BIG DATA

Big Data - Assignment 4

1) How can we set the block size for HBASE?

Using HBase blocksize

You must configure the HBase blocksize to set the smallest unit of data HBase can read from the column family's HFiles.

HBase data is stored in one (after a major compaction) or more (possibly before a major compaction) HFiles per column family per region. The blocksize determines:

- The blocksize for a given column family determines the smallest unit of data HBase can read from the column family's HFiles.
- The basic unit of measure cached by a RegionServer in the BlockCache.

The default blocksize is 64 KB. The appropriate blocksize is dependent upon your data and usage patterns. Use the following guidelines to tune the blocksize size, in combination with testing and benchmarking as appropriate.

Consider the average key/value size for the column family when tuning the blocksize. You can find the average key/value size using the HFile utility:

```
$ hbase org.apache.hadoop.hbase.io.hfile.HFile -f /path/to/HFILE -m -v

...

Block index size as per heapsize: 296

reader=hdfs://srv1.example.com:9000/path/to/HFILE, \

compression=none, inMemory=false, \

firstKey=US6683275_20040127/mimetype:/1251853756871/Put, \

lastKey=US6684814_20040203/mimetype:/1251864683374/Put, \

avgKeyLen=37, avgValueLen=8, \

entries=1554, length=84447

...
```

- Consider the pattern of reads to the table or column family. For instance, if it is common to scan for 500 rows on various parts of the table, performance might be increased if the blocksize is large enough to encompass 500-1000 rows, so that often, only one read operation on the HFile is required. If your typical scan size is only 3 rows, returning 500-1000 rows would be overkill.

  It is difficult to predict the size of a row before it is written, because the data will be compressed when it is written to the HFile. Perform testing to determine the correct blocksize for your data.

2) What is Zookeeper's purpose?

ZooKeeper is an open source Apache project that provides **a centralized service for providing configuration information, naming, synchronization and group services over large clusters in distributed systems**. The goal is to make these systems easier to manage with improved, more reliable propagation of changes.

3) Explain the architecture of Sqoop in detail.

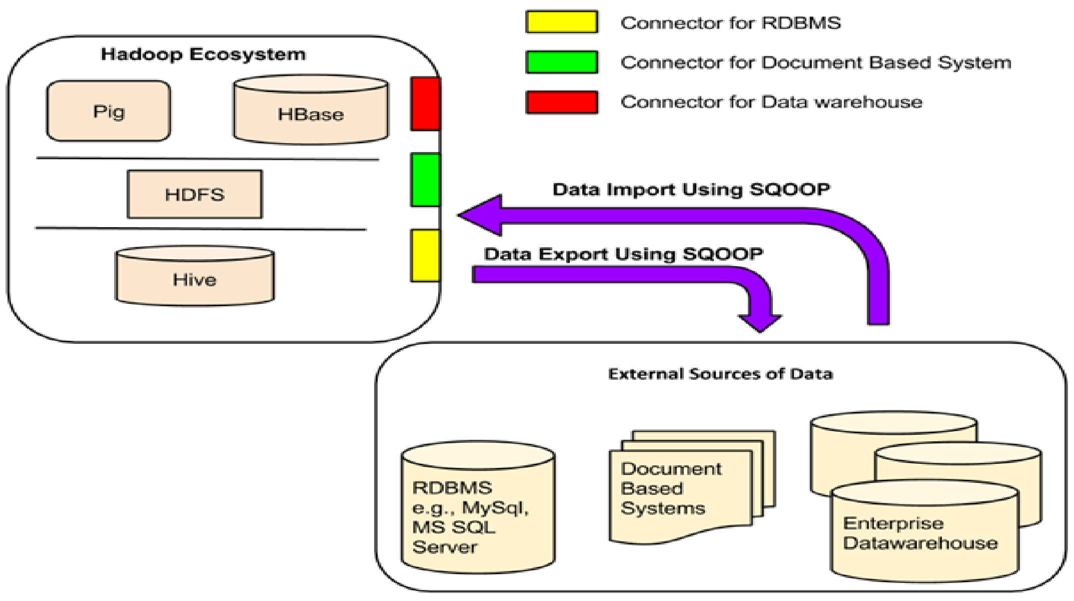
Apache SQOOP (SQL-to-Hadoop) is a tool designed to support bulk export and import of data into HDFS from structured data stores such as relational databases, enterprise data warehouses, and NoSQL systems. It is a data migration tool based upon a connector architecture which supports plugins to provide connectivity to new external systems.

