# HADOOP (2.7)

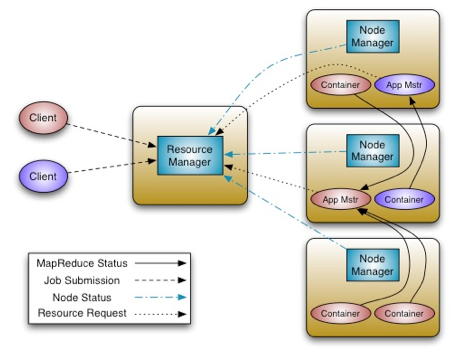
Hadoop Common: The common utilities that support the other Hadoop modules.

Hadoop Distributed File System (HDFS™): A distributed file system that provides high-throughput access to application data.

Hadoop YARN: A framework for job scheduling and cluster resource management.

Hadoop MapReduce: A YARN-based system for parallel processing of large data sets.

Hadoop 2.x (with YARN)



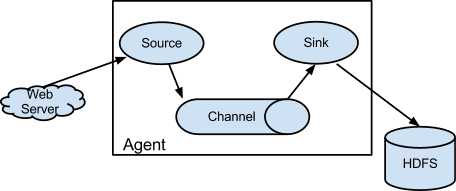
# EcoSystem

## Sqoop (v1.4)

Apache Sqoop is a tool designed for efficiently transferring bulk data between Apache Hadoop and structured datastores such as relational databases.

## Flume (v1.7)

Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data.



## Oozie (v4.3)

Workflow scheduler for Hadoop jobs.

Control flow nodes – start, kill, end

Action nodes can be – mapreduce, fs, java, pig, hive, sqoop, email

<workflow-app name='wordcount-wf' xmlns="uri:oozie:workflow:0.1">

<start to='wordcount'/>

<action name='wordcount'>

<map-reduce>

<job-tracker>${jobTracker}</job-tracker>

<name-node>${nameNode}</name-node>

<configuration>

<property>

<name>mapred.mapper.class</name>

<value>org.myorg.WordCount.Map</value>

</property>

<property>

<name>mapred.reducer.class</name>

<value>org.myorg.WordCount.Reduce</value>

</property>

<property>

<name>mapred.input.dir</name>

<value>${inputDir}</value>

</property>

<property>

<name>mapred.output.dir</name>

<value>${outputDir}</value>

</property>

</configuration>

</map-reduce>

<ok to='end'/>

<error to='kill'/>

</action>

<kill name='kill'>

<message>Something went wrong: ${wf:errorCode('wordcount')}</message>

</kill/>

<end name='end'/>

</workflow-app>

A fork node splits one path of execution into multiple concurrent paths of execution.

A join node waits until every concurrent execution path of a previous fork node arrives to it.

The fork and join nodes must be used in pairs. The join node assumes concurrent execution paths are children of the same fork node.

<workflow-app name="sample-wf" xmlns="uri:oozie:workflow:0.1">

...

<fork name="forking">

<path start="firstparalleljob"/>

<path start="secondparalleljob"/>

</fork>

<action name="firstparallejob">

<map-reduce>

<job-tracker>foo:9001</job-tracker>

<name-node>bar:9000</name-node>

<job-xml>job1.xml</job-xml>

</map-reduce>

<ok to="joining"/>

<error to="kill"/>

</action>

<action name="secondparalleljob">

<map-reduce>

<job-tracker>foo:9001</job-tracker>

<name-node>bar:9000</name-node>

<job-xml>job2.xml</job-xml>

</map-reduce>

<ok to="joining"/>

<error to="kill"/>

</action>

<join name="joining" to="nextaction"/>

...

</workflow-app>

## HIVE (1.2)

Is data warehouse software facilitates reading, writing, and managing large datasets residing in distributed storage and queried using SQL syntax. Designed for OLAP (Online Analytic Processing).

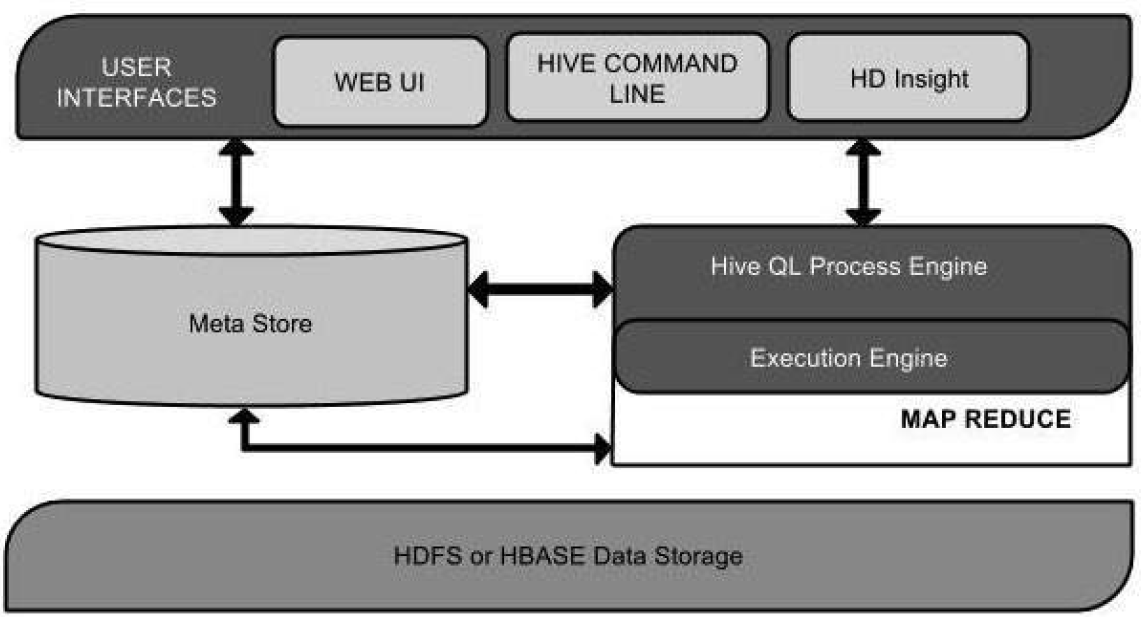
Features of Hive:

* Built on top of Apache Hadoop. Where scheme stored in DB (derby for single user, mysql for multi) and files stored either directly in Apache HDFS or in other data storage systems such as Apache HBase
* It provides SQL type language for querying called HiveQL or HQL for data warehousing tasks ETL. Query execution via Apache Tez™, Apache Spark™, or MapReduce.

Hive is not

* A relational database
* designed for OnLine Transaction Processing OLTP
* for real-time streaming and row-level updates (i.e. HIVE supports only inserts)

### Hive Architecture



Run Modes:

* Local
* Map-Reduce mode (if you have clusters for parallel processing)

Data Modes:

* Internal Table – scheme and data in metastore
* External Table – scheme in metastore data in HDFS/HBASE

Clients:

* Hive Client (JDBC, ODBC, Thrift)
* HiveServer2 – (interface for external clients to run HIVE queries, supports multi-client concurrency and authentication) HiveServer2 Client and Beeline

### Data Units

- In the order of granularity

* Databases
* Tables
* Partitions – divide table into slices based on portioning keys

Ex: table - Tab1 contains employee data such as id, name, dept, and yoj

/tab1/employeedata/file1

id, name, dept, yoj

1, gopal, TP, 2012

2, kiran, HR, 2012

3, kaleel,SC, 2013

4, Prasanth, SC, 2013

Portioning on the yoj column, splits it into below 2 files

/tab1/employeedata/2012/file2

1, gopal, TP, 2012

2, kiran, HR, 2012

/tab1/employeedata/2013/file3

3, kaleel,SC, 2013

4, Prasanth, SC, 2013

* Buckets (or Clusters) – further dividing the partitions, based on the hash value of each **column-values** of a particular column of the Table. If there are too many partitions, which would be delaying the query execution, so bucketing helps in reducing the processing time i.e. lesser units called buckets. Column values with same hash value will be in the same buckets and the rest will be ordered and distributed equally with all the buckets.

Hive> describe <table\_name>

Hive> describe extended <table\_name>

Hive> describe formatted <table\_name>

### Create Table – DDL

CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db\_name.]table\_name    -- (Note: TEMPORARY available in Hive 0.14.0 and later)

  [(col\_name data\_type [COMMENT col\_comment], ... [constraint\_specification])]

  [COMMENT table\_comment]

  [PARTITIONED BY (col\_name data\_type [COMMENT col\_comment], ...)]

  [CLUSTERED BY (col\_name, col\_name, ...) [SORTED BY (col\_name [ASC|DESC], ...)] INTO num\_buckets BUCKETS]

  [SKEWED BY (col\_name, col\_name, ...)                  -- (Note: Available in Hive 0.10.0 and later)]

     ON ((col\_value, col\_value, ...), (col\_value, col\_value, ...), ...)

     [STORED AS DIRECTORIES]

  [

   [ROW FORMAT row\_format]

   [STORED AS file\_format]

     | STORED BY 'storage.handler.class.name' [WITH SERDEPROPERTIES (...)]  -- (Note: Available in Hive 0.6.0 and later)

  ]

  [LOCATION hdfs\_path]

  [TBLPROPERTIES (property\_name=property\_value, ...)]   -- (Note: Available in Hive 0.6.0 and later)

  [AS select\_statement];   -- (Note: Available in Hive 0.5.0 and later; not supported for external tables)

CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db\_name.]table\_name

  LIKE existing\_table\_or\_view\_name

  [LOCATION hdfs\_path];

data\_type

  : primitive\_type

  | array\_type

  | map\_type

  | struct\_type

  | union\_type  -- (Note: Available in Hive 0.7.0 and later)

primitive\_type

  : TINYINT

  | SMALLINT

  | INT

  | BIGINT

  | BOOLEAN

  | FLOAT

  | DOUBLE

  | DOUBLE PRECISION -- (Note: Available in Hive 2.2.0 and later)

  | STRING

  | BINARY      -- (Note: Available in Hive 0.8.0 and later)

  | TIMESTAMP   -- (Note: Available in Hive 0.8.0 and later)

  | DECIMAL     -- (Note: Available in Hive 0.11.0 and later)

  | DECIMAL(precision, scale)  -- (Note: Available in Hive 0.13.0 and later)

  | DATE        -- (Note: Available in Hive 0.12.0 and later)

  | VARCHAR     -- (Note: Available in Hive 0.12.0 and later)

  | CHAR        -- (Note: Available in Hive 0.13.0 and later)

array\_type

  : ARRAY < data\_type >

map\_type

  : MAP < primitive\_type, data\_type >

struct\_type

  : STRUCT < col\_name : data\_type [COMMENT col\_comment], ...>

union\_type

   : UNIONTYPE < data\_type, data\_type, ... >  -- (Note: Available in Hive 0.7.0 and later)

row\_format

  : DELIMITED [FIELDS TERMINATED BY char [ESCAPED BY char]] [COLLECTION ITEMS TERMINATED BY char]

        [MAP KEYS TERMINATED BY char] [LINES TERMINATED BY char]

        [NULL DEFINED AS char]   -- (Note: Available in Hive 0.13 and later)

  | SERDE serde\_name [WITH SERDEPROPERTIES (property\_name=property\_value, property\_name=property\_value, ...)]

file\_format:

  : SEQUENCEFILE

  | TEXTFILE    -- (Default, depending on hive.default.fileformat configuration)

  | RCFILE      -- (Note: Available in Hive 0.6.0 and later)

  | ORC         -- (Note: Available in Hive 0.11.0 and later)

  | PARQUET     -- (Note: Available in Hive 0.13.0 and later)

  | AVRO        -- (Note: Available in Hive 0.14.0 and later)

  | INPUTFORMAT input\_format\_classname OUTPUTFORMAT output\_format\_classname

constraint\_specification:

  : [, PRIMARY KEY (col\_name, ...) DISABLE NOVALIDATE ]

    [, CONSTRAINT constraint\_name FOREIGN KEY (col\_name, ...) REFERENCES table\_name(col\_name, ...) DISABLE NOVALIDATE

### Row Formats & SerDe

You can create tables with a custom SerDe or using a native SerDe. A native SerDe is used if ROW FORMAT is not specified or ROW FORMAT DELIMITED is specified.

