Module 4
MapReduce, Hive and Pig

4.1 MapReduce map tasks , reduce tasks and MapReduce Execution

MapReduce is a framework using which we can write applications to process huge amounts of data, in parallel, on large clusters of commodity hardware in a reliable manner.

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.

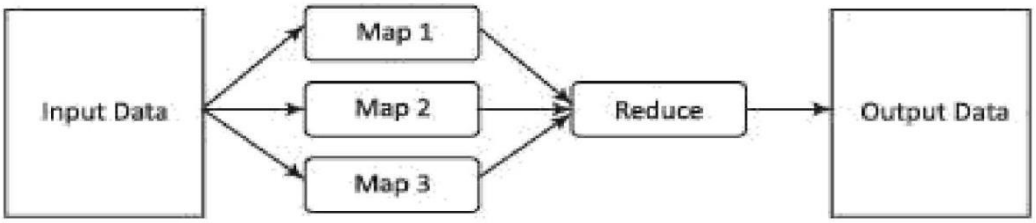


Figure : MapReduce programming model.

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 parallelized and execute on a large cluster.

MapReduce simplifies software development practice. It eliminates the need to write and manage parallel codes. The YARN resource managing framework takes care of scheduling the tasks, monitoring them and re-executing the failed tasks.

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.

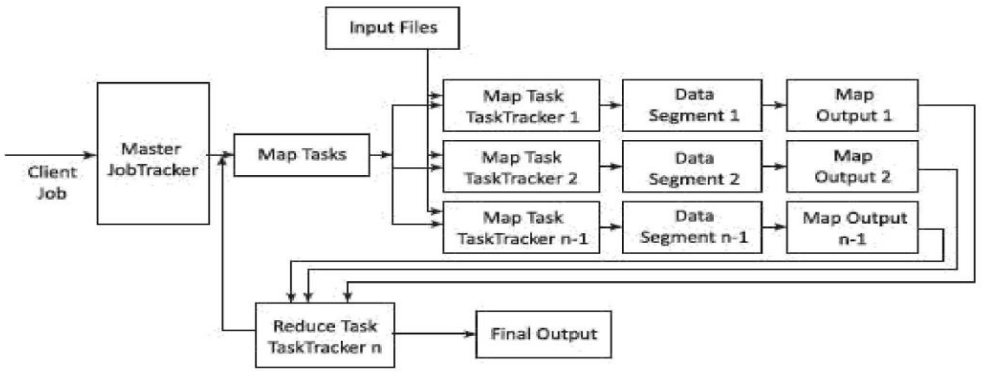


Figure: MapReduce process on client submitting a job JobTracker and Task Tracker

Figure shows MapReduce process when a client submits a job, and the succeeding actions by the JobTracker and TaskTracker. 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, possibly of different types.

4.1.1 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 : Logical view of functioning of map()

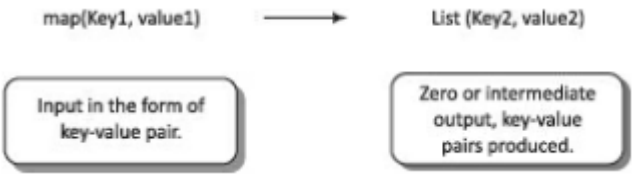


Figure shows logical view of functioning of map(). Hadoop Java API includes Mapper class. An abstract function map() is present in the Mapper class. Any specific Mapper implementation should be a subclass of this class and overrides the abstract function, map().

The Sample Code for Mapper Class

```
public class SampleMapper extends Mapper<k1, V1, k2, v2>
{
    void map (k1 key, V1 value, Context context) throws IOException,
    InterruptedException
   {..}
}
```

Individual Mappers do not communicate with each other.

4.1.2 key-value pairs

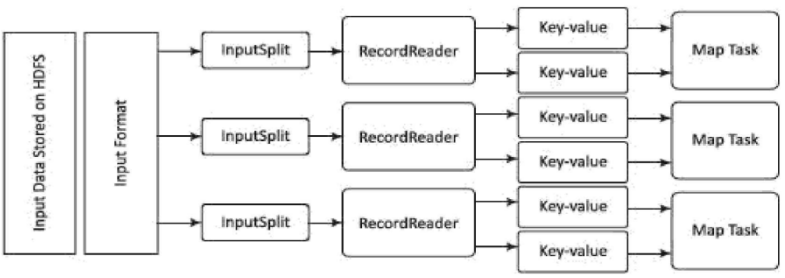


Figure : Key-value pairing in MapReduce

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 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.

4.1.3 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.

4.1.4 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. Typically, the same codes implement the Partitioner, Combiner as well as reduce() functions. 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.

4.1.5 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.

This limits the volume of data transfer between map and reduce tasks, and thus reduces the cost of data transfer across the network. 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.

