARY MACRO macro_name (colname coltype, col_name coltype) expression ;

CREATE ~~TEMPORARY~~ TABLE MACRO simple_add (~~x int~~ y int) x+y ;

DROP TEMPORARY MACRO [IF EXISTS] macro_name ;

FUNCTION

CREATE TEMPORARY FUNCTION func_name AS class_name ;

- temporary means its available only in current session
- The class should be in the classpath of Hive. (The jar can be added using ADD JAR CLI command

ADD (FILE | JAR | ARCHIVES) Filepath1, Filepath2 ;

DROP TEMPORARY FUNCTIONS [IF EXISTS] func_name

* We can also use permanent functions by registering them to the metastore. (which means they are not just for a session)

CREATE FUNCTION dbname.func_name AS classname
[USING JAR | FILE | ARCHIVE 'file_uri', JAR | FILE | ARCHIVE file2_uri]

When the function is referenced for first time Hive adds the jar to the environment classpath. The function will be added either to the specified database or current database

SHOW

SHOW TABLES [IN dbname] ['identifier_with_wildcards'];

SHOW PARTITIONS table_name;

SHOW TABLE EXTENDED [IN | FROM dbname] LIKE '
PARTITION (...)

SHOW TBLPROPERTIES table_name;

SHOW CREATE TABLE dbname.tablename;

SHOW [FORMATTED] (INDEX | INDEXES) ON tablename [FROM|IN dbname]

SHOW COLUMNS (FROM|IN) table_name

DESCRIBE DATABASE [EXTENDED] db_name

DESCRIBE [EXTENDED|FORMATTED] tablename

DESCRIBE [EXTENDED|FORMATTED] tablename PARTITION (...)

hive> SET mapreduce.jobs.reducers = 3

* ORDER BY - executes only on 1 reducer since it needs to order the overall dataset. For huge datasets this will represent a performance problem.
  Hence we use SORT BY which does ordering per reducer

* DISTRIBUTE BY - Use the specified column to distribute the records between multiple reducers

* If we are sort by & distribute by have same columns (s) we can instead use CLUSTER BY <column>

HIVE DMLs

LOAD

LOAD DATA [LOCAL] INPATH 'file_path' [OVERWRITE]

INTO TABLE tablename [PARTITION(col = val, col = val2)]

- LOCAL → local filesystem (In this case the local file will be copied (NOT Moved) to the location need while creating table
- OVERWRITE - contents of target table (partition) will be deleted & replaced by the files otherwise files in filepath will be added to the table.
  (If target table has a file with same name with any file in filepath, the existing file will be replaced by new file.

INSERT [OVERWRITE] TABLE tablename [PARTITION (...)]
[IF NOT EXISTS] Select_statements;

Multiple inserts FROM from_statement
INSERT [OVERWRITE] TABLE table_name
[PARTITION (..)[IF NOT EXIST]] select
[INSERT [OVERWRITE] ... ;

OVERWRITE - will overwrite any existing data unless IF NOT EXISTS is specified.

* A table can be made immutable by setting TBLPROPERTIES ('immutable' = 'true'). if OVERWRITE specified immutable is ignored.

* In multi-insert the output of each select statement can be written to a specific table or partition.

* We cannot use "LOAD Data INPATH" for loading data into bucketed tables.
We need to use Insert overwrite Table ... Select ... From to load bucketed table from another staging table
*

Dynamic Partition Inserts

configuration : hive.exec.dynamic.partition = false | true.

hive.exec.dynamic.partition.mode = strict | nonstrict

- strict → atleast one static partition
- nonstrict → all partition can be dynamic

Example

```
FROM page_view_stg pvs
INSERT OVERWRITE TABLE page_view PARTITION (dt='2008-06-08', country)
SELECT pvs.viewtime, pvs.userid, pvs.url,
       pvs.ref_url, null, null, pvs.ip, pvs.cnt;
```

(dt='2008-06-08' : static partition; country : dynamic partition)

* Each dynamic partition creation is determined by value of input column. The dynamic partition columns are specified last in the select clause & in the same order as in the partition clause specified during table creation.
* Dynamic partition values are selected by ordering and not name & taken as last columns from select clause. (example select column name cnt and partition colum name country)
* If input column is different than string, it's converted to string to be used in the hdfs path.
* If input column has Null values or empty string, the row will be put in special partition whose name is controlled by hive.exec.default.partition.name

Sometimes the number of distinct (dt, country) combinations may exceed the max configured value per node. This is because the mapper will take random set of rows.

* for using bucketing we need to set
> set hive.enforce.bucketing = true

To solve this problem we can group the rows by dynamic partition columns in the mapper & distribute them to the reducers where dynamic partitions will be created.

eg: set hive.exec.dynamic.partition.mode = nonstrict

```
FROM page_view_stg pvs
INSERT OVERWRITE TABLE page_view PARTITION(dt, country)
SELECT pvs.viewtime, pvs.userid, pvs.url, pvs.ref_url, null, pvs.ip
    from_unixtimestamp(pvs.viewtime, 'yyyy-mm-dd')ds, pvs.country
DISTRIBUTE BY ds, country;
```

This creates a map-reduce job rather than a map-only job. The select clause will be converted to a plan to the mappers and output will be distributed to the reducers based on (ds, country) pairs. The insert clause will be converted to the plan to the reducers which writes dynamic partitions

Inserting into Local files

```
INSERT OVERWRITE [LOCAL] DIRECTORY '/tmp/pv_gender' [ROW FORMAT]
[STORED AS]
SELECT pv_gender_sum.* ← all columns
FROM pv_gender_sum;
```

UNION

Merges rows from two or more tables [the select clauses of the two tables should return the same schema.

```
INSERT OVERWRITE action_users
SELECT u.id, actions.date
FROM (
    SELECT av.uid AS uid ←
    FROM action_video av
    WHERE av.date = '2008-06-08'
    UNION ALL
    SELECT ac.uid AS uid ←
    FROM action_comment ac
    WHERE ac.date = '2008-06-08'
) actions JOIN users u ON (u.id = actions.uid)
```

projected as same column name

already same column name

alias for schema returned by union

Array : CREATE TABLE arraytable (int_array_col ARRAY<INT>)
Select pv.friends[2] from page_views pv;

Map : INSERT OVERWRITE page_views_map
SELECT pv.userid, pv.properties['page_type']
FROM page_views pv;

INSERT VALUES directly

INSERT INTO TABLE tablename
[PARTITION (partcol1 = val1, part_col2 = val2)]
VALUES values_col1, val_col2 ---;

UPDATE tablename SET col = value,
col = value
WHERE expr;

DELETE FROM tablename WHERE expr;

SELECT :

order by, sort by, cluster by, distribute by.

SELECT expression, expr FROM tablename
ORDER BY (col, col) ASC/DESC

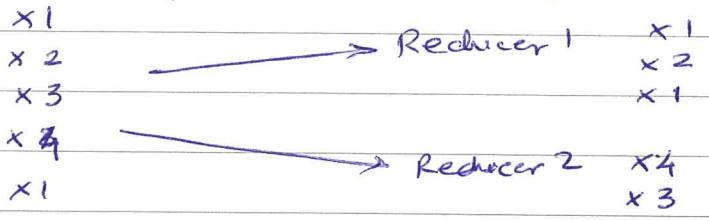
SELECT expr1, expr2 FROM src
SORT BY (col, col) ASC / DESC

difference: SORT BY sorts data per reducer
whereas ORDER BY guarantees total order of
the output

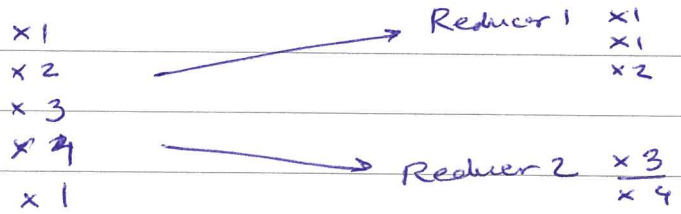
Cluster by and Distribute by are mainly used with Transform or Map-reduce scripts. But it can be used in Select also

Cluster By provides functionality of both DISTRIBUTE BY and sort by.

