Big Data Unit 2

Samyak Sarnayak

December 29, 2020

1 Hadoop Ecoystem

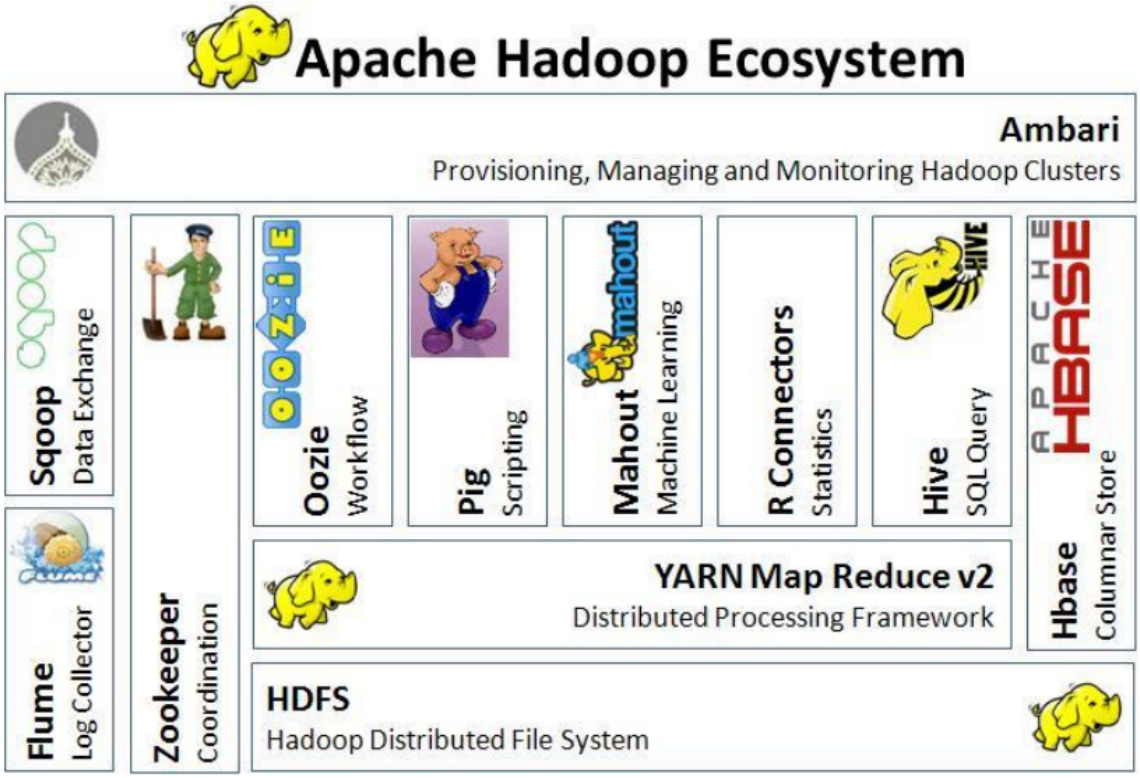


Figure 1: Hadoop Ecosystem

The hadoop ecosystem has:

1. HDFS: Distributed File System
2. Map Reduce
3. Oozie: workflow creation
4. Pig: write operations in a script -¿ gets converted to map-reduce tasks
5. Sqoop
6. Flume
7. Zookeeper
8. HBase

1.1 Oozie

- Oozie is a workflow engine for Hadoop
- Workflow is a sequence of steps. Steps in Oozie are Map-Reduce tasks
- User will specify steps, when to run steps, what to do when it fails, what to do next after a step succeeds
- Architecture
  - A Client application (Java): lets user create workflows
  - Oozie Web App: Server made in Apache Tomcat (which is like Flask / Django for Java). Workflows are submitted to this server.
  - Oozie DB: workflows are stored in a MySQL/Postgres database
  - Scheduler: Oozie schedules jobs according to the workflows. It supports many types of jobs - MapReduce, Hive, Spark, Pig, Sqoop, Java, etc.
- Workflows consists of nodes - a DAG (Directed Acyclic Graph)
  - Action Nodes: takes an action - run mapred task, etc. Has two exits - normal exit and error exit.
  - Control Nodes: Start (start of workflow), End, Kill (end with error), Others - Fork (start parallel task), Join (merge parallel tasks), Decision (ex: switch).