4.1.6 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

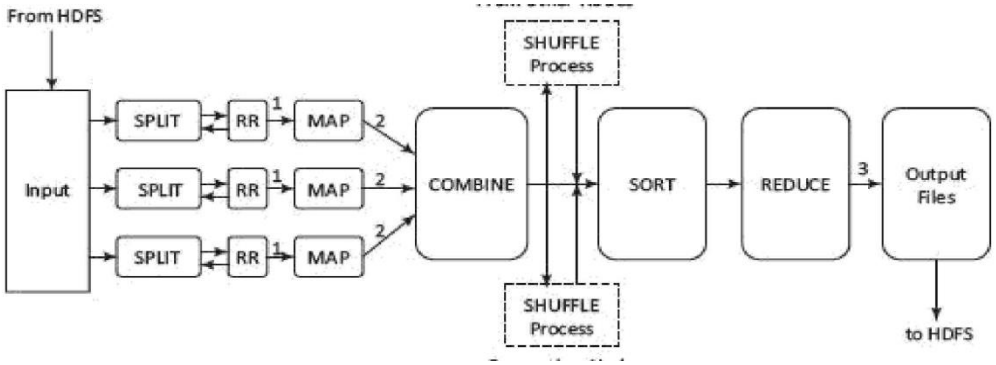
associated with a key and produces outputs such as aggregations and statistics. The reduce function sends output zero or another set of key-value pairs (k3, v3) to the final the output file. Reduce: {(k2, list (v2) -> list (k3, v3)}

Sample Code for Reducer Class

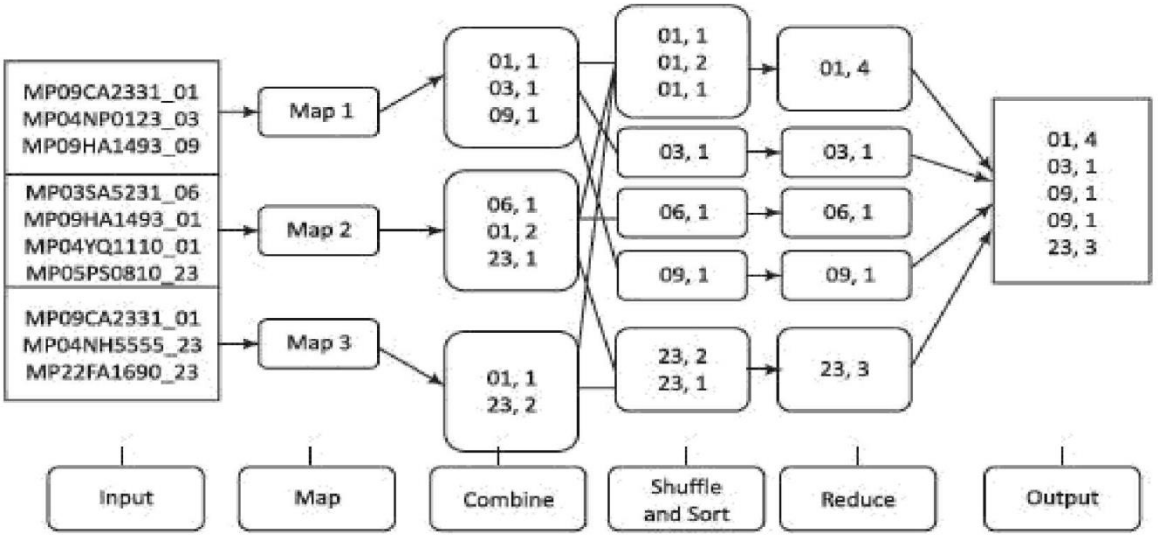
```
public class ExampleReducer extends Reducer<k2, v2, k3, v3>
{
    void reduce (k2 key, Iterable<v2> values, Context context) throw
    IOException, InterruptedException
    {...}
}
```

4.1.7 Details of MapReduce Processing Steps

Execution of MapReduce job does not consider how the distributed processing implements. Rather, the execution involves the formatting (transforming) of data at each step. Figure 4.6 shows the execution steps, data flow, splitting, partitioning and sorting on a map node and reducer on reducer node.



Let us explore an example, 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. The following figure shows the processing steps in a sample application data collected from an ACPAMS.



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.

EXAMPLE 4.5

Write pseudocodes for MapReduce algorithm for the ACPAMS.

SOLUTION

Figure 4.8 gives pseudocodes for the ACPAMS algorithm in MapReduce.

```
class Mapper {
      method Map (file id a; file f) {
            for all term i ∈ file f do {
                  t = Substring (i, 2, After_)
                  Emit (term t, count 1)}}}

class Reducer {
       method Reduce (term t, counts [c1, c2,…..]) {
              sum ← 0
              for all count c ∈ counts [c1, c2, ….] do {
              sum ← sum + c}
              Emit (term t, count sum)}}
```

Figure 4.8 Pseudocodes for the ACPAMS algorithm in MapReduce

Emit( ) function is for output in MapReduce. The Mapper emits an intermediate key-value pair for each alert/message in a document. The Reducer sums up all counts for each alert/message.

4.1.8 Coping with Node Failures

The primary way using which Hadoop achieves fault tolerance is through restarting the tasks. Each task nodes (TaskTracker) regularly communicates with the master node, JobTracker. If a TaskTracker fails to communicate with the JobTracker for a pre-defined period (by default, it is set to 10 minutes), a task node failure by the JobTracker is assumed. The JobTracker knows which map and reduce tasks were assigned to each TaskTracker.

If the job is currently in the mapping phase, then another TaskTracker will be assigned to re-execute all map tasks previously run by the failed TaskTracker. All completed map tasks also need to be assigned for re-execution if they belong to incomplete jobs. This is because the intermediate results residing in the failed TaskTracker file system may not be accessible to the reduce task.

If the job is in the reducing phase, then another TaskTracker will re-execute all reduce tasks that were in progress on the failed TaskTracker.

Once reduce tasks are completed, the output writes back to the HDFS. Thus, if a TaskTracker has already completed nine out of ten reduce tasks assigned to it, only the tenth task must execute at a different node.

