MODULE 4

MapReduce, Hive and Pig

LEARNING OBJECTIVES

After studying this chapter,you will be able to:

LO 4.1 Get understanding of MapReduce, map tasks using the key-value store, grouping by keys, reduce tasks using combiners, and coping with node failures

LO 4.2 Get knowledge of composing MapReduce programs for calculations, such as counting,
summing, and algorithms for relational algebraic operations, projections, unions, intersections, natural joins, grouping, aggregation operations and the matrix multiplication

LO 4.3 Get understanding of Hive, architecture, installation, comparison of Hive data store with traditional databases

LO 4.4 Apply HiveQL for querying, sorting, aggregating, querying scripts, MapReduce Joins and sub-queries

LO 4.5 Get knowledge of Pig architecture, Grunt shell commands, data model, Pig Latin, developing scripts and extensibility using UDFs.

RECALL FROM EARLIER CHAPTERS

Hadoop stores and processes data on large clusters (thousands of nodes) of commodity hardware. Hadoop is a software framework for writing distributed applications. Hadoop processes Big Data (multi-terabyte datasets) in parallel and in a reliable and fault-tolerant way. The Hadoop distribution model is a method in which both computations and the data distribute and handle large• sized data.
(Section 2.3)

MapReduce functions are an integral part of the Hadoop physical organization. MapReduce is a programming model for the distributed computing environment. Applications using MapReduce v2, process huge amounts of data, in parallel, on a large number of data nodes reliably (Sections 2.4 and 2.5).

Figure 2.2 showed the main components and the ecosystem components of Handoop, such as AVRO, Zookeeper, Ambari, HBase, Hive, Pig and Mahout (Section 2.6).

This chapter focuses on details of MapReduce, Hive and Pig programming and their use in Big Data applications.

4.1 ! 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 (Section 3.3.3). 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 (Section 2.6.4). Pig enables the usage of relational algebra in HDFS. MapReduce is the processing framework and YARN is the resource managing framework (Section 2.6.5).

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.

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 APIs 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.

Suppose stored files have key-value pairs. Mapping tells us whether the key is in file or in the value store, in a particular cluster and rack. Reduce task uses those values for further processing such as counting, sorting or aggregating.

Application sub-task assigned for processing needs only the outputs of reduce tasks. For example, a query needs the required response for a data store. In Example 2.3, a sub-task may just need total daily sales of specific chocolate flavours to compute the analytics and data visualization.

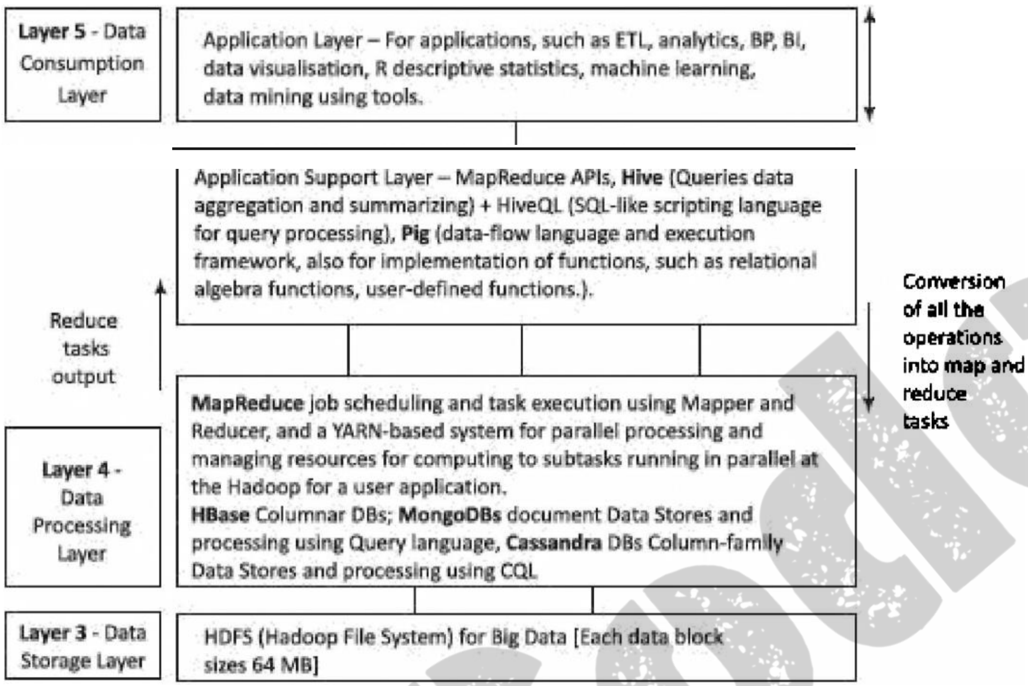


Figure 4.1 Big Data architecture design layers

A reader must learn the following new selected key terms and their meanings besides the key terms given in the previous three chapters.

MapReduce programming model refers to a programming paradigm for processing Big Data sets with a parallel and distributed environment using map and reduce tasks.

YARN refers to provisioning of running and scheduling parallel programs for map and reduce tasks and allocating parallel processing resources for computing sub-tasks running in parallel at the Hadoop for a user application. The YARN resources management enables large-scale data analytics using multiple machines (data nodes) in the HDFS cluster.

Script refers to a small program (codes up to few thousand lines of code) in a language used for purposes such as query processing, text processing, or refers to a small code written in a dynamic high-level general-purpose language, such as Python or PERL.

SQL-like scripting language means a language for writing script that processes queries similar to SQL. SQL lets us: (i) write structured queries for processing in DBMS, (ii) create and modify schema, and control the data access, (iii) create client for sending query scripts, and create and manage server databases, and (iv) view, query and change (update, insert or append or delete) databases.

NoSQL DBs refers to DBs with no prior fixed schema, schema-less models, and databases which possess increasing flexibility for data manipulation.

NoSQL data model refers to ones offering relaxation in one or more of the ACID properties (Atomicity, Consistence, Isolation and Durability) of the database. A theorem known as CAP ~onsistency, Availability and r.artitions) states that out of three properties, at least two must be present for the application/service/process. NoSQL relies upon another model known as the BASE model. This model has three principles: Basic availability (the availability of data even in the presence of multiple failures), Soft state (data consistency is the developer's problem and should not be handled by the database), Eventual consistency (when no new changes occur on existing data, eventually all accesses to that data will return the last updated value).

Data-architecture patterns refer to formats used in NoSQL DBs. The examples are Key-Value Data Stores, Object Data Stores, Column family Big Data Stores, Tabular Data Stores and Document Stores.

Key-Value Data Store refers to a simplest way to implement a schema-less database. A string called key maps to values in a large data string or BLOB (basic large object). Key-value stores use primary key access. Therefore, the storage easily scales up and data retrievals are fast.

Object Data Store refers to a repository which stores the (i) objects (such as files, images, documents, folders and business reports), (ii) system metadata which provides information such as filename, creation_date, last_modified, language_used (such as Java, C, C#, C++, Smalltalk, