**OLTP** (OnLine Transaction Processing Systems): systems designed to handle vast amounts of end user requests as fast as possible

**OLAP** (OnLine Analytical Processing Systems): systems designed to handle much more complex queries over vast amounts of data for the purpose of business intelligence or reporting rather than to process end-user transactions.

The Hadoop ecosystem is mostly an OLAP platform.

**Hadoop**

1. business outcome:
2. Data discovery
3. Single view
4. Predictive analysis
5. Cost saving:
6. Building active achieves
7. Optimize ETL on-board
8. Practicing data enrichment
9. Hadoop
10. HDFS+YARN
11. trading the extreme convenience of relational databases for the extreme scalability of Hadoop

**HDFS**

1. HDFS (Hadoop Distributed File System) - distributed, fault tolerant filesystem service implemented in Java

* HDFS Divide files into blocks and store them on different servers in the cluster. Each server is either a **master node, worker node** or **utility node.**
* Store multiple replicas of blocks for reliability (A default replication factor of 3 for every block) -> if a server hosting a block fails, HDFS will still give you that data back by reading it from a different server in the cluster
* Instead of sending data to the client, you send client computation to the data.

1. HDFS components

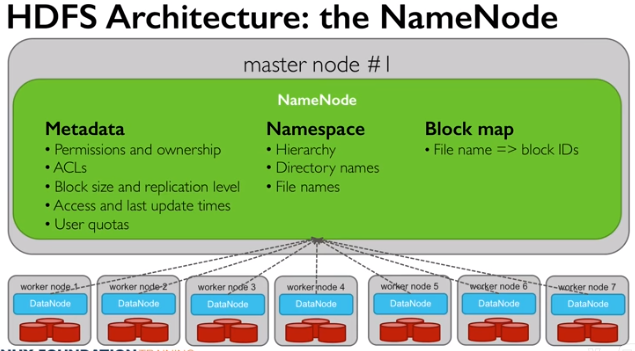
* **NameNode** (one or two per cluster) -> master

- Represents a single filesystem namespace rooted at /

- Keep track of what DataNodes are online by detecting “heartbeat” from DataNodes, and what DataNodes are providing storage services in form of block information (determines and maintains how the chunks of data are distributed across the DataNodes)

- Actual data never resides here, only metadata (e.g., maps of where blocks are distributed).

**- High Availability (HA) configuration**: With two NameNodes running in an Active/Standby configuration, the Standby can take over in cases where the Active one fails or needs to be brought down for maintenance.



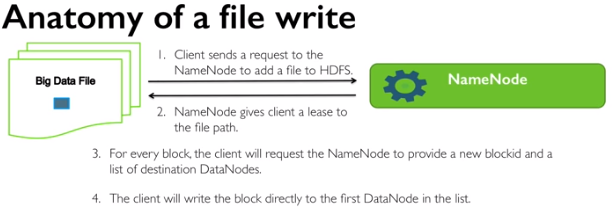
* **DataNode** (as many as you want per cluster) -> worker

**-** DataNode is a single storage unit connected to NameNode, no other way to access it, but to read it from the DataNode itself.

- Stores the chunks of data, and is responsible for replicating the chunks across other DataNodes

- Default block size on most clusters is 128MB.

3. File write in HDFS



**YARN**

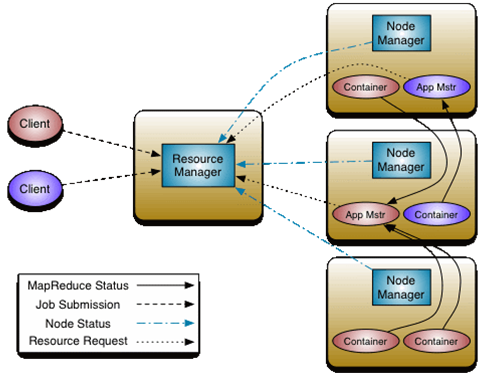
1. YARN (Yet Another Resource Negotiator) is a distributed resource management and scheduling framework written in Java, running on the same nodes in the cluster that HDFS runs.

* Allowing computation to run on each node in the cluster
* Using the same nodes for computation that HDFS was using for storing blocks
* Making sure that pieces of computation got scheduled on the nodes that hosted blocks required for computation locally, thus avoiding the need to send data over the network.