Map tasks are slightly more complicated. A node may have completed ten map tasks but the Reducers may not have copied all their inputs from the output of those map tasks. Now if a node fails, then its Mapper outputs are inaccessible. Thus, any complete map tasks must also be re-executed to make their results available to the remaining reducing nodes. Hadoop handles all of this automatically. MapReduce does not use any task identities to communicate between nodes or which reestablishes the communication with other task node. Each task focuses on only its own direct inputs and knows only its own outputs. The failure and restart processes are clean and reliable.

The failure of JobTracker (if only one master node) can bring the entire process down; Master handles other failures, and the MapReduce job eventually completes. When the Master compute-node at which the JobTracker is executing fails, then the entire MapReduce job must restart.

Following points summarize the coping mechanism with distinct Node Failures:

(i) Map TaskTracker failure:

- Map tasks completed or in-progress at TaskTracker, are reset to idle on failure
- Reduce TaskTracker gets a notice when a task is rescheduled on another TaskTracker

(ii) Reduce TaskTracker failure:

- Only in-progress tasks are reset to idle

(iii) Master JobTracker failure:

- Map-Reduce task aborts and notifies the client (in case of one master node).

4.2 Composing MapReduce for calculations and Algorithms

Counting and Summing Assume that the number of alerts or messages generated during a specific maintenance activity of vehicles need counting for a month. Figure 4.8 showed the pseudocode using emit() in the map() of *Mapper* class. *Mapper* emits 1 for each message generated. The reducer goes through the list of 1s and sums them. 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*. This requires only one MapReduce job but the process is not scalable, and hence has limited applicability in large data sets.

Collating 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.)

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 × 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 × 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.

4.3.2 Matrix–Vector Multiplication by MapReduce

Numbers of applications need multiplication of n × 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,

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

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

$$\text{Multiplication } \mathbf{C} = \mathbf{A} \times \mathbf{B} = \begin{bmatrix} 1\times4 + 5\times1 + 4\times3 \\ 2\times4 + 1\times1 + 3\times3 \\ 4\times4 + 2\times1 + 1\times3 \end{bmatrix}$$

$$\text{Hence, } \mathbf{C} = \begin{bmatrix} 21 \\ 18 \\ 21 \end{bmatrix} \quad \ldots (4$$

Algorithm for using MapReduce: The Mapper operates on **A** and emits row-wise multiplication of each matrix element and vector element ($a_{ij} \times b_j \ \forall$ i). The Reducer executes sum() for summing all values associated with each i and emits the element c_i. Application of the algorithm is found in linear transformation.

4.3.3 Relational–Algebra Operations

Explained ahead are the some approaches of algorithms for using MapReduce for relational algebraic operations on large datasets.

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

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

The test() tests the attribute values used for a selection after the binary operation of an attribute with the value(s) or value in an attribute name with value in another attribute name and the binary operation by which each tuple selects. Selection may also return *false* or *unknown*. The test condition then does not select any.

The *Mapper* calls test() for each tuple in a row. When test satisfies the selection criterion then emits the tuple. The *Reducer* transfers the received input tuple as the output.

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 × 82), (Oreo, 0.5 × 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.

Symmetric Difference Symmetric difference (notation is $A \Delta B$ (or $A \ominus B$)] is another relational entity. It means the set of elements in exactly one of the two relations A or B. R3 $\ominus$ 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 this tuple belongs to). The *Reducer* transfers only tuples that belong to neither set_1 or set_2.

4.3.3.5 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.

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.

4.3.3.6 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> ¡ <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 ¡ 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.

4.3.4 Matrix Multiplication

Consider matrices named A (i rows and j columns) and B (j rows and k columns) to produce the matrix C;(i rows and k columns). Consider the elements of matrices A, B and C as follows:

$$\mathcal{A} = \begin{matrix} a_{11}\ a_{12} \cdots a_{1j} \\ a_{21}\ a_{22} \cdots a_{2j} \\ \cdots\cdots \\ \cdots\cdots \\ a_{i1}\ a_{i2} \cdots a_{ij} \end{matrix} \quad \mathcal{B} = \begin{matrix} b_{11}\ b_{12} \cdots b_{1k} \\ b_{21}\ b_{22} \cdots b_{2k} \\ \cdots\cdots \\ \cdots\cdots \\ b_{j1}\ b_{i2} \cdots b_{jk} \end{matrix} \quad \mathcal{C} = \begin{matrix} c_{11}\ c_{12} \cdots c_{1k} \\ c_{21}\ c_{22} \cdots c_{2k} \\ \cdots\cdots \\ \cdots\cdots \\ c_{i1}\ c_{i2} \cdots c_{ik} \end{matrix} \qquad \ldots (4.3)$$

A.B = C; Each element evaluates as follow:

$$C_{ik} = \text{Sum } (a_{ij} \times b_{jk})_{j=1 \text{ to } j}. \; v_a = a_{ij} \text{ and } v_b = b_{jk} \quad \ldots (4.4)$$

First row of C

C first column element = $(a_{11}b_{11} + a_{12}b_{21} + \ldots + a_{1j}.b_{j1})$. Second column element = $(a_{11}b_{12} + a_{12}b_{22} + \ldots + a_{1j}.b_{j2})$.
The k^{th} column element = $(a_{11}b_{1k} + a_{12}b_{2k} + \ldots + a_{1j}\, b_{jk})$.

