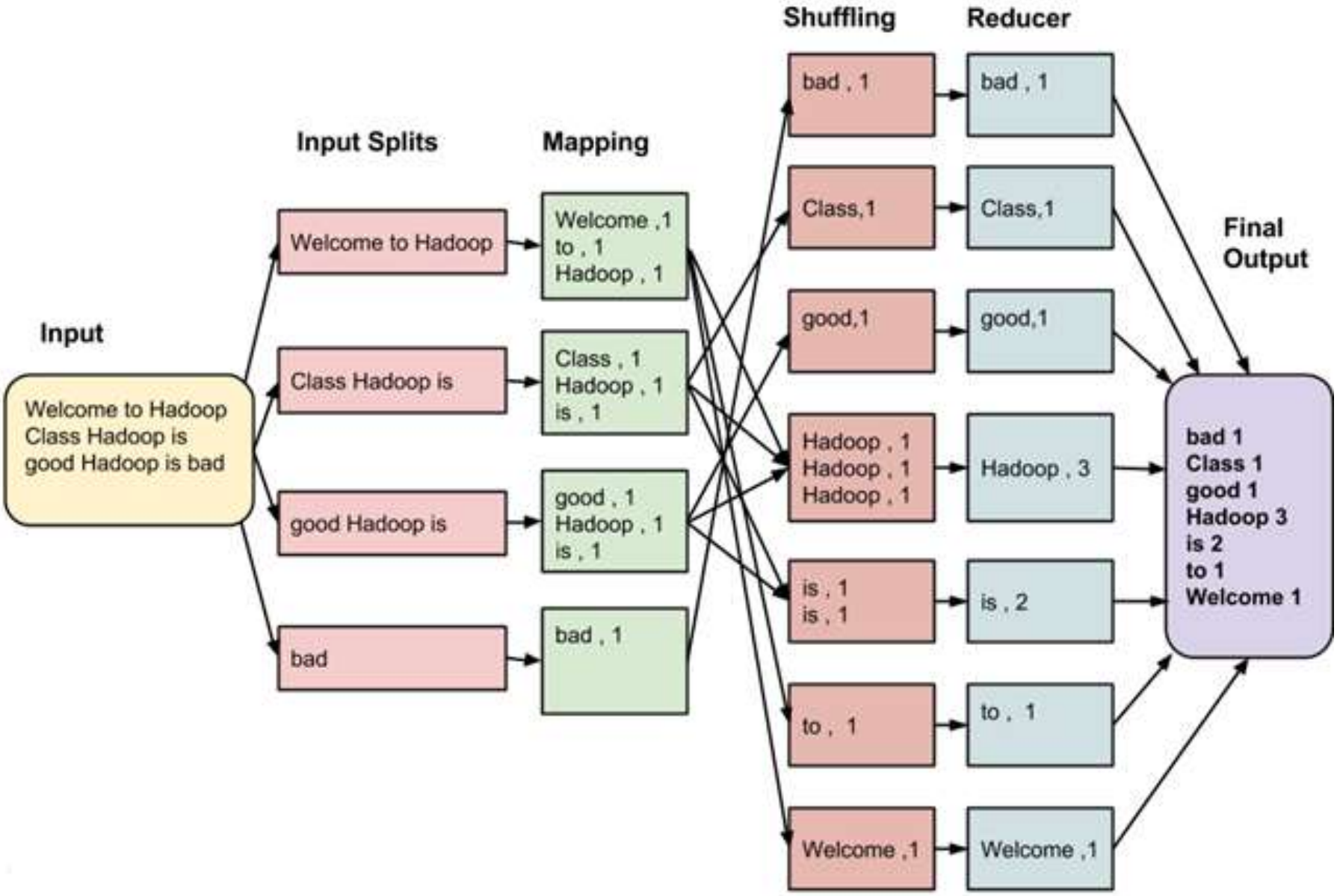
MODULE-4

MapReduce , Hive and Pig

MapReduce Architecture



INTRODUCTION

- The data processing layer is the application support layer, while the application layer is the data consumption layer in Big-Data architecture design (Figure 1.2).
- When using HDFS, the Big Data processing layer includes the APIs of Programs such as **MapReduce and Spark.**

- The application support layer includes HBase which creates column-family data store using other formats such as key-value pairs or JSON file
- HBase stores and processes the columnar data after translating into MapReduce tasks to run in HDFS.

- The support layer also includes Hive which creates SQL-like tables.
- Hive stores and processes table data after translating it into MapReduce tasks to run in HDFS.
- Hive creates SQL-like tables in Hive shell.
- Hive uses HiveQL processes queries, ad hoc (unstructured) queries, aggregation functions and summarizing functions, such as functions to compute maximum, minimum, average of selected or grouped datasets. HiveQL is a restricted form of SQL.

- The support layer also includes Pig. Pig is a data-flow language and an execution framework .
- Pig enables the usage of relational algebra in HDFS.
- MapReduce is the processing framework and YARN is the resource managing framework .

- Figure 4.1 shows Big Data architecture design layers: (i) data storage, (ii) data processing and data consumption, (iii) support layer APIs for MapReduce, Hive and Pig running on top of the HDFS Data Store, and (v) application tasks.
- Pig is a dataflow language, which means that it defines a data stream and a series of transformations.

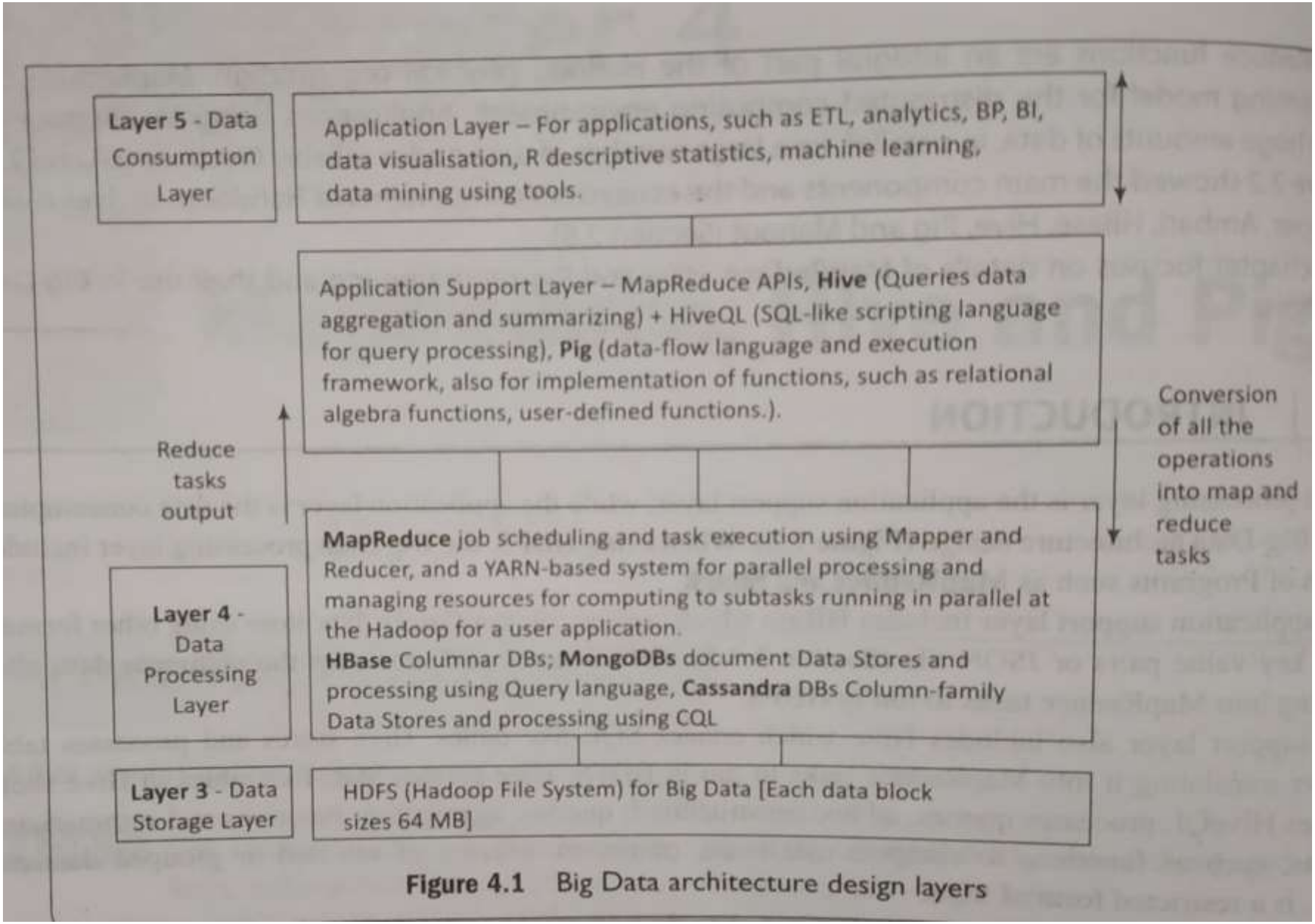


Figure 4.1 Big Data architecture design layers

- Hive and Pig are also part of the ecosystem (Figure 4.1).
- Big Data storage and application-support APIs can use Hive and Pig for processing data at HDFS.
- Processing needs mapping and finding the source file for data. File is in the distributed data store. Requirement is to identify the needed data-block in the cluster.
- Applications and AP Is run at the data nodes stored at the blocks.

- The smallest unit of data that can be stored or retrieved from the disk is a block. HDFS deals with the data stored in blocks.
- The Hadoop application is responsible for distributing the data blocks across multiple nodes.
- The tasks, therefore, first convert into map and reduce tasks. This requirement arises because the mapping of stored values is very important.
- The number of map tasks in an application is handled by the number of blocks of input files.

MAPREDUCE MAP TASKS, REDUCE TASKS AND MAPREDUCE EXECUTION

- Big data processing employs the MapReduce programming model.
- A Job means a MapReduce program. Each job consists of several smaller units, called MapReduce tasks.
- A software execution framework in MapReduce programming defines the parallel tasks. The tasks give the required result. The Hadoop MapReduce implementation uses Java framework.

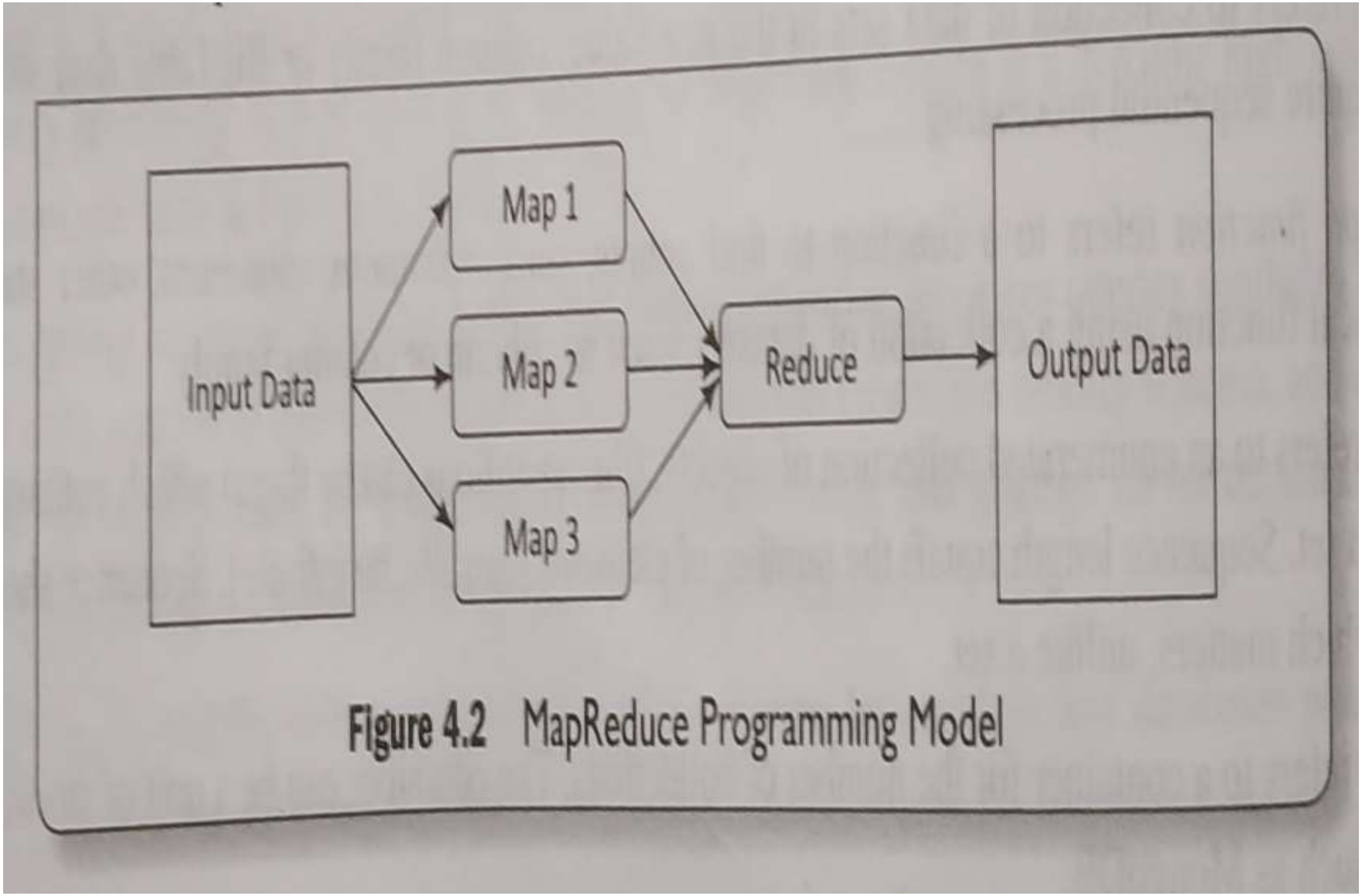


Figure 4.2 MapReduce Programming Model

- The model defines two important tasks, namely Map and Reduce.
- **Map** takes input data set as pieces of data and maps them on various nodes for parallel processing.
- The **reduce** task, which takes the output from the maps as an input and combines those data pieces into a smaller set of data. A reduce task always run after the map task (s).
- Many real-world situations are expressible using this model. Such Model describes the essence of MapReduce programming where the programs written are automatically parallelize and execute on a large cluster.