"Don't bring data to compute, bring compute to data",

1. YARN components

YARN consists of one ResourceManager service coordinating all the cluster compute resources, and a lot of NodeManager services that connect to the ResourceManager, receive computational tasks from it, and execute them on the worker nodes.



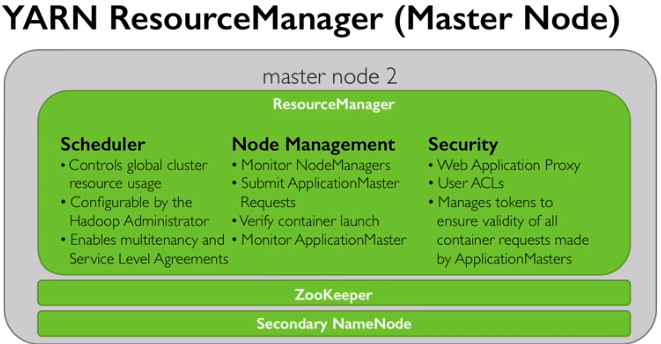
* **Resource Manager** (one or two per cluster) -> master process

- Global resource scheduler

- NodeManagers Coordination by monitoring “heartbeat”

- Hierarchical queues

- High Availability (HA) configuration: two Resource Managers running in an Active/Standby configuration, the Standby can take over in cases where the Active one fails or needs to be brought down for maintenance.



* **Node Manager** (running next to the DataNode)

- Encapsulates RAM and CPU resources available on a worker node into units called YARN **containers** (unit of work allocated specific CPU and memory resources by the NodeManager; runs ApplicationMaster job)

- NodeManagers are typically running on the same servers that DataNodes are running on -> In order to minimize the network traffic and use data that is stored on the node itself

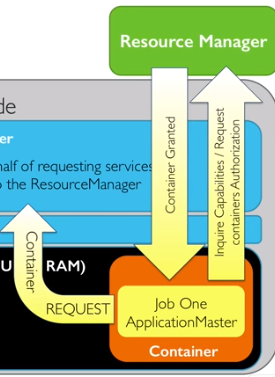
* **Application Master** (created on-demand)

- Bootstrap process that launches and manages everything for a YARN application.

- Manages application scheduling and task execution

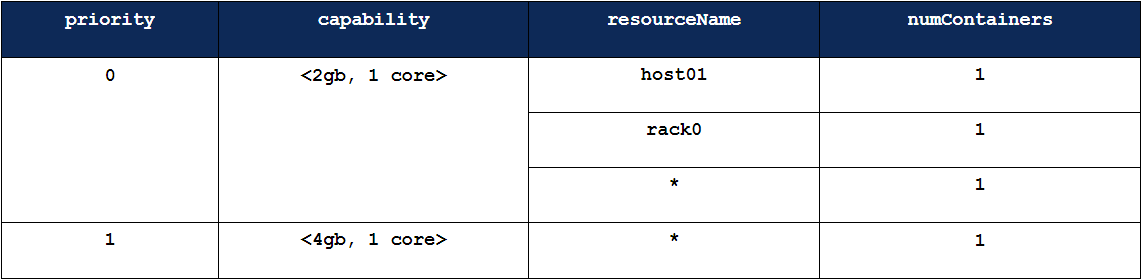
- Each Application Master, is an instance of an application's framework-specific library of code, and it comes from the client, not from Hadoop itself. Typically, specific to a higher-level framework (e.g. MapReduce, Spark).

- **Negotiate for resources** (memory, CPU, etc.) with ResourceManager: The ApplicationMaster requests additional resources from the ResourceManager via the NodeManager on which it is running. The request, after being checked against the policy, is typically granted, and an ApplicationMaster can now send a validated request to a NodeManager -> From a NodeManager standpoint, this is just a regular request for allocating a container. (It could have been a different NodeManager running on another worker node)



1. Resource requests

* Priority
* Capability: each request asks for a specific amount of resources (memory, CPU, etc.)
* resourceName: where to put the containers
* numContainers



1. To deal with multi-tenant environment: capacity Scheduler

**Deployment Mode**

1. **Non-Distributed/local** mode: using data from files stored in local filesystem

There are no APIs aside from what is embedded right in your application.