Second row of C

C first column element = $(a_{21}b_{11} + a_{22}b_{21} + \ldots + a_{2j}.b_{j1})$. Second column element = $(a_{21}b_{12} + a_{22}b_{22} + \ldots + a_{2j}.b_{j2})$.
The k^{th} column element = $(a_{21}b_{1k} + a_{22}b_{2k} + \ldots + a_{2j}.\, b_{jk})$.

The i^{th} row of C

C first column element = $(a_{i1}b_{11} + a_{i2}b_{21} + \ldots + a_{ij}.b_{j1})$. Second column element = $(a_{i1}b_{12} + ai_2b_{22} + \ldots + a_{ij}.b_{j2})$. The k^{th} column element = $(a_{i1}b_{1k} + a_{i2}b_{2k} + \ldots + a_{ij}.b_{jk})$.

Consider two solutions of matrix multiplication.

Matrix Multiplication with Cascading of Two MapReduce Steps

Table 4.2 gives the names, attributes, relations R_A, R_B and R_C, tuples in A, B, C, natural Join of R_A and R_B, keys and values, and seven steps for multiplication of A and B.

Table 4.2 Seven steps for multiplication of A and B for cascading of two MapReduce Steps

Step	Step description	Matrix A	Matrix B	Matrix C = A.B
1	Name	A	B	C
2	Specify attributes of (Key, Value pairs of each element [row number, column number, value]	(I, J, v_a)	(J, K, v_b)	(I, K, v_c)
3	Specify relations	R_A = A (I, J, v_a)	R_B = A (J, K, v_b)	R_C = A (I, K, v_c)
4	Consider tuples of A, B and C	(i, j, a_{ij})	(j, k, b_{jk}).	(i, k, c_{ik})
5	Find natural Join of R_A and R_B = Matrix elements (a_{ij}, b_{jk}) [j is common in both]	-	-	tuples (i, j, k, v_a, v_b)
6	Get tuples for finding Product C			Four-component tuple (i, j, k, $v_a \times v_b$)
7	Grouping and aggregation of tuples with attributes I and K			<I, K> SUM ($v_a \times v_b$).

The product A.B = Natural join of tuples in the relations R_A and R_B followed by grouping and aggregation. Natural Join of A (I, J, v_a) and B (J, K, v_b), having only attribute J in common = Tuples (i, j, k, v_a, v_b) from each tuple (i, j, v_a) in A and tuples (j, k, v_b) in B.

1. *MapReduce tasks for Steps 5 and 6*: Five-component tuple represents the pair of matrix elements
(a_{ij}, b_{jk}). Requirement is product of these elements. That means four-component tuple (i, j, k, $v_a \times v_b$),
from equation (4.4) for elements C_{ik} = Sum $(a_{ij}.b_{jk})_{j= 1 \text{ to } j}$.

   (a) Mapper Function: (i) *Mapper* emits the key-value pairs (j, (A, i, a_{ij})) for each matrix element a_{ij}, and (ii) *Mapper* emits the key-value pair (j, (B, k, b_{jk})) for each matrix element a_{ij}.

   (b) Reduce Function: Consider the tuples of A = (A, i, a_{ij}) for each key j, consider tuples of
   B = (B, k, b_{jk}) for each key j. Produce a key-value pair with key equal to (i, k) and
   value = $a_{ij} \times b_{jk}$. A and B are just the names, may be represented by 0101 and 1010.

2. *Next MapReduce Steps 7*: Perform <I, K> ¡ SUM ($v_a \times v_b$). That means do grouping and aggregation, with I and K as the grouping attributes and the sum of $v_a \times v_b$ as the aggregation.

   (c) The *Mapper* emits the key-value pairs (i, k, v_c) for each matrix element of C inputs with key i and k, and v_c from earlier task of the reducer $v_a \times v_b$.

   (d) *Reducer* groups (i, k, v_c) in C using [C, i, k, sum (v_c)] from aggregated values of v_c from sum (v_c). Aggregation uses the same memory locations as used by elements v_c. C is just the name, may be represented by 1111.

Matrix Multiplication with One MapReduce Step MapReduce tasks for Steps 5 to 7 in a single step.

   (e) Map Function: For each element a_{ij} of A, the *Mapper* emits all the key-value pairs
   [(i, k), (A, j, a_{ij})] for k = 1, 2, . . ., up to the number of columns of B. Similarly, emits all the key-value pairs [(i, k), (B, j, b_{jk})] for i = 1, 2, . . ., up to the number of rows of A. for each element b_{jk} of B.

   (f) Reduce Function: Consider the tuples of A = (A, i, a_{ij}) for each key j. Consider tuples of
   B = (B, k, b_{jk}) for each key j. Emits the key-value pairs with key equal to (i, k) and value = sum of ($a_{ij} \times b_{jk}$) for all values j.

