Introduction, MapReduce Map Tasks, Reduce Tasks and MapReduce Execution, Composing MapReduce for Calculations and Algorithms, Hive, HiveQL, Pig.

Module-4

Q. 07	a	Explain Map Reduce Execution steps with neat diagram.
	b	What is HIVE? Explain HIVE Architecture.
OR		
Q. 08	a	Explain Pig architecture for scripts dataflow and processing
	b	Explain Key Value pairing in Map Reduce.

1)With Neat diagram Explain the process in MapReduce when client submitting job?

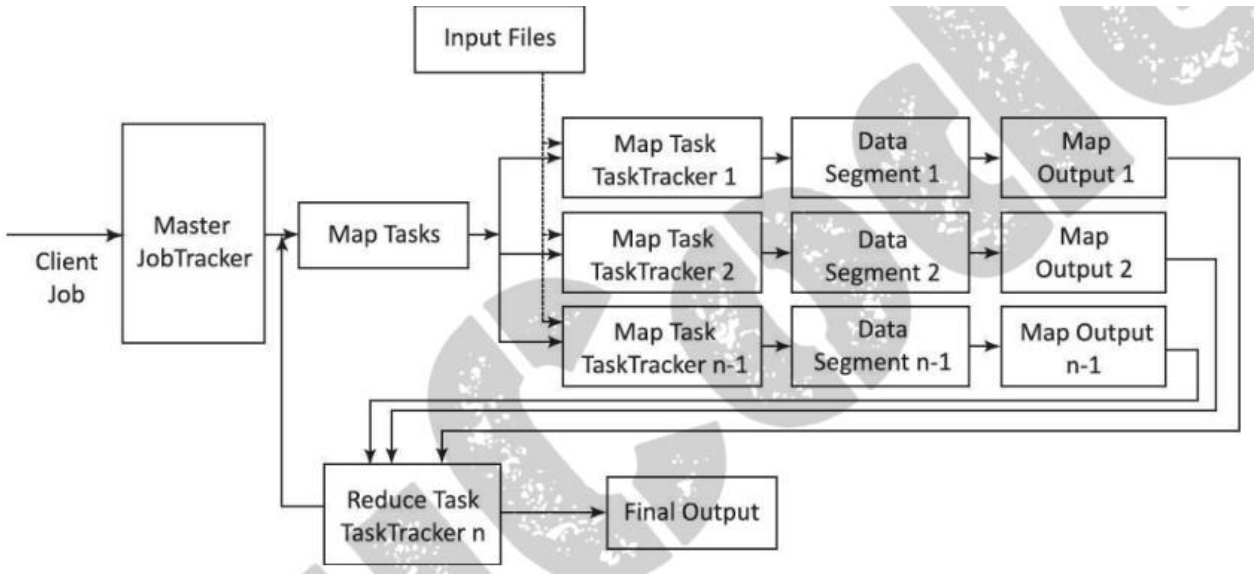

Figure: MapReduce process on client submitting a job

Steps in the MapReduce Process

1. Job Submission

- Client Interaction:
    - A client submits a MapReduce job to the JobTracker, which acts as the central controller in the cluster.

- The input data for the job is typically stored in HDFS.

- Job Specification:
  - The job submission includes configuration information, such as:
    - Input and output paths in HDFS.
    - Mapper and Reducer code.
    - Job-specific parameters.

2. JobTracker Responsibilities

- Job Scheduling:
  - The JobTracker divides the job into smaller sub-tasks (Map and Reduce tasks).
  - It identifies nodes in the cluster to execute these tasks based on data locality, reducing network overhead.
- Task Assignment:
  - Assigns Map tasks to TaskTrackers near the data and Reduce tasks to other nodes.
- Monitoring:
  - Continuously monitors TaskTrackers for task progress and failure.
  - In case of failure, reassigns tasks to other TaskTrackers.

3. TaskTracker Role

- TaskTrackers execute tasks as instructed by the JobTracker.
- Each TaskTracker performs the following:
  - Map Tasks:
    - Read input data blocks from HDFS.
    - Process each block using the Mapper code to generate intermediate key-value pairs.
  - Reduce Tasks:
    - Shuffle and sort the intermediate data from the Map phase.
    - Apply the Reducer logic to aggregate or process data and generate final results.
- Regularly send heartbeats to the JobTracker to report task progress.

4. Execution Phases

- Map Phase:
    - Splits the input data into smaller chunks (InputSplits).
    - Processes each split independently using the Mapper to produce intermediate key-value pairs.
- Shuffle and Sort Phase:
    - Groups intermediate key-value pairs by keys across all mappers.
    - Sorts the data to prepare it for Reducer input.
- Reduce Phase:
    - Receives grouped data.
    - Processes it using the Reducer function to generate the final output.
    - Writes the final output to HDFS.