1.2 Ambari

- To Deploy and manage hadoop clusters
- Provides UI to easily install, configure and manage hadoop software on a cluster
- Repeatable clusters - config can be copied to a new cluster
- Architecture
  - Ambari **Stacks:** software stack - applications to install, components, structure. Adds *services*. Custom stacks can also be made.
  - Ambari **Views:** Web UI. Can also be customised.
  - Ambari **Blueprints:** Defines layout, cluster configuration. Using blueprints, installation can be automated.
- Ambari Stacks
  - **Stack:** set of services, where to get them, managing their lifecycle. Ex: HDP 2.3, HDP 2.2
  - **Service:** components of stacks. Ex: HDFS, YARN
  - **Component:** building blocks of a service. Ex: Namenode, datanode
  - **Category:** category of component. Ex: Master, slave, client

1.3 Pig

- MapReduce too low level for data analysis
- Pig Latin is a scripting language to do this - scripts are converted to MapReduce jobs
- Interactive shell named Grunt shell
- Data transformations done using built-in operators - join, group, filter, limit
- `data = load 'filename' as (col1, col2, col3);`
- `dgroup = group data by col2;`
- `counts = foreach dgroup generate col1, count(visits);`
- (You are not expected to memorize the syntax of Pig)

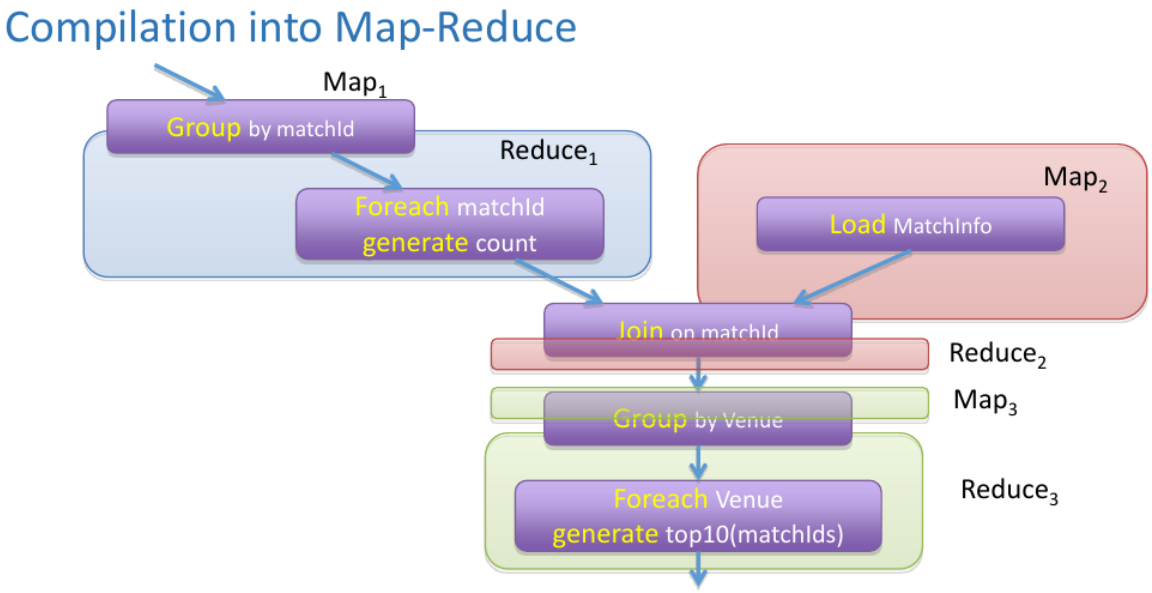


Figure 2: Example of pig script being converted to Map-Reduce tasks. (important)

1.4 SQOOP

- Need to get data from SQL Database, perform analytics and store back in DB
- SQl to hadOOP = SQOOP
- Can transfer voluminous data - Bulk Data Transfer Tool
- Can import/export data to/from SQL DB
- Integrates with Oozie as part of workflow (action node)
- Customizable - plugins for new DBs can be written
- Step 1: inspect DB and get metadata (table schema, etc.)
- Step 2: Transfer data using a map-only hadoop job, data is stored onto HDFS as a CSV file

1.5 Flume

- Service for efficiently collecting, aggregating, and moving large amounts of log data
- It is robust and fault tolerant
- Architecture
  - **Sources**: accept data from an app. Ex: source to collect web server logs
  - **Sinks**: receives data and stores into HDFS
  - **Channel**: connects sources to sinks
  - **Agent**: run sources and sinks within Flume

2 Page Rank

2.1 Representing pages as a matrix

- Represent pages as a directed graph. Vertices in the graph are hyperlinks and nodes are pages.