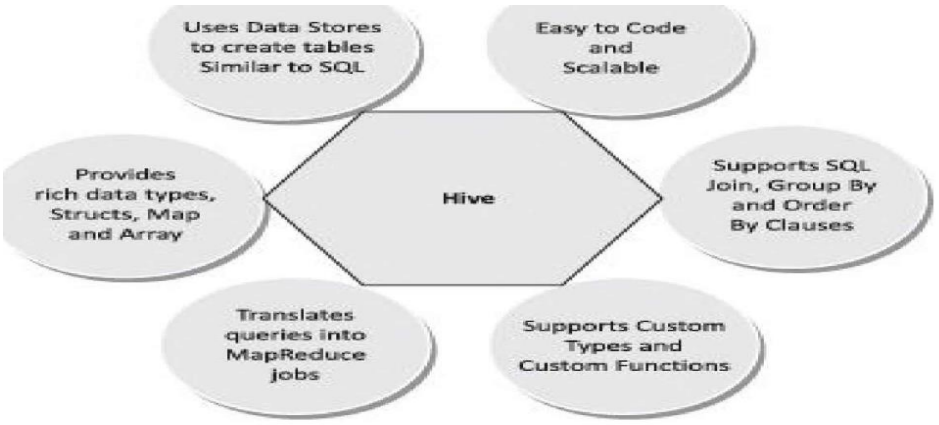
Memory required in one step MapReduce is large as compared to two steps in cascade. This is due to the need to store intermediate values of v_c and then sum them in the same *Reducer* step.

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.

Features of Hive:

The figure shows the main features of Hive



Characteristics of Hive:

- 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.
- Supports web interfaces as well. Application APIs as well as web-browser clients, can access the Hive DB server.
- Provides an SQL dialect (Hive Query Language, abbreviated HiveQL or HQL).

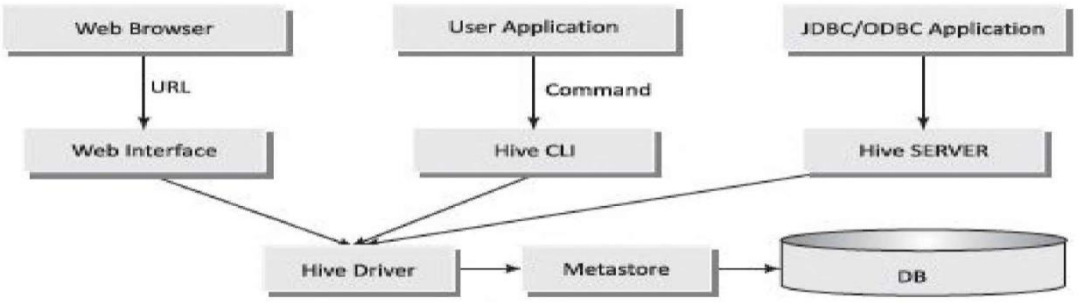
Results of HiveQL Query and the data load in the tables which store at the Hadoop cluster at HDFS.

Limitations

Hive is:

- Not a full database. Main disadvantage is that Hive does not provide update, alter and deletion of records in the database.
- Not developed for unstructured data.
- Not designed for real-time queries.

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.

Hive Installation

- 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)

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, file formats and data model:-

Hive data types are:

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

Hive Collection types are:

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')

File formats are

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).

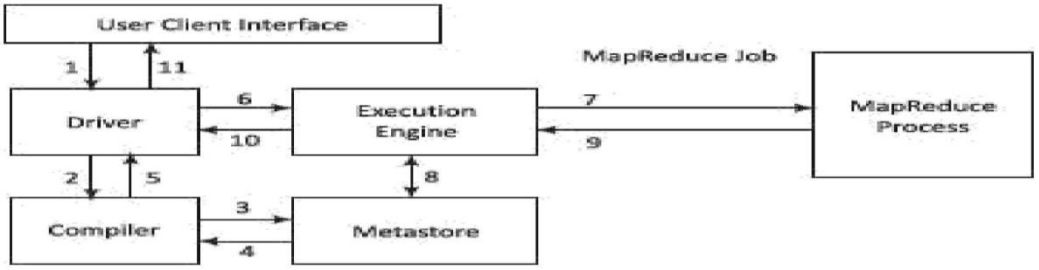
Data model are

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.

Hive integration and workflow steps:-

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

Figure : Dataflow sequences and workflow steps



Steps 1 to 11 are as follows:

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:-

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.

Following are the examples of the returned output:
SELECT floor(10.5) from marks;
Output = 10.0
SELECT ceil(10.5) from marks;
Output = 11.0

HiveQL (Hive Query Language or HQL):-

- 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.

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 gives the row formats in a Hive table.

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'

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
```

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 comma

SOLUTION

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

*Default database is test.

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

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:

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 .

```
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. The following example explains the concept of partitioning, columnar and file records formats.

  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_{r1}, t_{r2}, t_{r3} and t_{r4} horizontally row-wise. Each sub-table has one row and eight columns. Now, convert each sub-table t_{r1}, t_{r2}, t_{r3} and t_{r4} 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');
```

Advantages of Partition

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

Limitations of Partition

- 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.
- Partitions may optimize some queries based on Where clauses, but they may be less responsive for other important queries on grouping clauses.
- 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 $H(C_{bucket\_i})$ *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:

```
#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.

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.

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.

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. 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.