* DISTRIBUTE BY used to distribute rows among reducers. All rows with DISTRIBUTE BY columns will go to same reducer. However it does not guarantee sorting or clustering.



CLUSTER BY



Sort by is used instead of Cluster by if we want to distribute by columns and then sort by different columns

```
CREATE [EXTERNAL] TABLE hbase_table_emp (id int, name string
                                         role string)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES("hbase.columns.mapping" = ":key, ef1:name
                                                ef1:role")
TBLPROPERTIES ("hbase.table.name" = "emp");
```

TRANSFORM / MAP - REDUCE Scripts

- Used to plugin custom mappers - reducers Script

- By default, columns would be transformed to String and tab limited before feeding to user scripts (NULL values will be converted to \N char.
- The standard output of user script will be treated as TAB delimited string

Map- reduce

```
FROM
(
  FROM src
  MAP expr, expr---- [ROW FORMAT]
  USING 'my_map_script'
  ( AS colname, colname ---)
  [ROWFORMAT ... ] [CLUSTER BY | (DISTRIBUTE BY cols SORT BY cols
  src_alias
)
REDUCE expr, expr ... [ROWFORMAT ... ]
USING 'my_reduce_script'
  (AS colname, colname ... ) [ROW FORMAT ... ]
```

Transform ,

FROM (

FROM src

SELECT TRANSFORM (expr1, expr2 ...) [ROW FORMAT ...]

USING 'my-map-script'

(AS colname, colname ...) [ROW FORMAT]

[CLUSTER BY | DISTRIBUTE BY (cols) SORT BY (cols)] src_alias

)

SELECT TRANSFORM (expr1, expr2 ...)

[ROW FORMAT]

USING 'my-reduce-script'

AS colname, colname ... [ROW FORMAT ...]

Examples

FROM (

FROM pv_users

MAP pv_users.userid, pv_users.date

USING 'map-script'

AS dt, uid

CLUSTER BY dt ~~) map_output~~

) map_output

INSERT OVERWRITE TABLE pv_users_reduced

REDUCE map_output.dt, map_output.uid

USING 'reduce_script'

AS date, count;

FROM (

FROM pv_users

SELECT TRANSFORM (pv_users.userid, pv_users.date)

USING 'map_script'

AS dt, uid CLUSTER BY dt

) map_output

INSERT OVERWRITE TABLE pv_users_reduced

SELECT TRANSFORM (map_output.dt, map_output.uid)

USING 'reduce_script'

AS date, count;

UDF :

UDAF : (User Defined Aggregate function)

UDTF : (User defined Table generating function)

- normal UDF & UDAF take single input row and output single row.
- In contrast UDTF transform single input row to multiple output row

Ex : explode (ARRAY)
explode (MAP)

BEELINE CLI

%> beeline -u jdbc:hive2://<hostname>:<port>/db_name
-n <username> : / 10000

OR

%> beeline
beeline> !connect jdbc:hive2://<host>:<port> <username> <pw

jdbc:hive2:// ← Starts running HiveServer2 in embedded mode

HIVE CLI :

%> hive

hive> set -v → prints configs values

hive> set mapred.reduce.tasks = 32

hive -e "Query"
hive -f <filename> ← file containing hql.

> list FILES;
> list JARS
> list ARCHIVES

Beeline Connection URL when Hive Server 2 is running in HTTP mode

~~jdbc~~

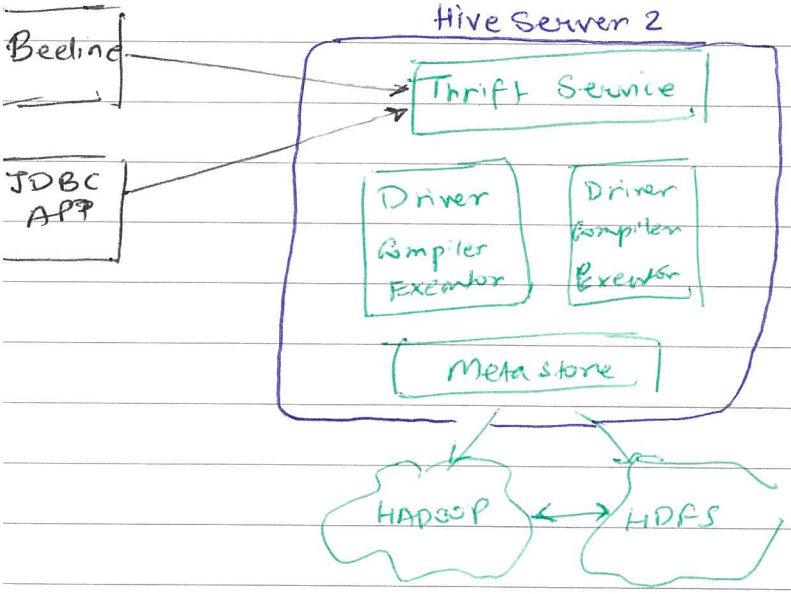
jdbc:hive2://<host>:<port>/<db>;transportMode=http;

~~10000~~ 10001

httpPath = <http-endpoint>

↑ Default CLIService

Hive Server 2 Architecture



Hive server 2 is a container for the Hive execution Engine (driver)

New client API

- TCLIService.thrift
- JDBC/ODBC drivers to be compatible with existing DB software

Packages: TCLIService.thrift is a binary protocol

JDBC driver: org.apache.hive.jdbc.HiveDriver

Beeline uses JDBC internally to connect to Hive

* Using JDBC/ODBC driver client like Tableau etc can access hive over JDBC.
* ~~Beeline~~ Can run multiple HiveServer2 instances for load balancing & high availability using zookeeper

HADOOP Admin details

Port	Name of Parameter	Description
50030	mapred.job.tracker.http.address	Application master admin web UI
50070	dfs.http.address	NM admin web GUI
50010	dfs.datanode.address	DN control port
50020	dfs.datanode.ipc.address	DN IPC port, used for block transfer
50060	mapred.task.tracker.http	address of per NM web UI
50075	dfs.datanode.http	address per datanode web
50090	dfs.secondary.http	address per SNN web GUI
50470	dfs.https	address NN web GUI https
50475	dfs.data node_https	address per DN web UI https

hadoop-env.sh - Used to set Hadoop Env settings such as java path etc.

core-site.xml - Used to configure tmp dir of NN

mapred-site.xml - Used to define no. of mappers, & other MR settings.

masters - used to specify SNN

Critical Configuration

→ defines prefix for all other dir/path info.

hadoop.tmp.dir = Used as temporary dir for local & HDFS

fs.default.name = used to specify NN host & port

mapred.job.tracker =

dfs.name.dir = Storage location of NN metadata in local FS

Comma or space seperated list of directories

All provided directories are used for redundant storage

dfs.data.dir = Storage location for DN blocks in the local file system
comma or space seperated list of directories

mapred.local.dir = local dir

Performance tuning configs

dfs.datanode.handler.count = Handles number of server threads in DN
dfs.datanode.du.reserved = reserved space in bytes per volume
dfs.replication = Sets the replication factor
fs.checkpoint.dir = Stores temporary images & merges them in need of a job in local FS of the DFS SNN.