- The input data is in the form of an HDFS file. The output of the task also gets stored in the HDFS.
- The compute nodes and the storage nodes are the same at a cluster, that is, the MapReduce program and the HDFS are running on the same set of nodes.
- This configuration results in effectively scheduling of the sub-tasks on the nodes where the data is already present.
- This results in high efficiency due to reduction in network traffic across the cluster.

- A user application specifies locations of the input/ output data and translates into map and reduces functions.
- A job does implementations of appropriate interfaces and/ or abstract-classes. These, and other job parameters, together comprise the job configuration.
- The Hadoop job client then submits the job (jar/ executable etc.) and configuration to the JobTracker, which then assumes the responsibility of distributing the software/configuration to the slaves by scheduling tasks, monitoring them, and provides status and diagnostic information to the job-client.

- Figure 4.3 shows MapReduce process when a client submits a job, and the succeeding actions by the JobTracker and TaskTracker.

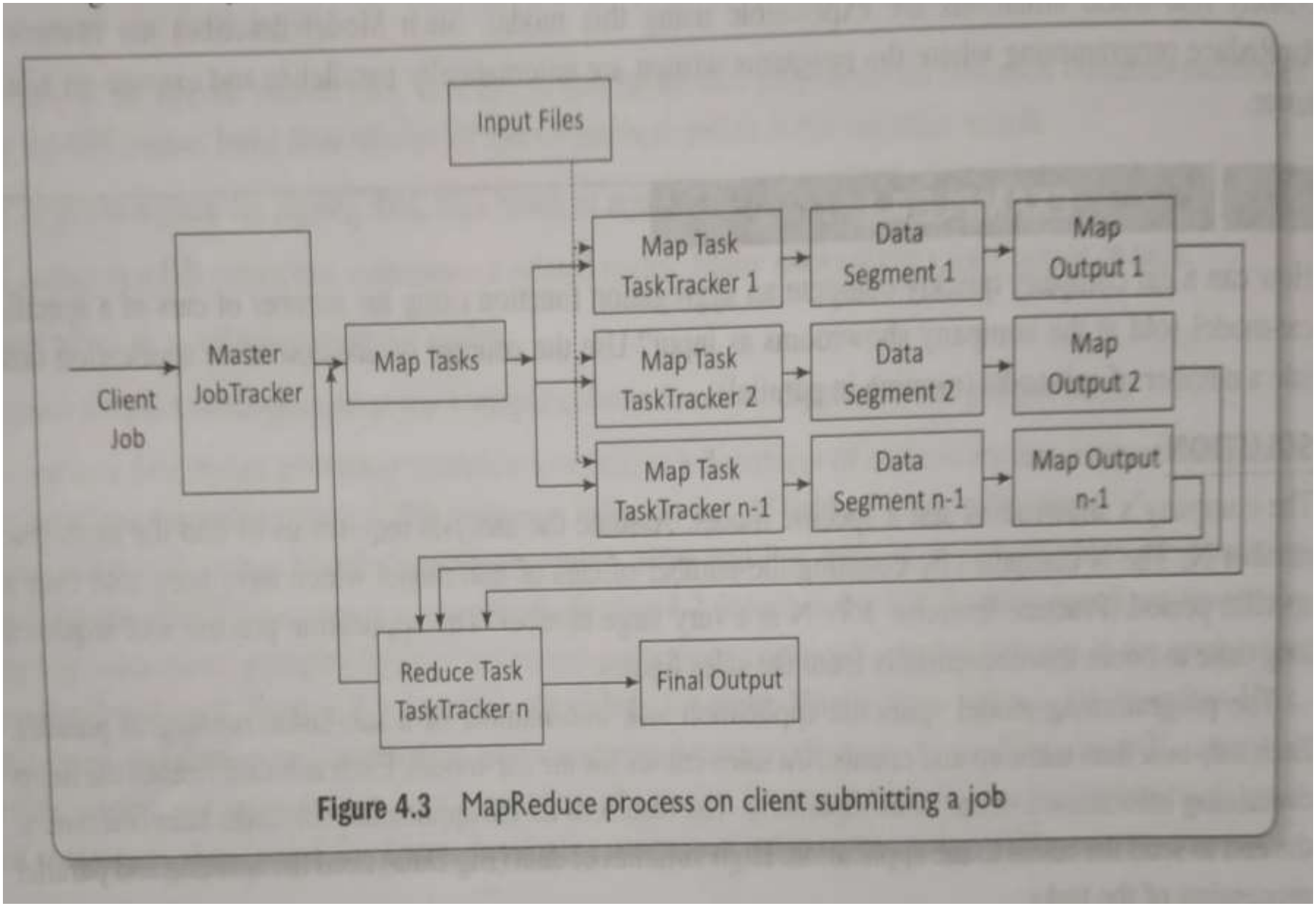


Figure 4.3 MapReduce process on client submitting a job

JobTracker and Task Tracker

- MapReduce consists of a single master JobTracker and one slave TaskTracker per cluster node.
- The master is responsible for scheduling the component tasks in a job onto the slaves, monitoring them and re-executing the failed tasks.
- The slaves execute the tasks as directed by the master.

- The data for a MapReduce task is initially at input files. The input files typically reside in the HDFS.
- The files may be line-based log files, binary format file, multi-line input records, or something else entirely different.
- These input files are practically very large, hundreds of terabytes or even more than it.

- Most importantly, the MapReduce framework operates entirely on key, value-pairs.
- The framework views the input to the task as a set of (key, value) pairs and produces a set of (key, value) pairs as the output of the task.

Map-Tasks

- ***Map task*** means a task that implements a map(), which runs user application codes for each key-value pair **(k1, v1).**
- **Key k1** is a set of keys. Key k1 maps to a group of data values .
- **Values v1** are a large string which is read from the input file(s).
- The output of map() would be zero (when no values are found) or intermediate key-value pairs (k2, v2).
- The value v2 is the information for the transformation operation at the reduce task using aggregation or other reducing functions.

- ***Reduce task*** refers to a task which takes the output v2 from the map as an input and combines those data pieces into a smaller set of data using a combiner. The reduce task is always performed after the map task.
- The Mapper performs a function on individual values in a dataset irrespective of the data size of the input. That means that the Mapper works on a single data set.
- Figure 4.4 shows logical view of functioning of map().

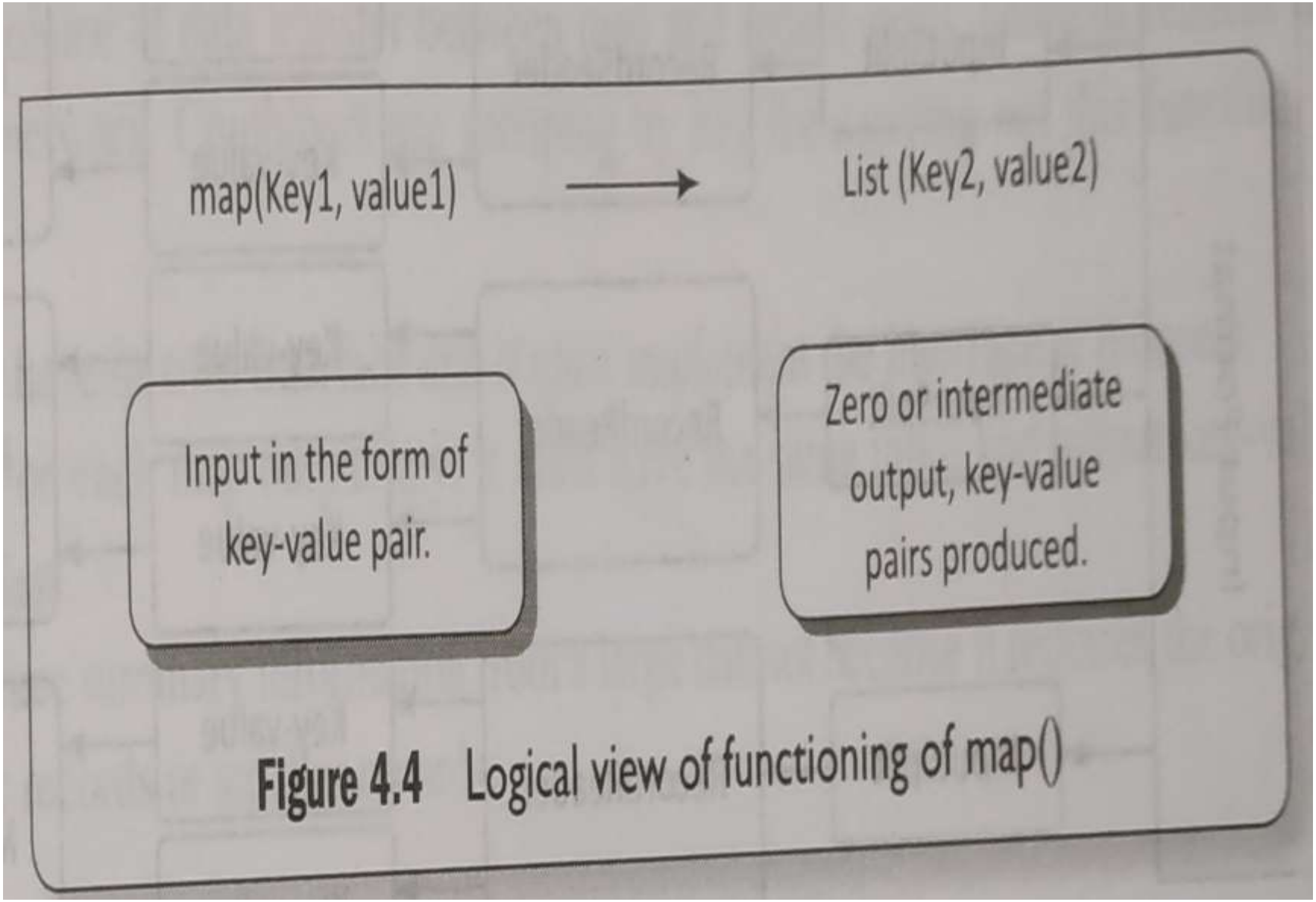


Figure 4.4 Logical view of functioning of map()

Key-Value Pair

- Each phase (Map phase and Reduce phase) of MapReduce has key-value pairs as input and output.
- Data should be first converted into key-value pairs before it is passed to the Mapper, as the Mapper only understands key-value pairs of data.

Key-value pairs in Hadoop MapReduce are generated as follows:

- ***InputSplit*** - Defines a logical representation of data and presents a Split data for processing at individual map().
- ***RecordReader*** - Communicates with the InputSplit and converts the Split into records which are in the form of key-value pairs in a format suitable for reading by the Mapper.
- RecordReader uses TextInputFormat by default for converting data into key-value pairs.
- RecordReader communicates with the InputSplit until the file is read.

- Figure 4.5 shows the steps in MapReduce key-value pairing.
- Generation of a key-value pair in MapReduce depends on the dataset and the required output.
- Also, the functions use the key-value pairs at four places: map() input, map() output, reduce() input and reduce() output.

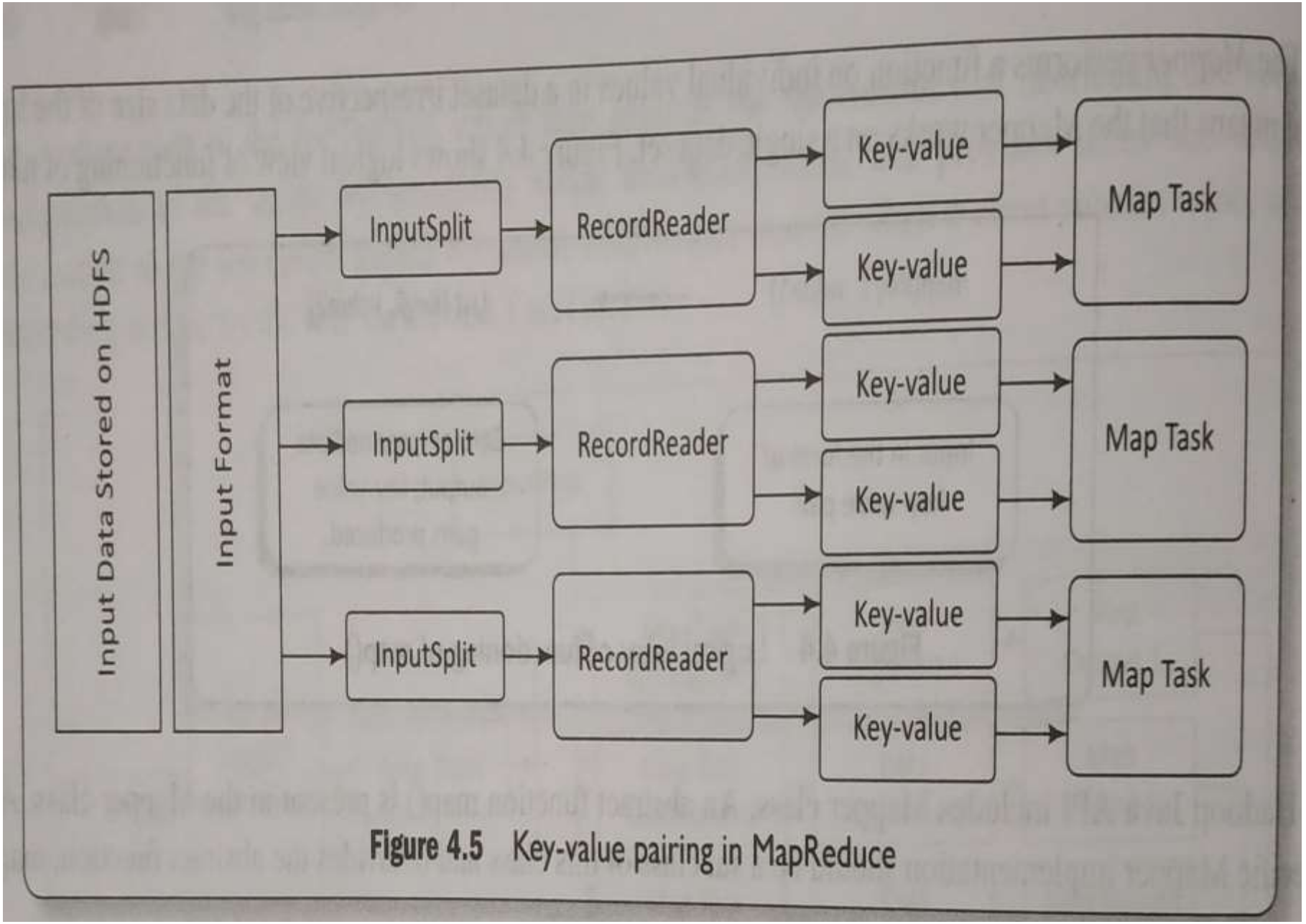


Figure 4.5 Key-value pairing in MapReduce

Grouping by Key

- When a map task completes, Shuffle process aggregates (combines) all the Mapper outputs by grouping the key-values of the Mapper output, and the value v2 append in a list of values.
- A "Group By" operation on intermediate keys creates v2.