5. Final Output

- The Reducer stores the processed output into HDFS at the specified output path.
- The client retrieves the results from HDFS.

The MapReduce process can be represented as follows:

1. Client Submits Job:
    - The client communicates with the JobTracker.
2. JobTracker Actions:
    - Splits job into Map and Reduce tasks.
    - Assigns tasks to TaskTrackers.
3. Task Execution:
    - TaskTrackers execute Map tasks on data blocks and send intermediate results.
    - Reduce tasks process these intermediate results.
4. Output:
    - Reducers write the final output to HDFS.

2)Describe MapReduce Execution Steps with Neat Sketch or Describe Map tasks, Reduce tasks and MapReduce execution process

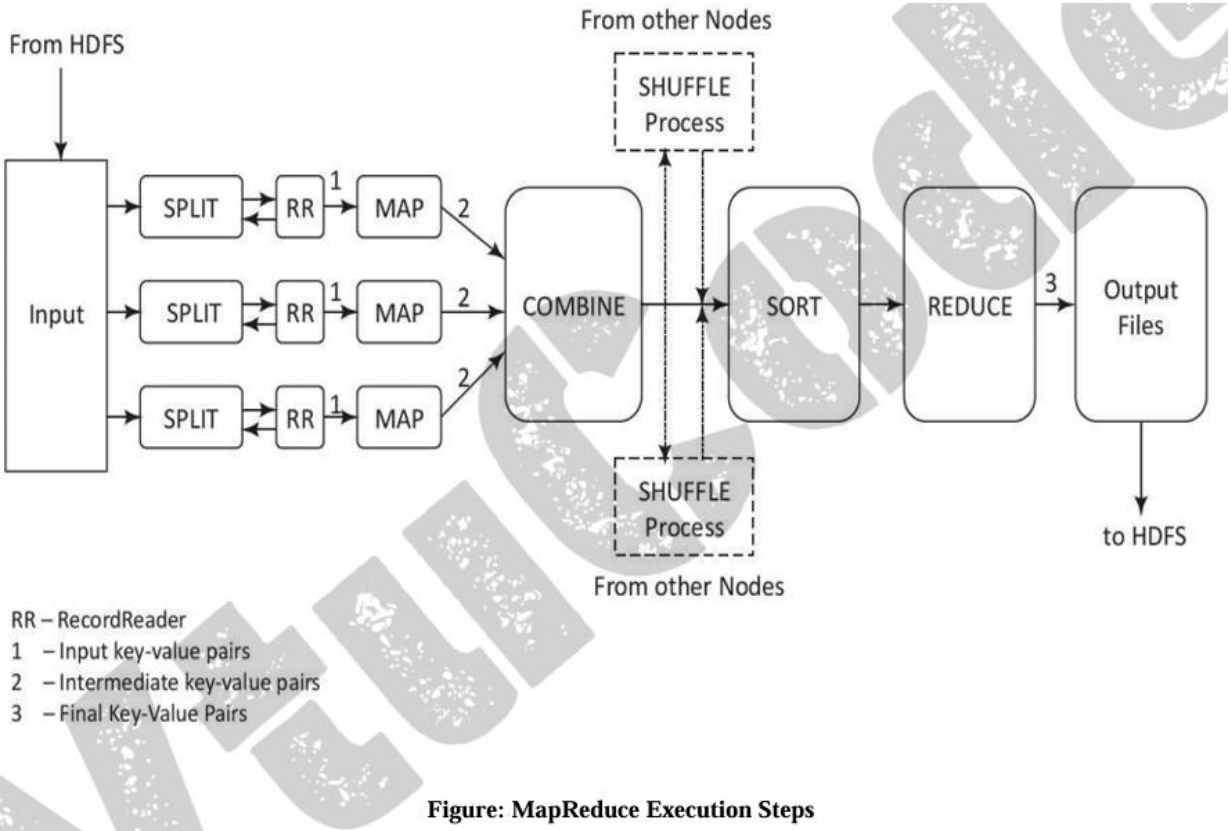

Figure: MapReduce Execution Steps

1. Input Phase (Data Splitting and Reading)

- The input data resides in HDFS (Hadoop Distributed File System).
- The input file is split into smaller chunks (Splits) for parallel processing.
- Each split is assigned to a Map Task.
- A RecordReader (RR) reads the data in each split and converts it into key-value pairs.
    - For example: If the input is a text file, the RecordReader might output line number as key and line content as value.
- The splits and key-value pairs are fed into the Map function for further processing.

---

2. Map Phase (Processing Input)

- Each Mapper processes the key-value pairs generated by the RecordReader.
- The Map function processes each input pair to generate intermediate key-value pairs.

- Multiple Map tasks run in parallel across different nodes, enabling distributed processing.
- The output of the Map function is:
    - Intermediate Key-Value Pairs (e.g., <key, list of values>).
- The intermediate data is stored in a local disk of the node temporarily.