**Custom SerDe**

| **Row Format** | **Description** |
| --- | --- |
| **RegEx**  ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe' WITH SERDEPROPERTIES  ( "input.regex" = "<regex>" ) STORED AS TEXTFILE; | Stored as plain text file, translated by Regular Expression.  The following example defines a table in the default Apache Weblog format.  CREATE TABLE apachelog (    host STRING,    identity STRING,    user STRING,    time STRING,    request STRING,    status STRING,    size STRING,    referer STRING,    agent STRING)  ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'  WITH SERDEPROPERTIES (    "input.regex" = "([^]\*) ([^]\*) ([^]\*) (-|\\[^\\]\*\\]) ([^ \"]\*|\"[^\"]\*\") (-|[0-9]\*) (-|[0-9]\*)(?: ([^ \"]\*|\".\*\") ([^ \"]\*|\".\*\"))?"  )  STORED AS TEXTFILE; |
| **JSON**   ROW FORMAT SERDE  'org.apache.hive.hcatalog.data.JsonSerDe'  STORED AS TEXTFILE | Stored as plain text file in JSON format.  In some distributions, a reference to hive-hcatalog-core.jar is required.  ADD JAR /usr/lib/hive-hcatalog/lib/hive-hcatalog-core.jar;  CREATE TABLE my\_table(a string, b bigint, ...)  ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe'  STORED AS TEXTFILE; |
| **CSV/TSV**  ROW FORMAT SERDE  'org.apache.hadoop.hive.serde2.OpenCSVSerde'  STORED AS TEXTFILE | Stored as plain text file in CSV / TSV format.  The following example creates a TSV (Tab-separated) file.  CREATE TABLE my\_table(a string, b string, ...) ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'  WITH SERDEPROPERTIES (     "separatorChar" = "\t",     "quoteChar"     = "'",     "escapeChar"    = "\\"  )  STORED AS TEXTFILE;  Default properties for SerDe is Comma-Separated (CSV) file  DEFAULT\_ESCAPE\_CHARACTER \  DEFAULT\_QUOTE\_CHARACTER  "  DEFAULT\_SEPARATOR        ,  **Limitations**  This SerDe treats all columns to be of type String. Even if you create a table with non-string column types using this SerDe, the DESCRIBE TABLE output would show string column type.  The type information is retrieved from the SerDe.  To convert columns to the desired type in a table, you can create a view over the table that does the CAST to the desired type.  The CSVSerde has been built and tested against Hive 0.14 and later, and uses [Open-CSV](http://opencsv.sourceforge.net/) 2.3 which is bundled with the Hive distribution. |

### Data Store

#### Managed Table

**Creation**

Example,

CREATE TABLE page\_view(viewTime INT, userid BIGINT, page\_url STRING, referrer\_url STRING, ip STRING COMMENT 'IP Address of the User')

COMMENT 'This is the page view table'

ROW FORMAT DELIMITED

        FIELDS TERMINATED BY '|'

STORED AS TEXTFILE;

* Notice that LOCATION is not specified, LOCATION needs to be specified only if it is an EXTERNAL table.

**Note** - that a change in the schema (such as the adding of the columns), preserves the schema for the old partitions of the table in case it is a partitioned table. All the queries that access these columns and run over the old partitions implicitly return a null value or the specified default values for these columns.

**Note** – In the CREATE TABLE statement, do not use Boolean as data type for any column unless the data is really only TRUE or FALSE, even with 1/0, hive will confuse and result in noise.

**Load Data**

Creation of internal table, will only create the scheme, i.e. no records in the table, until you load the data.

Loading of data doesn’t verify the scheme (so no error on LOAD), only when READ, results are returned as NULL for unmatched data columns.

**From local**

LOAD DATA LOCAL INPATH './examples/files/kv1.txt' OVERWRITE INTO TABLE page\_view;

* LOCAL signifies that the input file is on the local file system
* OVERWRITE will move any existing data file to trash. If the 'OVERWRITE' keyword is omitted, data files are appended to existing data sets
* Data will be **copied** from the local file system to HIVE-WAREHOUSE under the database directory

**From HDFS**

LOAD DATA INPATH '/user/myname/kv2.txt' OVERWRITE INTO TABLE page\_view;

* Data will be **moved** from current HDFS path to HIVE-WAREHOUSE under the database directory

#### External Table

**Creation**

CREATE **EXTERNAL** TABLE page\_view\_stg(viewTime INT, userid BIGINT,

                page\_url STRING, referrer\_url STRING,

                ip STRING COMMENT 'IP Address of the User',

                country STRING COMMENT 'country of origination')

COMMENT 'This is the staging page view table'

ROW FORMAT DELIMITED FIELDS TERMINATED BY '44' LINES TERMINATED BY '12'

STORED AS TEXTFILE

LOCATION '/user/data/staging/page\_view';

#### Create Table AS (CTAS)

– Using select query to load the data and specify the scheme

CREATE TABLE new\_key\_value\_store

   ROW FORMAT SERDE "org.apache.hadoop.hive.serde2.columnar.ColumnarSerDe"

   STORED AS RCFile

   AS

SELECT (key % 1024) new\_key, concat(key, value) key\_value\_pair

FROM key\_value\_store

SORT BY new\_key, key\_value\_pair;

CTAS has these restrictions:

* The target table cannot be a partitioned table.
* The target table cannot be an external table.
* The target table cannot be a list bucketing table.

Advantage of CTAS, is to select data from one table to another where Hive handles the conversion of the data from the source format to the destination format as the query is being executed.

#### Create Table LIKE

* Copy an existing table definition exactly (without copying its data), using default SerDe and file formats

CREATE TABLE empty\_key\_value\_store

LIKE key\_value\_store;

#### Sorted Bucketed Tables

CREATE TABLE page\_view(viewTime INT, userid BIGINT,

     page\_url STRING, referrer\_url STRING,

     ip STRING COMMENT 'IP Address of the User')

 COMMENT 'This is the page view table'

 PARTITIONED BY(dt STRING, country STRING)

 CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS

 ROW FORMAT DELIMITED

   FIELDS TERMINATED BY '\001'

   COLLECTION ITEMS TERMINATED BY '\002'

   MAP KEYS TERMINATED BY '\003'

 STORED AS SEQUENCEFILE;

#### Skewed Tables

#### Temporary Tables

### Static Partitioning

If the values for partition columns are known at compile time (i.e. well in advance of loading the data into a Hive table).

#### Managed Table

CREATE TABLE page\_views (eventTime String, userid String, page String)

PARTITIONED BY (dt String, applicationType String)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ‘\t’

STORED AS TEXTFILE;

LOAD DATA INPATH ‘/mydata/android/Aug\_10\_2013/pageviews/’

INTO TABLE page\_views

PARTITION (dt=’2013-08-10’, applicationType = ‘andriod’);

In case of table data loaded from local or HDFS file, to append data to partitioned managed tables, overwrite with updated file.

LOAD DATA INPATH ‘/mydata/android/Aug\_10\_2013/pageviews/’

OVERWRITE INTO TABLE page\_views

PARTITION (dt=’2013-08-10’, applicationType = ‘andriod’);

Create partitioned managed table with data from another table,

CREATE TABLE patents (citing\_patent INT, cited\_patent INT, assignee STRING, companyname STRING, publication\_date STRING)

PARTITIONED BY (year INT, month INT, day INT)

INSERT OVERWRITE INTO TABLE patents PARTITION (year=’1985’, month=’Jul’, day=’05’)

SELECT citing, cited, name, company

FROM patents\_raw\_data

WHERE year(publication\_date) = ‘1985’ AND month(publication\_date) = ‘Jul’ AND day(publication\_date)=’05’;

Partitioned columns are virtual columns, so you can query them same as normal columns

SELECT dt as eventDate, page, count(\*) as pviewCount FROM page\_views WHERE applicationtype = ‘iphone’;

#### External Table

CREATE EXTERNAL TABLE page\_views (eventTime String, userid String, page String)

PARTITIONED BY (dt String, applicationType String)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ‘\t’

LOCATION ‘somewhere/on/hdfs/data/’;

For existing data in the external source, you need to alter the table to add partition on the user defined location (standard notation of naming the path is using the partition values).

ALTER TABLE page\_views

ADD PARTITION (dt=’2013-09-09’, applicationType=’Windows Phone 8’)

LOCATION ‘/somewhere/on/hdfs/data/2013-09-09/wp8’;

ALTER TABLE page\_views

ADD PARTITION (dt=’2013-09-10’, applicationType=’Windows Phone 8’)

LOCATION ‘/somewhere/on/hdfs/data/2013-09-10/wp8’;

In case of structure or partitioning of an external table is changed, or data files are updated in the external location then, Use MSCK command for HIVE to the update the same in HIVE

MSCK REPAIR TABLE page\_views\_ext;

Use ALTER TABLE for Amazon Elastic Map Reduce (EMR)

ALTER TABLE table\_name RECOVER PARTITIONS;

### Dynamic Partitioning

If the values for partition columns are known only at run time (i.e. during loading of the data into a Hive table).

Table creation semantics are the same for both static and dynamic partitioning. With static partitioning, partitions are explicitly added or dropped by the user, using “ALTER TABLE ADD/DROP PARTITION …” queries. This updates the Hive metastore with table partition information. With dynamic partitioning, since partition values are not known in advance, the user does not need to perform the explicit “alter table” step. Hive automatically takes care of updating the Hive metastore when using dynamic partitions.

#### Managed Table

CREATE TABLE patents (citing\_patent INT, cited\_patent INT, assignee STRING, companyname STRING, publication\_date STRING)

PARTITIONED BY (year INT, month INT, day INT)

INSERT OVERWRITE INTO TABLE patents PARTITION (year, month, day)

SELECT citing, cited, name, company, year(publication\_date), month(publication\_date), day(publication\_date)

FROM patents\_raw\_data;

The table, “patents\_raw\_data”, is an external table, which points to patent raw data. Notice the order of the partition columns specified in the “SELECT” clause is in exactly the same order as the partition columns specified in the “PARTITIONED BY” clause in create table query. Also, the columns year, month, and day are purposefully specified at the very end in the “SELECT” clause. Hive splits the data into multiple partitions by year, month, and day values. It also updates the Hive metastore automatically without explicit user intervention.

#### External Table

There are 2 ways, we can do dynamic partitioning on External Table

1. If the data files can be split into multiple files (directory based on partition keys)
   1. Create directories in HDFS similar to the partitions created

/somewhere/year=2016/month=jan/log.txt

/month=feb/log.txt

/month=mar/log.txt

/somewhere/year=2017/month=jan/log.txt

* 1. Create External Table with Partitioned By clause and to the above root location

CREATE EXTERNAL TABLE app\_logs (appName String, numTimes Int)

PARTITIONED BY (year Int, month String)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ","

STORED AS TEXTFILE

LOCATION "/srini\_datasets/app\_logs";

* 1. Use MSCK REPAIR TABLE <table\_name> to load partitions

HIVE> MSCK REPAIR TABLE app\_logs;

OK

Partitions not in metastore: app\_logs:year=2016/month=Feb app\_logs:year=2016/month=Jan app\_logs:year=2016/month=Mar app\_logs:year=2017/month=Jan

Repair: Added partition to metastore app\_logs:year=2016/month=Feb

Repair: Added partition to metastore app\_logs:year=2016/month=Jan

Repair: Added partition to metastore app\_logs:year=2016/month=Mar

Repair: Added partition to metastore app\_logs:year=2017/month=Jan

Now add new files and new partitions, then use MSCK REPAIR TABLE command to load them.

hadoop fs -mkdir /srini\_datasets/app\_logs/year=2017/month=Feb

hadoop fs -put feb2017\_log.txt /srini\_datasets/app\_logs/year=2017/month=Feb

hive> MSCK REPAIR TABLE app\_logs;

OK

Partitions not in metastore: app\_logs:year=2017/month=Feb

Repair: Added partition to metastore app\_logs:year=2017/month=Feb

Time taken: 0.401 seconds, Fetched: 2 row(s)

hive> show partitions app\_logs;

OK

year=2016/month=Feb

year=2016/month=Jan

year=2016/month=Mar

year=2017/month=Feb

year=2017/month=Jan

Time taken: 0.157 seconds, Fetched: 5 row(s)

1. If the data files can’t be split into multiple files (directory based on partition keys)
   1. Create an staging - External Table without partitions (you should include to-be partitioned keys as columns in this staging table)

CREATE EXTERNAL TABLE globcountrytemp(dt DATE, LandAvg DOUBLE, LandAvgU DOUBLE, Country STRING)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ','

STORED AS TEXTFILE

LOCATION '/srini\_datasets/GlobalLandTemperatures/globcountrytemp';

* 1. Create managed table with partitioned by clause

CREATE TABLE globcountrytemp\_PART(dt DATE, LandAvg DOUBLE, LandAvgU DOUBLE)

PARTITIONED BY (Country STRING);

* 1. SET dynamic partition to TRUE and dynamic partition mode to non-strict

SET hive.exec.dynamic.partition=true;

SET hive.exec.max.dynamic.partitions=2048;

SET hive.exec.max.dynamic.partitions.pernode=256;

SET hive.exec.dynamic.partition.mode=non-strict;

* 1. Load data from staging external table

INSERT INTO TABLE globcountrytemp\_PART PARTITION (Country)

SELECT dt, LandAvg, LandAvgU, Country from globcountrytemp;

Partition columns should come last and in the same order in the SELECT clause

\*\* Important draw back of the second option is, any new data can’t be loaded into the final PARTITIONED TABLE (managed table) even through MSCK REPAIR TABLE. Only the staging external table gets refreshed with the new data on MSCK REPAIR TABLE but that doesn’t reflect on the PARTITIONED table.