- Graph can be represented as an Adjacency Matrix - element is 1 when link is present, 0 when it's not. This will be huge for the web with billions of pages.
- Adjacency Matrix will be sparse
- in HDFS, sparse matrix is stored as a CSV with each row having

$$\langle row\_number, column\_number, value \rangle$$

- Number of entries is equal to number of links.

2.2 Matrix Vector Multiplication with MapReduce

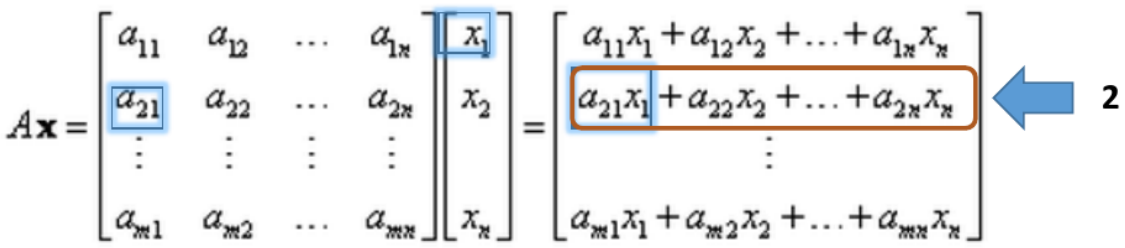
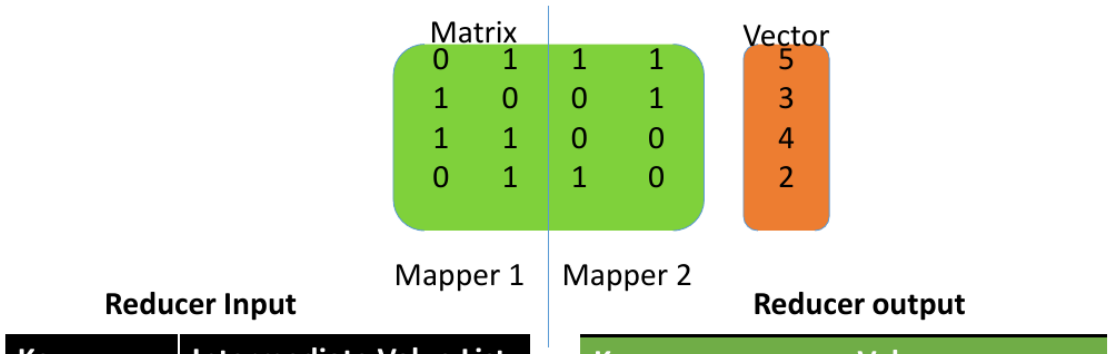


Figure 3: Matrix Vector Multiplication - Map step



Reducer Input

Key	Intermediate Value List
1	2,3,4
2	2, 5
3	3, 5
4	3, 4

Reducer output

Key	Value
1	9
1	7
2	8
4	7

Figure 4: Matrix Vector Multiplication - Mapper and Reducer outputs

- matrix M of size $n \times n$ is to be multiplied with n-element vector v, gives a vector of size n.
- v fits in memory and is loaded by all the mappers
- M is stored as a CSV file in HDFS (is distributed across all nodes)
- Map step:

- Computes partial product between the element it sees M_{ij} and the corresponding element in the vector v_j.
- In figure Figure 3, a_{21} has to be multiplied with x_1 of v and the result goes into row number 2. Hence the key will be 2.
- The key is i - index of row in final vector
- Output $\langle i, M_{ij}, v_j \rangle$.

- Reduce step:
  - Sum all partial products (same key as mapper)
- Figure 4 shows the mapper and reducer outputs for an example

2.2.1 When Vector v does not fit into memory

Partition M and v into stripes. Same algorithm can be used and stripes can be multiplied separately.