Shuffle and Sorting Phase

- Here, all pairs with the same group key (**k2**) collect and group together, creating one group for each key. So, the Shuffle output format will be a List of <**k2**, List (v2)>, Thus, a different subset of the intermediate key space assigns to each reduce node. These subsets of the intermediate keys (known as "partitions") are inputs to the reduce tasks.
- Each reduce task is responsible for reducing the values associated with partitions. HDFS sorts the partitions on a single node automatically before they input to the Reducer.

Partitioning

- The Partitioner does the partitioning .The partitions are the semi-mappers in MapReduce. Partitioner is an optional class.
- MapReduce driver class can specify the Partitioner. A partition processes the output of map tasks before submitting it to Reducer tasks.
- Partitioner function executes on each machine that performs a map task. Partitioner is an optimization in MapReduce that allows local partitioning before reduce-task phase.

- Functions for Partitioner and sorting functions are at the mapping node.
- The main function of a Partitioner is to split the map output records with the same key.

Combiners

- Combiners are semi-reducers in MapReduce. Combiner is an optional class.
- MapReduce driver class can specify the combiner.
- The combiner() executes on each machine that performs a map task.
- Combiners optimize MapReduce task that locally aggregates before the shuffle and sort phase. Typically, the same codes implement both the combiner and the reduce functions, combiner() on map node and reducer() on reducer node. .

- The main function of a Combiner is to consolidate the map output records with the same key.
- The output (key-value collection) of the combiner transfers over the network to the Reducer task as input.
- Combiners use grouping by key for carrying out this function. The combiner works as follows:

(i) It does not have its own interface and it must implement the interface at reduce().

(ii) It operates on each map output key. It must have the same input and output key-value types as the Reducer class.

(iii) It can produce summary information from a large dataset because it replaces the original Map output with fewer records or smaller records.

Reduce Tasks

- Java API at Hadoop includes Reducer class. An abstract function, reduce() is in the Reducer. Any specific Reducer implementation should be subclass of this class and override the abstract reduce().
- Reduce task implements reduce() that takes the Mapper output (which shuffles and sorts), which is grouped by key-values (k2, v2) and applies it in parallel to each group.
- Intermediate pairs are at input of each Reducer in order after sorting using the key.
- Reduce function iterates over the list of values associated with a key and produces outputs such as aggregations and statistics.
- The reduce function sends output zero or another set of key-value pairs (k3, v3) to the final the output file. Reduce: {(k2, list (v2) -> list (k3, v3)}

- ***Sample Code for Reducer Class***

```
public class ExampleReducer extends Reducer<k2,
   v2, k3, v3>
{
      void reduce (k2 key, Iterable<v2> values, Context
         context) throws IOBxception,InterruptedException
         { ... }
}
```

Details of MapReduce Processing Steps

- Figure 4.6 shows the execution steps, data flow, splitting, partitioning and sorting on a map node and reducer on reducer node.

- **ACVMs (Automatic Chocolate Vending Machines).**
- **Automotive Components and Predictive Automotive Maintenance Services (ACPAMS).** ACPAMS is an application of (Internet) connected cars which renders services to customers for maintenance and servicing of (Internet) connected cars.

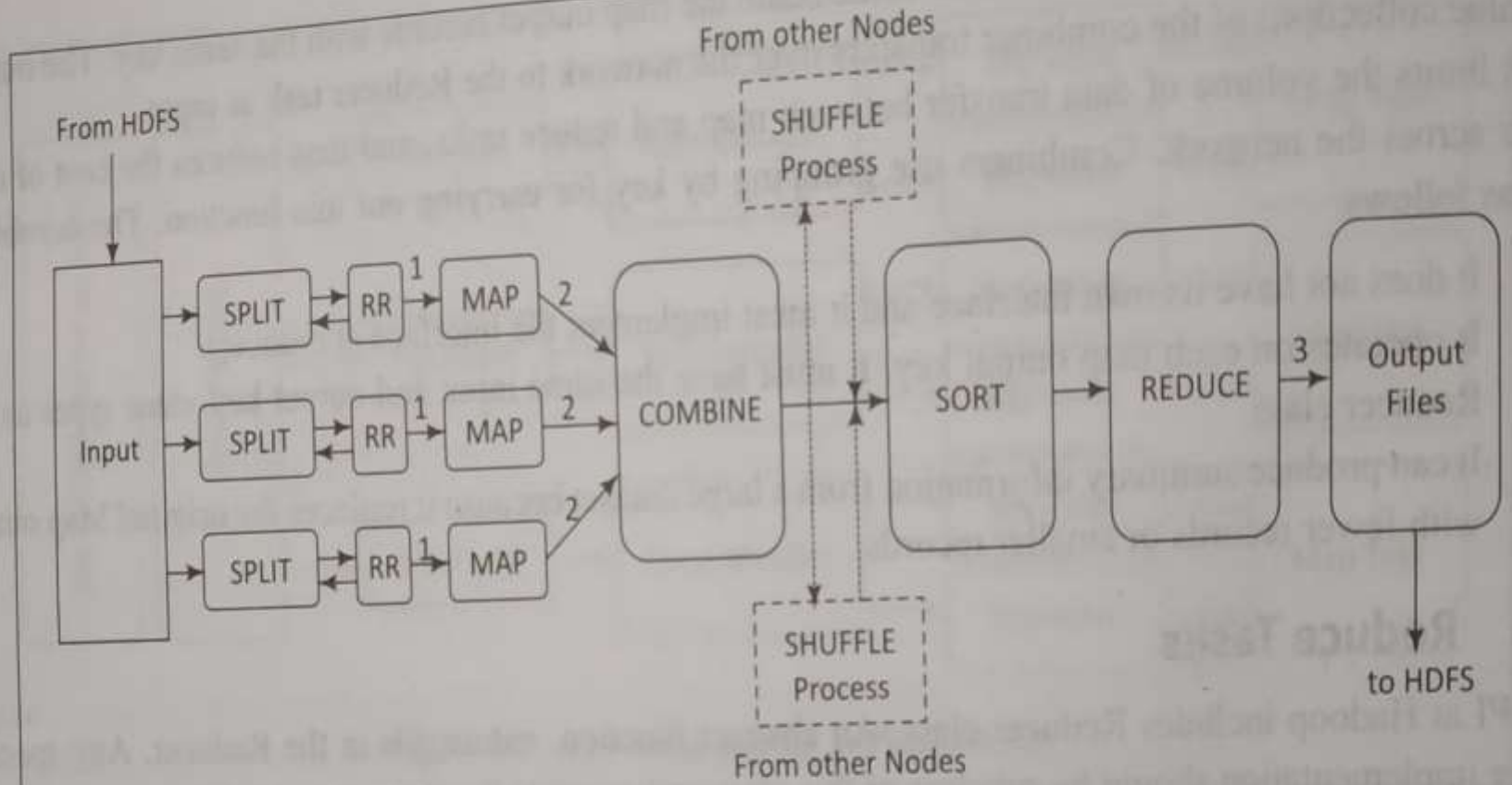


Figure 4.6 MapReduce execution steps

Describe the MapReduce processing steps of a task of ACPAMS.

SOLUTION

Figure 4.7 shows processing steps of an ACPAMS task in MapReduce. Steps are inputs, mapping, combining, shuffling and reducing for the output to application task.

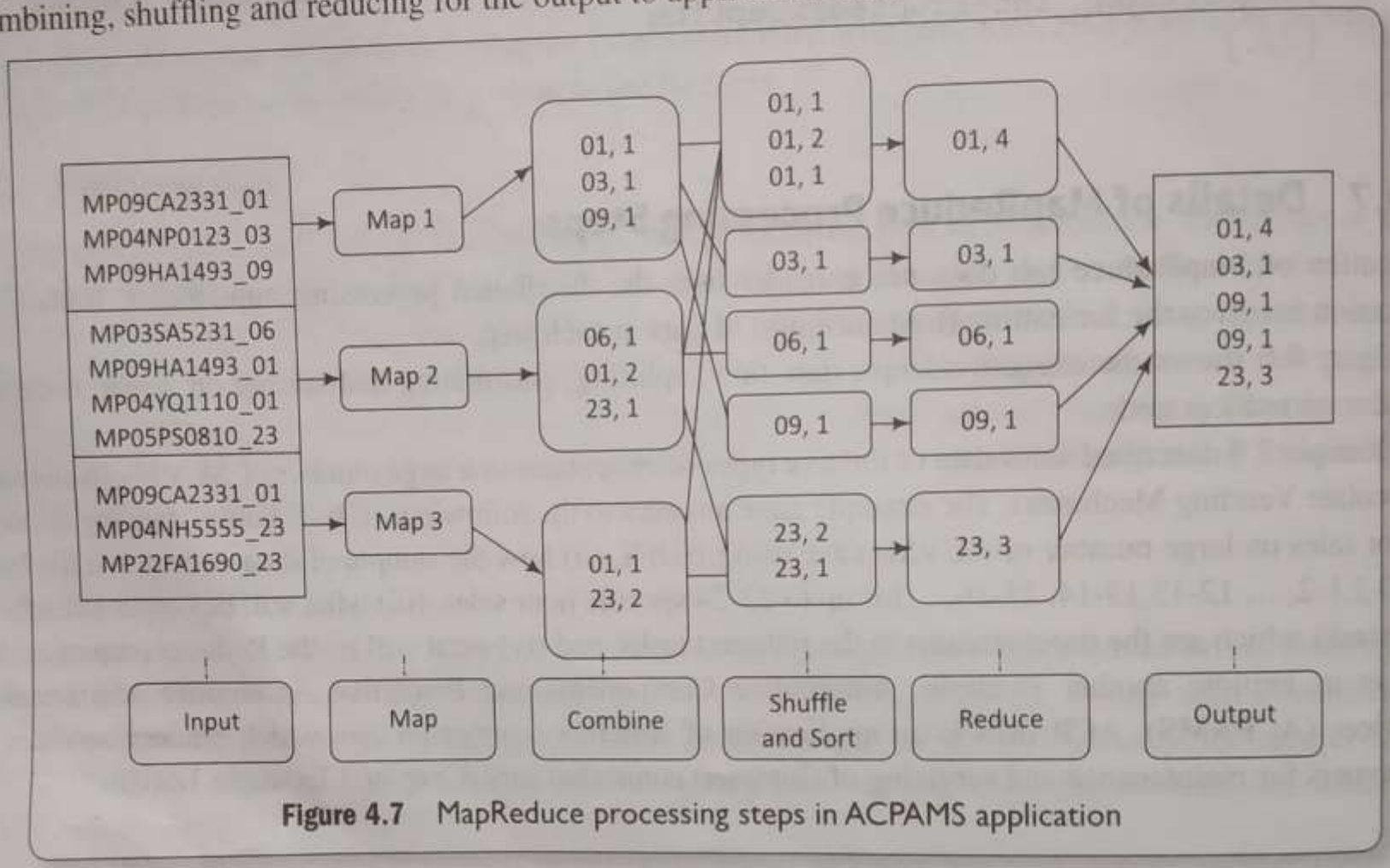


Figure 4.7 MapReduce processing steps in ACPAMS application

- The application submits the inputs. The execution framework handles all other aspects of distributed processing transparently, on clusters ranging from a single node to a few thousand nodes.
- The aspects include scheduling, code distribution, synchronization, and error and fault handling.

COMPOSING MapReduce FOR CALCULATIONS AND ALGORITHMS

- The following subsections describe the use of MapReduce program composition in counting and summing, algorithms for relational algebraic operations, projections, unions, intersections, natural joins, grouping and aggregation, matrix multiplication and other computations.

Composing Map-Reduce for Calculations

The calculations for various operations compose are:

- **Counting and Summing :** Assume that the number of alerts or messages generated during a specific maintenance activity of vehicles need counting for a month.
- **Counting** is used in the data querying application. For example, count of messages generated, word count in a file, number of cars sold, and analysis of the logs, such as number of tweets per month. Application is also in business analytics field.