mapred.local.dir.minspacestart = Limits the job tasks for exemption if the space is relatively less.

dfs.block.size = Changes default blocksize.
dfs.name.edits.dir = Storage location of NN edits file in Local FS

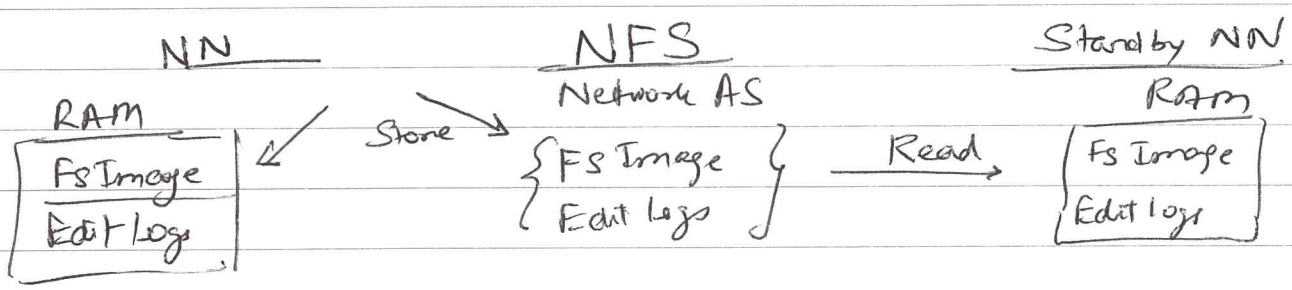
Security

hadoop.rpc.protection (core-site.xml) = Secure data transfer between Hadoop services & clients.
dfs.encrypt.data.transfer (set to true) = Secure transfer protocol of DN

CAP → Partition Tolerance → Replication & rack awareness
→ Availability - Node/Hardware Failure

Consistency - update to one copy of block automatically updates other copies of data. However Hadoop does not support this, since it uses Write Once Read many times -

Standby NN and Seconday NN are both optional. However, we cannot use both of them together, only one of them can be used at a time



*

SQOOP

- Used to import table or tables from RDBMS into Hadoop (HDFS, Hive, HBase)
- Used to export data from Hadoop to RDBMS
- Executes map-reduce job under the hood
- Supports importing data in delimited text file, Avro, sequence and many other formats

NOTE: Import process generates Java classes which encapsulates one record of import table. This class can be used for writing subsequent MR to process data.

Sqoop is collection of tools

Syntax: sqoop [tool name] [tool arguments]

toolnames: import

import-all-tables

export

help

create-hive-table - import table definition to hive

list-databases - list databases on server

list-tables - list tables in database

Sqoop Import:

[Common Arguments]

--connect <jdbc-url>

ex: --connect jdbc:mysql://host:port/db_name

--driver ~~com~~ - if Sqoop does not support a particular db directly (automatically) then we can connect to any JDBC compliant dbs by directly specifying driver class.

ex: --driver com.microsoft.jdbc.sqlserver.SqlServerDriver

--connect-manager <classname> - Specify connection manager class to use.

--password <password> - Database password (This approach is not recommended as it is insecure use --password-file option

--password-file <filename> Password file (echo -n <password> > pas

--password-file hdfs://< >/pass.txt

--username Database Username

[Generic Hadoop CLI Arguments]

- conf <config-file>
- D <property=value>
- files <csv list of files> - Specify files to copied to Cluster
- libjars <csv of jars> - Specify jars to include in CLASSPATH
- archives (csv of archives> - Specify Archive files to be unarchived on compute nodes.

sqoop import [Generic Args] [common-Args]

Using Option files to pass arguments

$) sqoop import --connect <jdbcurl> --username <BI> --table Test

can be written as

sqoop --options-file /user/work/import.txt --table Test

where import.txt contains

import
--connect
<jdbc-url>
--username

By default SQOOP stores imported in the /users home directory/<tablename> in HDFS. By default fields delimited by ','

Sqoop import

Command Sqoop import (generic-args) (import-args)

→ if target directory exist the sqoop-import fails unless this option is specified

--append	Append data to an existing dataset in HDFS
-as-avrodatafile	Avro file.
-as-sequencefile	
--boundary-query	Query whose results are used to create splits (Determine subset of records to be sent to different mappers)
--columns <col1, col2,...>	CSV of columns to import
--delete-target-dir	Delete the import target dir if exists.
--direct	Use direct connector to DB if exist (Faster than JDBC.)
--fetch-size	No of records to read from DB at once.
-m, --num-mappers <n>	number of map tasks to create
-e, --query <statement>	import results of the query.
--split-by	: Column of table used to create splits. If not specified, Sqoop uses PK of the source table. If ~~pk of~~ source table does not have a PK and if number of mappers is not 1 and --autoreset-to-one-mapper is not specified, it fails.
--table <tname)	source table to import
--target-dir	HDFS destination dir

- hive-import --hive-table ~~create~~ options create new hive table while ~~Cochin~~ sqoop import

-- By default hive import will insert 'null' string in String columns but hence does not work in Hive

--warehouse-dir HDFS parent for table destination

--where <clause> where condition to be used during import

-z, --compress Enable Compression

--compression-codec Default Gzip.

--null-string <char> The string to be written for NULL values in string columns of the same table

--null-non-string <char> The string to be written for NULL values of non-string columns.

Examples

This is VERY IMP!

① Free-Form Query Imports

> sqoop import --query 'select * from table t where $CONDITIONS' --split-by t.id --target-dir /user/foo/result

② ~~Direct~~ Import

> sqoop import --connect <jdbc> --username <> --password <>
--target dir /user/root/import-results/
files will be created under this dir

> sqoop import --connect --user --passwd --table foo
--warehouse-dir /user/root/import-results/
directory foo created /foo/(files)

Sqoop import to HDFS & Sqoop export are compatible without any special needs for delimited or null fields.

Isolation : By default Sqoop uses 'read committed' isolation but we can reduce isolation guarantees by --relaxed-isolation option

Controlling Column Type mapping

By default, sqoop automatically infers JAVA or Hive types from the Sql types from the RDBMS columns schema. We can override this by using.

--map-column-java <mapping>

--map-column-hive <mapping>

Eg: Sqoop import ... --map-column-java id=String,value=Integer

Incremental Imports

--check-column	Column to examine when doing import
--last-value	last of value of this column where increment import should begin
--incremental <mode>	Can be append or lastmodified used for column with increasing row id values (only new rows will be imported)

IMP
--append option is mandatory with this option

last modified - ~~each update~~ Use this option when source table may be updated and each such update will set the value of last-modified column to current timestamp. Rows where check column holds timestamp more recent than the timestamp specified in --last-value are imported.

(Used if we need to import new & updated records in src table. However the source table should have a timestamp column which gets updated everytime a record is inserted or updated.

* Importing to hive without specifying field-terminated-by ',' option makes '\001' as default field delimiter and hence this data cannot be exported back to RDBMS.

* Importing data to HDFS without specifying field-terminated-by will use ',' as default and hence such data can be exported back to RDBMS

Importing to Hive

--hive-import Import tables to hive

--hive-overwrite Overwrite existing data into Hive table.

--create-hive-table used to create source table schema in Hive if target table exists in Hive this will fail

--hive-table <tablename> By default Hive will import data in default database and use same name for target table as source table. We can override this by specifying --hive-table retail_db.customers.

--hive-database <dbname> target hive database (~~default~~)

--hive-partition-key Name of hive field to partition on

--hive-partition-value <v> - String value that serves as partition key for this imported into hive in this job.