#### Configuration

If the table has only dynamic partition columns, then the configuration setting hive.exec.dynamic.partition.mode should be set to non-strict mode:

SET hive.exec.dynamic.partition.mode=non-strict;

If it is set to “*strict*” then atleast one partition should be static, and the rest can be dynamic.

Hive enforces a limit on the number of dynamic partitions it can create. The default is 100 dynamic partitions per node, with a total (default) limit of 1000 dynamic partitions across all nodes. However, this setting is configurable,

SET hive.exec.dynamic.partition=true;

SET hive.exec.max.dynamic.partitions=2048;

SET hive.exec.max.dynamic.partitions.pernode=256;

To increase the max number of files a data node can service in (hdfs-site.xml)

SET dfs.datanode.max.xcievers=4096;

Modifying the number of partition columns:

1. Create a new table with all required partition columns
2. Loading data into the new table from the already existing partitioned table, dynamic partitioning should be used here.
3. Deleting the existing table.

Dynamic partition has a disadvantage, considering each partition will go to each reducer (reducers run as induvial JVM threads), then so many threads has to there in a node, which will have effect on the resources. If a particular partition has more data compared to other, then that particular reducer runs for more time than the one with lesser data, so we can go for bucketing.

### Bucketing

Distribute By

This will define which mapper output should go to which reducer.

SELECT x, y, z FROM t1 DISTRIBUTE BY y

Cluster By

Distribute By combined with Sort By is Cluster By, means mapper output key-values will be sorted before passing on to reducer

SELECT x, y, z FROM t1 CLUSTER BY y

**Bucketing** is an approach to distribute or cluster table data.

* More efficient sampling
* Better performance with map side joins
* Used with partitioning or w/o when partitioning doesn’t work for your data set
* Buckets can also be sorted (Sort-Merge-Bucket joins. SMB)

CREATE TABLE t1 (a INT, b STRING, c STRING)

CLUSTERED BY (b) INTO 256 BUCKETS

There are 2 approaches to load data to the bucketed tables

CREATE TABLE t1 (a INT, b STRING, c STRING)

PARTITIONED BY (dt STRING)

CLUSTERED BY (b) SORTED BY (c) INTO 64 BUCKETS

1. Setting the number of reducers to be same as number of buckets (given in CREATE statement)

Set.mapred.reduce.tasks = 64;

1. Enforcing the HIVE to use the number of reducers based on the number of buckets (given in CREATE statement) and also automatically find the clustered by column

Set hive.enforce.bucketing = true;

INSERT OVERWRITE TABLE t1

SELECT a,b,c FROM t2 CLUSTERED BY b;

### Functions

#### Built-in Table Functions

Mathematical

* abs(double a)
* round(double a, intd)
* floor(double a)

Collection

* size(Map<K.V>)
* map\_keys(Map<K.V>)
* map\_values(Map<K.V>)
* SELECT array\_contains(a, ‘test’) FROM t1;

Date

* unix\_timestamp()
* year(string d), month(string d), day(string d), hour, second
* datediff(string enddate, string startdate)
* date\_add(string startdate, intdays)
* date\_sub(string startdate, intdays)
* to\_date(string timestamp)

Conditional

* SELECT IF(a = b, ‘true result’, ‘false result’) FROM t1;
* SELECT COALESCE(a, b, c) FROM t1;
* SELECT CASE a WHEN 123 THEN ‘first’ WHEN 456 THEN ‘second’ ELSE ‘none’ END FROM t1;
* SELECT CASE WHEN a = 13 THEN c ELSE d END FROM t1;

String

* SELECTconcat(a, b) FROM t1;
* SELECT concat\_ws(sep, a, b) FROM t1;
* SELECT regex\_replace(“Hive Rocks”, “ive”, “adoop”) FROM dummy;
* substr(string|binaryA, intstart)
* substring(string|binaryA, intstart, intlength)
* SELECT sentences(“Loving this course! Hive is awesome.”) FROM dummy;
  + ((“Loving”, “this”, “course”), (“Hive”, “is”, “awesome”))

Explode – takes array as input and returns each item as an individual row. With LATERAL VIEW, functions can be included in the SELECT expression with other columns.

#### Built-in Aggregate Functions

* COUNT(\*), COUNT(expr), COUNT(DISTINCT expr)
* SUM(col), SUM(DISTINCT col)
* AVG, MIN, MAX, VARIANCE, STDDEV\_POP
* HISTOGRAM\_NUMERIC(col, b)
* returns array<struct{‘x’, ‘y’}>

#### User defined Functions (UDF)

UDAF (User defined aggregate functions)

UDTF (User defined table functions)

**Creation**

Import necessary packages

import org.apache.hadoop.hive.ql.exec.UDF;

import org.apache.hadoop.hive.ql.exec.Description;

add hive-exec-1.2.1.jar (same version as the HIVE you got)

Anything you need as part of your UDF

import org.apache.hadoop.io.Text

import java.util.\*;

add Hadoop-common-2.7.3.jar (same version as the HADOOP you got)

Add annotations

Description, Deterministic, Stateful, DistinctLike

Extend the UDF class

Provide an implementation of the **evaluate** function possibly with multiple overloads

package com.srny.hive;

import org.apache.hadoop.hive.ql.exec.Description;

import org.apache.hadoop.hive.ql.exec.UDF;

import org.apache.hadoop.io.Text;

@Description(name = "reverse",

value="\_FUNC\_(string) - reverses the input string",

extended = "Example:\n"

+ " SELECT \_FUNC\_(input\_string) FROM src;\n"

)

public final class srny1udf extends UDF{

@SuppressWarnings("unused")

public Text evaluate(final Text inputStr) {

if (inputStr == null) {

return null;

}

String revStr = new StringBuilder(inputStr.toString()).reverse().toString();

return new Text(revStr);

}

}

Compile and package code (on the same machine as hive to maintain same java version)

javac -cp $(ls /usr/lib/hive/lib/hive-exec\*.jar):/usr/lib/hadoop/hadoop-core.jar com/pluralsight/udf/MyReverse.java

jar cvf myudf.jar com/pluralsight/udf/MyReverse.class

Adds JAR to distributed cache & classpath

ADD JAR /path/to/jar/myudf.jar

Create TEMPORARY FUNCTION and reference class

Create temporary function MyReverse as ‘com.pluralsight.udf.MyReverse’;

Use the function in the query

Select MyReverse(appname) from app\_logs;

### Integration with HBase

Hive QL statements to access HBase tables for both read (SELECT) and write (INSERT).

To combine access to HBase tables with native Hive tables via joins and unions.

For integrating HBase with Hive, Storage Handlers (**HBaseStorageHandler**) in Hive is used. Storage Handlers are a combination of InputFormat, OutputFormat, SerDe, and specific code that Hive uses to identify an external table (in which case Hive will not create to drop that table directly).

*CREATE TABLE foo(rowkey STRING, a STRING, b STRING)*

*STORED BY ‘org.apache.hadoop.hive.hbase.HBaseStorageHandler’*

*WITH SERDEPROPERTIES (‘hbase.columns.mapping’ = ‘:key,f:c1,f:c2’)*

*TBLPROPERTIES (‘hbase.table.name’ = ‘bar’);*

The values provided in the mapping property correspond one-for-one with column names of the hive table. HBase column names are fully qualified by column family, and you use the special token :key to represent the rowkey. The above example makes rows from the HBase table bar available via the Hive table foo. The foo column rowkey maps to the HBase’s table’s rowkey, a to c1 in the f column family, and b to c2, also in the f family.

You can also associate Hive’s MAP data structures to HBase column families. In this case, only the STRING Hive type is used. The other Hive type currently supported is BINARY.

### Misc

#### Joins

JOIN (Inner Join)

SELECT a.val, b.val FROM a JOIN b ON (a.key = b.key);

LEFT, RIGHT, FULL [OUTER] JOIN

LEFT SEMI JOIN

CROSS JOIN

##### Map-side joins

More efficient joins

Buckets can be joined with each other when:

* Tables being joined are bucketed on join columns (Clustered)
* Number of buckets in one table is a multiple of the number of buckets in the other table
* Set hive.optimize.bucketmapjoin=true

Sort Merge Join

* Tables being joined are bucketed on join columns (Clustered)
* They have the same number of buckets
* Buckets are also sorted
* Set:
  + hive.input.format=org.apache.hadoop.hive.ql.io.BucketizedHiveInputFormat;
  + hive.optimize.bucketmapjoin = true;
  + hive.optimize.bucketmapjoin.sortedmerge = true;

#### Distributed Cache

ADD FILE mydata.txt;

ADD ARCHIVE sendme.zip;

ADD JAR myprogram.jar;

LIST FILES|JARS|ARCHIVES [filepath];

#### Lateral View

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **movie\_id** | **title** | **actors** | | | | | |
| **620** | **The King’s Speech** | **Colin Firth** | **Geoffrey Rush** | | **Helena Bonham Carter** | | **Freya Wilson** |
| **621** | **Elysium** | **Matt Damon** | | **Jodie Foster** | | **Sharlto Copley** | |

SELECT movie\_id, title, actor

FROM movies LATERAL VIEW explode(actors) actorTable AS actor;

|  |  |  |
| --- | --- | --- |
| **movie\_id** | **title** | **actor** |
| **620** | **The King’s Speech** | **Colin Firth** |
| **620** | **The King’s Speech** | **Geoffrey Rush** |
| **620** | **The King’s Speech** | **Helena Bonham Carter** |
| **620** | **The King’s Speech** | **Freya Wilson** |
| **621** | **Elysium** | **Matt Damon** |
| **621** | **Elysium** | **Jodie Foster** |
| **621** | **Elysium** | **Sharlto Copley** |

#### Batch Mode

CLI args –e and –f runs the queries in batch mode.

hive -e ‘select a, b, from t1 where c = 15’

hive -f /my/local/file/system/get-data.sql

args –S will export the console output to a file

hive -S -e ‘select a, b from t1’ > results.txt

#### Variable Substitution

4 namespaces

* hivevar -d, --define , --hivevar
  + - * set hivevar:name=value
* hiveconf --hiveconf
  + - * set hiveconf:property=value
* system set system:property=value
* env set env:property=value

$ hive -d srctable=movies

hive> set hivevar:cond=123;

hive> select a,b,cfrom pluralsight.${hivevar:srctable}where a = ${hivevar:cond};

To run periodic batch queries with variable substitution

$ hive -v -d src=movies -d db=pluralsight-e 'select \* from ${hivevar:db}.${hivevar:src} LIMIT 100;‘

#### Bucket Sampling

Sampling on the bucketed columns are more efficient, as they are sorted and can be easily clustered.

SELECT \* FROM page\_views TABLESAMPLE (BUCKET 3 OUT OF 64 ON userid);

SELECT \* FROM page\_views TABLESAMPLE (BUCKET 3 OUT OF 64 ON rand());

#### Block Sampling

Based on HDFS blocks (64/128/256 etc..)