- **Sorting** Figure 4.6 illustrated MapReduce execution steps, i.e., dataflow, splitting, partitioning and sorting on a map node and reduce on a reducer node.
- Example 4.3 illustrated the sorting method. Many applications need sorted values in a certain order by some rule or process.
- **Mappers** just emit all items as values associated with the sorting keys which assemble as a function of items.
- **Reducers** combine all emitted parts into a final list.

- **Finding Distinct Values (Counting unique values)** Applications such as web log analysis need counting of unique users. Evaluation is performed for the total number of unique values in each field for each set of records that belongs to the same group. Two solutions are possible:

(i) The **Mapper** emits the dummy counters for each pair of field and groupId, and the **Reducer** calculates the total number of occurrences for each such pair.

(ii) The **Mapper** emits the values and groupId, and the **Reducer** excludes the duplicates from the list of groups for each value and increments the counter for each group. The final step is to sum all the counters emitted at the Reducer.

- **Collating :** is a way to collect all items which have the same value of function in one document or file, or a way to process items with the same value of the function together.
- Examples of applications are producing inverted indexes and extract, transform and load operations.
- **Mapper** computes a given function for each item, produces value of the function as a key, and the item itself as a value.
- **Reducer** then obtains all item values using group-by function, processes or saves them into a list and outputs to the application task or saves them.

Filtering or Parsing

- Filtering or parsing collects only those items which satisfy some condition or transform each item into some other representation.
- Filtering/ parsing include tasks such as text parsing, value extraction and conversion from one format to another.
- Examples of applications of filtering are found in data validation, log analysis and querying of datasets.

- **Mapper** takes items one by one and accepts only those items which satisfy the conditions and emit the accepted items or their transformed versions.
- **Reducer** obtains all the emitted items, saves them into a list and outputs to the application.

Distributed Tasks Execution

- Large computations divide into multiple partitions and combine the results from all partitions for the final result. Examples of distributed running of tasks are physical and engineering simulations, numerical analysis and performance testing.
- **Mapper** takes a specification as input data, performs corresponding computations and emits results.
- **Reducer** combines all emitted parts into the final result.

Graph Processing using Iterative Message Passing

- Graph is a network of entities and relationships between them.
- A node corresponds to an entity. An edge joining two nodes corresponds to a relationship. Path traversal method processes a graph.
- Traversal from one node to the next generates a result which passes as a message to the next traversal between the two nodes.
- Cyclic path traversal uses iterative message passing.

- Web indexing also uses iterative message passing.
- Graph processing or web indexing requires calculation of the state of each entity. Calculated state is based on characteristics of the other entities in its neighborhood in a given network.
- (State means present value. For example, assume an entity is a course of study. The course may be Java or Python. Java is a state of the entity and Python is another state.)

- A set of nodes stores the data and codes at a network.
- Each node contains a list of neighbouring node IDs. MapReduce jobs execute iteratively.
- Each node in an iteration sends messages to its neighbors. Each neighbor updates its state based on the received messages.
- Iterations terminate on some conditions, such as completion of fixed maximal number of iterations or specified time to live or negligible changes in states between two consecutive iterations.

- **Mapper** emits the messages for each node using the ID of the adjacent node as a key. All messages thus group by the incoming node.
- **Reducer** computes the state again and rewrites a node new state.

Cross Correlation

- Cross-correlation involves calculation using number of tuples where the items co-occur in a set of tuples of items. If the total number of items is N, then the total number of values= N x N.
- Cross correlation is used in text analytics. (Assume that items are words and tuples are sentences).
- Another application is in market-analysis (for example, to enumerate, the customers who buy item x tend to also buy y).
- If N x N is a small number, such that the matrix can fit in the memory of a single machine, then implementation is straightforward.

Two solutions for finding cross correlations are:

(i) The ***Mapper*** emits all pairs and dummy counters, and the **Reducer** sums these counters. The benefit from using combiners is little, as it is likely that all pairs are distinct. The accumulation does not use in-memory computations as N is very large.

(ii) The ***Mapper*** groups the data by the first item in each pair and maintains an associative array ("stripe") where counters for all adjacent items accumulate. The ***Reducer*** receives all stripes for the leading item, merges them and emits the same result as in the pairs approach.

The grouping:

- Generates fewer intermediate keys. Hence, the framework has less sorting to do.
- Greatly benefits from the use of combiners.
- In-memory accumulation possible.
- Enables complex implementations.
- Results in general, faster computations using stripes than "pairs".

Matrix-Vector Multiplication by MapReduce

- Numbers of applications need multiplication of n x n matrix **A** with vector **B** of dimension n. Each element of the product is the element of vector C of dimension n. The elements of **C** calculate by relation,

Matrix-Vector Multiplication by MapReduce

$c_i = \sum_{j=1}^{n} a_{ij} b_j$. An example of calculations is given below.

Assume $\mathbf{A} = \begin{vmatrix} 1 & 5 & 4 \\ 2 & 1 & 3 \\ 4 & 2 & 1 \end{vmatrix}$ and $\mathbf{B} = \begin{vmatrix} 4 \\ 1 \\ 3 \end{vmatrix}$.

Multiplication $\mathbf{C} = \mathbf{A} \times \mathbf{B} = \begin{bmatrix} 1 \times 4 + 5 \times 1 + 4 \times 3 \\ 2 \times 4 + 1 \times 1 + 3 \times 3 \\ 4 \times 4 + 2 \times 1 + 1 \times 3 \end{bmatrix}$

Hence, $\mathbf{C} = \begin{bmatrix} 21 \\ 18 \\ 21 \end{bmatrix}$

Relational-Algebra Operations

- **Selection**
- Example of Selection in relational algebra is as follows: Consider the attribute names (ACVM_ID, Date, chocolate_flavour, daily_sales). Consider relation

R = {(524, 12122017, KitKat, 82), (524, 12122017, Oreo, 72), (525, 12122017, KitKat, 82), (525, 12122017, Oreo, 72), (526, 12122017, KitKat, 82), (526, 12122017, Oreo, 72)}.

Selection $\sigma_{ACVM\_ID <= 525}(R)$ selects the subset R= {(524, 12122017, KitKat, 82), (524, 12122017, Oreo, 72), (525, 12122017, KitKat, 82), (525, 12122017, Oreo, 72)}.

Selection $\sigma_{chocolate\_flavour = Oreo}$ selects the subset {(524, 12122017, Oreo, 72), (525, 12122017, Oreo, 72), (526, 12122017, Oreo, 72)}.

4.3.3.2 Projection

Example of *Projection* in relational algebra is as follows:

Consider attribute names (ACVM_ID, Date, chocolate_flavour, daily_sales).

Consider relation R = {(524, 12122017, KitKat, 82), (524, 12122017, Oreo, 72)}.

Projection $\Pi_{ACVM\_ID}(R)$ selects the subset {(524)}.

Projection, $\Pi_{chocolate\_flavour,\ 0.5*\ daily\_sales}$ selects the subset {(KitKat, 0.5 x 82), (Oreo, 0.5 x 72)}.

The test() tests the presence of attribute (s) used for projection and the factor by an attribute needs projection.

The *Mapper* calls test() for each tuple in a row. When the test satisfies, the predicate then emits the tuple (same as in selection). The *Reducer* transfers the received input tuples after eliminating the possible duplicates. Such operations are used in analytics.

4.3.3.3 Union

Example of Union in relations is as follows: Consider,

R1 = {(524, 12122017, KitKat, 82), (524, 12122017, Oreo, 72)}

R2 = {(525, 12122017, KitKat, 82), (525, 12122017, Oreo, 72)}

and R3 = {(526, 12122017, KitKat, 82), (526, 12122017, Oreo, 72)}

Result of Union operation between R1 and R3 is:

R1 U R3 = {(524, 12122017, KitKat, 82), (524, 12122017, Oreo, 72), (526, 12122017, KitKat, 82), (526, 12122017, Oreo, 72)}

The *Mapper* executes all tuples of two sets for union and emits all the resultant tuples. The *Reducer* class object transfers the received input tuples after eliminating the possible duplicates.

4.3.3.4 Intersection and Difference

Intersection Example of Interaction in relations is as follows: Consider,

R1 = {(524, 12122017, Oreo, 72)}

R2 = {(525, 12122017, KitKat, 82)}

and R3 = {(526, 12122017, KitKat, 82), (526, 12122017, Oreo, 72)}

Result of Intersection operation between R1 and R3 are

$R1 \cap R3$ = {(12122017, Oreo)}

The *Mapper* executes all tuples of two sets for intersection and emits all the resultant tuples. The *Reducer* transfers only tuples that occurred twice. This is possible only when tuple includes primary key and can occur once in a set. Thus, both the sets contain this tuple.

Difference Consider:

R1 = {(12122017, KitKat, 82), (12122017, Oreo, 72)}

and R3 = {(12122017, KitKat, 82), (12122017, Oreo, 25)}

Difference means the tuple elements are not present in the second relation. Therefore, difference set_1 is

R1 – R3 = (12122017, Oreo, 72) and set_2 is R3 – R1 = (12122017, Oreo, 25).

The *Mapper* emits all the tuples and tag. A tag is the name of the set (say, set_1 or set_2 to which a tuple belongs to). The *Reducer* transfers only tuples that belong to set_1.

Natural Join

- Consider two relations R1 and R2 for tuples a, b and c. Natural Join computes for R1 (a, b) with R2 (b, c). Natural Join is R (a, b, c). Tuples b joins as one in a Natural Join.
- The **Mapper** emits the key-value pair (b, (R1, a)) for each tuple (a, b) of R1, similarly emits (b, (R2, c)) for each tuple (b, c) of R2.
- The **Mapper** is mapping both with Key for b. The **Reducer** transfers all pairs consisting of one with first component R1 and the other with first component R2, say (R1, a) and (R2, c).
- The output from the key and value list is a sequence of key-value pairs. The key is of no use and is irrelevant. Each value is one of the triples (a, b, c) such that (R1, a) and (R2, c) are present in the input list of values.

The following example explains the concept of join, how the data stores use the INNER Join and NATURAL Join of two tables, and how the Join compute quickly.

EXAMPLE 4.6

An SQL statement "`Transactions INNER JOIN KitKatStock ON Transactions.ACVM_ID = KitKatStock.ACVM_ID`"; selects the records that have matching values in two tables for transactions of KitKat sales at a particular ACVM. One table is KitKatStock with columns (KitKat_Quantity, ACVM_ID) and second table is Transactions with columns (ACVM_ID, Sales_Date and KitKat_SalesData).

1. What will be INNER Join of two tables KitKatStock and Transactions?
2. What will be the NATURAL Join?

SOLUTION

1. The INNER JOIN gives all the columns from the two tables (thus the common columns appear twice). The INNER JOIN of two tables will return a table with five column: (i) KitKatStock. Quantity, (ii) KitKatStock. KitKat_ACVM_ID, (iii) Transactions.ACVM_ID, (iv)Transactions. KitKat_SalesDate, and (v) Transactions.KitKat_SalesData.
2. The NATURAL JOIN gives all the unique columns from the two tables. The NATURAL JOIN of two tables will return a table with four columns: (i) KitKatStock.Quantity, (ii) KitKatStock. ACVM_ID, (iii) Transactions.KitKat_SalesDate, and (iv) Transactions.KitKat_SalesData.

Values accessible by key in the first table KitKatStock merges with Transactions table accessible by the common key ACVM_ID.

NATURAL JOIN gives the common column once in the output of a query, while INNER JOIN gives common columns of both tables.

Join enables fast computations of the aggregate of the number of chocolates of specific flavour sold.

Grouping and Aggregation by MapReduce

Grouping means operation on the tuples by the value of some of their attributes after applying the aggregate function independently to each attribute. A Grouping operation denotes by <grouping attributes> ℑ <function-list> (R). Aggregate functions are count(), sum(), avg(), min() and max().

Assume *R*= {(524, 12122017, KitKat, 82), (524, 12122017, Oreo, 72), (525, 12122017, KitKat, 82), (525, 12122017, Oreo, 72), (526, 12122017, KitKat, 82), (526, 12122017, Oreo, 72)}. Chocolate_flavour ℑ count ACVM_ID, sum (daily_sales (chocolate_flavour)) will give the output (524, KitKat, sale_month), (525, KitKat, sale_month), ... and (524, Oreo, sale_month), (525, Oreo, sale_month), ... for all ACVM_IDs.

The *Mapper* finds the values from each tuple for grouping and aggregates them. The *Reducer* receives the already grouped values in input for aggregation.

HIVE

- Hive was created by Facebook. Hive is a data warehousing tool and is also a data store on the top of Hadoop.
- An enterprise uses a data warehouse as large data repositories that are designed to enable the tracking, managing, and analyzing the data.
- Hive processes structured data and integrates data from multiple heterogeneous sources. Additionally, also manages the constantly growing volumes of data.

Figure 4.9 shows the main features of Hive.