Sqoop Architecture

All the existing **Database Management Systems** are designed with SQL standard in mind. However, each DBMS differs with respect to dialect to some extent. So, this difference poses challenges when it comes to data transfers across the systems. Sqoop Connectors are components which help overcome these challenges.

Data transfer between Sqoop Hadoop and external storage system is made possible with the help of Sqoop's connectors.

Sqoop has connectors for working with a range of popular relational databases, including MySQL, PostgreSQL, Oracle, SQL Server, and DB2. Each of these connectors knows how to interact with its associated DBMS. There is also a generic JDBC connector for connecting to any database that supports Java's JDBC protocol. In addition, Sqoop Big data provides optimized MySQL and PostgreSQL connectors that use database-specific APIs to perform bulk transfers efficiently.



Sqoop Architecture

In addition to this, Sqoop in big data has various third-party connectors for data stores, ranging from enterprise data warehouses (including Netezza, Teradata, and Oracle) to NoSQL stores (such as Couchbase). However, these connectors do not come with Sqoop bundle; those need to be downloaded separately and can be added easily to an existing Sqoop installation.

4) In Hadoop, write the difference between Sqoop and HDFS.

SQOOP –

- Sqoop is a command-line interface application for transferring data between relational databases and Hadoop. The Apache Sqoop project was retired in June 2021 and moved to the Apache Attic.
- Sqoop is used **to transfer data from RDBMS (relational database management system)** like MySQL and Oracle to HDFS (Hadoop Distributed File System). Big Data Sqoop can also be used to transform data in Hadoop MapReduce and then export it into RDBMS.
- Apache Sqoop uses the YARN framework for importing and exporting the data.
- Apache Sqoop is highly robust in nature.

HDFS –

- The **Hadoop Distributed File System** (HDFS) is the primary data storage system used by Hadoop applications. HDFS employs a NameNode and DataNode architecture to implement a distributed file system that provides high-performance access to data across highly scalable Hadoop clusters.
- The way HDFS works is by **having a main « NameNode » and multiple « data nodes » on a commodity hardware cluster**. ... Data is then broken down into separate « blocks » that are distributed among the various data nodes for storage. Blocks are also replicated across nodes to reduce the likelihood of failure.
- HDFS comprises of 3 important components-**NameNode, DataNode and Secondary NameNode**
- Hadoop HDFS is a highly available file system.