---

3. Combine Phase (Local Aggregation)

- Before transferring data over the network, a Combiner (if defined) can aggregate the output locally on each Mapper node.
- A Combiner works like a mini-Reducer to reduce the size of intermediate data.
    - For example: Summing up counts or merging values locally to minimize network load.
- This step is optional but improves efficiency in scenarios with heavy intermediate output.

---

4. Shuffle Phase (Data Transfer and Grouping)

- The Shuffle phase begins as soon as the Map tasks are complete.
- The intermediate key-value pairs are grouped by their keys:
    - All values associated with the same key are brought together across different Mapper outputs.
- The Shuffle Process transfers data from Mapper nodes to the appropriate Reducer nodes:
    - If the same key appears on different nodes, it is moved and combined at the destination.
- This step ensures that all values for a specific key are available to the corresponding Reducer.

---

5. Sort Phase (Sorting by Key)

- Before data reaches the Reducer, the intermediate key-value pairs are sorted by key.
- Sorting ensures that the Reducer processes data in a predictable order.
- The framework automatically handles sorting during the Shuffle process.

---

6. Reduce Phase (Final Aggregation/Processing)

- The Reduce function processes the sorted intermediate key-value pairs received from the Shuffle phase.
- For each unique key, the Reducer applies a function to process or aggregate all associated values.
    - For example: Summing up counts, concatenating lists, or performing custom operations.
- The Reducer produces final output key-value pairs.

---

7. Output Phase (Writing Results to HDFS)

- The final output generated by the Reducer is written to HDFS.
- The output is stored as output files in a specified format (e.g., text, CSV, etc.).
- Each Reducer produces a separate output file, and these files together form the final result.

3)With Neat diagram describe Hive integration and workflow steps

Hive Integration and Workflow Steps:

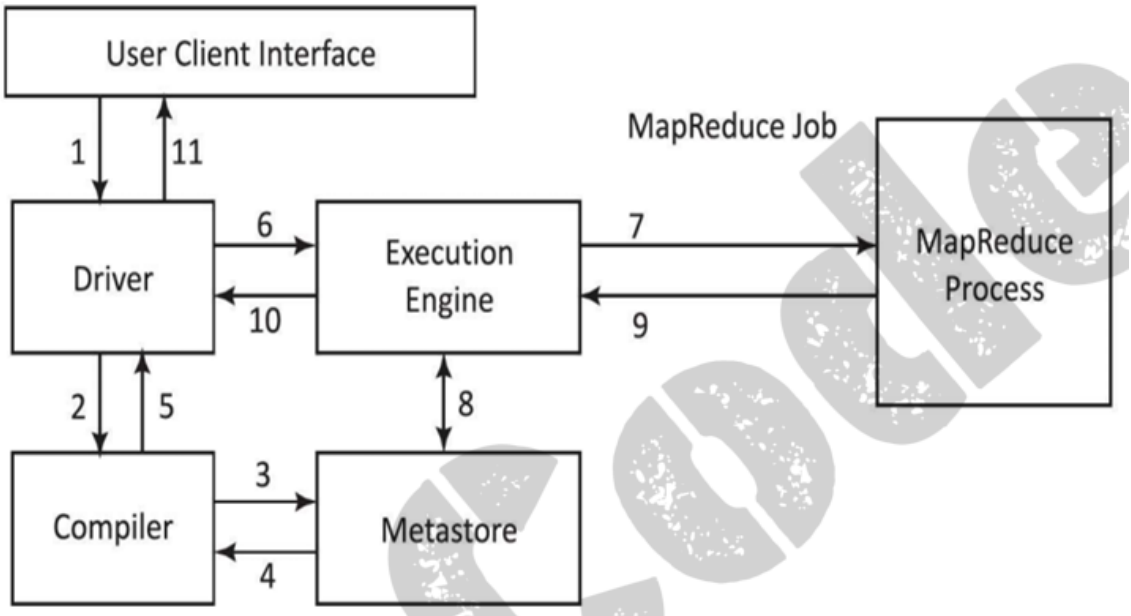

Figure: Dataflow sequences and workflow steps

1, **Execute Query**:

The user submits a query to Hive through the **User Client Interface** (such as Hive CLI or Web UI). This query is passed to the **Driver** to initiate execution.

2. **Get Plan**:
The Driver forwards the query to the **Compiler**. The Compiler parses the query to validate its syntax and generates a query plan based on the requirements.

3. **Get Metadata**:
The Compiler interacts with the **Metastore** to fetch metadata. The Metastore contains information about the tables, schemas, and data required for query execution.

4. **Send Metadata**:
The **Metastore** sends back the metadata to the Compiler, providing details such as column names, data types, and table structures.

5. **Send Plan**:
The Compiler verifies the requirements and sends the execution plan back to the Driver. At this stage, query parsing and compilation are complete.