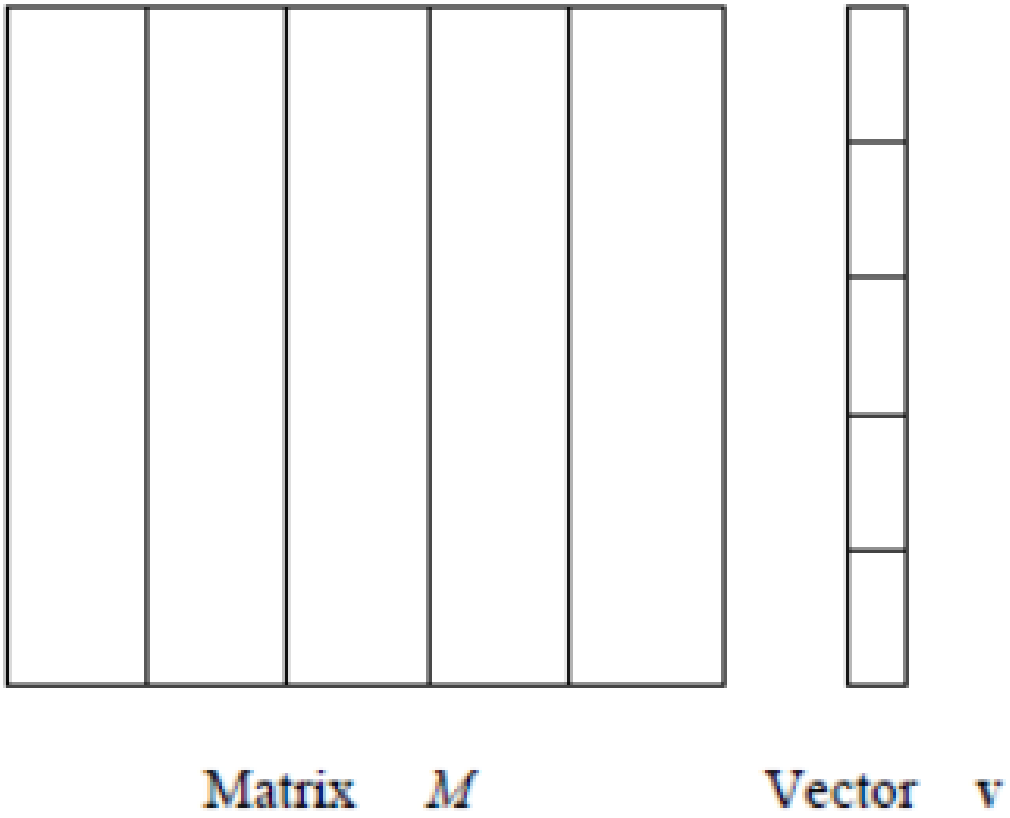


Figure 5: Division of matrix and vector into stripes

2.3 Search Engine working

- When a term is searched, an *inverted index* is looked up - maps words to documents that contain the words. Sort documents based on important which is based on usage of those words. Easy to spam the required word 1000 times to mark it as important.
- Instead of taking just words, see how many pages are pointing to that page. Easy to spam by creating many fake pages that point to my page.
- Page Rank - consider random surfers starting at random pages. What fraction of surfers end up at that page. This is called page rank.
- Take into account words near the link that points to my page - shows how relevant my page is.