NOTE: Hive using \N to denote NULL values hence we can specify --null-string & --null-non-string to control this

> Sqoop import ... --null-string '\\N' --null-non-string '\\N'

- -hive-import cannot be used with --sequence or avro files.

Examples:

> Sqoop import --connect --user --pass ...
--table < > --columns "emp_id, first_name, last_name"

Delimiters

> Sqoop import ...
--fields-terminated-by '\t'
--lines-terminated-by '\n'
--optionally-enclosed-by '\"'

sqoop import to Hive ~~CLI~~ with default option ~~none~~ use 'null' string to represent null values from RDBMS and '\001' as default field delimiter. Queries with is NULL or IS NOT NULL will not work since hive nulls are represented as '\\N'.

Where

> sqoop import ... --table Employees
--where "start_date > '2010-01-01'"

Changing default split column

> sqoop import ... --table-name Employees
--split-by dept_id.

Performing Incremental import after already importing 100,000 rows

> sqoop import ... --table sometable --where "id > 100000"
--target-dir /user/root/result --append

SQOOP Import All tables

* Most [common Arguments] are same as ~~Hive~~ Sqoop import command

Syntax :> sqoop import-all-tables (generic-args) (import-args)

↓

Hadoop generic args like -D, -conf etc

Output formatting

--escaped-by <ch> sets the escaping character

--fields-terminated-by <char> field seperator

eg: --fields-terminated-by ','

Input parsing arguments

--input-escaped-by <ch>

--input-fields-terminated-by <ch>

--hive-drop-import-delims	Drops \n, \r and \01 from string fields when importing to Hive
--hive-delims-replacement	Replace \n, \r and \01 from string fields with user defined string when importing to Hive

Sqoop Exports

* Export tool exports data from HDFS back to RDBMS
  The target table must already exist in RDBMS.
  The input files are read & parsed into records according to user specified ~~database~~ delimiters

Syntax

sqoop export (generic args) (export-args)
sqoop-export (generic args) (export args)

[Common Args]

--connect <jdbc url>
--connection-manager <class name>
--driver <class name>
--help
--password-file
--password <password>
-P Read password from console.
--username <user>
--relaxed-isolation

[Exp] * The target table & database must be already created before the export

* The staging-table option cannot be used in with ~~update only~~ mode
-- update mode param

[Export Control Arguments]

--columns <col1, col2, ...>	Columns to export.
--direct	
--export-dir <dir>	HDFS source path
-m, --num-mappers <n>	number of map task to use.
--table <tablename>	target RDBMS table to populate.
--call <stored-proc-name>	Stored proc to call
--update-key <col-name>	Anchor column to use for updates (CSV list) [Used for updating existing records based matching value in this column.
--update-mode <mode>	Specifies how updates are performed updateonly - only updates existing & matching records. If no match is found does not perform inserts. allowinsert - updates matching rows else inserts if no match
--input-null-string <string> --input-null-non-string <string>	How to treat NULLs when reading from source.
--staging-table <table-name>	The table in which data will be staged before inserting in destination table.
--clear-staging-table	Indicates any data in staging table can be deleted.

Since Sqoop breaks down export process into multiple transactions. It is possible that a failed export job may result in partial data being committed to the database. This can cause subsequent jobs also to fail due to collisions or duplicate data.

To overcome this we can use a staging table which acts as an auxillary table that is used to stage exported data. The staged data is then moved to destination table in single transaction.

→ To use this feature the staging table must be manually created before running export job. It should have the same schema as the target table.

→ The table should be empty before export job runs. We can ensure this by specifying --clear-staging-table options, which deletes existing data in the staging table before running the export.

Exports & Transactions

Sqoop uses multi-row insert upto 100 rows per statement. It commits after every 100 statements (ie after every 10000 rows)

✳ Therefore sqoop export is not a atomic process - Partial results will become available before export is complete.

✳ Each map task operates in a seperate transaction. If the transaction fails it rolled back. But transaction from other mappers may still be successful resulting in partially completed export.

Examples

> sqoop export --connect jdbc:mysql://----
--table foo --export-dir /results/foo-data

Stored proc.

> sqoop export --connect ...
--call ~~sto~~ foo-proc --export-dir /results/bar-data

Validation - determinates error margin between source & target is acceptable. (Absolute, %, etc)

ValidationThreshold
~~3 basic Interface~~.

Default implementation is AbsoluteValidationThreshold which ensures the row counts from source & target is same

~~org~~.

ValidationFailureHandler - handling failures.

Default LogOnFailureHandler that logs a warning message to configured logger

Validator - Drives validation logic by delegating the decision to ValidationThreshold & failure to ValidationFailureHandl

Default implementation is RowCountValidation

validation-threshold = org.apache.sqoop.validation.AbsoluteValidationThresho

validation-failurehandler = org.apache.sqoop.validation.
AbortOnFailureHandler

validator = org.apache.sqoop.validation.RowCountValidator.

> sqoop import (-- ...) --validate

> sqoop import (- - - -) --validate

--validation-threshold org.apache.----

--validator org.apache.---

--validation-failure handler org.apache.---

SQOOP Jobs

* Allows to save imports & export to be saved as jobs & executed multiple times.

sqoop job (generic-args) (job-args) [-- [subtool name] (subtool args)

sqoop job (generic args) (job-args) [-- Subtool] [subtool args]]

[Job Args]

--create <job-id>	Create a job
--delete <job-id>	delete a job
--exec <job-id>	executes a job
--show <job-id>	Show parameters for a job.
--list	lists all the jobs.

--meta-connect <jdbc url> Connection URL for the Metastore

--help

* NOTE: Sqoop metastore is not secure hence sqoop does not store passwords in metastore & prompts for it each time a job is executed.

We can enable password storage in sqoop metastore by setting sqoop.metastore.client.record.password = true. in the config.

This has to be done when automating sqoop jobs via Oozie.

Sqoop - create - hive - table

- This tool populates the Hive metastore with the definition for a table based on database table.
- Used if the data is already imported to HDFS
- Also used to create table definitions in Hive. Data can then be imported into the HDFS warehouse directory which corresponds to this table

Sqoop create-hive-table (generic args) (create-hive-table args)

[Common Args] --connect, --username, --password, --driver, etc are all same as before

[Hive Arguments] --hive-table, --hive-database, --hive-overwrite etc are same as before

[output formatting] --fields-terminated-by etc same as before

Example

> sqoop create-hive-table --connect ( )
--table employees --hive-table retail.emp

Sqoop-eval : Used to quickly run queries & print results to console

> Sqoop-eval (generic args) (eval-args)

ex:

> Sqoop eval --connect ( )
--query "Select * from employees Limit 10"

Sqoop - list - database : list databases on server

> sqoop list-database (generic args) (args)

ex:

> sqoop list-databases --connect (host:port)

Sqoop - list - tables : shows tables within db

> sqoop list_tables --connect (host:port/dbname

Sqoop -help [tool name]

Sqoop - version

Sqoop export from Partitioned Hive table

* Export individual partitions seperately
* In the export the partition column key will not be inserted you have to issue update statement on target database for doing this.