6. **Execute Plan**:
The Driver forwards the execution plan to the **Execution Engine** to process the query.

7. **Execute Job**:
Internally, the **Execution Engine** converts the query into a **MapReduce job**. The job is submitted to the **JobTracker** in the NameNode, which assigns it to **TaskTrackers** in the DataNodes. The TaskTrackers perform the actual query execution on the data.

8. **Metadata Operations**:
During execution, the Execution Engine can interact with the **Metastore** to perform any metadata operations (e.g., retrieving additional schema information).

9. **Fetch Results**:
Once the MapReduce job is complete, the Execution Engine collects the query execution results from the DataNodes.

10. **Send Results to Driver**:
The Execution Engine sends the results back to the Driver.

11. **Send Results to User**:
Finally, the Driver forwards the query results to the Hive Interface, and the user receives the output.

4) How node failure can be handled in Hadoop?

1. Regular Heartbeats

- Mechanism:
    - Each node (TaskTracker or DataNode) sends periodic heartbeats to the master node (JobTracker or NameNode) to confirm its health and availability.
- Action on Missing Heartbeats:
    - If a node fails to send heartbeats within a predefined time (e.g., 10 minutes), it is marked as failed.

2. TaskTracker Failure

- Impact:
    - All map or reduce tasks running on the failed TaskTracker are affected.
- Recovery:
    - The JobTracker assigns the tasks of the failed TaskTracker to other available TaskTrackers.
    - For completed map tasks, the associated intermediate data is re-generated if not already consumed by reducers.

3. DataNode Failure

- Impact:
    - Data blocks stored on the failed DataNode become inaccessible.
- Recovery:
    - Hadoop's HDFS replication mechanism ensures that each block of data is replicated across multiple DataNodes (default replication factor is 3).
    - If a DataNode fails, the NameNode detects the unavailability of the replicas.
    - The NameNode instructs other DataNodes with existing replicas to create additional copies to maintain the desired replication level.

4. JobTracker Failure

- Impact:
    - The JobTracker being a single point of failure can halt the entire MapReduce process.
- Recovery:
    - Typically, the entire job needs to be restarted as there is no automatic recovery mechanism for the JobTracker in older versions of Hadoop.
    - Later versions of Hadoop (e.g., YARN) address this limitation by separating resource management and job scheduling, providing better fault tolerance.

5. Reduce TaskTracker Failure

- Impact:
    - Only reduce tasks in progress are affected.
- Recovery:
    - The JobTracker reassigns the failed reduce tasks to other TaskTrackers.
    - Completed reduce tasks are not impacted.

6. HDFS Read/Write Failures

- Impact:
    - A client attempting to read or write to a failed DataNode may experience delays.
- Recovery:
    - For reads: The client retries the operation with another DataNode that has a replica of the required block.
    - For writes: The client re-attempts the operation on another DataNode.

7. Automatic Restart Mechanism

- Task Rescheduling:
    - Hadoop automatically restarts failed tasks on different nodes without manual intervention.
- Redundant Execution:
    - Speculative execution can run duplicate instances of slow tasks to mitigate delays caused by slow nodes.

5)with example explain working of MapReduce Programming model?

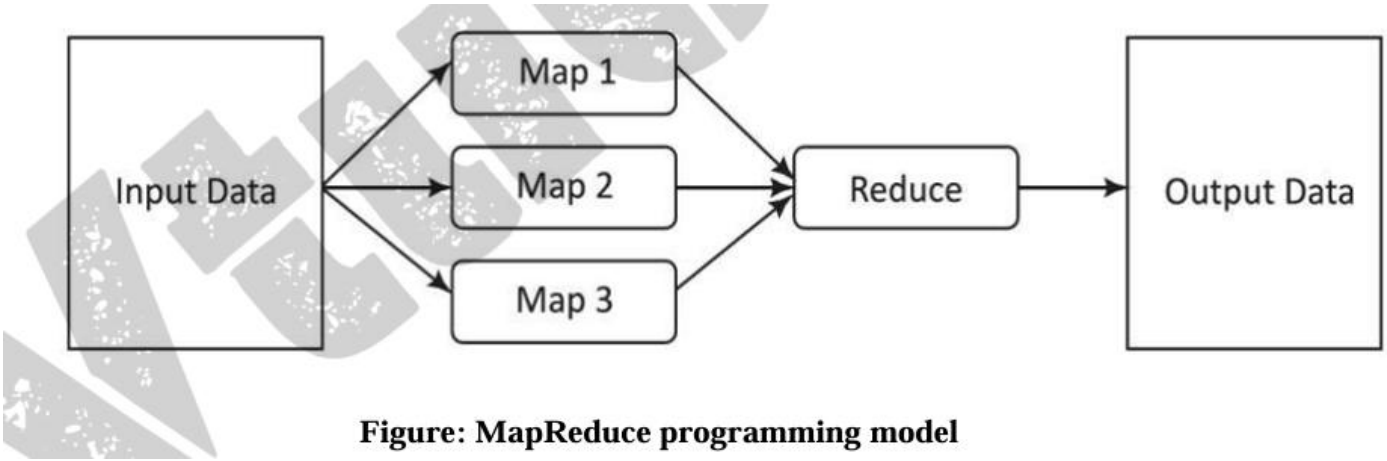

Figure: MapReduce programming model

The MapReduce programming model is designed for processing large datasets across a distributed system, such as a Hadoop cluster. It ensures efficient computation by running tasks in parallel while minimizing data movement across the network. Below is a detailed explanation:

Key Components

1. Parallel Execution:
    - The MapReduce framework divides the input dataset into smaller parts and processes them in parallel across different nodes.
2. Hadoop and Java:
    - Hadoop implements MapReduce using Java, providing APIs for developers to write map() and reduce() functions.
3. Two Core Tasks:
    - Map Task:
        - Splits the input data into smaller key-value pairs.
        - Each Mapper operates independently on its data partition.
    - Reduce Task:
        - Processes intermediate key-value pairs generated by the Mappers.
        - Aggregates and combines data into a final output.

Execution Flow

1. Input Data:
    - Input is stored in HDFS and split into smaller logical pieces called InputSplits.
2. Mapping:
    - The Map task processes each InputSplit and generates intermediate key-value pairs.
    - For example, in a word count program:
        - Input: *"hello world"*
        - Output: (hello, 1), (world, 1)
3. Shuffle and Sort:
    - The intermediate key-value pairs are grouped and sorted by keys.
    - Data is then transferred to the Reducer nodes.
4. Reducing:
    - Reducers process the grouped data to generate the final output.
    - Example: (hello, [1, 1]) → (hello, 2)
5. Output Data:
    - The final results are written back to HDFS.

Advantages of MapReduce

1. Data Locality:
    - The framework schedules tasks on the nodes where the data is stored, reducing data movement and improving efficiency.
2. Scalability:
    - Handles large datasets by distributing tasks across a cluster of nodes.
3. Fault Tolerance:
    - Automatically detects and reassigns failed tasks to other nodes.

4. Efficiency:
    - Minimizes network traffic by processing data locally on storage nodes

6)Describe the Hive Architecture Components along with Hive built in Functions? Or Explain Hive Architecture and its Characteristics

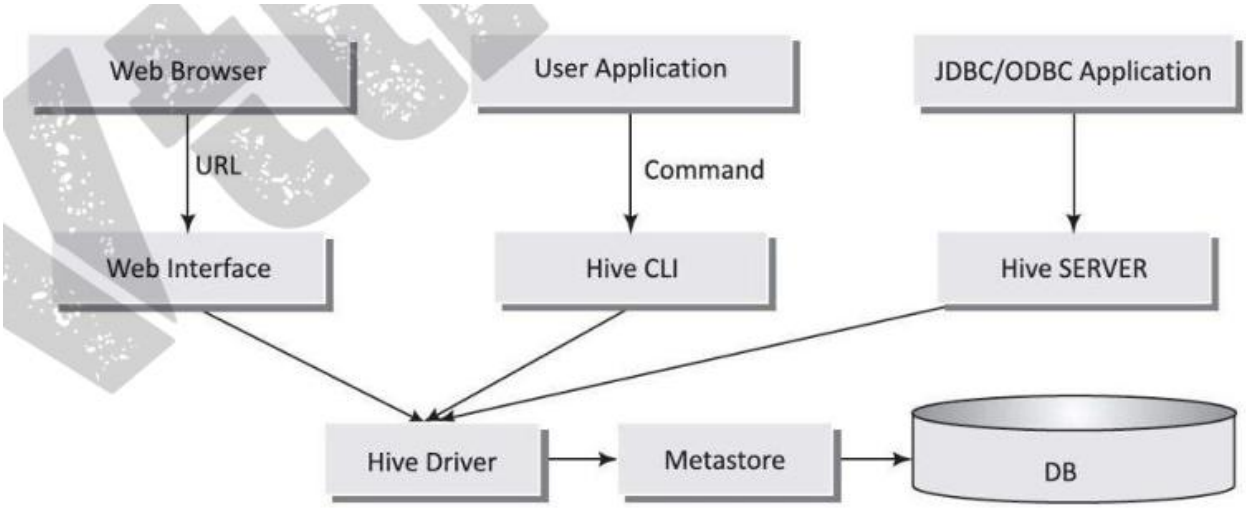

Figure: Hive Architecture

1. User Interfaces (UI)

Hive provides several interfaces for users to interact with it, including the Command Line Interface (CLI), a web interface, and ODBC/JDBC drivers. These interfaces allow users to run queries, interact with data, and integrate with external applications.

