Apache Hive

The [**Apache Hive™**](http://hive.apache.org/) data warehouse software facilitates reading, writing, and managing large datasets residing in distributed storage and queried using SQL syntax.

Built on top of Hadoop,

* Tools to easy access data via SQL
* Mechanism to impose structure on variety of formats
* **Extensible**and**scalable**
* Hive is not designed for online transaction processing (OLTP) workloads. It is best used for traditional data warehousing tasks.   
  Hive is designed to maximize scalability (scale out with more machines added dynamically to the Hadoop cluster), performance, extensibility, fault-tolerance, and loose-coupling with its input formats.

Layhmen term : Apache Hive is a data warehouse system built on top of Hadoop and is used for analyzing structured and semi-structured data.

Use:

Apache Hive takes advantage of both the worlds i.e. SQL Database System and [***Hadoop – MapReduce***](https://www.edureka.co/blog/mapreduce-tutorial/)framework. Therefore, it is used by a vast multitude of companies. It is mostly used for data warehousing where you can perform analytics and data mining that does not require real time processing. Some of the fields where you can use Apache Hive are as follows:

* Data Warehousing
* Ad-hoc Analysis

Components of Hive include HCatalog and WebHCat.

* [**HCatalog**](https://cwiki.apache.org/confluence/display/Hive/HCatalog) is a component of Hive. It is a table and storage management layer for Hadoop that enables users with different data processing tools — including Pig and MapReduce — to more easily read and write data on the grid.
* [**WebHCat**](https://cwiki.apache.org/confluence/display/Hive/WebHCat) provides a service that you can use to run Hadoop MapReduce (or YARN), Pig, Hive jobs or perform Hive metadata operations using an HTTP (REST style) interface.

Hive Architecture:



* UI – The user interface for users to submit queries and other operations to the system. As of 2011 the system had a command line interface and a web based GUI was being developed.
* Driver – The component which receives the queries. This component implements the notion of session handles and provides execute and fetch APIs modeled on JDBC/ODBC interfaces.
* Compiler – The component that parses the query, does semantic analysis on the different query blocks and query expressions and eventually generates an execution plan with the help of the table and partition metadata looked up from the metastore.
* Metastore – The component that stores all the structure information of the various tables and partitions in the warehouse including column and column type information, the serializers and deserializers necessary to read and write data and the corresponding HDFS files where the data is stored.
* Execution Engine – The component which executes the execution plan created by the compiler. The plan is a DAG of stages. The execution engine manages the dependencies between these different stages of the plan and executes these stages on the appropriate system components.

Steps:

The UI calls the execute interface to the Driver .

The Driver creates a session handle for the query and sends the query to the compiler to generate an execution plan (step 2).

The compiler gets the necessary metadata from the metastore (steps 3 and 4). This metadata is used to typecheck the expressions in the query tree as well as to prune partitions based on query predicates.

The plan generated by the compiler (step 5) is a DAG of stages with each stage being either a map/reduce job, a metadata operation or an operation on HDFS. For map/reduce stages, the plan contains map operator trees (operator trees that are executed on the mappers) and a reduce operator tree (for operations that need reducers).

The execution engine submits these stages to appropriate components (steps 6, 6.1, 6.2 and 6.3). In each task (mapper/reducer) the deserializer associated with the table or intermediate outputs is used to read the rows from HDFS files and these are passed through the associated operator tree. Once the output is generated, it is written to a temporary HDFS file though the serializer (this happens in the mapper in case the operation does not need a reduce). The temporary files are used to provide data to subsequent map/reduce stages of the plan. For DML operations the final temporary file is moved to the table's location. This scheme is used to ensure that dirty data is not read (file rename being an atomic operation in HDFS).

For queries, the contents of the temporary file are read by the execution engine directly from HDFS as part of the fetch call from the Driver (steps 7, 8 and 9).

### Metastore Architecture

Metastore is an object store with a database or file backed store. The database backed store is implemented using an object-relational mapping (ORM) solution called the [DataNucleus](http://www.datanucleus.org/). The prime motivation for storing this in a relational database is queriability of metadata. Some disadvantages of using a separate data store for metadata instead of using HDFS are synchronization and scalability issues. Additionally there is no clear way to implement an object store on top of HDFS due to lack of random updates to files. This, coupled with the advantages of queriability of a relational store, made our approach a sensible one.

The metastore can be configured to be used in a couple of ways: remote and embedded. In remote mode, the metastore is a [Thrift](https://thrift.apache.org/) service. This mode is useful for non-Java clients. In embedded mode, the Hive client directly connects to an underlying metastore using JDBC. This mode is useful because it avoids another system that needs to be maintained and monitored. Both of these modes can co-exist. (Update: Local metastore is a third possibility. See [Hive Metastore Administration](https://cwiki.apache.org/confluence/display/Hive/AdminManual+MetastoreAdmin) for details.)

### Metastore Interface

Metastore provides a [Thrift interface](https://thrift.apache.org/docs/idl) to manipulate and query Hive metadata. Thrift provides bindings in many popular languages. Third party tools can use this interface to integrate Hive metadata into other business metadata repositories.

Hive Metastore: The Metastore provides two important but often overlooked features of a data warehouse: data abstraction and data discovery.

Without the data abstractions provided in Hive, a user has to provide information about data formats, extractors and loaders along with the query.