> sqoop export --connect "host:port/retail_db" --username --password
--table ~~depts~~ employees
--export-dir /apps/hive/warehouse/retail_db.db/employees/gender=M
~~gender=M~~

if we need to update existing ROWS table

> sqoop export --connect ( ) --username ( ) --password ( )
--table ~~employees~~ departments
--export-dir /apps/hive/warehouse/retail_db.db/departments
--update-key department_id
-- update-mode (update-only | allowinsert)

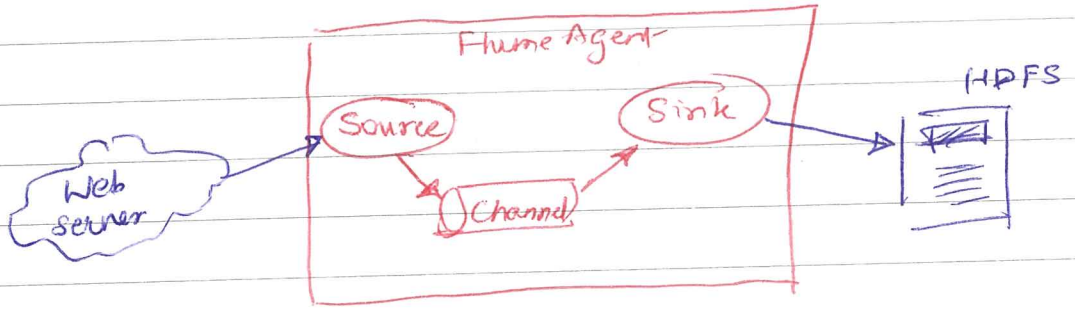
This indicates column to use to compare records from source & target

update-only → Only update existing rows

allowinsert → upsert - update existing
- insert new

FLUME :

- is a reliable, distributed and available system for efficiently collecting, aggregating and moving large amount of log data from many different sources to a centralised data store



Source :- A flume source consumes events delivered to it by external source like Web server.

- The external source sends events to Flume in a format that is recognised by the Flume source.

eg: Avro source is used to receive events from Avro clients. OR other Flume Agent that send events to Avro sink.

Sink : The sink removes the event from a channel & puts it into external repository like HDFS (via Flume HDFS sink) OR forwards it to the Flume source of next Flume agent in the chain.

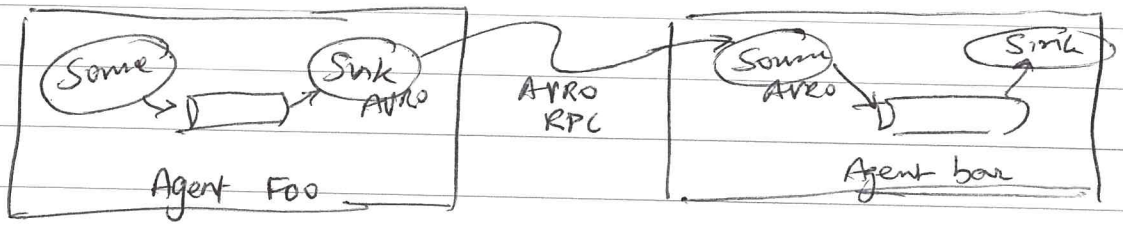
* The source & sink run async respective to one another with events staged in the channel.

Reliability : The events are removed from a channel only after they are stored in the channel of the next agent or in the final destination.

Additionally the sources & sinks encapsulate the storage and retrieval of data in a transaction.

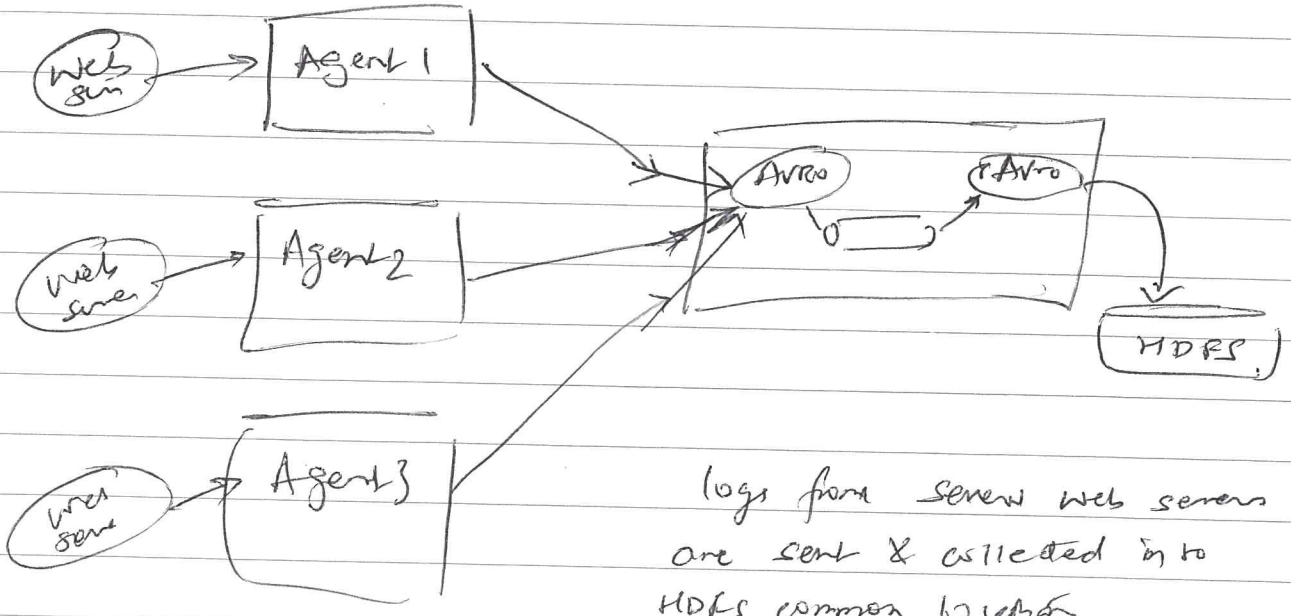
Recoverability - Flume supports durable file channel which is backed by the local file system.
(Memory channels are transient)

Multi-Agent flow



Sink of Agent "foo" & source of agent "bar" should be compatible (eg: in this AVRO) with the sink pointing to the host & port of the source of "Bar" Agent.

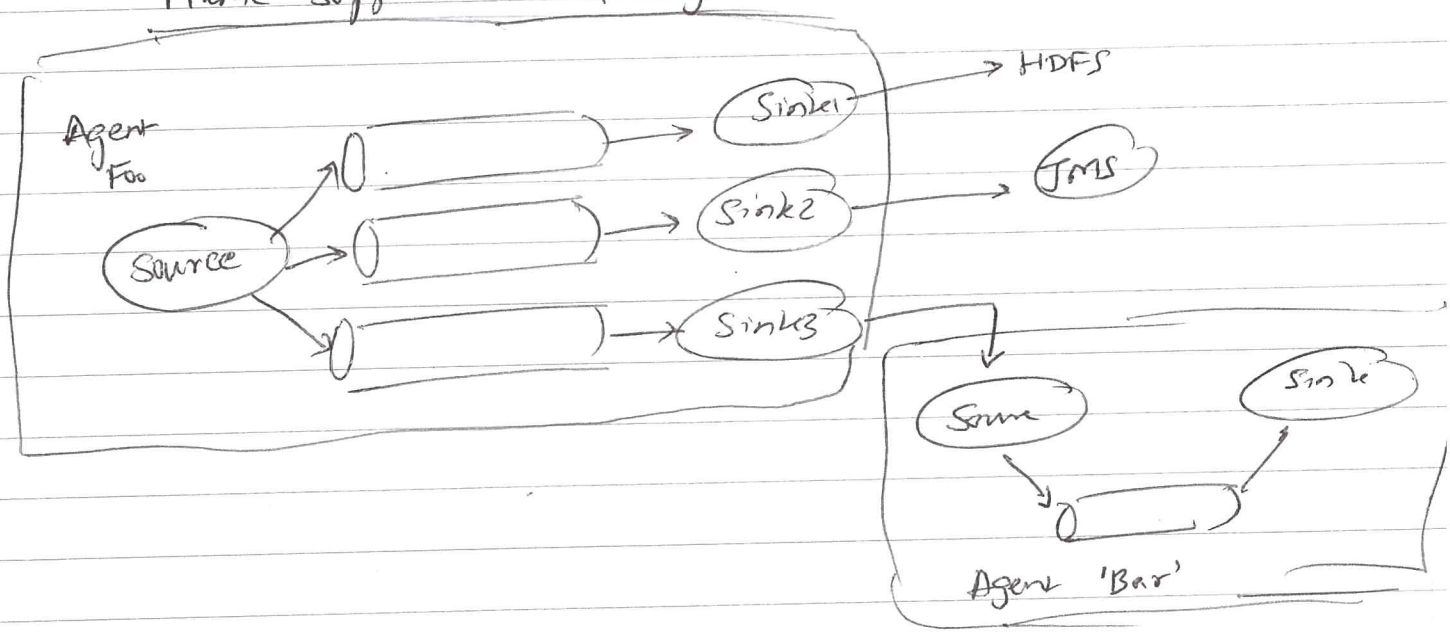
Consolidation



logs from several web servers are sent & collected into HDFS common location

Multiplexing :