2.4 Transition Matrix

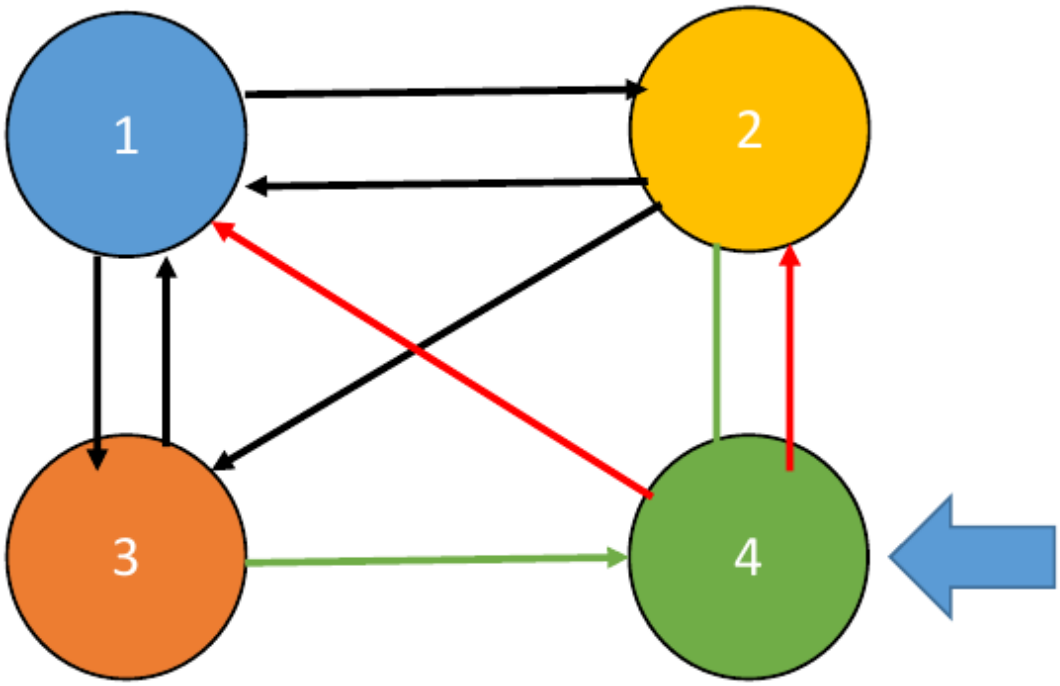


Figure 6: A graph of four inter-linked pages

- Adjacency matrix shows which pages are reachable (element is 1) from which pages.
- Probability of a random surfer at node i going to node j is equal to $1/(number\ of\ outlinks\ at\ node\ i)$
- In Figure 6, probability of transition from node 4 to 1 is $1/2$
- The probability of transitions to every node from a given node can be represented as a column vector. For node 4, the vector will be:

$$\begin{bmatrix} 1/2 \\ 1/2 \\ 0 \\ 0 \end{bmatrix}$$

- For the entire graph, this will be a matrix of size $n \times n$ and is called the **Transition Matrix** M.
- Each entry in a transition matrix represents the probability of transition from source to destination.
- For M_{ij}, source is *Column number* j and destination is *Row number* i
- Fro Figure 6, the transition matrix will be:

$$\begin{bmatrix} 0 & 1/3 & 1/2 & 1/2 \\ 1/2 & 0 & 0 & 1/2 \\ 1/2 & 1/3 & 0 & 0 \\ 0 & 1/3 & 1/2 & 0 \end{bmatrix}$$

2.5 Random Surfer

2.5.1 Initialisation

- The random surfer can start at any node
- The probability of it being at a node is equal for all nodes
- Let vector v represent the relative **importance** of each node.
- Initially v will have all equal elements of value $1/n$ where n is the number of nodes
- This is called v_0
- For graph in Figure 6

$$v_0 = \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}$$

2.5.2 Movement

- Let the random surfer move once
- We need to find the chance of it being at each node, which will be represented by v_1
- For each node i, we multiply the current probability by the transition probability to node j - this gives new probability of it being in node j. The probabilities for node j will be summed up. This is nothing but matrix multiplication.
- Multiply transition matrix M and v

$$\begin{bmatrix} 0 & 1/3 & 1/2 & 1/2 \\ 1/2 & 0 & 0 & 1/2 \\ 1/2 & 1/3 & 0 & 0 \\ 0 & 1/3 & 1/2 & 0 \end{bmatrix} \times \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/4 \\ 5/24 \\ 5/24 \end{bmatrix} = v_1$$

- With each movement, the multiplication is done again.

2.6 Eigenvalues and Eigenvectors

- An $n \times n$ matrix A multiplied by n-element vector v gives another n-element vector.
- When multiplying A by v gives the same vector v with a scale change

$$Av = \lambda v$$

- λ is called the eigenvalue and v is the eigenvector

2.6.1 Page rank as eigenvector problem

- Initialise v to be a vector of equal values
- Do $v_{i+1} = Av_i$ in a loop
- until $v_{i+1} \approx v_i$ i.e., $v = Av$
- Thus, v is an eigenvector and the final page rank

Consider a page with importance I which links to n other pages. It distributes the importance I equally to every page equally i.e., I/n importance added to each linked page.