Percentage of data size (notice this is not # of rows)

Returns at least the percentage specified

Doesn’t always work - Depends on compression and input format (CombineHiveInputFormat)

SELECT \* FROM page\_views TABLESAMPLE (0.1 PERCENT);

SELECT \* FROM page\_views TABLESAMPLE (90M);

#### Custom Map-Reduce Scripts

|  |
| --- |
| FROM (       FROM pv\_users       MAP pv\_users.userid, pv\_users.date       USING 'map\_script'       AS dt, uid       CLUSTER BY dt) map\_output     INSERT OVERWRITE TABLE pv\_users\_reduced       REDUCE map\_output.dt, map\_output.uid       USING 'reduce\_script'       AS date, count; |

Sample map script (weekday\_mapper.py )

|  |
| --- |
| import sys  import datetime    for line in sys.stdin:    line = line.strip()    userid, unixtime = line.split('\t')    weekday = datetime.datetime.fromtimestamp(float(unixtime)).isoweekday()    print ','.join([userid, str(weekday)]) |

**Hive HCatalog Streaming API – (**not real-time streaming, as the latency is high due to batch size**)**

Traditionally adding new data into Hive requires gathering a large amount of data onto HDFS and then periodically adding a new partition. This is essentially a “batch insertion”. Insertion of new data into an existing partition is not permitted. Hive Streaming API allows data to be pumped continuously into Hive. The incoming data can be continuously committed in small batches of records into an existing Hive partition or table. Once data is committed it becomes immediately visible to all Hive queries initiated subsequently.

**What’s new on Hive 2? -** Hive 2.x doesn’t support Hadoop 1.x

## PIG (v0.16)

Pig is an abstraction over MapReduce. It is a tool/platform which is used to analyze larger sets of data representing them as data flows. Pig is generally used with Hadoop.

UDF’s − Pig provides the facility to create User-defined Functions in other programming languages such as Java and invoke or embed them in Pig Scripts.

Handles all kinds of data − Apache Pig analyzes all kinds of data, both structured as well as unstructured. It stores the results in HDFS.

**Apache Pig Vs MapReduce**

Listed below are the major differences between Apache Pig and MapReduce.

|  |  |
| --- | --- |
| **Apache Pig** | **MapReduce** |
| Apache Pig is a data flow language. | MapReduce is a data processing paradigm. |
| It is a high level language. | MapReduce is low level and rigid. |
| Performing a Join operation in Apache Pig is pretty simple. | It is quite difficult in MapReduce to perform a Join operation between datasets. |
| Any novice programmer with a basic knowledge of SQL can work conveniently with Apache Pig. | Exposure to Java is must to work with MapReduce. |
| Apache Pig uses multi-query approach, thereby reducing the length of the codes to a great extent. | MapReduce will require almost 20 times more the number of lines to perform the same task. |
| There is no need for compilation. On execution, every Apache Pig operator is converted internally into a MapReduce job. | MapReduce jobs have a long compilation process. |

**Apache Pig Vs SQL**

Listed below are the major differences between Apache Pig and SQL.

|  |  |
| --- | --- |
| **Pig** | **SQL** |
| Pig Latin is a procedural language. | SQL is a declarative language. |
| In Apache Pig, schema is optional. We can store data without designing a schema (values are stored as $01, $02 etc.) | Schema is mandatory in SQL. |
| The data model in Apache Pig is nested relational. | The data model used in SQL is flat relational. |
| Apache Pig provides limited opportunity for Query optimization. | There is more opportunity for query optimization in SQL. |

In addition to above differences, Apache Pig Latin −

* Allows splits in the pipeline.
* Allows developers to store data anywhere in the pipeline.
* Declares execution plans.
* Provides operators to perform ETL (Extract, Transform, and Load) functions.

**Apache Pig Vs Hive**

Both Apache Pig and Hive are used to create MapReduce jobs. And in some cases, Hive operates on HDFS in a similar way Apache Pig does. In the following table, we have listed a few significant points that set Apache Pig apart from Hive.

|  |  |
| --- | --- |
| **Apache Pig** | **Hive** |
| Apache Pig uses a language called Pig Latin. It was originally created at Yahoo. | Hive uses a language called HiveQL. It was originally created at Facebook. |
| Pig Latin is a data flow language. | HiveQL is a query processing language. |
| Pig Latin is a procedural language and it fits in pipeline paradigm. | HiveQL is a declarative language. |
| Apache Pig can handle structured, unstructured, and semi-structured data. | Hive is mostly for structured data. |

## TEZ (v0.8)

A generalized data-flow programming framework, built on Hadoop YARN, which provides a powerful and flexible engine to execute an arbitrary DAG of tasks to process data for both batch and interactive use-cases. Tez is being adopted by Hive™, Pig™ and other frameworks in the Hadoop ecosystem, and also by other commercial software (e.g. ETL tools), to replace Hadoop™ MapReduce as the underlying execution engine.

## Impala (v2.8)

Impala raises the bar for SQL query performance on Apache Hadoop. Impala uses the same metadata, SQL syntax (Hive SQL), ODBC driver, and user interface (Hue Beeswax) as Apache Hive, providing a familiar and unified platform for batch-oriented or real-time queries. (For that reason, Hive users can utilize Impala with little setup overhead.)

**Architecture**

To avoid latency, Impala circumvents/eliminates MapReduce to directly access the data through a specialized distributed query engine that is very similar to those found in commercial parallel RDBMSs. The result is order-of-magnitude faster performance than Hive, depending on the type of query and configuration.

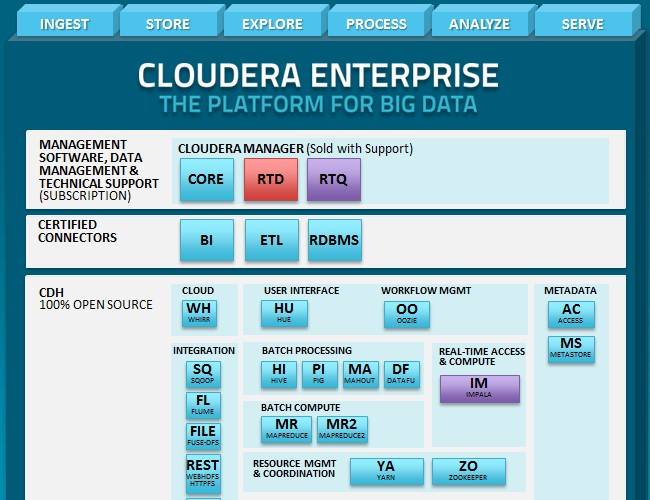
Advantages:

* Thanks to local processing on data nodes, network bottlenecks are avoided.
* A single, open, and unified metadata store can be utilized.
* Costly data format conversion is unnecessary and thus no overhead is incurred.
* All data is immediately query-able, with no delays for ETL.
* All hardware is utilized for Impala queries as well as for MapReduce.
* Only a single machine pool is needed to scale.

# Hadoop Distributions

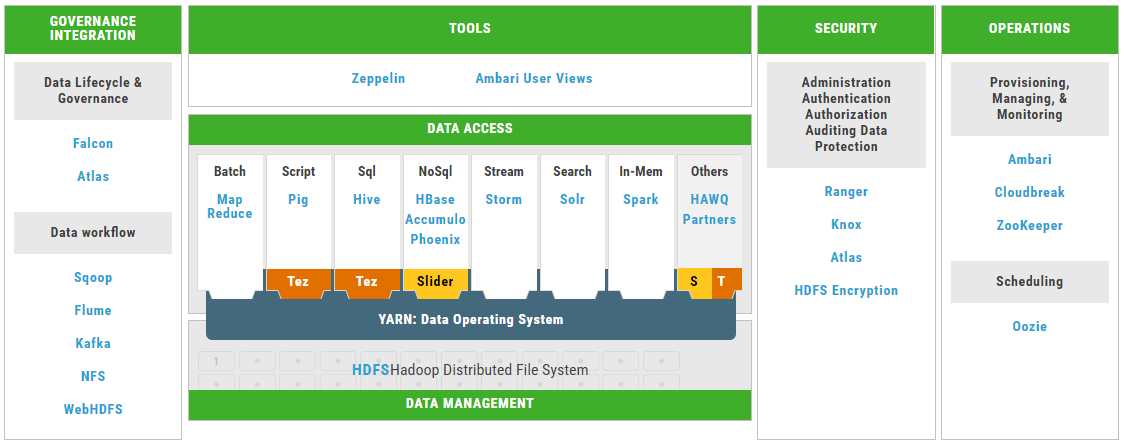
## Cloudera (5.5/.7/.8)

Proprietary features namely, Cloudera Manager, Impala, Cloudera Search



## Hortonworks (2.3/.4/.5)

100% open source.



## MapR

MapR replaces HDFS component and instead uses its own proprietary file system, called MapRFS.

Proprietary features namely, JobTracker HA, NameNode HA, NFS-HA, Mirroring, Snapshot

# Spark (1.5, 1.6, 2.0, 2.1)

RDD’s use either JAVA serialization or Kyro

A fast and general compute engine for Hadoop data. Spark provides a simple and expressive programming model that supports a wide range of applications, including ETL, machine learning, stream processing, and graph computation.

Broadcast Variables – read-only, used to share across all the worker nodes from driver node

Accumulators – counter/addition, limitation – addition should be associative

**Transformations**: map(), flatMap(), filter(), mapPartitions(), mapPartitionsWithIndex(), sample(), union(), distinct(), groupByKey(), reduceByKey(), sortByKey(), join(), cogroup(), pipe(), coalesce(), repartition(), partitionBy(), …

**Actions**: reduce(), collect(), count(), first(), take(), takeSample(), takeOrdered(), saveAsTextFile(), saveAsSequenceFile(), saveAsObjectFile(), countByKey(), foreach(), …

Prefer reducebykey over groupbykey. Refer - <https://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/prefer_reducebykey_over_groupbykey.html>

## Facts & Alternates

* Spark 2.0 runs on Java 7+, Python 2.6+/3.4+, R 3.1+ and Scala 2.11.
  + Note that support for Java 7 and Python 2.6 are deprecated as of Spark 2.0.0, and support for Scala 2.10 and versions of Hadoop before 2.6 are deprecated as of Spark 2.1.0, and may be removed in Spark 2.2.0.
* Spark 2.0 has got 2nd generation tungsten engine (using whole-stage code generation, a technique that blends state-of-the-art from modern compilers and MPP-massively parallel processing databases), which has increased processing speed, compared to Spark 1.6. And also more streamlined APIs like SprakSession has replaced HiveContext and SQLContext.
* <spark-home>/conf/spark-env.sh
  + # - SPARK\_EXECUTOR\_INSTANCES, Number of executors to start (Default: 2)
  + # - SPARK\_EXECUTOR\_CORES, Number of cores for the executors (Default: 1).
  + # - SPARK\_EXECUTOR\_MEMORY, Memory per Executor (e.g. 1000M, 2G) (Default: 1G)
  + # - SPARK\_DRIVER\_MEMORY, Memory for Driver (e.g. 1000M, 2G) (Default: 1G)

## Spark SQL

In Spark 2.0, DataFrames are just Dataset of Rows in Scala and Java API. These operations are also referred as “untyped transformations” in contrast to “typed transformations” come with strongly typed Scala/Java Datasets.

When reading from and writing to Hive metastore Parquet tables, Spark SQL will try to use its own Parquet support instead of Hive SerDe for better performance. This behavior is controlled by the spark.sql.hive.convertMetastoreParquet configuration, and is turned on by default.