Flume supports multiplexing event to one or more destination



Multiple flows in a single Agent

```
# list sources, sinks, channels
agent_foo.sources = avro-AppServer-src1 exec-tail-source2
agent_foo.sinks = hdfs-cluster-sink1 avro-forward-sink2
agent_foo.channels = mem-channel-1 file-channel-2

# flow1 config
agent_foo.sources.avro-AppServer-src1.channels = mem-channel-1
agent_foo.sinks.hdfs-cluster-sink1.channel = mem-channel-1

# flow2 config
agent_foo.sources.exec-tail-source2.channels = file-chanel-1
agent_foo.sinks.avro-forward_sink2.channel = file-channel-1
```

Multi-Agent flow

Weblog agent config

```
# list sources, sinks, channels
agent-foo.sources = avro-AppServer-source
agent-foo.sinks = avro-forward-sink
agent-foo.channels = file-channel

# define flow
agent-foo.sources.avro-AppServer-source.channels = file-channel
agent-foo.sinks.avro-forward-sink.channel = file-channel

# avro sink properties
agent-foo.sources.avro-forward-sink.type = avro
agent-foo.sources.avro-forward-sink.hostname = 10.1.1.100
                                      .port = 10000
```

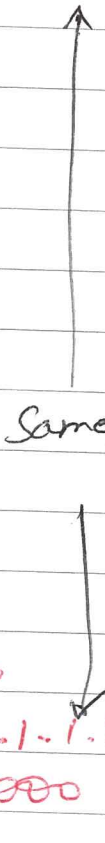
---- other config

HDFS agent config

```
# list sources, sinks, channels
agent-bar.sources = avro-collection-source
agent-bar.sinks = hdfs-sink
agent-bar.channels = mem-channel

# define flow
...
..

# avro source properties
agent-foo.sources.avro-collection-source.type = avro
                                      .bind = 10.1.1.100
                                      .port = 10000
```



Third party plugins

* Flume automatically picks up plugins jars from the directory $FLUME_HOME/plugins.d.

* Each plugin (subdirectory) under plugins.d can have 3 subdirectories
  1. lib - the plugin's jar(s)
  2. libext - the plugin's dependency jars
  3. native - require native libraries like (.so) files

Command-to-run

> flume-ng agent | help | avro-client | version | password
↳ create password file for use in flume conf

Global options

-c, --conf <conf>	Use config in <conf> directory
--classpath, -C <cp>	append to classpath
--plugins-path <dirs>	csv list of directories (plugins.d)
-D prop=value	

Agent Option

--conf-file, -f <file>	flume agent configuration.
--name, -n <name>	name of flume agent

Avro-client options

--rpcProps, -P <file>	RPC client properties file with server connection params
--host, -H <host>	hostname to which events will be sent
--port, -p <port>	port of the AVRO source
--dirname <dir>	directory to stream to AVRO source
--filename <file>	Text file to stream to AVRO source

--headerFile, -R <file> File containing event headers as key/value pairs on each new line.

Password options

--outfile The file in which encoded password is stored

> flume-ng -n "myagent -c /etc/flume/conf
-f myflume.properties
-Dflume.root.logger = DEBUG, console

Interceptors :

* Can modify or drop events based on any criteria specified by developer
* It is possible to chain interceptors by provider space seperated list. (They are invoked in same order as they appear in configuration
* Interceptors are specified only on Sources.

Selectors :

* Are use to select a particular channel to send events to, based on certain criteria
* Can be applied only on Sources.
* Default is "Replicating" selector if not specified. It forwards the event on each specified output channel.

Common Sources :

~~AVRO Source~~ : * Common Source properties

→ channels = Space Seperated list of channels to send events to

selector.type = Selector to use

selector.* = Selector specific properties

interceptors = Space seperated list of interceptors

interceptors.* = interceptor specific properties

→ type = type of source

AVRO Source : listens to AVRO port & receives events from Avro client streams

type = avro

bind = host/IP

port = port to bind to

Thrift source

type = thrift

bind = host/IP to listen

port = port to bind to

Exec Source : Runs a unix command on startup & expects that process to continously produce data on std out
If process exits for any reason, the source also exits & produces no further data.

type = exec
command = Command to execute
restart = true | false whether command should be re-executed if it dies.

JMS source : reads messages from JMS destination such as queue or topic.

Spooling-directory Sources

* allows ingestion of files placed into this directory.
* Watches the directory for new files and parse events out as new files appear.
* Parsing logic is pluggable.
* Only uniquely named files can be dropped in the dir.

type = spooldir
SpoolDir = directory to watch
fileSuffix = Suffix to append to completely read files (.completed)
deletePolicy = Whether to delete files after consuming (never | immediate)
fileHeader = true | false (Whether to add header storing the absolute path name)
fileHeaderKey = Header key to use when appending absolute path filename to event header.
basenameHeader = true | false Whether to add a header storing name of file
basenameHeaderKey = basename (accessed as %{basename}) Key to use when storing filename in header

ignorePattern = ^$ Regex to specify files to skip

trackerDir Directory to store metadata about which files were consumed

deserializer : Specify the deserializer to parse files into events. Default is LINE which means to parse each line as event.
The custom deserializer should implement- EventDeserializer.Builder

deserializer.* : deserializer specific properties

Event Deserializers

LINE

deserializer.maxLineLength - no. of characters to read in one event (2048)

AVRO : deserialize AVRO container file & reads one AVRO record into one event.

Blob Deserializer - reads BLOB per event - The blob is buffered in RAM hence not suitable for very large objects

Twitter Source: continously download tweets, converts them to AVRO format and sends AVRO events to a downstream Flume sink.

type = org.apache.flume.source.twitter.TwitterSource
consumerKey = OAuth consumer key.
consumerSecret = OAuth Consumer secret.
accessToken = OAuth Access token
accessSecret = OAuth Token secret

Kafka Source:

Netcat Source: listens to given port & turns each line of text into event.

type = netcat
bind = host/ip
port =
max-line-length = 512

Sequence Generator Source - used for testing

type = sequence

Syslog sources

Syslog Tcp Source
Multiport Syslog Tcp Source
Syslog UDP source

HTTP Source - accepts Flume events by HTTP POST or GET

type = http
bind = host/IP
port
handler = FQCN of handler class
default : org.apache.flume.source.http.JSONHandler
handler.*

FLUME Sinks

HDFS Sink : writes events to HDFS. Supports text & sequence files

* It can be rolled (based on count, size, duration)
* can be partitioned data based on timestamp, host etc.

type = hdfs
hdfs.path =
hdfs.prefix = FlumeData
hdfs.suffix =
hdfs.rollInterval = 30
hdfs.rollSize = 1024
hdfs.rollCount = 10
hdfs.inUsePrefix = -
hdfs.inUseSuffix = .tmp
hdfs.writeFormat = Text | Writable.