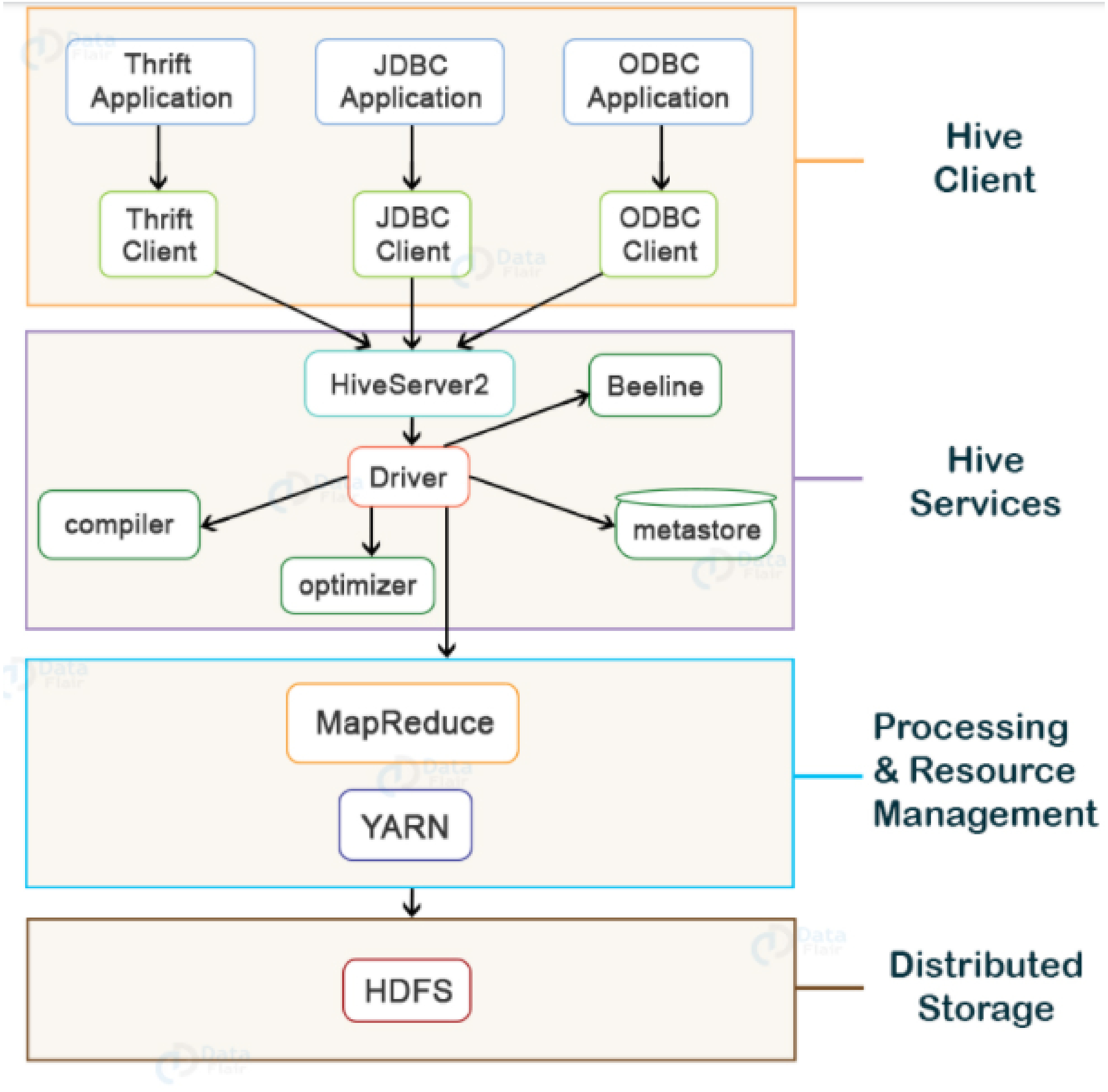
5) Describe the Hive architecture's essential components.

Apache Hive is an **open-source data warehousing tool** for performing distributed processing and data analysis. It was developed by **Facebook** to reduce the work of writing the Java MapReduce program.

Apache Hive uses a **Hive Query language**, which is a declarative language similar to SQL. Hive translates the hive queries into MapReduce programs.
It supports developers to perform processing and analyses on structured and semi-structured data by replacing complex java MapReduce programs with hive queries.
One who is familiar with SQL commands can easily write the hive queries.

Hive makes the job easy for performing operations like

- Analysis of huge datasets
- Ad-hoc queries
- Data encapsulation



Hive Architecture & Its Components

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**

[ps2id id='Hive-Client' target=''/]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.

[ps2id id='Hive-Services' target=''/]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

2. Hive Server 2

3. Hive Driver

4. Hive Compiler

5. Optimizer

6. Execution Engine

7. Metastore

8. HCatalog

9. WebHCat

- **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.