2. Driver

The Hive Driver is responsible for managing the lifecycle of Hive queries. It handles parsing, compiling, and optimizing HiveQL statements. It generates execution plans and coordinates the entire process, including session management and executing the query.

3. Compiler

The Compiler is responsible for converting HiveQL queries into a logical plan. It parses the query, optimizes it for performance, and generates tasks that are executable by the Hadoop cluster. It applies optimizations such as predicate pushdown, partition pruning, and reordering of joins to improve the efficiency of query execution.

4. Metastore

The Metastore is a central repository for storing metadata about the Hive database, including information about databases, tables, columns, partitions, and their mappings to HDFS files. It uses an RDBMS (e.g., MySQL) to store this metadata and provides it to other components of Hive when needed.

5. Execution Engine

The Execution Engine is responsible for executing the tasks generated by the Compiler. It interacts with Hadoop's JobTracker (or YARN in newer versions) to submit MapReduce or Spark jobs. It manages task execution, monitors progress, and handles error recovery by rescheduling failed tasks.

6. Storage

Hive stores data in Hadoop's HDFS (Hadoop Distributed File System), where data is organized in various file formats, including TextFile, SequenceFile, ORC, Avro, and Parquet. It supports partitioning and bucketing to improve the management and retrieval of large datasets, helping optimize performance by organizing data into smaller, more manageable pieces.

2. Built-in Functions in Hive

Hive provides several built-in functions for data transformation and analysis, categorized as follows:

2.1 String Functions

- **Examples**:
    - concat(string1, string2, ...) - Concatenates strings.
    - substr(string, start, length) - Extracts a substring.
    - lower(string) - Converts a string to lowercase.
    - upper(string) - Converts a string to uppercase.

2.2 Mathematical Functions

- **Examples**:
    - round(double a) - Rounds a number to the nearest integer.
    - floor(double a) - Returns the largest integer less than or equal to a number.
    - ceil(double a) - Returns the smallest integer greater than or equal to a number.
    - rand() - Generates a random number between 0 and 1.

2.3 Date Functions

- **Examples**:
  - year(string date) - Extracts the year from a date.
  - month(string date) - Extracts the month from a date.
  - day(string date) - Extracts the day from a date.

2.4 Aggregation Functions

- **Examples**:
  - count(*) - Counts the number of rows.
  - sum(column) - Sums the values in a column.
  - avg(column) - Calculates the average of values in a column.
  - max(column) - Finds the maximum value in a column.
  - min(column) - Finds the minimum value in a column.

2.5 Conditional Functions

- **Examples**:
  - if(boolean test, T value1, T value2) - Returns value1 if the condition is true, otherwise value2.
  - case when condition1 then value1 when condition2 then value2 else default end - Implements conditional logic.

7)Explain Pig architecture for scripts data flow and processing?