Hive Sink : Sinks events containing delimited text or JSON directly into Hive table/partition.

* Uses Hive transactions to write ~~Hive~~ data.

type = hive

hive.metastore = URL of metastore (eg: thrift://.../

hive.database

hive.table

hive.partition = CSV list of partition values

serializer - class responsible to parsing events (JSON or Delimited

AVRO sink : Flume events are converted to AVRO events & sent to configured Host/Port.

type = avro

hostname =

port.

Thrift sink : Similar to AVRO but uses thrift events

type = thrift.

File Roll Sink : Stores events in local FS.

type = file_roll

sink.directory = path in Local FS.

sink.rollInterval

NULL sink. : Discards events. Used to drain a channel.

Logger sink : Used to send events to logger. Used for testing.

HBase Sink : Writes events to HBase.

* User class implementing HBaseEventSerializer to convert events into HBase puts/increments.

type = hbase

table =

columnFamily = name of column family

Serializer - org.apache.flume.sink.hbase.SimpleHBaseEventSerializer

FLUME Channels

Memory channel : event stored in memory.

: data stored in channel is lost if agent fails.

type = memory

capacity = 100 (max events stored in channel)

transactionCapacity = 100 (max event taken from source or given to sink in one transaction)

JDBC channel : events persisted in database.

* Currently supports embedded Derby

type = jdbc

db.type = DERBY

driver.class = org.apache.derby.jdbc.EmbeddedDriver

driver.url JDBC URL

db.username

db.password

create.schema = true

create.index = true

create.foreignkey = true

transaction.isolation = Read Committed

maximum.connections = 10

maximum.capacity = 0 (unlimited)

Kafka Channel

File Channel : Stores the events in files until consumed by sink. . Protects against Agent failures.

type = file.
checkpointDir = dir where checkpoint file is stored
dataDirs = CSV list of dirs for storing log files
maxFileSize - max size in bytes for single log file
checkpointInterval = time between checkpoints.
transactionCapacity = max transaction size (events per txn.)
capacity - max ~~capacity~~ events stored in channel

Spillable Memory Channel

FLUME channel selectors

default = replicating.

selector.type = replicating

Multiplexing

* selects one of the available channels based on certain criteria. (example header)

selector.type = multiplexing

selector.header = (header name, key)

selector.mapping.* (property per value of the header key)

eg:

- .header = State
- .mapping.NY = Channel 1
- .mapping.IL = Channel 2
- .mapping.CA = Channel 3
- .mapping.default = Channel 4

FLUME Interceptors : modify event in transit

Timestamp Interceptor : inserts event header containing time in millis at which event is processed

type = timestamp

preserveExisting = true/false. if a timestamp header already exist preserve it

Hostname Interceptor

type = host

hostHeader = header key.

useIP = true/false

preserveExisting = true/false

Static Interceptor : append static header key with a static value to all events

type = static

key = Header key.

value = static value

preserve Existing

HIVE

Insert into using sql literals
Update
Delete

Limitation of Inserts

* Insert into Table <Tablename> Values (, , ,)
  Values must be provided for each column of the table
* Dynamic partitioning supported same as Insert ... select
* Insert, Update, Delete not supported on Sorted tables (i.e. tables with SORTED BY clause)

Limitation of Update

* Update <Tablename> SET = <Value> Where expr
* Subqueries not supported
* Partitioning columns cannot be updated
* Bucketing columns cannot be updated.
* Vectorization needs to be turned off for Update (This is automatic, Updated tables can be still queried using Vectorization)

Limitations of Delete

DELETE FROM <table> WHERE Expr

Overall Limitations for Insert, Update, delete (Hive Transactions)

→ Begin, Commit, Rollback are not supported, All language operations are Auto-commit

→ Tables must be bucketed to use these features (Table must support ACID)

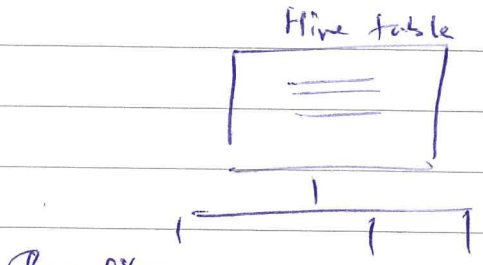
→ hive.txn.manager should be DbTxnManager

(TBLPROPERTIES ('transactional' = 'true'));

Table creation should have Clustered by
Stored as orc
TBLPROPERTIES('transactional' = 'true');

→ Only Snapshot Isolation level supported.
→ Only ORC file format is supported. (Only for Update & Delete
~~Insert~~ can be done with other file format

Architecture



For every transaction

During read (query) the base files & delta files (contain update & delete) are combined to provide result.
However in this case the result is only in memory & not stored.
Therefore periodically we can use compaction

Minor Compaction : combines many delta's into one delta. per bucket.

Major Compaction : takes one or more delta files & base file per bucket & writes them into new base file per bucket.

Conf

hive.txn.manager = org.apache.hadoop.hive.ql.lockmgr.DbTxnManager

hive.support.concurrency = true.
hive.enforce.bucketing = true.
hive.exec.dynamic.partition.mode = nonstrict

VECTORIZED QUERY EXECUTION IN HIVE

* → Normal query execution processes one row at a time.

→ Vectorized query execution streamlines this by processing 1024 rows at a time. (blocks).

→ Within each block each column is stored as vector

Configuration

> SET hive.vectorized.execution.enabled = true

VARCHAR datatype is NOT supported in vectorization.

* Vectorized query execution requires data to be stored in ORC format.

MapReduce Imp Conf

mapreduce.map.output.compress = true

mapreduce.map.output.compress.codec = org.apache.hadoop.io.compress.GZipCodec

mapreduce.output.fileoutputformat.compress = true

mapreduce.output.fileoutputformat.compress.type = NONE | BLOCK | RECORD

mapreduce.output.fileoutputformat.compress.codec = org.apache.hadoop.io.compress.GZipCodec

mapreduce.job.maps = 2

mapreduce.job.reduces = 3

org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputFormat

org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat

Important Configuration Properties

Hive

hive.execution.engine = mr | tez

hive.exec.dynamic.partition = true

hive.exec.dynamic.partition.mode = nonstrict | strict

hive.exec.max.dynamic.partitions = 1000

hive.exec.max.dynamic.partitions.pernode = 100

-> hive.mapred.mode = strict (Does not allow queries on partitioned table without having atleast 1 partition col. in the where clause)

hive.exec.parallel = true

hive.enforce.bucketing = true.

hive.exec.compress.intermediate = true

hive.exec.compress.output = true.

hive.support.concurrency = true.

hive.txn.manager = org.apache.hadoop.hive.ql.lockmgr.DbTxnManager

hive.vectorized.execution.enabled = true.

hive.vectorized.execution.reduce.enabled = true.

io.seqfile.compression.type = BLOCK | RECORD | NONE

Pig Imp Config

SET execType mapreduce | tez ;

SET default_parallel 3 ;

org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat

- AvroContainerOutputFormat

org.apache.hadoop.hive.serde2.avro.AvroSerDe

org.apache.hadoop.hive.ql.io.orc.OrcSerDe

- OrcInputFormat
- OrcOutputFormat

IMPALA

→ Integrates with Hive metastore. So tables created by Hive can be queried by Impala & vice versa.

→ High level of integration with Hive QL.
→ optimized for Parquet file format.