2.7 Map-Reduce Implementation

- Need support for handling multiple files in mapper - transition matrix M and page rank vector v
- MapReduce tasks have to be iterated over many times

2.7.1 Handling Multiple input files

- Have two separate mapper classes to handle the 2 input files. Mappers output key-value pairs.
- Only one reducer - gets output of both mappers. Reducer uses key to merge values.
- In the job configuration (Java), files are added using `MultipleInput.addInputPath` with the appropriate mapper class passed to it.

2.7.2 Iteration in MapReduce

- The output of one MapReduce task v is stored back into HDFS
- Compare this value of v with previous v using an external script
- Use new v as the input for the next MapReduce task

3 Relational Operations using MapReduce

3.1 Relational operators

Rows are tuples $R(A_1, A_2, ..., A_n)$. Columns are attributes.

- *Selection*: select subset of R according to condition C (row selection)
- *Projection*: select subset of attributes of R (column selection)
- *Union, Intersection, Difference*
- *Natural Join*
- *Grouping*
- *Aggregation*: sum, count, avg, max, min

3.2 Selection

3.2.1 Map

- Read each row t
- Output $\langle t, t \rangle$ if it satisfies condition C

3.2.2 Reduce

Do nothing

3.3 Projection

3.3.1 Map

- Read each row t
- Calculate subset of attributes t'
- Output $\langle t', t' \rangle$

3.3.2 Reduce

Eliminate duplicates.
$\langle t', [t', t', t'] \rangle \rightarrow \langle t', t' \rangle$

3.4 Union

$R \cup S$
R and S have same structure. Need to read 2 input files, 1 output file.

3.4.1 Map

Mapper 1 reads each row t of R and outputs $\langle t, t \rangle$
Mapper 2 reads each row t of S and outputs $\langle t, t \rangle$

3.4.2 Reduce

Eliminate duplicates
$\langle t, [t, t, t] \rangle \rightarrow \langle t, t \rangle$

3.5 Intersection

$R \cap S$

3.5.1 Map

Mapper 1 reads each row t of R and outputs $\langle t, t \rangle$
Mapper 2 reads each row t of S and outputs $\langle t, t \rangle$

3.5.2 Reduce

Output *only duplicates*
$\langle t, [t, t, t] \rangle \rightarrow \langle t, t \rangle$

3.6 Difference

$R - S$
All rows present in R, but not in S

3.6.1 Map

Mapper 1 reads each row t of R and outputs $\langle t, R \rangle$
Mapper 2 reads each row t of S and outputs $\langle t, S \rangle$

3.6.2 Reduce

Output *only*
$\langle t, [R] \rangle \rightarrow \langle t, t \rangle$

3.7 Join

Join R and S on attributes B. A and C are leftover attributes in R and S.

3.7.1 Map

Mapper 1 reads (a, b) of R and outputs $\langle b, (R, a) \rangle$
Mapper 1 reads (b, c) of R and outputs $\langle b, (S, c) \rangle$

3.7.2 Reduce

For each pair $\langle b, (R, a) \rangle$ and $\langle b, (S, c) \rangle$, output $\langle a, b \cdot c \rangle$.

3.8 Grouping and Aggregation

For relation $R(A, B, C)$, group by A and aggregate by a function $f(B)$.

3.8.1 Map

For each line $\langle a, b, c \rangle$, output $\langle a, b \rangle$.

3.8.2 Reduce

Aggregate $\langle a, [b1, b2, b3, \ldots] \rangle$ to $\langle a, f(b1, b2, b3, \ldots) \rangle$.

4 HIVE

SQL for Hadoop

- System for querying and managing structured data, using MapReduce and Hadoop.
- Data is stored in HDFS
- Has a **Meta Store** to store table schemas
- User submitted SQL queries are converted to MR jobs