\*\*\* Hive buckets are still not supported in Spark SQL

Upgrading from 1.6 to 2.0

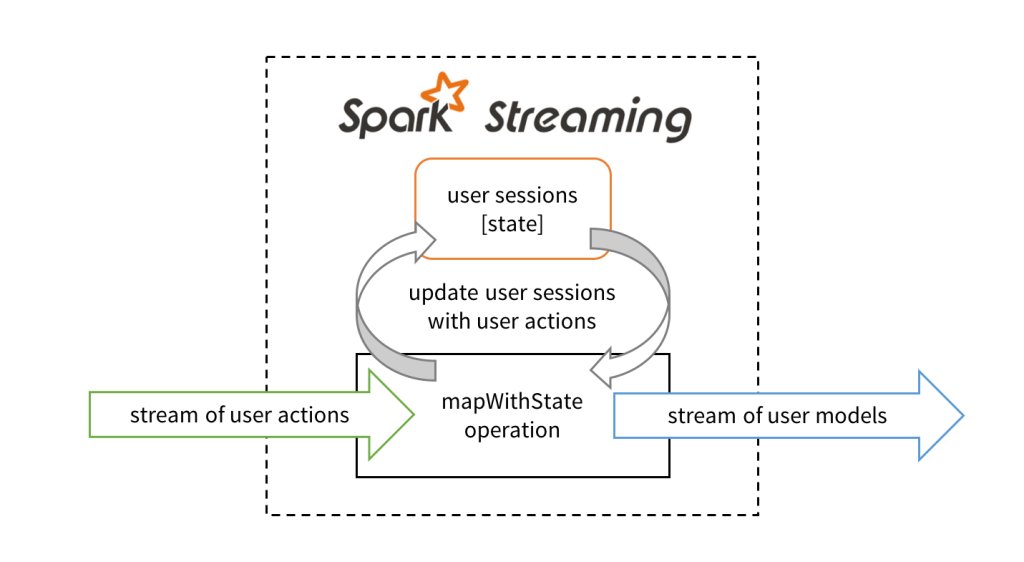
* SparkSession is now the new entry point of Spark that replaces the old SQLContext and HiveContext. Note that the old SQLContext and HiveContext are kept for backward compatibility. A new catalog interface is accessible from SparkSession - existing API on databases and tables access such as listTables, createExternalTable, dropTempView, cacheTable are moved here.
* Dataset API and DataFrame API are unified. In Scala, DataFrame becomes a type alias for Dataset[Row], while Java API users must replace DataFrame with Dataset<Row>. Both the typed transformations (e.g., map, filter, and groupByKey) and untyped transformations (e.g., select and groupBy) are available on the Dataset class. Since compile-time type-safety in Python and R is not a language feature, the concept of Dataset does not apply to these languages’ APIs. Instead, DataFrame remains the primary programing abstraction, which is analogous to the single-node data frame notion in these languages.
* Dataset and DataFrame API unionAll has been deprecated and replaced by union
* Dataset and DataFrame API explode has been deprecated, alternatively, use functions.explode() with select or flatMap
* Dataset and DataFrame API registerTempTable has been deprecated and replaced by createOrReplaceTempView

## Spark Streaming

### State Maintenance/Stateful Streaming

Earlier to 1.6, updateStateByKey API is used to maintain the state

From 1.6, mapWithState API is introduced, which has better optimization (i.e. 10x faster than updateStateByKey).



def stateUpdateFunction(

userId: UserId,

newData: UserAction,

stateData: State[UserSession]): UserModel = {

val currentSession = stateData.get() // Get current session data

val updatedSession = … // Compute updated session using newData

stateData.update(updatedSession) // Update session data

val userModel = … // Compute model using updatedSession

return userModel // Send model downstream

}

Then, we define the mapWithState operation on a DStream of user actions. This is done by creating a StateSpec object which contains all the specification of the operation.

// Stream of user actions, keyed by the user ID

val userActions = … // stream of key-value tuples of (UserId, UserAction)

// Stream of data to commit

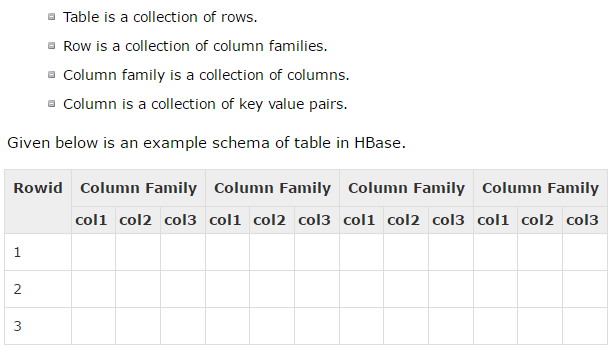
val userModels = userActions.mapWithState(StateSpec.function(stateUpdateFunction))

# No SQL Databases

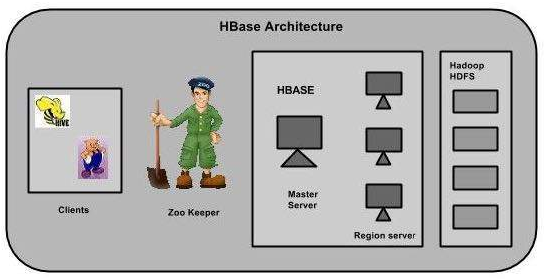
## HBase (1.1/.2)

HBase is a distributed column-oriented database built on top of the Hadoop file system

|  |  |
| --- | --- |
| **HDFS** | **HBase** |
| HDFS is a distributed file system suitable for storing large files. | HBase is a database built on top of the HDFS. |
| HDFS does not support fast individual record lookups. | HBase provides fast lookups for larger tables. |
| It provides high latency batch processing; no concept of batch processing. | It provides low latency access to single rows from billions of records (Random access). |
| It provides only sequential access of data. | HBase internally uses Hash tables and provides random access, and it stores the data in indexed HDFS files for faster lookups. |



Architecture:



Regions are nothing but tables that are split up and spread across the region servers.

Region server

* Communicate with the client and handle data-related operations.
* Handle read and write requests for all the regions under it.
* Decide the size of the region by following the region size thresholds.

When we take a deeper look into the region server, it contain regions and stores as shown below:



## Cassandra

|  |  |
| --- | --- |
| Specific characteristics | Apache Cassandra is the leading NoSQL, distributed database management system driving many of today's modern business applications by offering continuous availability, high scalability and performance, strong security, and operational simplicity while lowering overall cost of ownership. |
| Competitive advantages | No single point of failure ensures **100% availability**. Operational simplicity for **lowest total cost of ownership**. **Best-in-class scalability** of NoSQL platforms. |
| Typical application scenarios | Internet of Things (IOT), fraud detection applications, recommendation engines, product catalogs and playlists and messaging applications. |

## MongoDB

MongoDB maintains the most valuable features of relational databases: strong consistency, expressive query language and secondary indexes. As a result, developers can build highly functional applications faster than NoSQL databases.

MongoDB provides the data model flexibility, elastic scalability, and high performance and availability of NoSQL databases. As a result, engineers can continuously enhance applications, and deliver them at almost unlimited scale on commodity hardware.

Typical application scenarios - Internet of Things, Mobile, Single View, Real Time Analytics, Personalization, Catalogs, Content Management

## Cassandra vs HBase vs MongoDB

|  |  |  |  |
| --- | --- | --- | --- |
| Name | [Cassandra  X](http://db-engines.com/en/system/HBase%3BMongoDB) | [HBase  X](http://db-engines.com/en/system/Cassandra%3BMongoDB) | [MongoDB  X](http://db-engines.com/en/system/Cassandra%3BHBase) |
| Description | |  | | --- | | Wide-column store based on ideas of BigTable and DynamoDB | | Wide-column store based on Apache Hadoop and on concepts of BigTable | One of the most popular document stores |
| Database model | [Wide column store](http://db-engines.com/en/article/Wide+Column+Stores) | [Wide column store](http://db-engines.com/en/article/Wide+Column+Stores) | [Document store](http://db-engines.com/en/article/Document+Stores) |
| |  | | --- | | Data scheme | | |  | | --- | | schema-free | | |  | | --- | | schema-free | | |  | | --- | | schema-free | |
| |  | | --- | | XML support | |  | no |  |
| Secondary indexes | |  | | --- | | restricted | | no | yes |
| |  | | --- | | SQL | | no | no | no |
| APIs and other access methods | |  | | --- | | Proprietary protocol | | | Java API | proprietary protocol using JSON |
| RESTful HTTP API |
| Thrift |
| |  | | --- | | Partitioning methods | | |  | | --- | | Sharding | | |  | | --- | | Sharding | | Sharding |
| |  | | --- | | Replication methods | | |  | | --- | | selectable replication factor | | selectable replication factor | Master-slave replication |
| |  | | --- | | MapReduce | | yes | yes | yes |
| |  | | --- | | Consistency concepts | | | Eventual Consistency | Immediate Consistency | Eventual Consistency |
| Immediate Consistency | Immediate Consistency |
| |  | | --- | |  | | |  | | --- | |  | |
| |  | | --- | | Concurrency | | |  | | --- | | yes | | yes | |  | | --- | | yes | |
| |  | | --- | | Durability | | yes | yes | |  | | --- | | yes | |
| |  | | --- | | In-memory capabilities | |  | no | |  | | --- | | yes | |
| |  | | --- | | User concepts | | Access rights for users can be defined per object | |  | | --- | | Access Control Lists (ACL) | | Access rights for users and roles |

# Data Formats

Text: CSV, JSON record, not good for querying the data Also do not support the block compression, in most case they are also not splittable.

Sequence file: Row Based, used to transfer data between map/reduce phases. They are splittable which means they are good for map/reduce

Avro:

* Row based.
* Mainly used for serialization, fast binary format.
* Supports block compression and splittable.
* Most important they support schema evolution. Means frequent adding columns, renaming columns and removing columns.

Parquet:

* Column oriented when specific columns needs to be retrieved they are excellent.
* Each data file contains the value for a set of rows

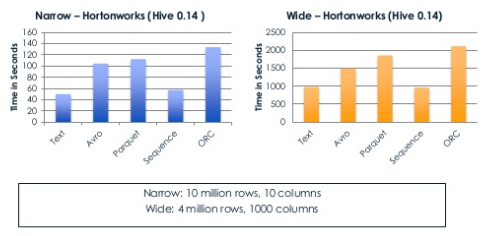
ORC (Optimized Row Columnar):

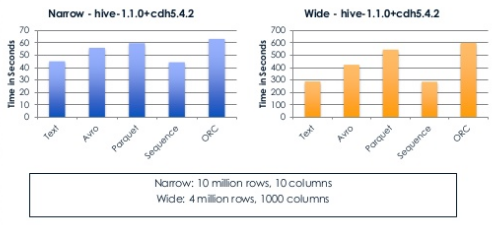
* Mixed of row and column format, that means stores collections of rows and within the collections the rows the data is stored in columnar format.
* Splittable that means parallel operations can be performed easily
* It comes with basic statistics on columns (min, max, sum and count)
* Introduces a lightweight indexing that enables skipping of irrelevant blocks of rows.

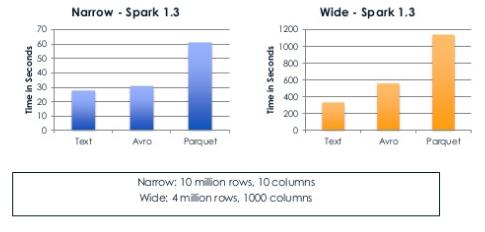
## Choosing the right format

### For WRITE

1. What type of data?
2. Is data format is compatible with the processing and querying tools?
3. File sizes?
4. Do you have schema that change over time (clickstream/Event)?
5. Frequency of write and the size of the files, let’s say if you dump each clickstream event then file size will be very small and you need to merge for better performances.
6. Speed concern while writing how fast you want to write your data?
   1. Parquet and ORC usually needs some additional parsing to format the data which increases the overall read time







\*\* More the columns, more time is required to WRITE

Avro – Event data that can change over time. In the large datasets, avro\_snappy writes in lesser time than avro\_deflate

Sequence file – Datasets shared between MR jobs

Text – Adding large amounts of data to HDFS quickly

### For READ

Types of queries, if queries needs to retrieve few or group of columns user either Parquet or ORC they are very good for read. Parquet and ORC optimize read performance at the expenses of write performance.

Compression of the file regardless the format increases query speed times. Snappy and LZO are commonly used compression technologies that enable efficient block storage and processing, so check which the combination of support let’s say parquet with Snappy compression work best in Spark.

## Conclusion

Each of the data formats has its own strengths, weaknesses, and trade-offs, so the decision on which format to use should be based on your specific use cases and systems.

If your main focus is to be able to write data as fast as possible and you have no concerns about space, then it might be acceptable to just store your data in text format with the understanding that query times for large data sets will be longer.

If your main concern is being able to handle evolving data in your system, then you can rely on Avro to save schemas. Keep in mind, though, that when writing files to the system Avro requires a pre-populated schema, which might involve some additional processing at the beginning.

Finally, if your main use case is analysis of the data and you would like to optimize the performance of the queries, then you might want to take a look at a columnar format such as Parquet or ORC because they offer the best performance in queries, particularly for partial searches where you are only reading specific columns. However, the speed advantage might decrease if you are reading all the columns.

There is a pattern in the mentioned uses cases: if a file takes longer to write, it is because it has been optimized to increase speed during reads.

Avro for – query datasets that have changed over time

Parquet – query a few columns on a wide table

For complex tables with common strings, Avro with Snappy is good fit.

References:

<http://www.slideshare.net/StampedeCon/choosing-an-hdfs-data-storage-format-avro-vs-parquet-and-more-stampedecon-2015>

<https://www.svds.com/how-to-choose-a-data-format/>

<http://www.svds.com/dataformats/>

# Distributed Messaging System

Distributed messaging is based on the concept of reliable message queuing. Messages are queued asynchronously between client applications and messaging systems. A distributed messaging system provides the benefits of reliability, scalability, and persistence.