3 Main Components

① Impala Daemon

→ runs on each DataNode of the cluster
→ impalad process
→ reads & write data files, accepts queries transmitted from the impala-shell command, Hue, JDBC or ODBC parallelizes the query & distributes work across the cluster & transmits intermediate query results back to central coordinator node.
→ The query can be submitted to ~~any~~ Impala Daemon running on any data Node & this instance serves as the coordinator node for that query. The other nodes transmit partial results back to the coordinator, which constructs final result set for the query.

→ The impala daemons are in constant communication with the impala statestore., to confirm which nodes are healthy & can accept more ~~tasks~~ work

java -jar <jarfile> getschema > dep.avsc

/ /

(2) Impala Statestore

→ Checks the health of all impala daemons on all DN of the cluster and relays findings to each daemon.

→ proces = statestored

→ only needed to run on one node

→ if any impala daemon goes offline due to any reason the statestore informs all other daemons about this.

→ If the statestore itself fails, the impala daemons continue to run & distributed work as usual but will become less robust

(3) Impala Catalog Service

→ relays changes in metadata to all DN.

→ process = catalogd

→ required to run on only one node.

→ since all request to/from catalog are passed through the statestore its normal practice to have statestore and catalog service running on same host.

→ when there is change to SQL objects like schema change, drop tables, create tables etc catalog services informs about this to all DN. thus avoiding the need to ~~issue~~ issue INVALIDATE METADATA or REFRESH <table name> when meta data changes are made through Impala

→ However any meta data change done through Hive will not be communicated. Here we have to issue INVALIDATE METADATA OR REFRESH command.

Impala SQL Reference

→ high degree of compatibility with Hive QL.
→ Same datatypes as Hive
→ many built-in functions with same name & parameters as Hive
→ Similar DML to Hive

Operators

+, -, *, /, %, &, |, ^

expr BETWEEN x AND y.

=, !=

EXISTS - subquery has any results (returns true if subquery returns NULL)

expr IN (expr1, expr2, ...) -

IS NULL, IS NOT NULL

LIKE

AND, OR, NOT

* show functions in _impala_builtins like '*substring*'

CREATE DATABASE [IF NOT EXISTS] db_name [LOCATION hdfs_path]

CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [dbname.] tablename
(colname datatype, ...)
[PARTITIONED BY (colname datatype, ...)
[WITH SERDEPROPERTIES ('key1'='value', ...)
[
ROW FORMAT DELIMITED FIELDS TERMINATED BY 'char' ESCAPED BY '.'
[LINES TERMINATED BY ' ']
]
[STORED AS PARQUET | AVRO | TEXTFILE | SEQUENCE FILE | RCFILE]
[LOCATION 'hdfs_path']
['TBLPROPERTIES ('key1'='value', ...)]

CTAS

CREATE [EXTERNAL] TABLE [IF NOT EXISTS] dbname.tablename
ROWFORMAT DELIMITED ...
STORED AS PARQUET | TEXTFILE
LOCATION hdfs_path.
AS
SELECT ...

LOAD DATA INPATH 'hdfs_file or dir' [OVERWRITE] INTO TABLE tablename
[PARTITION (partcol1=value1, ...)]

INVALIDATE METADATA [[dbname.][tablename]] ← more expensive

REFRESH [db_name.] tablename ← one table only, less expens

Subqueries:

→ Scalar subquery produces result set with one row, containing single column (basically single value)

→ Uncorrelated queries do not refer to any tables from outer block of query.
eg. SELECT x FROM t1 WHERE x IN (SELECT y FROM t2);

→ Correlated subqueries compare one or more values from outer query block. to values referenced in the where clause of the subquery.
eg: SELECT empName, empId FROM Employees one
WHERE salary > (SELECT Avg(salary) FROM Employees two
WHERE one.dept_id = two.dept_id)

→ Subquery in FROM clause (to create temporary tables)
SELECT select_list FROM (SELECT statements) table_ref
WHERE ----

ORDER_BY, GROUP_BY, HAVING, LIMIT

SELECT selectlist FROM table
WHERE cond
GROUP BY col_expr
HAVING cond
ORDER BY col ASC/DESC
LIMIT num;

DISTINCT : removes duplicates .

Built-in Function Reference

* Mathematical functions

abs (numeric type)

ceil (double)

floor (double)

round (double)

max_int(), max_smallint() ---> returns largest value of corresponding data type

min_int(), min_smallint() ->

rand(), rand(seed) -> generates random number.

* Type Conversion

Cast (expr as type)

eg: ~~eee~~ select cast(empid as string) from employees

✱ DateTime functions

now() - gets current time. returns timestamp datatype

datediff (timestamp enddate, timestamp startdate)

day (timestamp)
dayname (timestamp) - return name of day 'Sunday', ----
month (timestamp)
year (timestamp)
hour (timestamp)
minute (timestamp)
second (timestamp)

extract (unit from timestamp)

eg: extract (year from now())
extract (now(), "year")

from_unixtime (bigint unixtime, string format)
- returns string in specified format from unixtime

unix_timestamp (string datetime, string format)

unix_timestamp () - returns current Unix time

To convert string to timestamp simply cast

eg:
cast ('2016-06-02 16:25:31' As timestamp)

* String functions

concat (str1, str2, str3 ...)

instr (str , substr) → returns index (starting from 1) of substr in the str.

length (str)
lower (str)
ltrim (str)
upper (str)
rtrim (str)
trim (str)
reverse (str)

Substr (str, startIndex , len) - gets substring from str.

* Conditional functions (Switch case)

```
SELECT CASE X
    WHEN 1 THEN 'one'
    WHEN 2 THEN 'Two'
    WHEN 0 THEN 'zero'
    ELSE 'OUT OF RANGE'
END <ALIAS>   (column)
FROM <table name>
```

✳ ANALYTIC FUNCTIONS

→ Also called 'Window' functions

→ work on multiple rows

→ but rather than be limited to one result set per "GROUP BY" group they operate on Windows where the input rows are ordered and grouped using flexible conditions expressed using OVER() clause

→ function like LAG() or RANK() are purely used in analytic context.

→ functions such as SUM, MAX, MIN, AVG etc can be used to calculate overall aggregates or GROUP BY aggregates but can be used with OVER() clause to do these computations in a window

OVER() clause

→ required for pure analytic functions LEAD(), LAG, RANK etc

→ can be optionally used with aggregate func.

Syntax

func(args) OVER ([partition by clause] [order by clause] [window clause])

→ partition by - SPLIT the input rows based on columns supplied.

→ order by - order the rows within the partitions based on the column specified

→ window clause - only allowed in combination with order by

DENSE_RANK() OVER ([partition-by] order-by)

→ used for top N or bottom N queries

→ partition-by is optional

eg: // Select top 3 priced products per category.

```
SELECT product_cat_id, prod_id, prod_name, prod_price
FROM
(
    SELECT p.*, dense_rank
    dense_rank() OVER (PARTITION BY p.prod_cat_id
                       ORDER BY p.prod_price DESC) dr
    FROM products p
) t1 ← subquery table alias
WHERE dr <= 3
```

dr ↓ dense rank column alias

AVG ([DISTINCT | ALL] expr) [OVER (analytic clause)]

(Commulative Avg product price per category.

```
SELECT prod_cat_id, avg(prod_price) over(partition by prod_cat
                                         order_by prod_price)
                                         as Cummulative_Avg.
```

COUNT ([DISTINCT | ALL] expr) [OVER (analytic clause)]

FIRST_VALUE(expr) OVER ([partition by ..] order_by -- [window])

- first value of expression within the window.

LAST_VALUE(exp) OVER (--)

MAX (expr) OVER (---)

MIN (expr) OVER (---)

RANK () OVER (--)
- Sparse rank.

SUM (expr OVER (----)