The above figure shows the architecture of Apache Hive and its major components. The major components of Apache Hive are:

1. **Hive Client**
2. **Hive Services**
3. **Processing and Resource Management**
4. **Distributed Storage**

Hive Client

Hive supports applications written in any language like Python, Java, C++, Ruby, etc. using JDBC, ODBC, and Thrift drivers, for performing queries on the Hive. Hence, one can easily write a hive client application in any language of its own choice.

Hive clients are categorized into three types:

1. Thrift Clients

The Hive server is based on Apache Thrift so that it can serve the request from a thrift client.

2. JDBC client

Hive allows for the Java applications to connect to it using the JDBC driver. JDBC driver uses Thrift to communicate with the Hive Server.

3. ODBC client

Hive ODBC driver allows applications based on the ODBC protocol to connect to Hive. Similar to the JDBC driver, the ODBC driver uses Thrift to communicate with the Hive Server.

Skip Ad

Hive Service

To perform all queries, Hive provides various services like the Hive server2, Beeline, etc. The various services offered by Hive are:

1. Beeline

The Beeline is a command shell supported by HiveServer2, where the user can submit its queries and command to the system. It is a **JDBC** client that is based on **SQLLINE CLI** (pure Java-console-based utility for connecting with relational databases and executing SQL queries).

2. Hive Server 2

HiveServer2 is the successor of HiveServer1. HiveServer2 enables clients to execute queries against the Hive. It allows multiple clients to submit requests to Hive and retrieve the final results. It is basically designed to provide the best support for open API clients like JDBC and ODBC.

**Note:** Hive server1, also called a Thrift server, is built on Apache Thrift protocol to handle the cross-platform communication with Hive. It allows different client applications to submit requests to Hive and retrieve the final results.

It does not handle concurrent requests from more than one client due to which it was replaced by HiveServer2.

3. Hive Driver

The Hive driver receives the **HiveQL** statements submitted by the user through the command shell. It creates the session handles for the query and sends the query to the compiler.

4. Hive Compiler

Hive compiler parses the query. It performs semantic analysis and type-checking on the different query blocks and query expressions by using the metadata stored in metastore and generates an execution plan.

The execution plan created by the compiler is the **DAG(Directed Acyclic Graph)**, where each stage is a map/reduce job, operation on HDFS, a metadata operation.

5. Optimizer

Optimizer performs the transformation operations on the execution plan and splits the task to improve efficiency and scalability.

6. Execution Engine

Execution engine, after the compilation and optimization steps, executes the execution plan created by the compiler in order of their dependencies using Hadoop.

7. Metastore

Metastore is a central repository that stores the metadata information about the structure of tables and partitions, including column and column type information.

It also stores information of serializer and deserializer, required for the read/write operation, and HDFS files where data is stored. This metastore is generally a relational database.

Metastore provides a Thrift interface for querying and manipulating Hive metadata.

We can configure metastore in any of the two modes:

* **Remote:** In remote mode, metastore is a Thrift service and is useful for non-Java applications.
* **Embedded:** In embedded mode, the client can directly interact with the metastore using JDBC.

8. HCatalog

HCatalog is the table and storage management layer for Hadoop. It enables users with different data processing tools such as Pig, MapReduce, etc. to easily read and write data on the grid.

It is built on the top of Hive metastore and exposes the tabular data of Hive metastore to other data processing tools.

9. WebHCat

WebHCat is the REST API for HCatalog. It is an HTTP interface to perform Hive metadata operations. It provides a service to the user for running Hadoop MapReduce (or YARN), Pig, Hive jobs.

Processing Framework and Resource Management

Hive internally uses a **MapReduce** framework as a defacto engine for executing the queries.

MapReduce is a software framework for writing those applications that process a massive amount of data in parallel on the large clusters of commodity hardware. MapReduce job works by splitting data into chunks, which are processed by map-reduce tasks.

Distributed Storage

Hive is built on top of Hadoop, so it uses the underlying Hadoop Distributed File System for the distributed storage.

Let us now see how to process data with Apache Hive.

Working of Hive

**Step 1: executeQuery:** The user interface calls the execute interface to the driver.

**Step 2: getPlan:** The driver accepts the query, creates a session handle for the query, and passes the query to the compiler for generating the execution plan.

**Step 3: getMetaData:** The compiler sends the metadata request to the metastore.

**Step 4: sendMetaData:** The metastore sends the metadata to the compiler.

The compiler uses this metadata for performing type-checking and semantic analysis on the expressions in the query tree. The compiler then generates the execution plan (**Directed acyclic Graph**). For Map Reduce jobs, the plan contains **map operator trees** (operator trees which are executed on mapper) and **reduce operator tree** (operator trees which are executed on reducer).

**Step 5: sendPlan:** The compiler then sends the generated execution plan to the driver.

**Step 6: executePlan:** After receiving the execution plan from compiler, driver sends the execution plan to the execution engine for executing the plan.

**Step 7: submit job to MapReduce:**The execution engine then sends these stages of DAG to appropriate components.

For each task, either mapper or reducer, the deserializer associated with a table or intermediate output is used in order to read the rows from HDFS files. These are then passed through the associated operator tree.

Once the output gets generated, it is then written to the HDFS temporary file through the serializer. These temporary HDFS files are then used to provide data to the subsequent map/reduce stages of the plan.

For DML operations, the final temporary file is then moved to the table’s location.

**Step 8,9,10: sendResult:** Now for queries, the execution engine reads the contents of the temporary files directly from HDFS as part of a fetch call from the driver. The driver then sends results to the Hive interface.