No YARN service running in local mode.

hdfs dfs –mkdir input Create a folder in hdfs

hdfs dfs –put /tmp/zip\_codes.csv input put a file from local filesystem into Hadoop

1. **Pseudo-distributed** mode (UI address: localhost:50070)

The APIs are provided by genuine Hadoop services, but they are all running on the same host -> a cluster with just one node.

1. **HDFS**
2. **XML configuration** to tell HDFS to use just a single DataNode for each block (it uses 3 by default) and to tell NameNode what port to answer on:

Placing the following XML in $HADOOP\_HOME/etc/hadoop/hdfs-site.xml:

<configuration>

<property>

<name>dfs.replication</name>

<value>1</value> -> single node

</property>

</configuration>

Placing the following XML in $HADOOP\_HOME/etc/hadoop/core-site.xml:

<property>

<name>fs.defaultFS</name>

<value>hdfs://localhost:9000</value>

</property>

1. Formatting file system

$ hdfs namenode –format

1. Start a NameNode and a DataNode services, so we can store some data:

$ hdfs namenode > name\_node.log 2>&1 &

$ hdfs datanode > data\_node.log 2>&1 &

Showing what processes are running on your system: $ jobs

1. **Yarn** (UI address: localhost:8088)

$ yarn resourcemanager > resource\_manager.txt 2>&1 &

$ yarn nodemanager > node\_manager.txt 2>&1 &

To kill a node manager (no. 4): $ kill -9 %4

Run Spark on it: (deploy master using yarn in a client mode.)

$ YARN\_CONF\_DIR=$HADOOP\_HOME/etc/hadoop spark-shell --master yarn --deploy-mode client

scala> sc.textFile("input").flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(\_ + \_). collect().foreach(println)

1. **MapReduce**

XML configuration: $HADOOP\_HOME/etc/hadoop/yarn-site.xml

<configuration>

<property>

<name>yarn.nodemanager.aux-services</name>

<value>mapreduce\_shuffle</value>

</property>

</configuration>

Make MapReduce default to YARN as an execution engine: $HADOOP\_HOME/etc/hadoop/mapred-site.xml

<configuration>

<property>

<name>mapreduce.framework.name</name>

<value>yarn</value>

</property>

</configuration>

Submit a MapReduce job:

$ hadoop jar $HADOOP\_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.3.jar wordcount input output

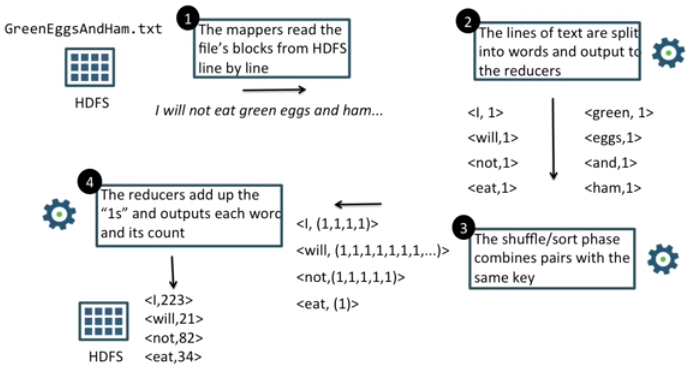
1. **Distributed mode**

The APIs are provided by genuine Hadoop services, running either on a cluster or in the cloud.

**Parallel Processing**

1. **MapReduce**: Map-Sort/Shuffle-Reduce
2. Mapper: the processes doing parallel computation of some sorts. Each Mapper (no. of Mapper = no. of blocks of input data in HDFS) produces a value or a stream of values that is then given to Reducer, so that a Reducer can arrive at a final answer.
   * The Mapper reads data in the form of key-value pairs (KVPs), and it outputs zero or more KVPs. This stream of KVPs from a Mapper (intermediate data) is not stored in HDFS, but rather consumed over the network by one or more Reducers.
3. Reducer: After the Map phase is over, all the intermediate values for a given intermediate key are combined together into a list. This list is given to a Reducer.
   * All values associated with a particular intermediate key are guaranteed to go to the same Reducers. The intermediate keys, as well as their value lists, are passed in a sorted order. The invisible MapReduce phase that makes sure the Reducer gets the keys in a sorted order is called a **Shuffle phase**.
   * No. of Reducers is governed by a configuration option, and is 1 by default. -> determine the number of the final output files for the MapReduce job.
   * The Reducer outputs zero or more KVPs. These KVPs are written to HDFS.