4.1 Components

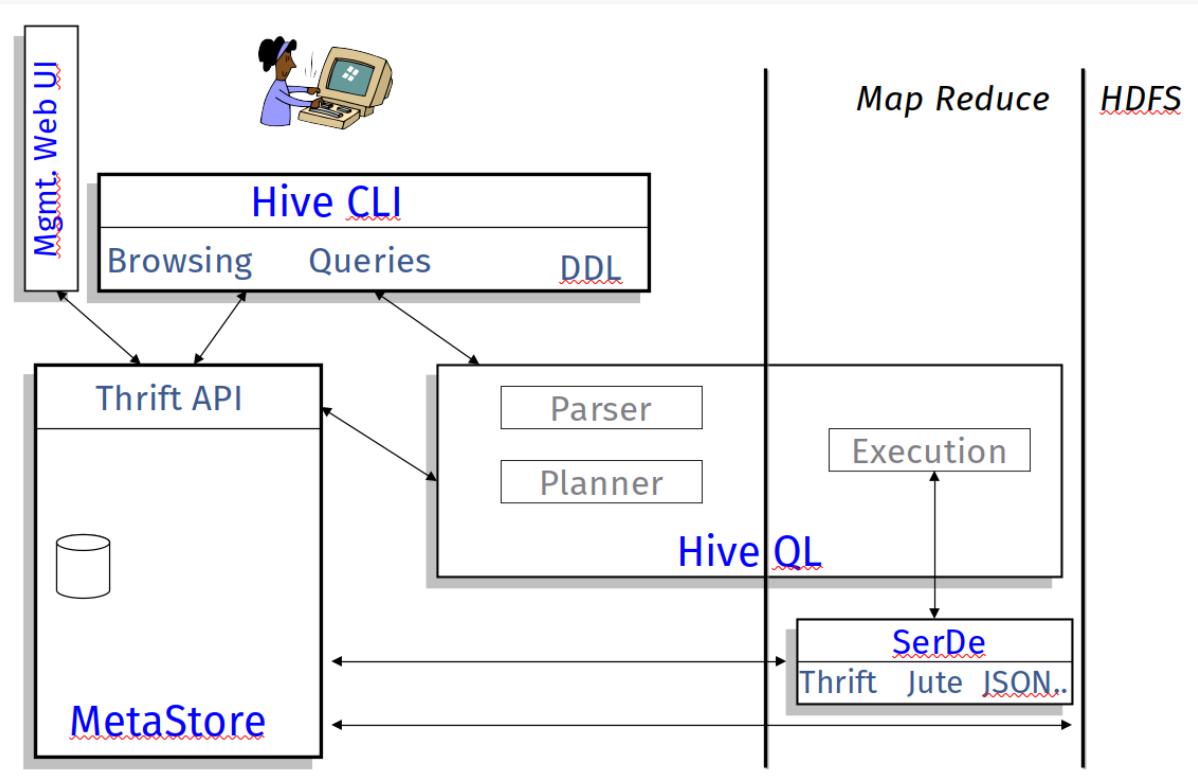


Figure 7: HIVE Architecture

Figure 7

- **External Interfaces**: CLI, Web UI
- **Metastore**: metadata of tables and data
- **SerDe**: serialisation, de-serialisation. Converts raw data from HDFS files to HIVE format during reads and vice versa during writes
- **Compiler**: Converts SQL statements to DAG of map-reduce jobs
- **Execution Engine**: Hadoop

4.2 Data Model

- Data stored as HDFS files
- **Table**: mapped to a HDFS directory. Ex: table *clicks* is available in `/hive/clicks`
- **Partition**: Part of tables are partitioned as subdirectories. The sub-directory name is the column name and value on which it is partitioned. Ex: If *clicks* is partitioned on column `ds` with a particular `ds` value `2008-03-25` will be stored in `/hive/clicks/ds=2008-03-25`. Further partitions will be nested subdirectories.
- **Bucket**: subdivision of partition based on hash of a specified column.

5 Columnar Databases

5.1 Advantages and Limitations of HDFS

Advantages:

- Good for batch processing
- Handles large files efficiently

Limitations:

- Not good for updates
- Not good for looking up specific records
- Not good for incremental addition of small batches

5.2 Advantages and Limitations of HIVE

Advantages:

- Doesn't store data itself - instead uses HDFS/Hbase for storage
- Provides SQL interface for querying

Limitations:

- Data must be structured - definite schema
- Not good for unstructured/semi-structured data
- Other limitations same as HDFS

5.3 Hbase and Cassandra

- Based on the **BigTable** architecture
- Record lookup is fast
- Record level insertion
- Support for updates
- Support for unstructured/semi-structured data