## Model / Patterns

### Message Queue Model

Examples: RabbitMQ, ActiveMQ, ZeroMQ, JMS

### Pub-Sub Model

Where the senders of the messages are called publishers and those who want to receive the messages are called subscribers (note: It is a very loosely coupled architecture; even the senders don’t know who their subscribers are). Once the message has been published by the sender, the subscribers can receive the selected message with the help of a filtering option. There are two types of filtering, one is topic-based filtering and another one is content-based filtering.

Examples: Kafka, Kestrel

### Log Aggregation System

Only when something happens, consumers read messages from these systems.

Examples: Flume, Scribe

## Real Time Processing

2 kinds of (near) real time processing

* Event-at-a-time or Event stream processing (ESP).
* Micro-batched event processing.

**Event-at-a-time Stream processing** - is a stateless, straight through processing of time-series data in motion or in flight in a distributed fashion, typically without reading external data sources for data enrichment. Typical end state is persistence for later deeper processing and/or real-time actions. Usually data sources are expected to be reliable and/or durable queue/topic - (and I love how already most of you are picturing Apache Kafka in your head). My usage on the word "typically" suggests you can always implement a variant.

Classical use case - Stock market tickers showing stock performances with a Green up arrow or Red down arrow in real time.

**Micro-batched streaming**- is a stateful, micro/mini batched, complex processing, where state maintenance would call out for some form of persistence not necessarily disk, could be memory. End state could be anything - ranging from feeding into another system for an alert or simply persistence or could feed into another system for deeper processing.

Classical use case - End user setting up an alert to the stock market saying "let me know if GOOG stocks went up by 10% and stayed up for 3 hours or more". Might look like a rule engine use case? Exactly, only you don't use rule engines anymore you use distributed CEPs.

Twitter trending topics rolling count by hashtags.

### Facts & Alternates

* Apache Spark is an in-memory distributed data analysis platform-- primarily targeted at speeding up batch analysis job
* Hadoop has HDFS, Hadoop/MapReduce, HBase et al, and all done on large datasets in a batch mode with ~10-15 minutes granularity

# RabbitMQ vs Kafka

## Differences

|  |  |
| --- | --- |
| **RabbitMQ** | **Kafka** |
| RabbitMQ is **broker-centric**, focused around delivery guarantees between producers and consumers, with transient preferred over durable messages | Kafka is **producer-centric**, based around partitioning a fire hose of event data into durable message brokers with cursors, supporting batch consumers that may be offline, or online consumers that want messages at low latency |
| RabbitMQ uses message acknowledgments to ensure delivery state on the broker itself. It uses Erlang's Mnesia to maintain delivery state around the broker cluster. | Kafka doesn't have message acknowledgements, it assumes the consumer tracks of what's been consumed so far. Both Kafka brokers/producers & consumers use Zookeeper to reliably maintain their state across a cluster. |
| RabbitMQ presumes that consumers are mostly online, and any messages "in wait" (persistent or not) are held opaquely (i.e. no cursor). RabbitMQ pre-2.0 (2010) would fall over if your consumers were too slow, but now it's robust for online and batch consumers - but clearly large amounts of persistent messages sitting in the broker was not the main design case for AMQP in general. | Kafka was based from the beginning around both online and batch consumers, and also has producer message batching - it's designed for holding and distributing large volumes of messages. |
| RabbitMQ provides rich routing capabilities with AMQP 0.9.1's exchange, binding and queuing model. | Kafka has a very simple routing approach - in AMQP parlance it uses topic exchanges only. |
| RabbitMQ's philosophy is to make the cluster transparent, as if it were a virtual broker. Messages across several nodes are almost always unordered delivery (the AMQP 0.9.1 model says "one producer channel, one exchange, one queue, one consumer channel" is required for in-order delivery). | Kafka makes it explicit, by forcing the producer to know it is partitioning a topic's messages across several nodes, this has the benefit of preserving ordered delivery within a partition. |
| RabbitMQ's claim to that attribute is around **20k+ events per second**. | Kafka can handle **100k+ events per second**. |

## Conclusion

Use Kafka if you have a fire hose of events (100k+/sec) you need delivered in partitioned order 'at least once' with a mix of online and batch consumers, you want to be able to re-read messages, you can deal with current limitations around node-level HA (or can use trunk code), and/or you don't mind supporting incubator-level software yourself via forums/IRC.

Use Rabbit if you have messages (20k+/sec) that need to be routed in complex ways to consumers, you want per-message delivery guarantees, you don't care about ordered delivery, you need HA at the cluster-node level now, and/or you need 24x7 paid support in addition to forums/IRC.

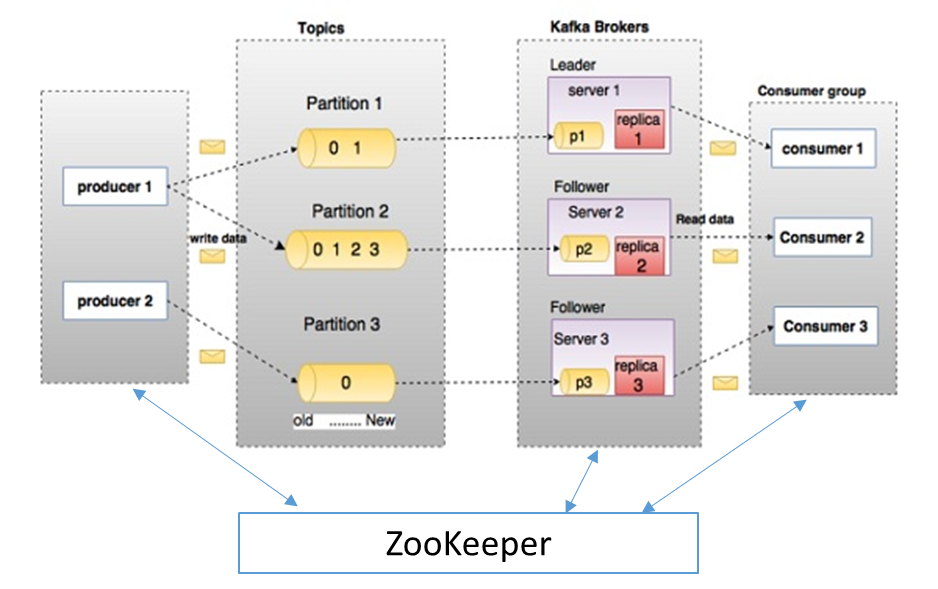
Neither offers great "filter/query" capabilities - if you need that, consider using Storm on top of one of these solutions to add computation, filtering, querying, on your streams. Or use something like Cassandra as your queryable cache.

Performance-wise, if you require ordered durable message delivery, currently it looks like there's no comparison: Kafka currently blows away RabbitMQ in terms of performance on synthetic benchmarks. This paper indicates 500,000 messages published per second and 22,000 messages consumed per second on a 2-node cluster with 6-disk RAID 10 (<http://research.microsoft.com/en-us/um/people/srikanth/netdb11/netdb11papers/netdb11-final12.pdf>)

# Kafka (0.8/.9/.10)

Apache Kafka is a distributed publish-subscribe messaging system and a robust queue that can handle a high volume of data and enables you to pass messages from one end-point to another. Kafka is suitable for both offline and online message consumption.

Kafka messages are persisted on the disk and replicated within the cluster to prevent data loss. Kafka is built on top of the ZooKeeper synchronization service. It integrates very well with Apache Storm and Spark for real-time streaming data analysis.



## Topic

A topic is a category or feed name to which records are published. Topics in Kafka are always multi-subscriber.

For each topic, the Kafka cluster maintains a partitioned log that looks like this:



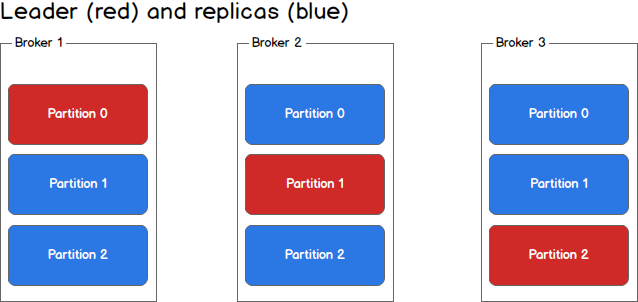
Each partition is an ordered, immutable sequence of records that is continually appended to a structured commit log. The records in the partitions are each assigned a sequential id number called the **offset** that uniquely identifies each record within the partition.

The Kafka cluster retains all published records whether or not they have been consumed using a configurable retention period.

## Partitions and Brokers

The partitions of the log are distributed over the (broker) servers in the Kafka cluster with each server handling data and requests for a share of the partitions. Each partition is replicated across a configurable number of servers for fault tolerance.

Each broker holds a number of partitions and each of these partitions can be either a leader or a replica for a topic. All writes and reads to a topic go through the leader (partition) and the leader coordinates updating replicas with new data. If a leader fails, a replica takes over as the new leader.



## Producers

Producers publish data to the topics of their choice. The producer is responsible for choosing which record to assign to which partition within the topic. This can be done in a round-robin fashion simply to balance load or it can be done according to some semantic partition function (say based on some key in the record)

## Consumers and Consumer Groups

Multiple Consumers with same group-id forms a consumer group to read only a particular topic, this consumer group model provides scalability feature to the Kafka. Recommended scenario - each consumer within the group reads from a unique partition and the group as a whole consumes all messages from the entire topic. If you have more consumers than partitions then some consumers will be idle because they have no partitions to read from. If you have more partitions than consumers then consumers will receive messages from multiple partitions. If you have equal numbers of consumers and partitions, each consumer reads messages in order from exactly one partition.

The following picture describes the situation with multiple partitions of a single topic. Server 1 holds partitions 0 and 3 and server 3 holds partitions 1 and 2. We have two consumer groups, A and B. A is made up of two consumers and B is made up of four consumers. Consumer Group A has two consumers of four partitions — each consumer reads from two partitions. Consumer Group B, on the other hand, has the same number of consumers as partitions and each consumer reads from exactly one partition.



## Consistency and availability

Applies only as long as you are producing to one partition and consuming from one partition.

Kafka makes the following guarantees about data consistency and availability: (1) Messages sent to a topic partition will be appended to the commit log in the order they are sent, (2) a single consumer instance will see messages in the order they appear in the log, (3) a message is ‘committed’ when all in sync replicas have applied it to their log, and (4) any committed message will not be lost, as long as at least one in sync replica is alive.

More on this <https://sookocheff.com/post/kafka/kafka-in-a-nutshell/>

For a **producer** we have three choices. On each message we can (1) wait for all in sync replicas to acknowledge the message, (2) wait for only the leader to acknowledge the message, or (3) do not wait for acknowledgement. Each of these methods have their merits and drawbacks and it is up to the system implementer to decide on the appropriate strategy for their system based on factors like consistency and throughput.

On the **consumer** side, we can only ever read committed messages (i.e., those that have been written to all in sync replicas). Given that, we have three methods of providing consistency as a consumer: (1) receive each message **at most once**, (2) receive each message **at least once**, or (3) receive each message **exactly once**.

For at most once message delivery, the consumer reads data from a partition, commits the offset that it has read, and then processes the message. If the consumer crashes between committing the offset and processing the message it will restart from the next offset without ever having processed the message. This would lead to potentially undesirable message loss.

A better alternative is at least once message delivery. For at least once delivery, the consumer reads data from a partition, processes the message, and then commits the offset of the message it has processed. In this case, the consumer could crash between processing the message and committing the offset and when the consumer restarts it will process the message again. This leads to duplicate messages in downstream systems but no data loss.

Exactly once delivery is guaranteed by having the consumer process a message and commit the output of the message along with the offset to a transactional system. If the consumer crashes it can re-read the last transaction committed and resume processing from there. This leads to no data loss and no data duplication. In practice however, exactly once delivery implies significantly decreasing the throughput of the system as each message and offset is committed as a transaction.

In practice most Kafka consumer applications choose at least once delivery because it offers the best trade-off between throughput and correctness. It would be up to downstream systems to handle duplicate messages in their own way.

### Exactly-Once semantics

Use a single-writer per partition and every time you get a network error check the last message in that partition to see if your last write succeeded

Include a primary key (UUID or something) in the message and deduplicate on the consumer.