e.g. Word count



$ hadoop jar $HADOOP\_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.3.jar wordcount input output

$ hadoop fs -getmerge output word\_count.txt

$ cat word\_count.txt

1. **Spark** (UI address: http://10.0.2.15:4040)

All the intermediate results are stored in the **memory** of each Spark node in the cluster -> don’t need to read/write HDFS. Iterative applications in MapReduce require intermediate data to be written to HDFS, and then, read again for every iteration.

RDD (Resilient Distributed Dataset): each RDDs is a fault-tolerant collection of elements that can be operated on in parallel.

Two ways to create RDDs: a. parallelizing an existing collection b. referencing a dataset in an external storage system, such as a shared filesystem, HDFS, Amazon's S3, etc.

Two types of operations: a. **transformations**, which derives a new RDD from an existing one. All transformations in Spark do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result to be returned to the client -> enables Spark to run more efficiently. b. **actions**, which return a value to the client after running a computation on the dataset.

e.g. wordcount

JavaRDD<String> textFile = sc.textFile("hdfs://...."); -> create an RDD called textFile by pointing at a dataset in HDFS.

JavaPairRDD<String, Integer> counts = textFile

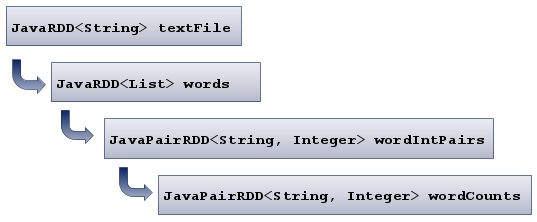
.flatMap(s -> Arrays.asList(s.split(" ")).iterator())

.mapToPair(word -> new Tuple2<>(word, 1))

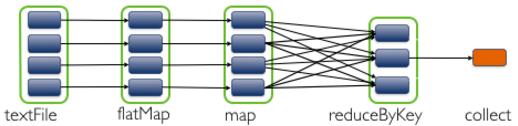
.reduceByKey((a,b) -> a + b); -> 3 transfomations: flatMap, mapToPair, reduceByKey

counts.saveAsTextFile("hdfs://...."); -> action: forces the content of this RDD to be written to HDFS and make computation kick-in

RDD transformation chain:

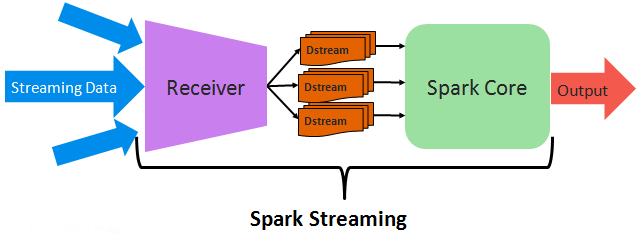


RDD partitioning maps to physical nodes in the cluster:



Spark streaming: allows for in-flight data processing with a latency as low as 1-2s.

The receivers listen to data streams, and create batches of data called Dstreams, that are then processed by the Spark Core engine.



1. **Hive**

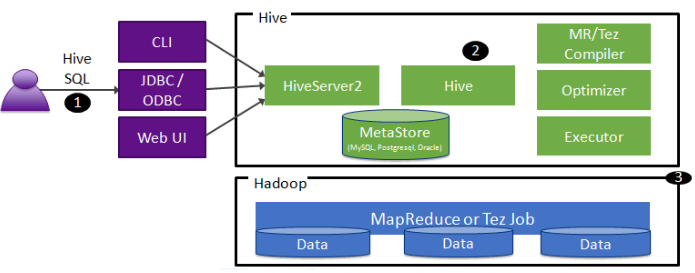
Two major component: a. **HiveServer2** (for managing clients connections, just like a regular relational database would) b. **MetaStore** (for mapping unstructured data stored in HDFS into relational schema).