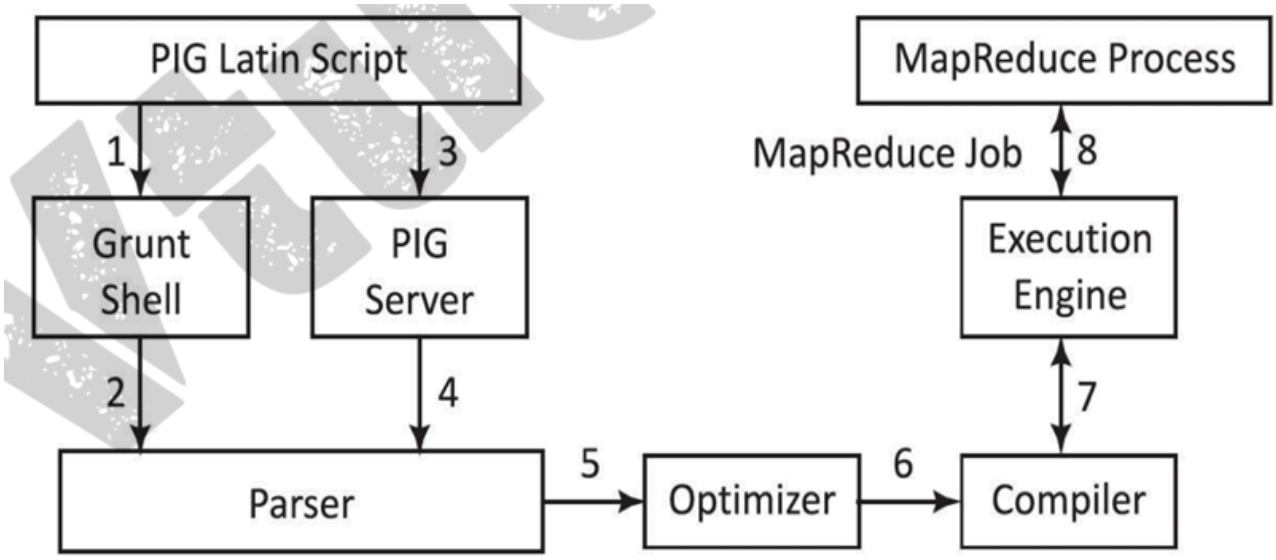

Figure:Pig architecture for scripts data flow and processing

1. Ways to Execute Pig Scripts:

- **Grunt Shell**:
  The Grunt shell is an interactive command-line interface for executing Pig Latin scripts. It allows the user to type commands and execute them immediately, making it suitable for testing and running small scripts.
- **Script File**:
  Pig scripts can be written in a file, and these commands are then executed through the **Pig Server**. This method is useful for running larger, more structured sets of commands.
- **Embedded Script**:
  Users can write **User Defined Functions (UDFs)** for operations that are not available in Pig's built-in functions. UDFs can be written in languages such as Java, Python, or others, and these functions can be embedded in a Pig Latin script file to extend Pig's capabilities.

---

2. Parser in Pig Architecture:

- The **Parser** is responsible for processing Pig scripts after they are input via the **Grunt Shell** or **Pig Server**.
- It performs **type checking** and **syntax validation** to ensure that the script is correct.
- The output of the Parser is a **Directed Acyclic Graph (DAG)**.
  - **DAG** represents the relationships between the logical operators and statements in the Pig script.
  - **Nodes** represent the logical operators (such as filter, group, etc.).
  - **Edges** between the nodes represent data flows.
  - The acyclic nature ensures that each operator is processed only once.

---

3. Optimizer in Pig Architecture:

- After the DAG is created by the Parser, it is passed to the **Optimizer** for optimization.
- **Optimization activities** such as **split**, **merge**, **transform**, and **reordering** of operators occur in this phase.

- The goal of optimization is to reduce the amount of data being processed at any stage, improving performance and efficiency.

Optimization Techniques Include:

- **Push Up Filter**:
  If multiple filter conditions are present, Pig splits them and pushes them up to earlier stages of processing. This reduces the number of records that pass through subsequent operations.
- **Push Down For Each Flatten**:
  The **flatten** operator, which creates a cross-product of complex types like tuples or bags, is applied as late as possible to reduce the number of records in the pipeline.
- **Column Pruner**:
  This technique removes unused columns, minimizing the size of the records and making processing more efficient.
- **Map Key Pruner**:
  Removes unused map keys, thereby reducing the size of the record.
- **Limit Optimizer**:
  When a **limit** operator follows a **load** or **sort** operator, Pig optimizes the process by implementing a limit-sensitive version of these operators, so that the entire dataset is not processed.

---

4. Compiler in Pig Architecture:

- After the DAG is optimized, the **Compiler** takes the optimized plan and generates a series of **MapReduce jobs**.
- The compiler ensures that the Pig Latin script is translated into MapReduce tasks that can be executed on a Hadoop cluster.

---

5. Execution Engine in Pig Architecture:

- The **Execution Engine** is responsible for executing the MapReduce jobs that were generated by the compiler.
- It schedules, monitors, and executes these jobs on the Hadoop cluster, producing the final result.

Apache Pig – Grunt Shell:

- The **Grunt Shell** is the interactive environment used for running Pig scripts.
- It allows you to execute Pig commands and includes support for shell commands such as **sh** and **ls** for running system commands within the Pig environment.

Example commands in the Grunt shell:

  - **sh shell command parameters**: Executes a shell command within the Grunt shell.
  - **ls**: Lists files in the current directory.

- The Grunt Shell also includes utility commands like **clear**, **help**, **history**, **quit**, and **set**, which help manage and control the Pig environment.

8)Differentiate between Pig and MapReduce?

Aspect	Pig	MapReduce
Level of Abstraction	High-level scripting language	Low-level programming model
Language	Uses **Pig Latin**, a high-level data flow language	Uses **Java** (or other languages like Python) for writing custom code
Ease of Use	Easier for developers with less programming experience, as it provides a simpler abstraction	Requires in-depth programming knowledge to write custom MapReduce jobs
Data Processing	Provides a rich set of built-in operators for processing large datasets (e.g., **filter**, **join**, **group** etc.)	Requires writing custom code for each task (Mapper and Reducer functions)
Optimization	Built-in **optimizer** that automatically optimizes the execution plan	No automatic optimization; developers have to manually optimize code
Execution Model	Generates **MapReduce jobs** behind the scenes but abstracts the complexity for the user	Directly runs **MapReduce** jobs, where the user has to manage all aspects of the job execution
Flexibility	Limited to the operations provided by Pig Latin, though you can extend with UDFs	Highly flexible, as you can define any operation in the Map and Reduce functions
Use Case	Ideal for **ETL processes**, **data processing**, and **ad-hoc queries** where speed of development is important	Ideal for **complex, custom processing** and when fine-grained control over job execution is required
Performance	Less control over optimization, but usually faster for simple tasks due to higher-level abstraction	More control over optimization and performance, but requires more effort from the developer
Data Flow	Handles data flow between operators automatically through Pig Latin	Data flow needs to be explicitly handled by the developer in the form of Map and Reduce operations