The second functionality, data discovery, enables users to discover and explore relevant and specific data in the warehouse. Other tools can be built using this metadata to expose and possibly enhance the information about the data and its availability.

**Metadata Objects**

* Database – is a namespace for tables. It can be used as an administrative unit in the future. The database 'default' is used for tables with no user-supplied database name.
* Table – Metadata for a table contains list of columns, owner, storage and SerDe information. It can also contain any user-supplied key and value data. Storage information includes location of the underlying data, file inout and output formats and bucketing information. SerDe metadata includes the implementation class of serializer and deserializer and any supporting information required by the implementation. All of this information can be provided during creation of the table.
* Partition – Each partition can have its own columns and SerDe and storage information. This facilitates schema changes without affecting older partitions.

## Compiler

* Parser – Transform a query string to a parse tree representation.
* Semantic Analyser – Transform the parse tree to an internal query representation, which is still block based and not an operator tree. As part of this step, the column names are verified and expansions like \* are performed. Type-checking and any implicit type conversions are also performed at this stage. If the table under consideration is a partitioned table, which is the common scenario, all the expressions for that table are collected so that they can be later used to prune the partitions which are not needed. If the query has specified sampling, that is also collected to be used later on.
* Logical Plan Generator – Convert the internal query representation to a logical plan, which consists of a tree of operators. Some of the operators are relational algebra operators like 'filter', 'join' etc. But some of the operators are Hive specific and are used later on to convert this plan into a series of map-reduce jobs. One such operator is a reduceSink operator which occurs at the map-reduce boundary. This step also includes the optimizer to transform the plan to improve performance – some of those transformations include: converting a series of joins into a single multi-way join, performing a map-side partial aggregation for a group-by, performing a group-by in 2 stages to avoid the scenario when a single reducer can become a bottleneck in presence of skewed data for the grouping key. Each operator comprises a descriptor which is a serializable object.
* Query Plan Generator – Convert the logical plan to a series of map-reduce tasks. The operator tree is recursively traversed, to be broken up into a series of map-reduce serializable tasks which can be submitted later on to the map-reduce framework for the Hadoop distributed file system. The reduceSink operator is the map-reduce boundary, whose descriptor contains the reduction keys. The reduction keys in the reduceSink descriptor are used as the reduction keys in the map-reduce boundary. The plan consists of the required samples/partitions if the query specified so. The plan is serialized and written to a file.

## Optimizer

More plan transformations are performed by the optimizer. The optimizer is an evolving component. As of 2011, it was rule-based and performed the following: column pruning and predicate pushdown. However, the infrastructure was in place, and there was work under progress to include other optimizations like map-side join. (Hive 0.11 added several [join optimizations](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+JoinOptimization).)  
  
The optimizer can be enhanced to be cost-based (see [Cost-based optimization in Hive](https://cwiki.apache.org/confluence/display/Hive/Cost-based+optimization+in+Hive) and [HIVE-5775](https://issues.apache.org/jira/browse/HIVE-5775)). The sorted nature of output tables can also be preserved and used later on to generate better plans. The query can be performed on a small sample of data to guess the data distribution, which can be used to generate a better plan.

## ****Metastore Configuration****

Metastore stores the meta data information using RDBMS and an open source ORM (Object Relational Model) layer called Data Nucleus which converts the object representation into relational schema and vice versa. The reason for choosing RDBMS instead of HDFS is to achieve low latency. We can implement metastore in following three configurations:

### ****1. Embedded Metastore:****



## Data Units

In the order of granularity - Hive data is organized into:

* **Databases**: Namespaces function to avoid naming conflicts for tables, views, partitions, columns, and so on.  Databases can also be used to enforce security for a user or group of users.
* **Tables**: Homogeneous units of data which have the same schema. An example of a table could be page\_views table, where each row could comprise of the following columns (schema):
  + timestamp—which is of INT type that corresponds to a UNIX timestamp of when the page was viewed.
  + userid —which is of BIGINT type that identifies the user who viewed the page.
  + page\_url—which is of STRING type that captures the location of the page.
  + referer\_url—which is of STRING that captures the location of the page from where the user arrived at the current page.
  + IP—which is of STRING type that captures the IP address from where the page request was made.
* **Partitions**: Each Table can have one or more partition Keys which determines how the data is stored. Partitions—apart from being storage units—also allow the user to efficiently identify the rows that satisfy a specified criteria; for example, a date\_partition of type STRING and country\_partition of type STRING. Each unique value of the partition keys defines a partition of the Table. For example, all "US" data from "2009-12-23" is a partition of the page\_views table. Therefore, if you run analysis on only the "US" data for 2009-12-23, you can run that query only on the relevant partition of the table, thereby speeding up the analysis significantly. Note however, that just because a partition is named 2009-12-23 does not mean that it contains all or only data from that date; partitions are named after dates for convenience; it is the user's job to guarantee the relationship between partition name and data content! Partition columns are virtual columns, they are not part of the data itself but are derived on load.
* **Buckets** (or **Clusters**): Data in each partition may in turn be divided into Buckets based on the value of a hash function of some column of the Table. For example the page\_views table may be bucketed by userid, which is one of the columns, other than the partitions columns, of the page\_view table. These can be used to efficiently sample the data.

Note that it is not necessary for tables to be partitioned or bucketed, but these abstractions allow the system to prune large quantities of data during query processing, resulting in faster query execution.