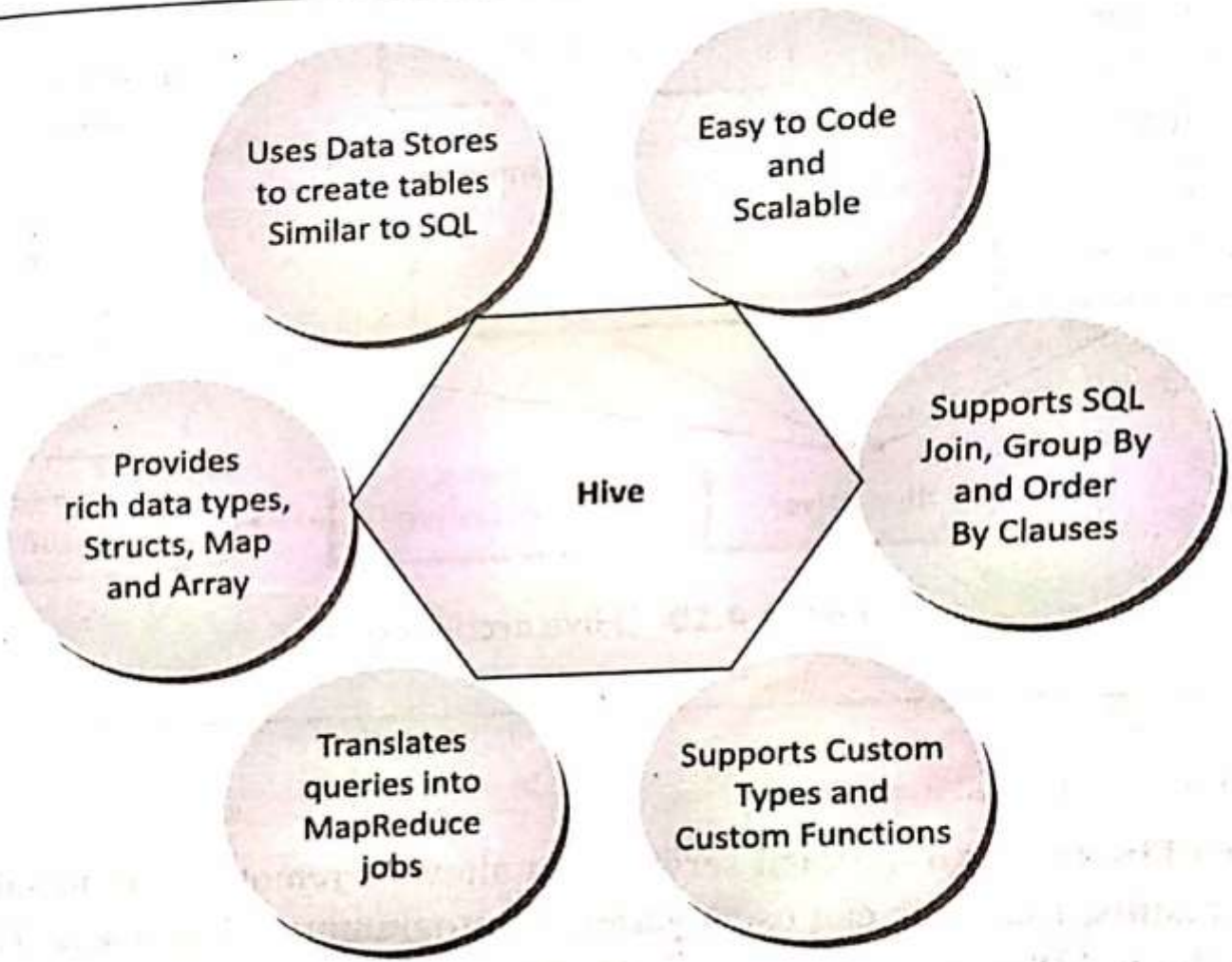


Figure 4.9 Main features of Hive

Hive Characteristics

1. Has the capability to translate queries into MapReduce jobs. This makes Hive scalable, able to handle data warehouse applications, and therefore, suitable for the analysis of static data of an extremely large size, where the fast response-time is not a criterion.
2. Supports web interfaces as well. Application APIs as well as web-browser clients, can access the Hive DB server.
3. Provides an SQL dialect (Hive Query Language, abbreviated HiveQL or HQL).

Limitations of Hive

Hive is:

1. Not a full database. Main disadvantage is that Hive does not provide update, alter and deletion of records in the database.
2. Not developed for unstructured data.
3. Not designed for real-time queries.
4. Performs the partition always from the last column.

Hive Architecture

4.4.1 Hive Architecture

Figure 4.10 shows the Hive architecture.

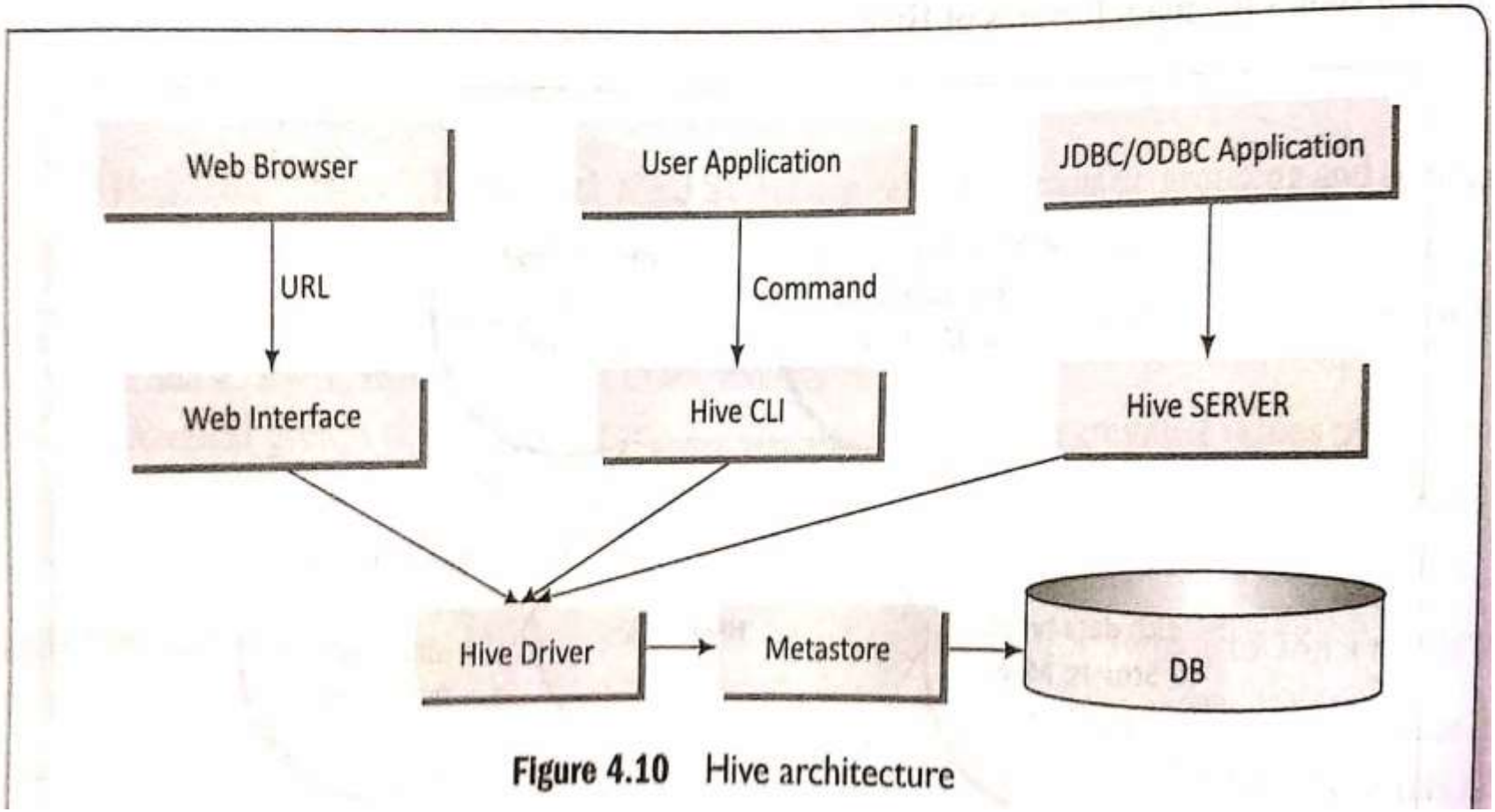


Figure 4.10 Hive architecture

Components of Hive architecture are:

- **Hive Server (Thrift) -** An optional service that allows a remote client to submit requests to Hive and retrieve results. Requests can use a variety of programming languages. Thrift Server exposes a very simple client API to execute HiveQL statements.
- **Hive CLI (Command Line Interface**)- Popular interface to interact with Hive. Hive runs in local mode that uses local storage when running the CLI on a Hadoop cluster instead of HDFS.

- **Web Interface -** Hive can be accessed using a web browser as well. This requires a HWI Server running on some designated code. The URL ***http:// hadoop:<port no> / hwi*** command can be used to access Hive through the web.
- **Metastore-** It is the system catalog. All other components of Hive interact with the Metastore. It stores the schema or metadata of tables, databases, columns in a table, their data types and HDFS mapping.
- **Hive Driver -** It manages the life cycle of a HiveQL statement during compilation, optimization and execution.

4.4.2 Hive Installation

Hive can be installed on Windows 10, Ubuntu 16.04 and MySQL. It requires three software packages:

- Java Development kit for Java compiler (Javac) and interpreter
- Hadoop
- Compatible version of Hive with Java– Hive 1.2 onward supports Java 1.7 or newer.

Steps for installation of Hive in a Linux based OS are as follows:

1. Install Javac and Java from Oracle Java download site. Download jdk 7 or a later version from http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html, and extract the compressed file.
   All users can access Java by Make java available to all users. The user has to move it to the location "/usr/local/" using the required commands
2. Set the path by the commands for jdk1.7.0_71, export JAVA_HOME=/usr/local/jdk1.7.0_71, export PATH=$PATH: $JAVA_HOME/bin
   (Can use alternative install /usr/bin/java usr/local/java/bin/java 2)
3. Install Hadoop http://apache.claz.org/hadoop/common/hadoop-2.4.1/
4. Make shared HADOOP, MAPRED, COMMON, HDFS and all related files, configure HADOOP and set property such as replication parameter.
5. Name the yarn.nodemanager.aux-services. Assign value to mapreduce_shuffle. Set namenode and datanode paths.
6. Download http://apache.petsads.us/hive/hive-0.14.0/. Use ls command to verify the files $ tar zxvf apache-hive-0.14.0-bin.tar.gz, $ ls
   OR
   Hive archive also extracts by the command apache-hive-0.14.0-bin apache-hive-0.14.0-bin.tar.gz. , $ cd $HIVE_HOME/conf, $ cp hive-env.sh.template hive-env.sh, export HADOOP_HOME=/usr/local/hadoop
7. Use an external database server. Configure metastore for the server.

Comparison with RDBMS (Traditional Database)

- Hive is a DB system which defines databases and tables. Hive analyzes structured data in DB. Hive has certain differences with RDBMS.
- Table 4.3 gives a comparison of Hive database characteristics with RDBMS.

Table 4.3 Comparison of Hive database characteristics with RDBMS

Characteristics	Hive	RDBMS
Record level queries	No Update and Delete	Insert, Update and Delete
Transaction support	No	Yes
Latency	Minutes or more	In fractions of a second
Data size	Petabytes	Terabytes
Data per query	Petabytes	Gigabytes
Query language	HiveQL	SQL
Support JDBC/ODBC	Limited	Full

Hive Data Types and File Formats

- Hive defines various primitive, complex, string, date/time, collection data types and file formats for handling and storing different data formats.
- Table 4.4 gives primitive, string, date/time and complex Hive data types and their descriptions.

Table 4.4 Hive data types and their descriptions

Data Type Name	Description
TINYINT	1 byte signed integer. Postfix letter is Y.
SMALLINT	2 byte signed integer. Postfix letter is S.
INT	4 byte signed integer
BIGINT	8 byte signed integer. Postfix letter is L.
FLOAT	4 byte single-precision floating-point number
DOUBLE	8 byte double-precision floating-point number
BOOLEAN	True or False
TIMESTAMP	UNIX timestamp with optional nanosecond precision. It supports java.sql.Timestamp format "YYYY-MM-DD HH:MM:SS.fffffffff"
DATE	YYYY-MM-DD format
VARCHAR	1 to 65355 bytes. Use single quotes (' ') or double quotes (" ")
CHAR	255 bytes
DECIMAL	Used for representing immutable arbitrary precision. DECIMAL (precision, scale) format
UNION	Collection of heterogeneous data types. `Create union`
NULL	Missing values representation

Table 4.5 gives Hive three Collection data types and their descriptions.

Table 4.5 Collection data-types and their descriptions

Name	Description
STRUCT	Similar to 'C' struc, a collection of fields of different data types. An access to field uses dot notation. For example, struct ('a', 'b')
MAP	A collection of key-value pairs. Fields access using [] notation. For example, map ('key1', 'a', 'key2', 'b')
ARRAY	Ordered sequence of same types. Accesses to fields using array index. For example, array ('a', 'b')

Table 4.6 gives the file formats and their descriptions.

Table 4.6 File formats and their descriptions

File Format	Description
Text file	The default file format, and a line represents a record. The delimiting characters separate the lines. Text file examples are CSV, TSV, JSON and XML (Section 3.3.2).
Sequential file	Flat file which stores binary key-value pairs, and supports compression.
RCFile	Record Columnar file (Section 3.3.3.3).
ORCFILE	ORC stands for Optimized Row Columnar which means it can store data in an optimized way than in the other file formats (Section 3.3.3.4).

Record columnar file means one that can be partitioned in rows and then partitioned with columns. Partitioning in this way enables serialization.

4.4.5 Hive Data Model

Table 4.7 below gives three components of Hive data model and their descriptions.

Table 4.7 Components (also called data units) of Hive Data Model

Name	Description
Database	Namespace for tables
Tables	Similar to tables in RDBMS Support filter, projection, join and union operations The table data stores in a directory in HDFS
Partitions	Table can have one or more partition keys that tell how the data stores
Buckets	Data in each partition further divides into buckets based on hash of a column in the table. Stored as a file in the partition directory.

4.4.6 Hive Integration and Workflow Steps

Hive integrates with the MapReduce and HDFS. Figure 4.11 shows the dataflow sequences and workflow steps between Hive and Hadoop.