## Tools & Settings

### Kafka Streams Application Reset Tool @Broker

Basically, Kafka Streams does not recommend to change the number of input topic partitions during its "life time". If you stop a running Kafka Streams application, change the number of input topic partitions, and restart your app it will most likely break with an exception as described in FAQ "What does exception "Store <someStoreName>'s change log (<someStoreName>-changelog) does not contain partition <someNumber>" mean?". This reset tool is used to fix this problem, but it is tricky to fix this for production use cases and it is highly recommended to not change the number of input topic partitions. The reason is you need to reprocess the whole input topic from beginning. This is of course only possible (definitely not in Production case), if all input data is still available and nothing got deleted by brokers that applying topic retention time/size policy.

Yes, new brokers can be added online to a cluster. Those new brokers won't have any data initially until either some new topics are created or some replicas are moved to them using this tool.

### Rebalancing @Consumer

During the rebalance process, each consumer will execute the same deterministic algorithm to range partition a sorted list of topic-partitions over a sorted list of consumer instances.

### ConsumerOffsetChecker @Consumer

bin/kafka-run-class.sh kafka.tools.ConsumerOffsetChecker --group consumer-group1 --zkconnect zkhost:zkport --topic topic1

consumer-group1,topic1,0-0 (Group,Topic,BrokerId-PartitionId)

Owner = consumer-group1-consumer1

Consumer offset = 70121994703

= 70,121,994,703 (65.31G)

Log size = 70122018287

= 70,122,018,287 (65.31G)

Consumer lag = 23584

= 23,584 (0.00G)

If consumer offset is not moving after some time, then consumer is likely to have stopped. If consumer offset is moving, but consumer lag (difference between the end of the log and the consumer offset) is increasing, the consumer is slower than the producer. If the consumer is slow, the typical solution is to increase the degree of parallelism in the consumer. This may require increasing the number of partitions of a topic.

### Rewind Offset @Consumer

Seek API to set to next position that will be fetched. You can either seek to the earliest position with seekToBeginning(), the latest with seekToEnd(), or to an arbitrary offset with seek().

### Auto Offset Commit @Consumer

* Set dual.commit.enabled=false and offsets.storage=zookeeper (Commit offsets to Zookeeper only).
* Set dual.commit.enabled=true and offsets.storage=kafka (Commit offsets to Zookeeper and Kafka).
* Set dual.commit.enabled=false and offsets.storage=kafka (Commit offsets to Kafka only).

Set auto.offset.reset in ConsumerConfig to "earliest" to read from the latest

### Manual Offset Commit @Consumer

auto.commit.enable=false in your consumer's config. Using commitOffsets API handle in the code.

### To Consume Large Messages

message.max.bytes @Broker

max.partition.fetch.bytes @Consumer

## Exceptions

QueueFullException – @Broker

This typically happens when the producer is trying to send messages quicker than the broker can handle. If the producer can't block, one will have to add enough brokers so that they jointly can handle the load. If the producer can block, one can set queue.enqueueTimeout.ms in producer config to -1. This way, if the queue is full, the producer will block instead of dropping messages.

InvalidMessageSizeException – @Consumer

This typically means that the "fetch size" of the consumer is too small. The default fetch.size is 300,000 bytes. For the new consumer starting v0.9, the property to adjust is "max.partition.fetch.bytes," and the default is 1MB.

In case of setting compression.codec to 1 in ProducerConfig – make sure you add snappy jar in the producer classpath

## Known Issues

### Multi-Threading

Kafka 0.10.0 and 0.10.1 does have some bugs with regard to multi-threading. If you cannot (or don't want to) update, you can apply the following workaround:

Switch to single threaded execution

* if you want to scale your app, start multiple instances (instead of going multi-threaded with one instance)
* if you start multiple instances on the same host, use a different state directory (state.dir config parameter) for each instance (to "isolate" the instances from each other)

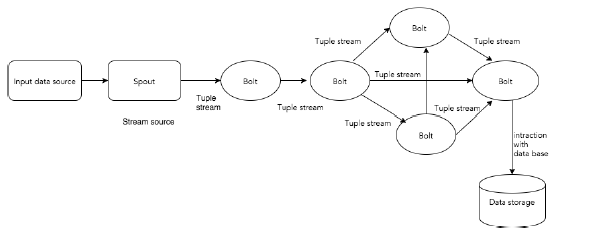
It might also be necessary, to delete the state directory manually before starting the application. This will not result in data loss – the state will be recreated from the underlying changelog topic.0

## Kafka Connect

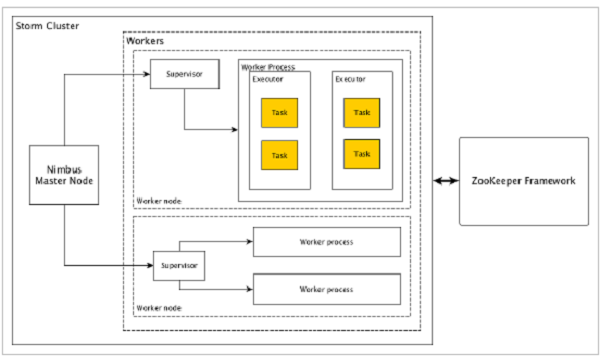
Kafka connectors are 2 types – source connector (connect source system to kafka) and sink connector (connect kafka to target system). In the source connector, the offset ID is controlled by the source, for example in case of RDBMS as source, offset ID can be each transaction ID or a table itself, in case of log aggregation as source, timestamp can be the offset ID. In case of sink connector, the offset ID is the kafka offset ID itself.

# Storm (1.0/.1/.2)

Apache Storm became a standard for distributed real-time processing system that allows you to process a huge volume of data. Storm is very fast and a benchmark clocked it at over a million tuples processed per second per node. Apache Storm runs continuously, consuming data from the configured sources (Spouts) and passes the data down the processing pipeline (Bolts). Combined, Spouts and Bolts make a Topology.



Architecture:



Benefits:

* Storm is fault tolerant, flexible, reliable, and supports any programming language.
* Allows real-time stream processing.
* Storm is unbelievably fast because it has enormous power of processing the data.
* Storm can keep up the performance even under increasing load by adding resources linearly. It is highly scalable.
* Storm performs data refresh and end-to-end delivery response in seconds or minutes depends upon the problem. It has very low latency.
* Storm has operational intelligence.
* Storm provides guaranteed data processing even if any of the connected nodes in the cluster die or messages are lost.

## Facts & Alternates

Storm guarantees that every spout tuple will be fully processed by the topology. It does this by tracking the tree of tuples triggered by every spout tuple and determining when that tree of tuples has been successfully completed. Every topology has a "message timeout" associated with it. If Storm fails to detect that a spout tuple has been completed within that timeout, then it fails the tuple and replays it later.

Storm is stateless in nature. Even though stateless nature has its own disadvantages, it actually helps Storm to process real-time data in the best possible and quickest way.

Storm is not entirely stateless though. It stores its state in Apache ZooKeeper. Since the state is available in Apache ZooKeeper, a failed nimbus can be restarted and made to work from where it left. Usually, service monitoring tools like monit will monitor Nimbus and restart it if there is any failure.

Apache Storm also have an advanced topology called Trident Topology with state maintenance and it also provides a high-level API like Pig.

## Trident

Trident is an extension of Storm, provides a high-level abstraction on top of Storm along with stateful stream processing and low latency distributed querying. Trident uses spout and bolt, but these low-level components are auto-generated by Trident before execution. Trident has functions, filters, joins, grouping, and aggregation. Trident processes streams as a series of batches which are referred as transactions. Generally the size of those small batches will be on the order of thousands or millions of tuples, depending on the input stream. This way, Trident is different from Storm, which performs tuple-by-tuple processing.

**When to Use Trident?**

As in many use-cases, if the requirement is to process a query only once, we can achieve it by writing a topology in Trident. On the other hand, it will be difficult to achieve exactly once processing in the case of Storm. Hence Trident will be useful for those use-cases where you require exactly once processing. Trident is not for all use cases, especially high-performance use-cases because it adds complexity to Storm and manages the state.

# Storm vs Spark Streaming

Apache storm (core) - ideal for Stream processing or ESP cases - (Spark streaming can be used here but then you will be using a batch processor for stream processing)

Spark Streaming - ideal for micro-batching and CEP use cases. (Storm Trident or Storm DRPC can also be used here)

Choosing a tool for stream processing depends on whether we want the data processed in "realtime" or "near realtime", deciding on fault tolerance between "atleast once semantics" and "exactly once semantics" and between "sub-second" latency and "few seconds" latency should help us decide between Storm and Spark.

Here's Storm and Spark in conversation about what they are capable of:

***Storm****: I process one incoming event at a time*

***Spark****: I collect data over small time window as a micro batch and process them*

***Storm****: I can provide sub-second latency*

***Spark****: I only provide few seconds latency. But hey, I ensure that data is read exactly once and guarantee better fault tolerance.*

***Storm****: I guarantee that I read data at least once.*

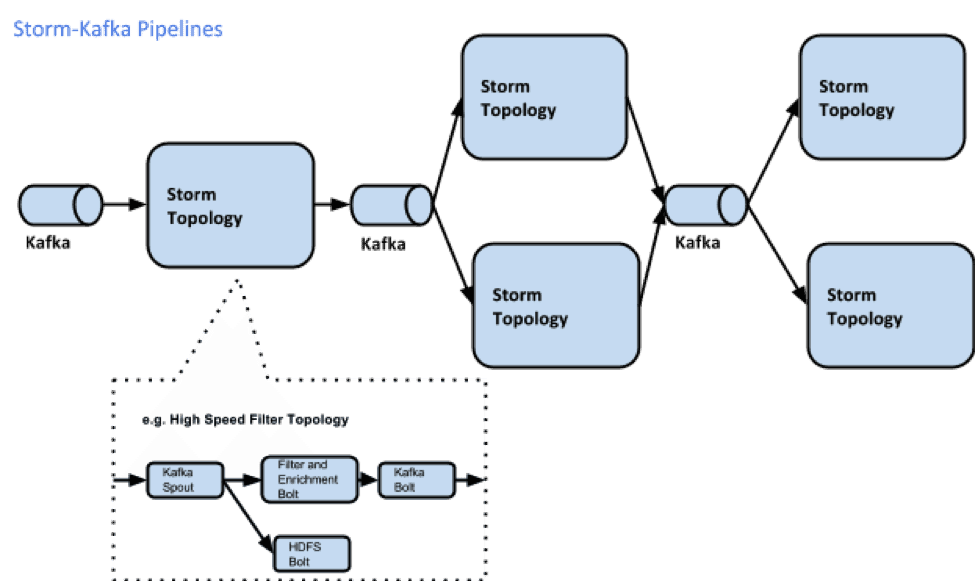
***Spark****: But that may allow duplicate data once you recover from faults.*

***Storm****: Ya. But that's a compromise you have to make if you need faster processing.*

# Kafka + Storm

Kafka is used as persistent store for storing stream of messages. Storm is realtime computational system which can process the stream of messages very fast.

Building the Data Refinery with Topologies

To perform real-time computation on Storm, we create “topologies.” A topology is a graph of a computation, containing a network of nodes called “Spouts” and “Bolts.” In a Storm topology, a Spout is the source of data streams and a Bolt holds the business logic for analyzing and processing those streams.

The first topology ingests raw data streams from Kafka and fans out to HDFS, which serves as persistent store for raw events. Next, a filter Bolt emits the enriched event to a downstream Kafka Bolt that publishes it to a Kafka Topic. As events flow through these stages, the system can keep track of data lineage that allows drill-down from aggregated events to its constituents and can be used for forensic analysis. In a multi-stage pipeline architecture, providing right cluster resources to most intense part of the data processing stages is very critical, an “Isolation Scheduler” in Storm provides the ability to easily and safely share a cluster among many topologies.

# Kafka + Spark Streaming

Spark streaming is used with kafka is to do the transformation job in the ETL pipeline.

As of spark 1.3, spark streaming can connect directly to kafka, earlier to it zookeeper was used for the connection.

*Create DStream’s from a kafka topic*

//create spark streaming context

Val ssc = new StreamingContext(“local[\*]”, “KafkaExample”, Seconds(1))

//hostname:port for kafka brokers, not zookeeper

Val kafkaparams = Map(“metadata.broker.list” -> “localhost:9092”)

//list of topics you want to listen for from kafka

Val topics = List(“testtopic”).toSet

//create our kafka stream, which will contain (topic\_name, message) pairs. We tack a map(\_.\_2) at the end in order to only get the messages. Which contain individual lines of data.