Any client connecting to Hive goes through the following steps:

(1) Issue an SQL query, either through a command line tool or a JDBC connection.

(2) HiveServer2 parses the query and prepares an execution plan, based on the data in the MetaStore.

(3) Hive executor converts the query into a series of **MapReduce** jobs or a **Tez** job (Apache Hive community has explored using **Spark** as an alternative execution layer).



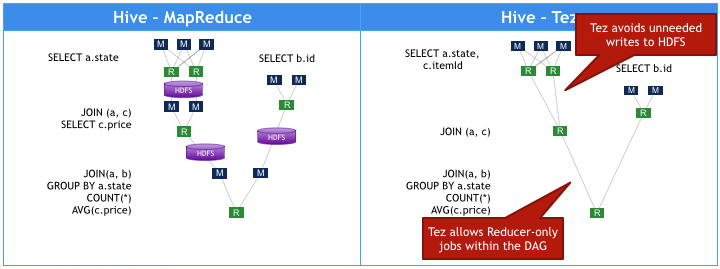
e.g select sales information (Table a is the main table with all the sales transactions)

SELECT a.state, COUNT(\*), AVG(c.price) FROM a

JOIN b ON (a.id = b.id)

JOIN c ON (a.itemId = c.itemId)

GROUP BY a.state



When Hive is running on the MapReduce engine, the only way to exchange data between different stages of the pipeline is to write it all out into HDFS and read it back later. Spark and Tez (shown on the right) don't have that limitation. They can make different stages of the query execution pipeline feed off of each other simply by exchanging data **in-memory**. On top of that, neither Tez nor Spark force Hive to create Mappers when there is no need for them (like in the final stage of our query execution pipeline that is all about a single Reducer).

Create a **native Hive table**:

CREATE TABLE customer (

customerID INT,

firstName STRING,

lastName STRING

birthday TIMESTAMP,

) ROW FORMAT DELIMITED

FIELDS TERMINATED BY ',';

LOCATION '/user/roman/customers'; -> specify where exactly in the HDFS file tree does Hive manage these files.

-> A bunch of CSV files (this is what FIELDS TERMINATED BY ',' tells Hive) in HDFS. All of those files will be deleted when you drop the table.

Create an external table: add EXTERNAL before TABLE

Load data:

a. From a local filesystem:

LOAD DATA LOCAL INPATH '/tmp/customers.csv' OVERWRITE INTO TABLE customers;

b. from a set of folders in HDFS:

LOAD DATA INPATH '/user/train/customers.csv' OVERWRITE INTO TABLE customers;

c. A more SQL-like way of loading data from an existing query:

INSERT INTO customers

SELECT firstName, lastName, birthday

FROM users

WHERE birthday IS NOT NULL;

-> Results: CSV files with data created in HDFS.

Create views to facilitate access to frequently issued queries:

CREATE VIEW 2017\_visitors AS

SELECT fname, lname, time\_of\_arrival, info\_comment

FROM wh\_visits

WHERE

cast(substring(time\_of\_arrival, 6, 4) AS int) >= 2017

AND

cast(substring(time\_of\_arrival, 6, 4) AS int) < 2018;

$ hive

hive> CREATE EXTERNAL TABLE zips (zip int, city String) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LOCATION ’/tutorial/input'; -> tell Hive how to interpret data in HDFS to table abstractions. (create an external table called zips)

hive> select city, count(\*) from zips group by city;

Other EDW support: Spark SQL, Apache Drill, Apache HAWQ (incubating), Apache Impala (incubating), and Presto

**Hadoop Security**

1. Comprehensive enterprise security is based on five fundamental pillars:

- Data protection: How can data on-disk and data on-wire be protected from snooping?

- Authentication: How can we make sure that users are who they say they are?

- Authorization: How can we easily manage what users can and cannot do?

- Audit: How can we track what users do?

- Administration: How do we centralize all security management around Hadoop?

2. Kerberos provides strong authentication

Apache Knox provides perimeter security and additional centralized authentication

Apache Ranger provides authorization and audit (in conjunction with Apache Atlas)

Strong encryption is used to protect data at-rest and data in-motion

Deploying secure Hadoop clusters can be facilitated by Apache Ambari.