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

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 DISTRIBUTE BY, CLUSTER BY are HiveQL clauses. An example of using the clauses is given below:

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;
```

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 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

(i) 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. It provides an interactive shell known as Grunt to write Pig Latin scripts. Programmers write scripts using Pig Latin to analyze data. The scripts are internally converted to Map and Reduce tasks with the help of the component known as Execution Engine, that accepts the Pig Latin scripts as input and converts these scripts into MapReduce jobs. Writing MapReduce tasks was the only way to process the data stored in HDFS before the Pig.

(ii) 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, JRuby. They easily embed into Pig scripts written in Pig Latin. UDFs provide extensibility to the Pig.

(iii) Process any kind of data, structured, semi-structured or unstructured data, coming from various sources.

(iv) 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.

(v) Handles inconsistent schema in case of unstructured data as well.

(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).

(vii) Performs automatic optimization of tasks before execution.

(viii) Programmers and developers can concentrate on the whole operation without a need to create mapper and reducer tasks separately.

(ix) 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.

(x) Pig characteristics are data reading, processing, programming the UDFs in multiple languages and programming multiple queries by fewer codes. This causes fast processing.

(xi) 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.

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

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

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 below shows Pig architecture for scripts dataflow and processing in the HDFS environment.

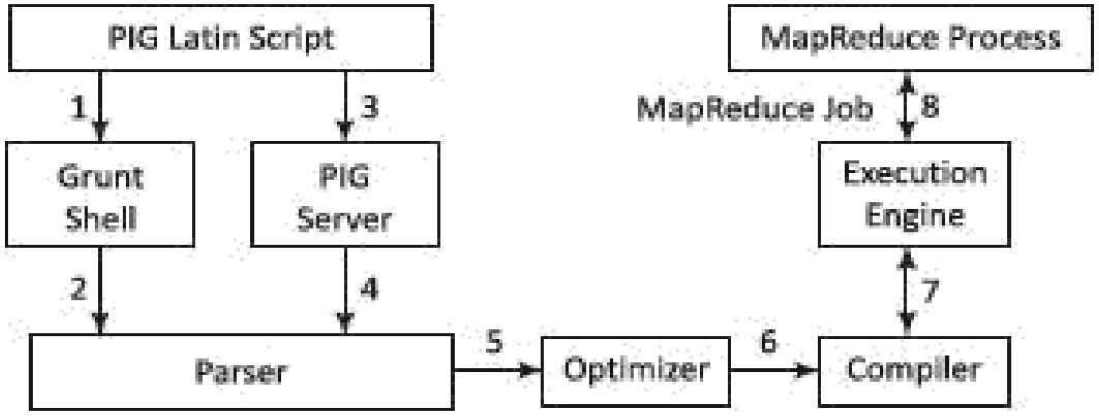


Figure : Pig architecture for scripts dataflow and processing

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.

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.

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:

PushUpFilter: If there are multiple conditions in the filter and the filter can be split, Pig splits the conditions and pushes up each condition separately. Selecting these conditions at an early stage helps in reducing the number of records remaining in the pipeline.

PushDownForEachFlatten: Applying flatten, which produces a cross product between a complex type such as a tuple, bag or other fields in the record, as late as possible in the plan. This keeps the number of records low in the pipeline.

ColumnPruner: Omitts never used columns or the ones no longer needed, reducing the size of the record. This can be applied after each operator, so that the fields can be pruned as aggressively as possible.

MapKeyPruner: Omitts never used map keys, reducing the size of the record.

LimitOptimizer: If the limit operator is immediately applied after *load* or *sort* operator, Pig converts the load or sort into a limit-sensitive implementation, which does not require processing the whole dataset. Applying the limit earlier reduces the number of records.

Compiler The compiler compiles after the optimization process. The optimized codes are a series of MapReduce jobs.

Execution Engine Finally, the MapReduce jobs submit for execution to the engine. The MapReduce jobs execute and it outputs the final result.

Apache Pig – Grunt Shell

Main use of Grunt shell is for writing Pig Latin scripts. Any shell command invokes using sh and ls. Syntax of sh command is:

grunt> sh shell command parameters

Syntax of ls command:

grunt> sh ls

Installing Pig

Following are the steps for installing Pig:

1. Download the latest version from − https://pig.apache.org/
2. Download the tar files and create a Pig directory

$ cd Downloads/
$ tar zxvf pig-0.15.0-src.tar.gz
$ tar zxvf pig-0.15.0.tar.gz
$ mv pig-0.15.0-src.tar.gz/* /home/Hadoop/Pig/

3. Configure the Pig

export PIG_HOME = /home/Hadoop/Pig
export PATH = $PATH:/home/Hadoop/pig/bin
export PIG_CLASSPATH = $HADOOP_HOME/conf

Pig Latin Data Model

Pig Latin supports primitive data types which are atomic or scalar data types. Atomic data types are int, float, long, double, char [], byte []. The language also defines complex data types. Complex data types are tuple, bag and map. Table 4.16 gives data types and examples.

Data type	Description	Example
bag	Collection of tuples	{(1,1), (2,4)}
tuple	Ordered set of fields	(1,1)
map (data map)	Set of key-value pairs	[Number#1]
int	Signed 32-bit integer	10
long	Signed 64-bit integer	10L or 10l