Val lines = KafkaUtils.CreateDirectStream[String, String, StringDecoder, StringDecoder](ssc, kafkaparams, topics).map(\_.\_2)

# Samza (v 0.11)

Apache Samza is a distributed stream processing framework. It uses Apache Kafka for messaging, and Apache Hadoop YARN to provide fault tolerance, processor isolation, security, and resource management.

**Managed state** - Samza manages snapshotting and restoration of a stream processor’s state. When the processor is restarted, Samza restores its state to a consistent snapshot. Samza is built to handle large amounts of state (many gigabytes per partition).

**Fault tolerance** - Whenever a machine in the cluster fails, Samza works with YARN to transparently migrate your tasks to another machine.

**Durability** - Samza uses Kafka to guarantee that messages are processed in the order they were written to a partition, and that no messages are ever lost.

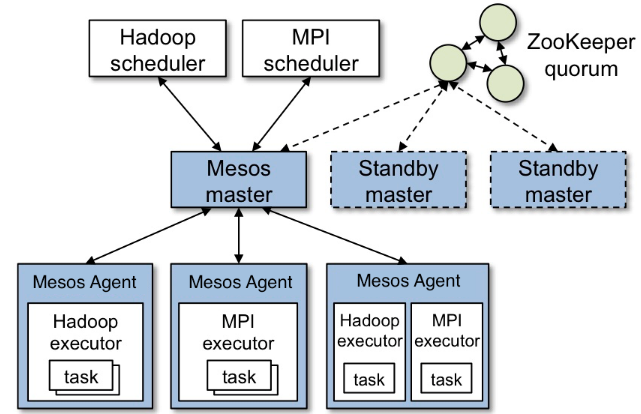
**Scalability** - Samza is partitioned and distributed at every level. Kafka provides ordered, partitioned, replayable, fault-tolerant streams. YARN provides a distributed environment for Samza containers to run in.

# Apache Solr

Apache Solr is an open source search platform built upon a Java library called Lucene. Solr is a popular search platform for Web sites because it can index and search multiple sites and return recommendations for related content based on the search query's taxonomy.

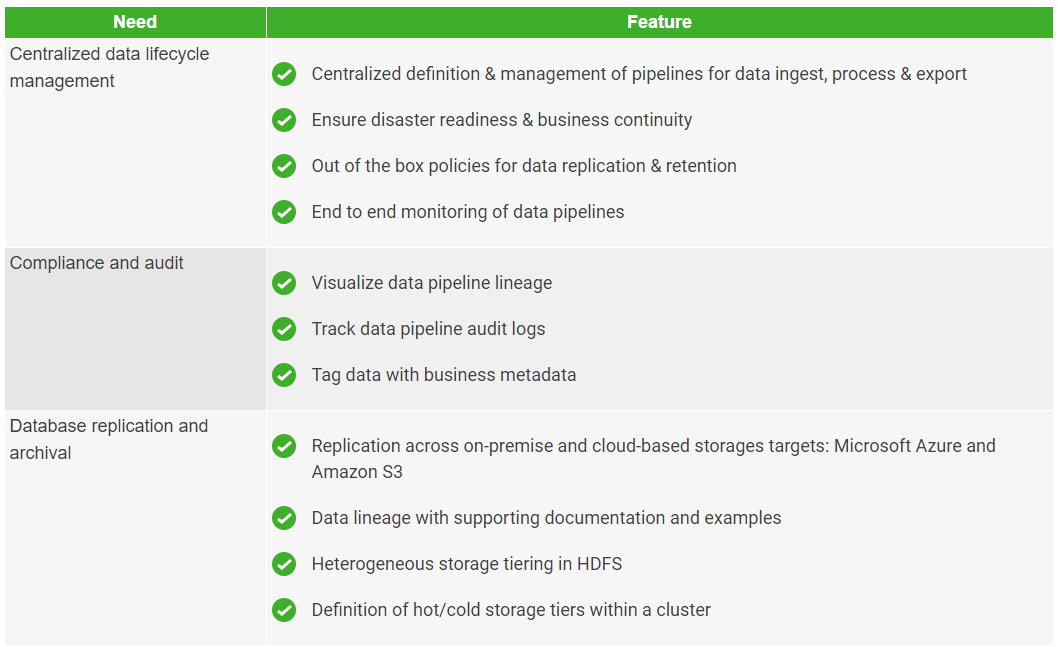
# Apache Mesos

Apache Mesos is an open-source cluster manager. It provides efficient resource isolation and sharing across distributed applications, or frameworks. The software enables resource sharing in a fine-grained manner, improving cluster utilization.



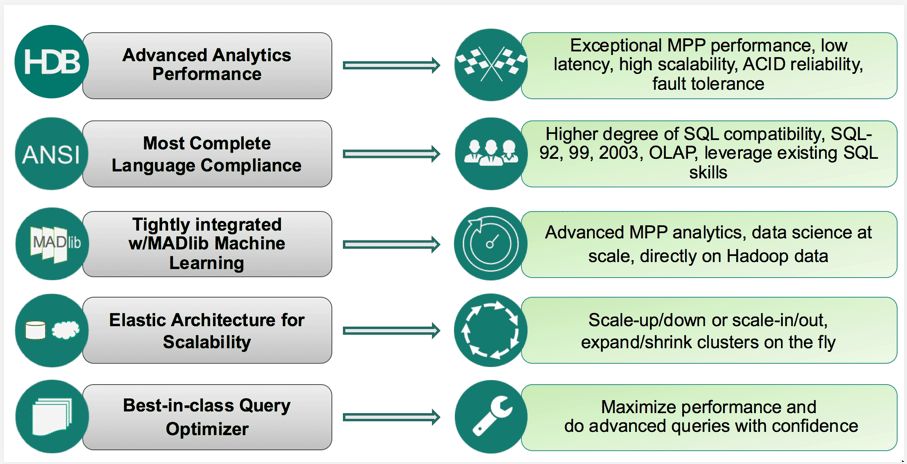
# Apache Falcon

A framework for managing data lifecycle in Hadoop clusters.



# Apache Hawq

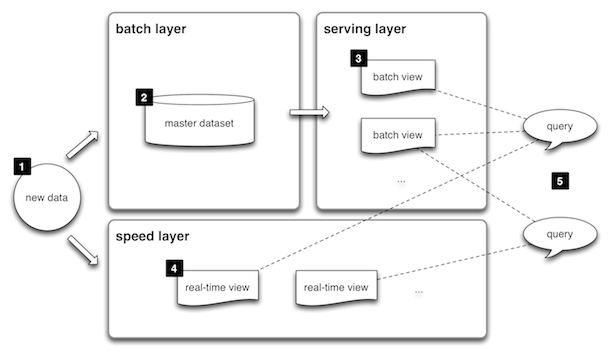
Apache HAWQ (incubating) provides native SQL on Apache Hadoop based on an advanced MPP elastic query engine. HAWQ represents a new generation of high performance, advanced analytics that transforms Hadoop into an enterprise analytic database. Move and analyze entire workloads, while simplifying management and expanding the breadth of data access and analytics, all natively in Hadoop.



# Big Data Architectures

## Lambda

The Lambda Architecture aims to satisfy the needs for a robust system that is fault-tolerant, both against hardware failures and human mistakes, being able to serve both stream and batch processing, wide range of workloads and use cases, and in which low-latency reads and updates are required. The resulting system should be linearly scalable, and it should scale out rather than up.



Lambda architecture describes a system consisting of three layers: batch processing, speed (or real-time) processing, and a serving layer for responding to queries. The processing layers ingest from an immutable master copy of the entire data set.

**Batch layer**

The batch layer precomputes results using a distributed processing system that can handle very large quantities of data. The batch layer aims at perfect accuracy by being able to process all available data when generating views. This means it can fix any errors by recomputing based on the complete data set, then updating existing views. Output is typically stored in a read-only database, with updates completely replacing existing precomputed views.

Apache Hadoop is the de facto standard batch-processing system used in most high-throughput architectures.

**Speed layer**

Diagram showing the flow of data through the processing and serving layers of lambda architecture. Example named components are shown.

The speed layer processes data streams in real time and without the requirements of fix-ups or completeness. This layer sacrifices throughput as it aims to minimize latency by providing real-time views into the most recent data. Essentially, the speed layer is responsible for filling the "gap" caused by the batch layer's lag in providing views based on the most recent data. This layer's views may not be as accurate or complete as the ones eventually produced by the batch layer, but they are available almost immediately after data is received, and can be replaced when the batch layer's views for the same data become available.

Stream-processing technologies typically used in this layer include Apache Storm, SQLstream and Apache Spark. Output is typically stored on fast NoSQL databases.

**Serving layer**

Output from the batch and speed layers are stored in the serving layer, which responds to ad-hoc queries by returning precomputed views or building views from the processed data.

Examples of technologies used in the serving layer include Druid, which provides a single cluster to handle output from both layers. Dedicated stores used in the serving layer include Apache Cassandra or Apache HBase for speed-layer output, and Elephant DB or Cloudera Impala for batch-layer output.

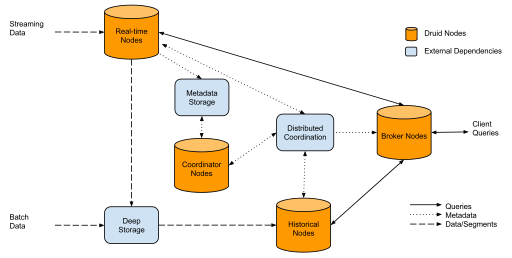
## Kappa

Kappa Architecture is a simplification of Lambda Architecture, with the batch processing system removed. Rather than using a relational DB like SQL or a key-value store like Cassandra, the canonical data store in a Kappa architecture system is an append-only immutable log. From the log, data is streamed through a computational system and fed into auxiliary stores for serving. To replace batch processing, data is simply fed through the streaming system quickly.

# Druid

Druid is a column-oriented, open-source, distributed data store written in Java. Druid is designed to quickly ingest massive quantities of event data, and provide low-latency queries on top of the data. Druid is commonly used in business intelligence/OLAP applications to analyze high volumes of real-time and historical data.

Druid runs as a cluster of specialized processes (called nodes in Druid) to support a fault-tolerant architecture where data is stored redundantly, and there is no single point of failure. The cluster includes external dependencies for coordination (Apache ZooKeeper), metadata storage (e.g. MySQL, PostgreSQL, or Derby), and a deep storage facility (e.g. HDFS, or Amazon S3) for permanent data backup.



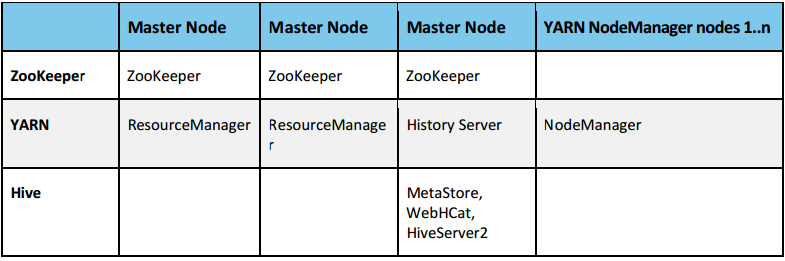
# Infrastructure

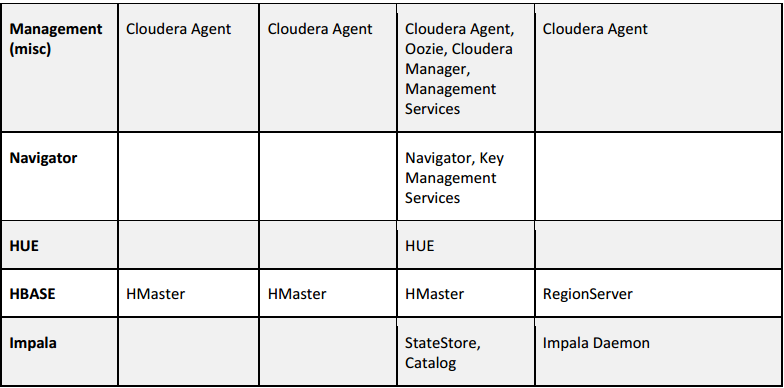
Each cluster will have master and compute nodes

The minimum requirements to build out the cluster are:

* Three master nodes
* The number of compute nodes depends on the cluster size (see sizing considerations section below)

The following table identifies service roles for different node types (CDH).





The following table provides size recommendations for the physical nodes.