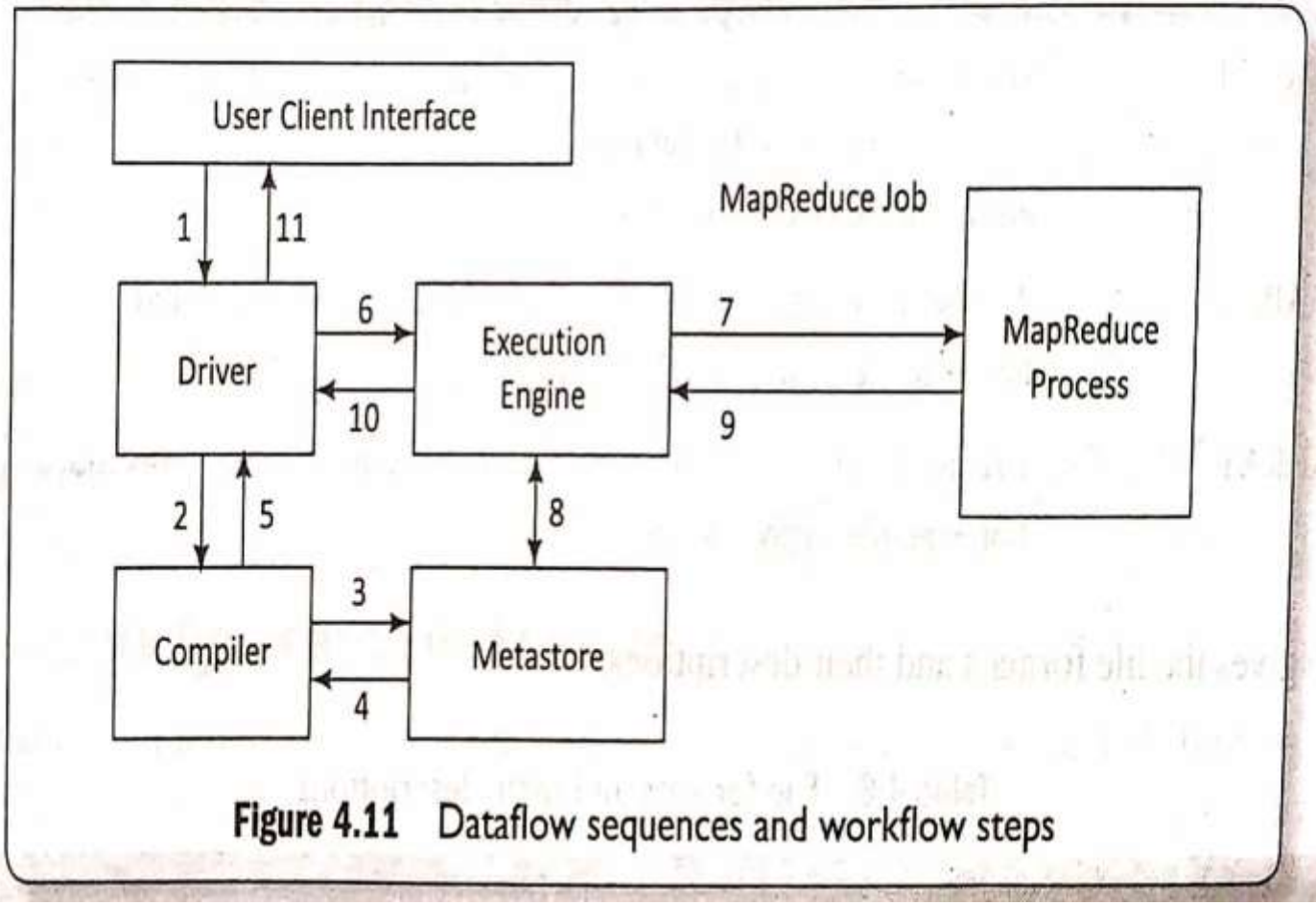


Figure 4.11 Dataflow sequences and workflow steps

Steps 1 to 11 are as follows:

STEP No.	OPERATION
1	**Execute Query**: Hive interface (CLI or Web Interface) sends a query to Database Driver to execute the query.
2	**Get Plan**: Driver sends the query to query compiler that parses the query to check the syntax and query plan or the requirement of the query.
3	**Get Metadata**: Compiler sends metadata request to Metastore (of any database, such as MySQL).
4	**Send Metadata**: Metastore sends metadata as a response to compiler.
5	**Send Plan:** Compiler checks the requirement and resends the plan to driver. The parsing and compiling of the query is complete at this place.
6	**Execute Plan:** Driver sends the execute plan to execution engine.
7	**Execute Job:** Internally, the process of execution job is a MapReduce job. The execution engine sends the job to JobTracker, which is in Name node and it assigns this job to TaskTracker, which is in Data node. Then, the query executes the job.
8	**Metadata Operations:** Meanwhile the execution engine can execute the metadata operations with Metastore.
9	**Fetch Result:** Execution engine receives the results from Data nodes.
10	**Send Results:** Execution engine sends the result to Driver.
11	**Send Results:** Driver sends the results to Hive Interfaces.

Hive Built-in Functions

- Hive supports a number of built-in functions. Table 4.8 gives the return types, syntax and descriptions of the examples of these functions.
- **Table 4.8 shows the Return types, syntax, and descriptions of the functions**

Table 4.8 Return types, syntax, and descriptions of the functions

Return Type	Syntax	Description
BIGINT	round(double a)	Returns the rounded BIGINT (8 Byte integer) value of the 8 Byte double-precision floating point number a
BIGINT	floor(double a)	Returns the maximum BIGINT value that is equal to or less than the double.
BIGINT	ceil(double a)	Returns the minimum BIGINT value that is equal to or greater than the double.
double	rand(), rand(int seed)	Returns a random number (double) that distributes uniformly from 0 to 1 and that changes in each row. Integer seed ensured that random number sequence is deterministic.
string	concate(string str1, string str2, ...)	Returns the string resulting from concatenating str1 with str2, ...
string	substr(string str, int start)	Returns the substring of str starting from a start position till the end of string str.
string	substr(string str, int start, int length)	Returns the substring of str starting from the start position with the given length.
string	upper(string str), ucase (string str)	Returns the string resulting from converting all characters of str to upper case.
string	lower(string str), lcase(string str)	Returns the string resulting from converting all characters of str to lower case.
string	trim(string str)	Returns the string resulting from trimming spaces from both ends. trim ('12A34 56') returns '12A3456'
string	ltrim(string str); rtrim(string str)	Returns the string resulting from trimming spaces (only one end, left or right hand side or right-handside spaces trimmed). ltrim('12A34 56') returns '12A3456' and rtrim(' 12A34 56 ') returns '12A3456'.
string	rtrim(string str)	Returns the string resulting from trimming spaces from the end (right hand side) of str.
int	year(string date)	Returns the year part of a date or a timestamp string.
int	month(string date)	Returns the month part of a date or a timestamp string.
int	day(string date)	Returns the day part of a date or a timestamp string.

HiveQL

- Hive Query Language (abbreviated HiveQL) is for querying the large datasets which reside in the HDFS environment.
- HiveQL script commands enable data definition, data manipulation and query processing.
- HiveQL supports a large base of SQL users who are acquainted with SQL to extract information from data warehouses.

HiveQL Process Engine	HiveQL is similar to SQL for querying on schema information at the Metastore. It is one of the replacements of traditional approach for MapReduce program. Instead of writing MapReduce program in Java, we can write a query for MapReduce job and process it.
Execution Engine	The bridge between HiveQL process Engine and MapReduce is Hive Execution Engine. Execution engine processes the query and generates results same as MapReduce results. It uses the flavor of MapReduce.

The subsections ahead give the details of data definition, data manipulation and querying data examples.

4.5.1 HiveQL Data Definition Language (DDL)

HiveQL database commands for data definition for DBs and Tables are CREATE DATABASE, SHOW DATABASE (list of all DBs), CREATE SCHEMA, CREATE TABLE. Following are HiveQL commands which create a table:

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [<database name>.] <table name>
[(<column name> <data type> [COMMENT <column comment>], ...)]
[COMMENT <table comment>]
[ROW FORMAT <row format>]
[STORED AS <file format>]
```

Table 4.9 Hive Table Row Formats

DELIMITED	Specifies a delimiter at the table level for structured fields. This is default. Syntax: FIELDS TERMINATED BY, LINES TERMINATED BY
SERDE	Stands for Serializer/Deserializer. SYNTAX: SERDE 'serde.class.name'

HiveQL database commands for data definition for the DBs and Tables are CREATE DATABASE, SHOW DATABASE (list of all DBs), CREATE SCHEMA, CREATE TABLE.

The following example uses HiveQL commands to create a database `toys_companyDB`.

EXAMPLE 4.7

How do you create a database named toys_companyDB and table named toys_tbl?

SOLUTION

```
$HIVE_HOME/binhive – service cli
hive>set hive.cli.print.current.db=true;
hive> CREATE DATABASE toys_companyDB
hive>USE toys_companyDB
hive (toys_companyDB)> CREATE TABLE toys_tbl (
>puzzle_code STRING,
>pieces SMALLINT
>cost FLOAT);
hive (toys_company)> quit;
&ls/home/binadmin/Hive/warehouse/toys_companyDB.db
```

The following example uses the command CREATE TABLE to create a table toy_products.

EXAMPLE 4.8

How do you create a table toy_products with the following fields?

Field	Data type
ProductCategory	string
ProductId	int
ProductName	string
ProductPrice	float

SOLUTION

```
CREATE TABLE IF NOT EXISTS toy_products (ProductCategory String, ProductId int, ProductName
String, ProductPrice float)
  COMMENT 'Toy details'
  ROW FORMAT DELIMITED
  FIELDS TERMINATED BY '\t'
  LINES TERMINATED BY '\n'
  STORED AS TEXTFILE;
```

The option IF NOT EXISTS, Hive ignores the statement in case the table already exists.

Consider the following command:

A command is

```
CREATE DATABASE|SCHEMA [IF NOT EXISTS] <database name>;
```

IF NOT EXISTS is an optional clause. The clause notifies the user that a database with the same name already exists. SCHEMA can be also created in place of DATABASE using this command

A command is written to get the list of all existing databases.

```
SHOW DATABASES;
```

A command is written to delete an existing database.

```
DROP (DATABASE|SCHEMA) [IF EXISTS] <database name> [RESTRICT|CASCADE];
```

EXAMPLE 4.9

Give examples of usages of database commands for CREATE, SHOW and DROP.

SOLUTION

```
CREATE DATABASE IF NOT EXISTS toys_companyDB;
SHOW DATABASES;
default toys_companyDB
```

*Default database is test.

```
Delete dtabase using the command:
Drop Database toys_companyDB.
```

4.5.2 HiveQL Data Manipulation Language (DML)

HiveQL commands for data manipulation are USE <database name>, DROP DATABASE, DROP SCHEMA, ALTER TABLE, DROP TABLE, and LOAD DATA.

The following is a command for inserting (loading) data into the Hive DBs.

```
LOAD DATA [LOCAL] INPATH '<file path>' [OVERWRITE] INTO TABLE <table
name> [PARTITION (partcol1=val1, partcol2=val2 ...)]
```

LOCAL is an identifier to specify the local path. It is optional. OVERWRITE is optional to overwrite the data in the table. PARTITION is optional. val1 is value assigned to partition column 1 (partcol1) and val2 is value assigned to partition column 2 (partcol2).

Command	Functionality	Script Example
LOAD DATA	Insert data in a table	LOAD DATA LOCAL INPATH '/home/user/ jigsaw_puzzle_info.txt' OVERWRITE INTO TABLE toy_tbl;

The following is an example for usages of data manipulation commands, INSERT, ALTER, and DROP.

EXAMPLE 4.10

Consider an example of a toy company selling Jigsaws. Consider a text file named jigsaw_puzzle_info.txt in */home/user* directory. The file is text file with four fields: Toy-category, toy-id, toy-name, and Price in US$ as follows:

```
puzzle_Garden 10725 Fantasy 1.35
puzzle_Jungle 31047 Animals 2.85
puzzle_School 81049 Nursery 4.45
```

How will you use (i) LOAD (insert), (ii) ALTER and (iii) DROP commands?

SOLUTION

(i) Insert the data of this file into a table using the following commands:

```
LOAD DATA LOCAL INPATH '/home/user/ jigsaw_puzzle_info.txt'
OVERWRITE INTO TABLE jigsaw_puzzle;
```

(ii) Alter the table using the following commands:

```
ALTER TABLE <name> RENAME TO <new name>
ALTER TABLE <name> ADD COLUMNS (<col spec> [, <col spec> ...])
ALTER TABLE <name> DROP [COLUMN] <column name>
ALTER TABLE <name> CHANGE <column name> <new name> <new type>
ALTER TABLE <name> REPLACE COLUMNS (<col spec> [, <col spec> ...])
```

The following query renames the table from jigsaw_puzzle to toy_tbl:

```
ALTER TABLE jigsaw_puzzle RENAME TO toy_tbl;
```

The following query renames the column name ProductCategory to ProductCat:

```
ALTER TABLE toy_tbl CHANGE ProductCategory ProductCat String;
```

The following query renames data type of ProductPrice from float to double:

```
ALTER TABLE toy_tbl CHANGE ProductPrice ProductPrice Double;
```

The following query adds a column named ProductDesc to the toy_tbl table:

```
ALTER TABLE toy_tbl ADD COLUMNS (ProductDesc String COMMENT 'Product Description');
```

The following query deletes all the columns from the toy_tbl table and replaces it with ProdCat and ProdName columns:

```
ALTER TABLE toy_tbl REPLACE COLUMNS (ProductCategory INT ProdCat Int, ProductName
STRING ProdName String);
```

(iii) The following query deletes a column named ProductDesc from the toy_tbl table:

```
ALTER TABLE toy_tbl DROP COLUMN ProductDesc;
```

A table DROP using the following command: DROP TABLE [IF EXISTS] table_name;

The following query drops a table named jigsaw_puzzle:

```
DROP TABLE IF EXISTS jigsaw_puzzle;
```

HiveQL For Querying the Data

- Partitioning and storing are the requirements. A data warehouse should have a large number of partitions where the tables, files and databases store. Querying then requires sorting, aggregating and joining functions.

- Querying the data is to SELECT a specific entity *satisfying a condition, having* presence of an entity or selecting specific entity using GroupBy .

Querying the data is to SELECT a specific entity *satisfying* a condition, *having* presence of an entity or selecting specific entity using GroupBy .