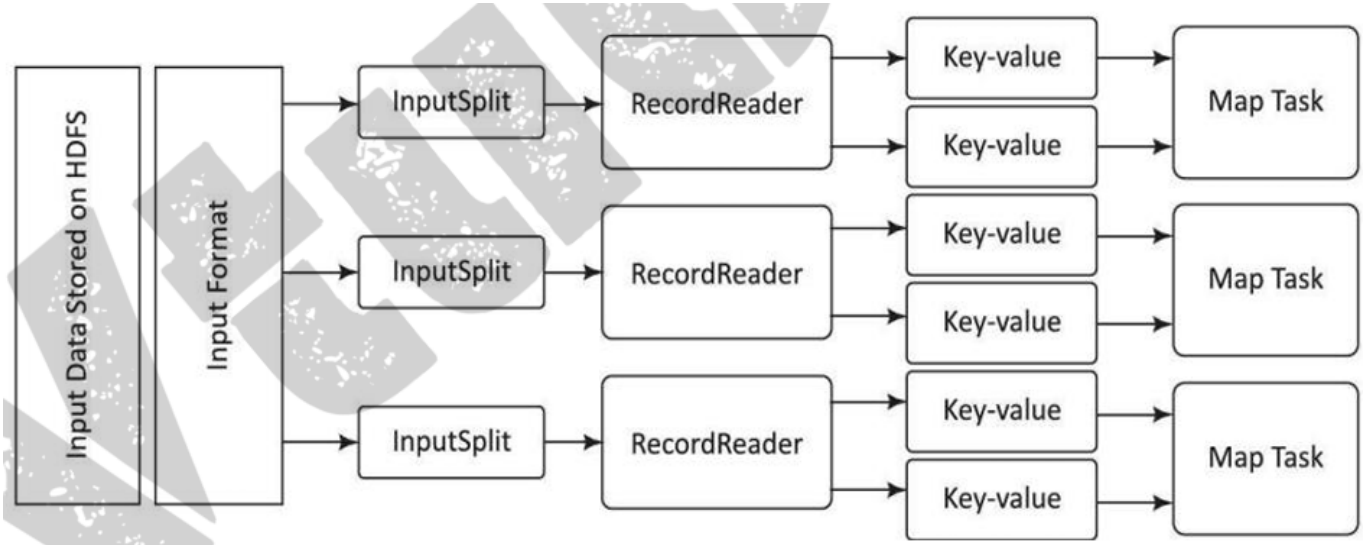

Figure: Key-value pairing in MapReduce

Key-Value Pairing in MapReduce

In Hadoop MapReduce, the processing of data occurs in the form of key-value pairs. Every phase of MapReduce, i.e., the Map phase and Reduce phase, operates on input and output as key-value pairs. Before the data is passed to the mapper, it must be converted into key-value pairs because the mapper only understands this format.

Steps for Key-Value Pair Generation in MapReduce

1. InputSplit:
    - InputSplit is the logical representation of the data that needs to be processed.
    - The input data is split into smaller chunks (InputSplits) for parallel processing across mappers.
    - Each InputSplit represents a portion of the input data to be processed by an individual map() function.
2. RecordReader:
    - The RecordReader interacts with the InputSplit and converts the split data into records.
    - These records are presented in the form of key-value pairs.
    - The TextInputFormat is used by default to convert input data into key-value pairs where:
        - The key represents the byte offset of a line within the file.

        - The value is the actual content of that line (a text line).
    - The RecordReader reads the split data and communicates with the InputSplit until the entire data is read.

Key-Value Pair Usage in MapReduce

Key-value pairs are used in four major phases of MapReduce:

1. map() input:
    - The mapper takes input as a key-value pair.
    - Example: Key = byte offset, Value = a line of text.
2. map() output:
    - After processing, the mapper produces intermediate key-value pairs as output.
    - Example: Key = a word, Value = 1 (for word count).
3. reduce() input:
    - The shuffle and sort phase groups the intermediate key-value pairs based on keys.
    - The input to the reducer consists of keys and a list of values.
    - Example: Key = a word, Values = [1, 1, 1, …] (grouped list from multiple mappers).
4. reduce() output:
    - The reducer processes the grouped key-value pairs and produces the final key-value pair output.
    - Example: Key = a word, Value = count of occurrences.

Grouping by Key in MapReduce

- After the map task completes, the Shuffle process aggregates all mapper outputs by grouping intermediate key-value pairs using the keys.
- The values for each key are appended into a list.
- This creates an intermediate key-value format where:
    - Key = group identifier (e.g., a word).
    - Value = list of values associated with that key (e.g., [1, 1, 1]).