float	32-bit floating point	22.7F or 22.7f
double	64-bit floating point	3.4 or 3.4e2 or 3.4E2
chararray	Char [], Character array	data analytics
bytearray	BLOB (Byte array)	ff00

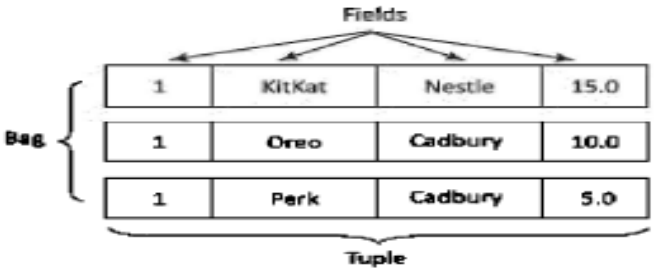


Figure 4.13 Pig Data Model with fields, Tuple and Bag

- **Tuple** Tuple is a record of an ordered set of fields. A tuple is similar to a row in a table of RDBMS. The elements inside a tuple do not necessarily need to have a schema associated to it. A tuple represents by '()' symbol. For example, (1, Oreo, 10, Cadbury).
- **Bag** A bag is an unordered set of tuples. A bag can contain duplicate tuples as it is not mandatory that they need to be unique. Each tuple can have any number of fields (flexible schema). A bag can also have tuples with different data types. {} symbol represents a bag.
  For example, {(Oreo, 10), (KitKat, 15, Cadbury)}
  There are two types of bag: outer bag or relations and inner bag. Outer bag or relation is a bag of tuples. Here relations are similar as relations in relational databases. To understand it better let us take an example: {(Oreo, Cadbury), (KitKat, Nestle), (Perk, Cadbury)}. This bag explains the relation between the Chocolate brand and their brand company.
  A bag can be a field in a relation; in that context, it is known as an inner bag. Thus, an inner bag contains a bag inside a tuple. Figure 4.14 shows a relation and keys and their values: (Cadbury, {(Oreo, 10), (Perk,5)}) (Nestle {(Kitkat,15)})

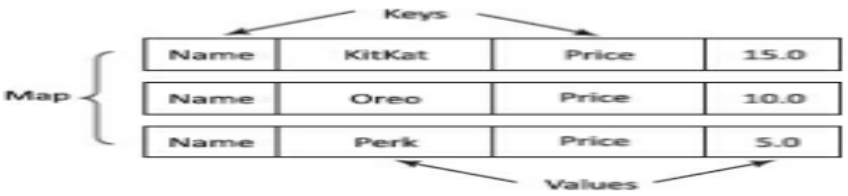


Figure 4.14 Relation and corresponding keys and their values (key-value pairs)

- **Relation** A relation is a bag of tuples. The relations in Pig Latin are unordered (there is no guarantee that tuples are processed in any particular order).
- **Map** A map (or data map) is a set of key-value pairs. The key needs to be of type chararray and should be unique (similar to a column name). Map can be indexed and value associated with it can be accessed from the keys. The value might be of any type. [] symbol represents Map. The key and value separate by '#' symbol. For example, [type#Oreo, price#10].

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:

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

Figure 4.15 shows the order of processing Pig statements—Load, dump and store.

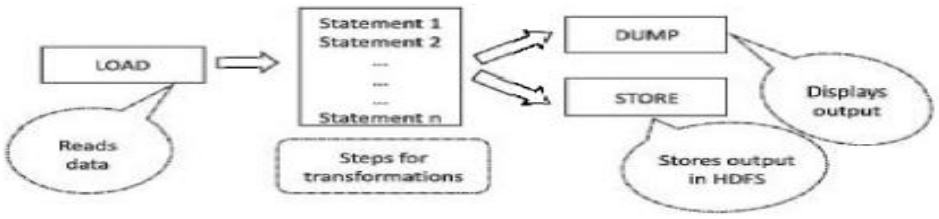


Figure 4.15 Order of processing Pig statements—Load, dump, and store

➢ **Apache Pig Execution**

Pig Execution Modes

- Local Mode: All the data files install and run from a local host using the local file system. Local mode is mostly used for testing purpose.

  COMMAND: pig –x local
- MapReduce Mode: All the data files load or process that exists in the HDFS. A MapReduce job invokes in the back-end to perform a particular operation on the data that exists in the HDFS when a Pig Latin statement executes to process the data.

  COMMAND: pig -x mapreduce or pig

Pig Latin Script Execution Modes

- Interactive Mode − Using the Grunt shell.
- Batch Mode − Writing the Pig Latin script in a single file with .pig extension.
- Embedded Mode − Defining UDFs in programming languages such as Java, and using them in the script.

➢ **Commands**

- To get the list of pig commands: pig –help;
- To get the version of pig: pig –version.
- To start the Grunt shell, write the command: pig

LOAD Command The first step to a dataflow is to specify the input. Load statement in Pig Latin loads the data from `PigStorage`.

To load data from HBase: `book = load 'MyBook' using HBaseStorage();`

For reading CSV file, `PigStorage` takes an argument which indicates which character to use as a separator. For example, `book = LOAD 'PigDemo/Data/Input/myBook.csv' USING PigStorage (,);`

For reading text data line by line: `book = LOAD 'PigDemo/Data/Input/myBook.txt' USING PigStorage() AS (lines: chararray);`

To specify the data-schema for loading: `book = LOAD 'MyBook' AS (name, author, edition, publisher);`

Store Command Pig provides the store statement for writing the processed data after the processing is complete. It is the mirror image of the load statement in certain ways.

By default, Pig stores data on HDFS in a tab-delimited file using PigStorage:

`STORE processed into '/PigDemo/Data/Output/Processed';`

To store in HBaseStorage with a *using* clause: `STORE processed into 'processed' using HBaseStorage();`

To store data as comma-separated text data, PigStorage takes an argument to indicate which character to use as a separator: `STORE processed into 'processed' using PigStorage(',');`

Dump Command Pig provides dump command to see the processed data on the screen. This is particularly useful during debugging and prototyping sessions. It

can also be useful for quick adhoc jobs.

The following command directs the output of the Pig script on the display screen:

```
DUMP processed;
```

Relational Operations The relational operations provided at Pig Latin operate on data. They transform data using sorting, grouping, joining, projecting and filtering. Followings are the basic relational operators:

Foreach

FOREACH gives a simple way to apply transformations based on columns. It is Pig's projection operator. Table 4.17 gives examples using FOREACH..

Load an entire record, but then remove all but the name and phone fields from each record	`A = load 'input' as (name: chararray, rollno: long, address: chararray, phone: chararray, preferences: map []);` `B = foreach A generate name, phone;`
Tuple projection using dot operator	`A = load 'input' as (t:tuple (x:int, y:int));` `B = foreach A generate t.x, t.$1;`
Bag projection	`A = load 'input' as (b:bag{t:(x:int, y:int)});` `B = foreach A generate b.x;`
Bag projection	`A = load 'input' as (b:bag{t:(x:int, y:int)});` `B = foreach A generate b.(x, y);`

Add all integer values	`A = load 'input' as (x:chararray, y:int, z:int);`
	`A1 = foreach A generate x, y + z as yz;`
	`B = group A1 by x;`
	`C = foreach B generate SUM(A1.yz);`

Filter

FILTER gives a simple way to select tuples from a relation based on some specified conditions (predicate). It is Pig's select command**.**

Loads an entire record, then selects the tuples with marks more than 75 from each record	`A = load 'input' as (name:chararray, rollno:long, marks:float);` `B = filter A by marks > 75.0;`
Find name (chararray) that do not match a regular expression by preceding the text without a given character string. Output is all names that do not start with P.	`A = load 'input' as (name:chararray, rollno:long, marks:float);` `B = filter A by not name matches 'P.*';`

Group

GROUP statement collects records with the same key. There is no direct connection between group and aggregate functions in Pig Latin unlike SQL.

Collects all records with the same value for the provided key into a bag. Then it can pass to aggregate function, if required or do other things with that.	`A = load 'input' as (name: chararray, rollno:long, marks: float);`
	`grpd = group A by marks;`
	`B = foreach grpd generate name, COUNT(A);`

Order by

ORDER statement sorts the data based on a specific field value, producing a total order of output data.

The syntax of order is similar to group. Key indicates by which the data sort.	`A = load 'input' as (name: chararray, rollno: long, marks: float);` `B = order A by name;`
To sort based on two or more keys (For example, first sort by, then sort by), indicate a set of keys by which the data sort. No parentheses around the keys when multiple keys indicate in order	`A = load 'input' as (name:chararray, rollno:long, marks:float);` `B = order A by name, marks;`

Distinct

DISTINCT removes duplicate tuples. It works only on entire tuples, not on individual fields:

Removes the tuples having the same name and city.	`A = load 'input' as (name: chararray, city: chararray);` `B = distinct A;`

Join

JOIN statement joins two or more relations based on values in the common field. Keys indicate the inputs. When those keys are equal, two tuples are joined. Tuples for which no match is found are dropped.

Join selects tuples from one input to put together with tuples from another input.	`A = load 'input1' as (name:chararray, rollno:long);`
	`B = load 'input2' as (rollno:long, marks:float);`
	`C = join A by rollno, B by rollno`

Also based on multiple keys join. All cases must have the same number of keys, and they must be of the same or compatible types.	`A = load 'input1' as (name: chararray, fathername: chararray, rollno: long);`
	`B = load 'input2' as (name: chararray, rollno: long, marks: float);`
	`C = join A by (name, rollno), B by (name, rollno)`

Limit

LIMIT gets the limited number of results.

Outputs only first five tuples from the relation.	`A = load 'input' as (name: chararray, city: chararray);` `B = Limit A 5;`

Sample

SAMPLE offers to get a sample of the entire data. It reads through all of the data but returns only a percentage of rows on random basis. Thus, results of a script with sample will vary with every execution. The percentage it will return is expressed as a double value, between 0 and 1. For example, 0.2 indicates 20%.

Outputs only 10% tuples from the relation	`A = load 'input' as (name:chararray, city: chararray);` `B = sample A 0.1;`

Split

SPLIT partitions a relation into two or more relations

Outputs A relation A splits into two relations P and Q	`A = load 'input' as (name:chararray, rollno:long, marks:float);` `Split A into P if marks >50.0, Q if marks ≤ 50.0;`

Parallel

PARALLEL statement is for parallel data processing.

Generating MapReduce job with 10 reducers	`A = load 'input' as (name: chararray, marks: float);` `B = group A by marks parallel 10;`

EVAL Functions

Following are the evaluation functions:

Function Name	Description
AVG	Compute the average of the numeric values in a in a single-column bag
SUM	Compute the sum of the numeric values in a single-column bag
MAX	Get the maximum of numeric values or chararrays in a single-column bag
MIN	Get the minimum of numeric values or chararrays in a single-column bag
COUNT and COUNT_STAR	Count the number of tuples in a bag
CONCAT	Concatenate two fields. The data type of the two fields must be the same, either chararray or bytearray.
DIFF	Compare two fields in a tuple
IsEmpty	Check if a bag or map is empty (has no data)
SIZE	Compute the number of elements based on the data type
TOKENIZE	Split a string and output a bag of words

Piggy Bank

Pig users share their functions from Piggy Bank. Register is keyword for using Piggy bank functions.

User-Defined Functions (UDFs)

A programmer defines UDFs which perform functionalities not present as built-in Pig function. A programmer can use UDFs for filtering data or performing further analysis. A programmer can write UDF using a programming language, such as Java, Python, Ruby, Jython or JRuby.