```
SELECT [ALL | DISTINCT] <select expression>, <select expression>, ...
FROM <table name>
[WHERE <where condition>]
[GROUP BY <column List>]
[HAVING <having condition>]
[CLUSTER BY <column List>| [DISTRIBUTE BY <column List>] [SORT BY <column
List>]]
[LIMIT number];
```

Partitioning

- Hive organizes tables into partitions. Table partitioning refers to dividing the table data into some parts based on the values of particular set of columns.
- Partition makes querying easy and fast. This is because SELECT is then from the smaller number of column fields.
- Section 3.3.3.3 described RC columnar format and serialized records.
- The following example explains the concept of partitioning, columnar and file records formats.

EXAMPLE 4.11

Consider a table T with eight-column and four-row table. Partition the table, convert in RC columnar format and serialize.

SOLUTION

Firstly, divide the table in four parts, t_{T1}, t_{T2}, t_{T3} and t_{T4} horizontally row-wise. Each sub-table has one row and eight columns. Now, convert each sub-table t_{T1}, t_{T2}, t_{T3} and t_{T4} into columnar format, or RC File records [Recall Example 3.7 on how RC file saves each row-group data in a format using SERDE (serializer/des-serializer)].

Each sub-table has eight rows and one column. Each column can serially send data one value at an instance. A column has eight key-value pairs with the same key for all the eight.

Table Partitioning

Create a table with Partition using command:

```
CREATE [EXTERNAL] TABLE <table name> (<column name 1> <data type 1>,
……)
PARTITIONED BY (<column name n> <data type n> [COMMENT <column comment>],
...);
```

Rename a Partition in the existing Table using the following command:

```
ALTER TABLE <table name> PARTITION partition_spec RENAME TO PARTITION
partition_spec;
```

Add a Partition in the existing Table using the following command:

```
ALTER TABLE <table name> ADD [IF NOT EXISTS] PARTITION partition_spec
[LOCATION 'location1'] partition_spec [LOCATION 'location2'] ...;
```

```
partition_spec: (p_column = p_col_value, p_column = p_col_value, ...)
```

Drop a Partition in the existing Table using the following command:

```
ALTER TABLE <table name> DROP [IF EXISTS] PARTITION partition_spec, PARTITION
partition_spec;
```

EXAMPLE 4.12

How will you add, rename and drop a partition to a table, toys_tbl?

SOLUTION

(i) Add a partition to the existing toy table using the command:

```
ALTER TABLE toy_tbl ADD PARTITION (category='Toy_Airplane') location
'/Toy_Airplane/partAirplane';
```

(ii) The following query renames a partition:

```
ALTER TABLE toy_tbl PARTITION (category='Toy_Airplane') RENAME TO
PARTITION (name='Fighter');
```

(iii) Drop a Partition in the existing Table using the command:

```
ALTER TABLE toy_tbl DROP [IF EXISTS] PARTITION (category='Toy_
Airplane');
```

- The following example **4.13** explains how querying is facilitated by using partitioning of a table.
- A query processes faster when using partition. Selection of a product of a specific category from a table during query processing takes lesser time when the table has a partition based on a category.

EXAMPLE 4.13

Assume that following file contains toys_tbl.

```
/table/toy_tbl/file1
Category, id, name, price
Toy_Airplane, 10725, Lost Temple, 1.25
Toy_Airplane, 31047, Propeller Plane, 2.10
Toy_Airplane, 31049, Twin Spin Helicopter, 3.45
Toy_Train, 31054, Blue Express, 4.25
Toy_Train, 10254, Winter Holiday Toy_Train, 2.75
```

A table *toy_tbl* contains many values for categories of toys. Query is required to identify all toy_airplane fields. Give reasons why partitioning reduces query processing time.

SOLUTION

Here, a table named toy_tbl contains several toy details (category, id, name and price). Suppose it is required to identify all the airplanes. A query searches the whole table for the required information. However, if a partition is created on the toy_tbl, based on category and stores it in a separate file, then it will reduce the query processing time.

Let the data partitions into two files, file 2 and file 3, using category.

```
/table/toys/toy_airplane/file2
toy_airplane, 10725, Lost Temple, TP, 1.25
toy_airplane, 31047, Propeller Plane, 2.10
toy_airplane, 31049, Lost Temple, 3.45
/table/toys/toy_train/file3
Toy_Train, 31054, Blue Express, 4.25
Toy_Train, 10254, Winter Holiday Toy_Train, 2.75
```

hive queries examples

- https://www.edureka.co/blog/hive-commands-with-examples

Advantages of Partition

1. Distributes execution load horizontally.
2. Query response time becomes faster when processing a small part of the data instead of searching the entire dataset.

Limitations of Partition

1. Creating a large number of partitions in a table leads to a large number of files and directories in HDFS, which is an overhead to NameNode, since it must keep all metadata for the file system in memory only.
2. Partitions may optimize some queries based on Where clauses, but they may be less responsive for other important queries on grouping clauses.

Limitations of Partition

3. A large number of partitions will lead to a large number of tasks (which will run in separate JVM) in each MapReduce job, thus creating a **lot of overhead in maintaining** JVM start up and tear down (A separate task will be used for each file). The overhead of JVM start up and tear down can exceed the actual processing time in the worst case.

Bucketing

A partition itself may have a large number of columns when tables are very large. Tables or partitions can be sub-divided into buckets. Division is based on the hash of a column in the table.

Consider bucketed column C_{bucket_i}. First, define a hash_function **H**() according to type of the bucketed column. Let the total number of buckets = $N_{buckets}$. Let C_{bucket_i} denote i^{th} bucketed column. The hash value h_i = hashing function $\mathbf{H}(C_{bucket\_i})$ *mod* $(N_{buckets})$.

Buckets provide an extra structure to the data that can lead to more efficient query processing when compared to undivided tables or partition. Buckets store as a file in the partition directory. Records with the same bucketed column will always be stored in the same bucket. Records kept in each bucket provide sorting ease and enable Map task Joins. A Bucket can also be used as a sample dataset.

CLUSTERED BY clause divides a table into buckets. A coding example on Buckets is given below:

EXAMPLE 4.14

A table *toy_tbl* contains many values for categories of toys. Assume the number of buckets to be created = 5. Assume a table for `Toy_Airplane` of product code 10725.

1. How will the bucketing enforce?
2. How will the bucketed table partition `toy_airplane_10725` create five buckets?
3. How will the bucket column load into `toy_tbl`?
4. How will the bucket data display?

SOLUTION

#Enforce bucketing

```
set hive.enforce.bucketing=true;
```

#Create bucketed Table for toy_airplane of product code 10725 and create cluster of 5 buckets

```
CREATE TABLE IF NOT EXISTS toy_airplane_10725(ProductCategory
STRING, ProductId INT, ProductName STRING, PrdocutMfgDate YYYY-MM-
DD, ProductPrice_US$ FLOAT) CLUSTERED BY (Price) into 5 buckets;
```

Load data to bucketed table.

```
FROM toy_airplane_10725 INSERT OVERWRITE TABLE toy_tbl SELECT
ProductCategory, ProductId, ProductName, PrdocutMfgDate,
ProductPrice;
```

- To display the contents for Price_US$ selected for the ProductId from the second bucket.

```
SELECT DISTINCT ProductId FROM toy_tbl_buckets TABLE FOR
10725(BUCKET 2 OUT OF 5 ON Price_US$);
```

Views

- A program uses functions or objects. Constructing an object instance enables layered design and encapsulating the complexity due to methods and fields.
- Similarly, Views provide ease of programming. Complex queries simplify using reusable Views. A HiveQL View is a logical construct.

A View provisions the following:

- Saves the query and reduces the query complexity
- Use a View like a table but a View does not store data like a table
- Hive query statement when uses references to a view, the Hive executes the View and then the planner combines the information in View definition with the remaining actions on the query (Hive has a query planner, which plans how a query breaks into sub-queries for obtaining the right answer.)
- Hides the complexity by dividing the query into smaller, more manageable pieces.

Sub-Queries (Using Views)

Consider the following query with a nested sub-query.

EXAMPLE 4.15

A table *toy_tbl* contains many values for categories of toys. Assume a table for Toy_Airplane of product code 10725. Consider a nested query:

```
FROM (
SELECT * toy_tbl_Join people JOIN Toy_Airplane
    ON (Toy_Airplane.ProductId= productId.id) WHERE productId=10725
    ) toys_ catalog SELECT prdocutMfgDate WHERE prdocutMfgDate = '2017-10-23';
```

Create a View for using that in a nested query.

SOLUTION

```
# create a view named toy_tbl_MiniJoin
CREATE VIEW toy_tbl_MiniJoin AS
    SELECT * toy_tbl_Join people JOIN Toy_Airplane
    ON (Toy_Airplane.ProductId= productId.id) WHERE productId=10725
    ) toys_ catalog SELECT prdocutMfgDate WHERE prdocutMfgDate = '2017-10-23';
```

Aggregation

- Hive supports the following built-in aggregation functions. The usage of these functions is same as the SQL aggregate functions.
- **Table 4.10** lists the functions, their syntax and descriptions.

Table 4.10 Aggregate functions, their return type, syntax and descriptions

Return Type	Syntax	Description
BIGINT	count(*), count(expr)	Returns the total number of retrieved rows.
DOUBLE	sum(col), sum(DISTINCT col)	Returns the sum of the elements in the group or the sum of the distinct values of the column in the group.
DOUBLE	avg (col), avg (DISTINCT col)	Returns the average of the elements in the group or the average of the distinct values of the column in the group.
DOUBLE	min (col)	Returns the minimum value of the column in the group.
DOUBLE	max(col)	Returns the maximum value of the column in the group.

Usage examples are:

Example: SELECT ProductCategory, count (*) FROM toy_tbl GROUP BY ProductCategory;

Example: SELECT ProductCategory, sum(ProductPrice) FROM toy_tbl GROUP BY ProductCategory;

Join

- A JOIN clause combines columns of two or more tables, based on a relation between them. HiveQLJoin is more or less similar to SQL JOINS. Following uses of two tables show the Join operations.
- Table 4.11 gives an example of a table named toy_tbl of Product categories, Productid and Product name.

4.5.5 Join

A JOIN clause combines columns of two or more tables, based on a relation between them. HiveQL Join is more or less similar to SQL JOINS. Following uses of two tables show the Join operations.

Table 4.11 gives an example of a table named *toy_tbl* of Product categories, ProductId and Product name.

Table 4.11 Table of Product categories, Product Id and Product name

ProductCategory	ProductId	ProductName
Toy_Airplane	10725	Lost temple
Toy_Airplane	31047	Propeller plane
Toy_Airplane	31049	Twin spin helicopter
Toy_Train	31054	Blue express
Toy_Train	10254	Winter holiday Toy_Train

Table 4.12 Table of ID and Product Cost

Id	ProductPrice
10725	100.0
31047	200.0
31049	300.0
31054	450.0
10254	200.0

Different types of joins are follows:

- **JOIN**
- **LEFT OUTER JOIN**
- **RIGHT OUTER JOIN**
- **FULL OUTER JOIN**

JOIN

- Join clause combines and retrieves the records from multiple tables.
- Join is the same as OUTER JOIN in SQL. A JOIN condition uses primary keys and foreign keys of the tables.

SELECT t.Productid, t.ProductName, p.ProductPrice FROM toy_tbl t JOIN price p ON (t.Productid = p.Id);

LEFT OUTER JOIN

- **A LEFT JOIN** returns all the values from the left table, plus the matched values from the right table, or NULL in case of no matching JOIN predicate.
- ***SELECT t.Productid, t.ProductName, p.ProductPrice FROM toy_tbl t LEFT OUTER JOIN price p ON (t.Productid = p.Id);***

- **RIGHT OUTER JOIN**
- A **RIGHT JOIN** returns all the values from the right table, plus the matched values from the left table, or NULL in case of no matching join predicate.
- **SELECT t.Productid, t.ProductName, p.ProductPrice FROM toy_tbl t RIGHT OUTER JOIN price p ON (t.Productid = p.Id);**

FULL OUTER JOIN

- **HiveQL FULL OUTER JOIN** combines the records of both the left and the right outer tables that fulfill the JOIN condition. The joined table contains either all the records from both the tables, or fills in NULL values for missing matches on either side.
- ***SELECT t.Productid, t.ProductName, p.ProductPrice FROM toy_tbl t FULL OUTER JOIN price p ON (t.Productid = p.Id);***

Group by Clause

- GROUP BY, HAVING,ORDER BY DISTRIBUTEBY, CLUSTER BY are HiveQL clauses. An example of using the clauses is given below:

EXAMPLE 4.16

How do SELECT statement uses GROUP BY, HAVING, DISTRIBUTE BY, CLUSTER BY? How does clause GROUP BY used in queries on toy_tbl?

SOLUTION

(i) Use of SELECT statement with WHERE clause is as follows:

```
SELECT [ALL | DISTINCT] <select expression>, <select expression>, ...
FROM <table name>
[WHERE <where condition>]
[GROUP BY <column List>]
[HAVING <having condition>]
[CLUSTER BY <column List>| [DISTRIBUTE BY <column List>] [SORT BY
<column List>]]
[LIMIT number];
```

(ii) Use of the clauses in queries to toy_tbl is as follows:

```
SELECT * FROM toy WHERE ProductPrice > 1.5;
SELECT ProductCategory, count (*) FROM toy_tbl GROUP BY
ProductCategory;
SELECT ProductCategory, sum(ProductPrice) FROM toy_tbl GROUP BY
ProductCategory;
```

HiveQL - Select-Joins

- https://www.tutorialspoint.com/hive/hiveql_joins.htm

PIG

Apache developed Pig, which:

- Is an abstraction over MapReduce
- Is an execution framework for parallel processing
- Reduces the complexities of writing a MapReduce program
- Is a high-level dataflow language. Dataflow language means that a Pig operation node takes the inputs and generates the output for the next node
- Is mostly used in HDFS environment
- Performs data manipulation operations at files at data nodes in Hadoop.

Applications of Apache Pig

Applications of Pig are:

- Analyzing large datasets
- Executing tasks involving adhoc processing
- Processing large data sources such as web logs and streaming online data
- Data processing for search platforms. Pig processes different types of data
- Processing time sensitive data loads; data extracts and analyzes quickly .

  For example, analysis of data from twitter to find patterns for user behavior and recommendations.

Features of PIG

1. Apache PIG helps programmers write complex data transformations using scripts (without using Java). Pig Latin language is very similar to SQL and possess a rich set of built-in operators, such as group.join, filter, limit, order by, parallel, sort and split.
2. Creates user defined functions (UDFs) to write custom functions which are not available in Pig. A UDF can be in other programming languages, such as Java, Python, Ruby, Jython,]Ruby. They easily embed into Pig scripts written in Pig Latin. UDFs provide extensibility to the Pig.
3. Process any kind of data, structured, semi-structured or unstructured data, coming from various sources.
4. (Reduces the length of codes using multi-query approach. Pig code of 10 lines is equal to MapReduce code of 200 lines. Thus, the processing is very fast.
5. Handles inconsistent schema in case of unstructured data as well.
6. (vi) Extracts the data, performs operations on that data and dumps the data in the required format in HDFS. The operation is called ETL (Extract Transform Load).
7. Performs automatic optimization of tasks before execution.
8. Programmers and developers can concentrate on the whole operation without a need to create mapper and reducer tasks separately.

Features of PIG (contd..)

9. Reads the input data files from HDFS or the data files from other sources such as local file system, stores the intermediate data and writes back the output in HDFS.
10. Pig characteristics are data reading, processing, programming the UDFs in multiple languages and programming multiple queries by fewer codes. This causes fast processing.
11. Pig derives guidance from four philosophies, live anywhere, take anything, domestic and run as if flying. This justifies the name Pig, as the animal pig also has these characteristics.

Table 4.13 Differences between Pig and MapReduce

Pig	MapReduce
A dataflow language	A data processing paradigm
High level language and flexible	Low level language and rigid
Performing Join, filter, sorting or ordering operations are quite simple	Relatively difficult to perform Join, filter, sorting or ordering operations between datasets
Programmer with a basic knowledge of SQL can work conveniently	Complex Java implementations require exposure to Java language
Uses multi-query approach, thereby reducing the length of the codes significantly	Require almost 20 times more the number of lines to perform the same task
No need for compilation for execution; operators convert internally into MapReduce jobs	Long compilation process for Jobs
Provides nested data types like tuples, bags and maps	No such data types

Table 4.14 gives differences between Pig and SQL.

Table 4.14 Differences between Pig and SQL

Pig	SQL
Pig Latin is a procedural language	A declarative language
Schema is optional, stores data without assigning a schema	Schema is mandatory
Nested relational data model	Flat relational data model
Provides limited opportunity for Query optimization	More opportunity for query optimization

Table 4.15 Differences between Pig and Hive

Pig	Hive
Originally created at Yahoo	Originally created at Facebook
Exploits Pig Latin language	Exploits HiveQL
Pig Latin is a dataflow language	HiveQL is a query processing language
Pig Latin is a procedural language and it fits in pipeline paradigm	HiveQL is a declarative language
Handles structured, unstructured and semi-structured data	Mostly used for structured data

Pig Architecture

- Firstly, Pig Latin scripts submit to the Apache Pig Execution Engine.
- Figure 4.12 shows Pig architecture for scripts dataflow and processing in the HDFS environment.

Pig Architecture

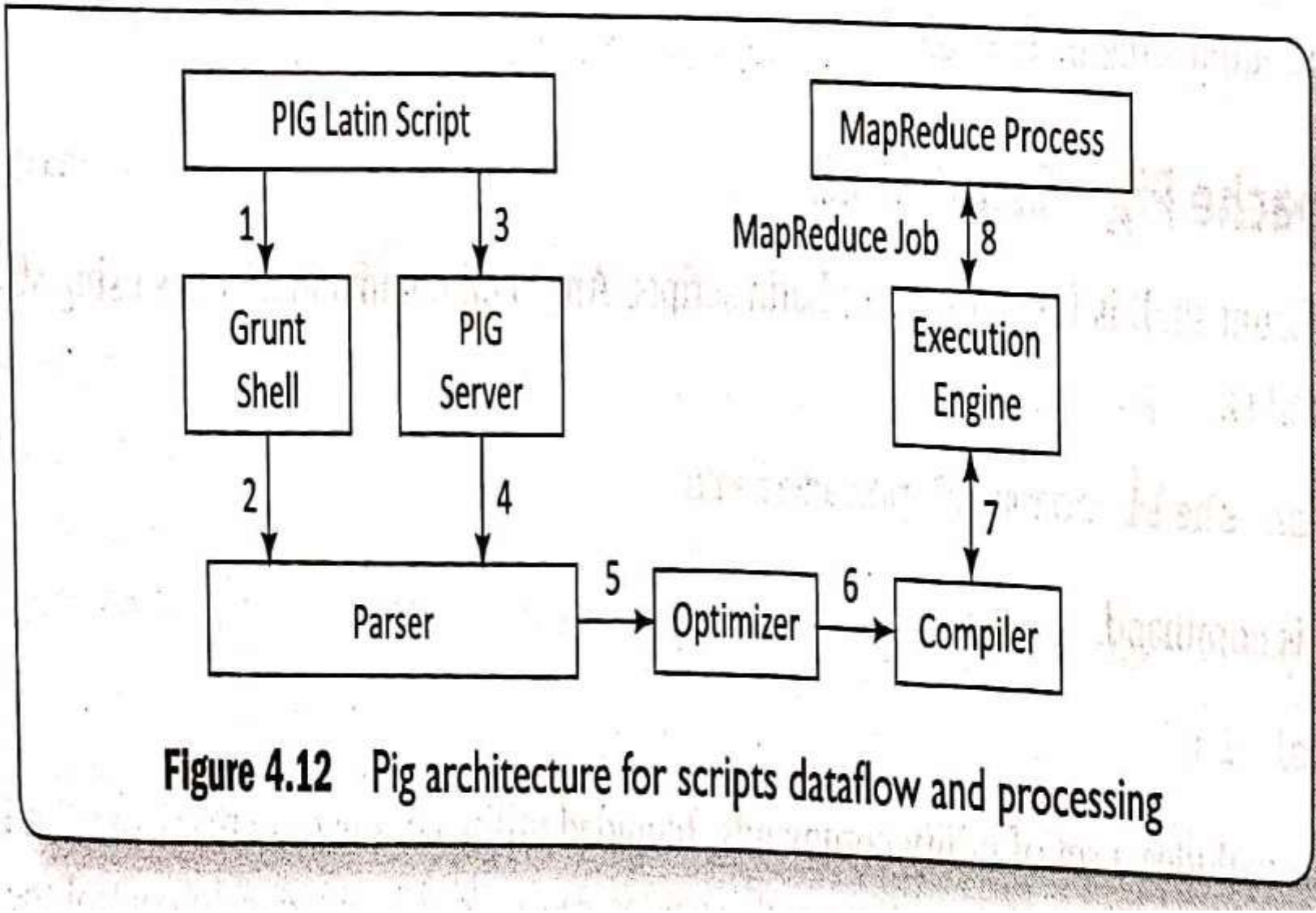


Figure 4.12 Pig architecture for scripts dataflow and processing

The three ways to execute scripts are:

1. **Grunt Shell:** An interactive shell of Pig that executes the scripts.
2. **Script File:** Pig commands written in a script file that execute at Pig Server.
3. **Embedded Script:** Create UDFs for the functions unavailable in Pig built-in operators. UDF can be in other programming languages. The UDFs can embed in Pig Latin Script file.

Pig Architecture(contd..)

Parser

- A parser handles Pig scripts after passing through Grunt or Pig Server.
- The Parser performs type checking and checks the script syntax. The output is a Directed Acyclic Graph (DAG). Acylic means only one set of inputs are simultaneously at a node, and only one set of output generates after node operations. DAG represents the Pig Latin statements and logical operators.
- Nodes represent the logical operators. Edges between sequentially traversed nodes represent the dataflows.

Pig Architecture(contd..)

Optimizer

- The DAG is submitted to the logical optimizer. The optimization activities, such as split, merge, transform and reorder operators execute in this phase.
- The optimization is an automatic feature. The optimizer reduces the amount of data in the pipeline at any instant of time, while processing the extracted data.
- It executes certain functions for carrying out this task, as explained as follows:

Pig Latin and Developing Pig Latin Scripts

- Pig Latin enables developing the scripts for data analysis. A number of operators in Pig Latin help to develop their own functions for reading, writing and processing data.
- Pig Latin programs execute in the Pig run-time environment.

Pig Latin

Statements in Pig Latin:

1. Basic constructs to process the data.
2. Include schemas and expressions.
3. End with a semicolon.
4. LOAD statement reads the data from file system, DUMP displays the result and STORE stores the result.
5. Single line comments begin with - - and multiline begin with/* and end with*/
6. Keywords (for example, LOAD, STORE, DUMP) are not case-sensitive.
7. Function names, relations and paths are case-sensitive.

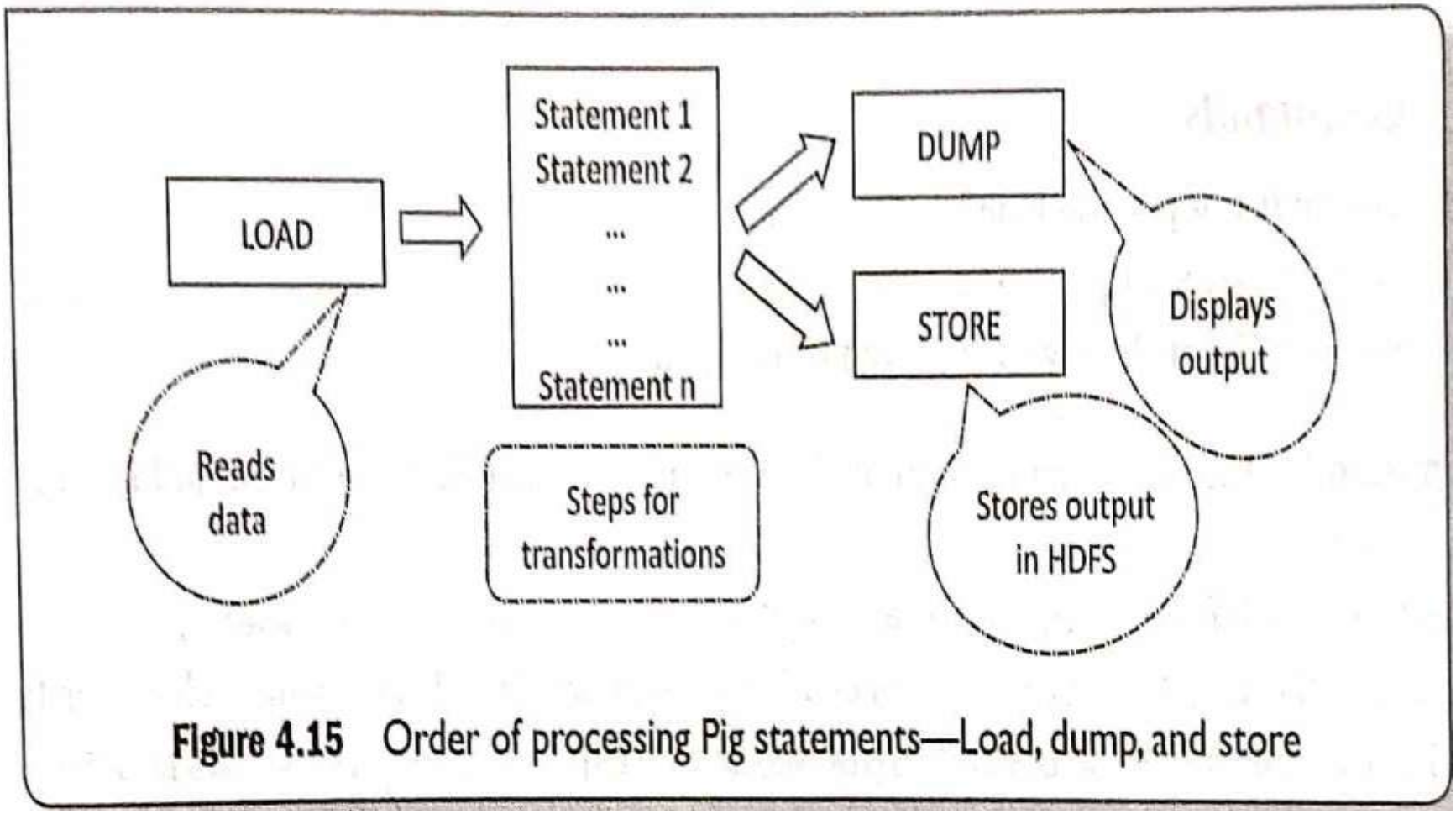


Figure 4.15 Order of processing Pig statements—Load, dump, and store

Operators in Pig Latin

Arithmetic Operators	+	-	*	/	%
Used for	Addition	Subtraction	Multiplication	Division	Remainder

Comparison Operators	==	!=	<	>	≤	≥
Used for	Equality	Not equal	Less than	Greater than	Less than and equal to	Greater than and equal to

Boolean Operators	AND	OR	NOT
Used for	Logical AND	Logical OR	Logical NOT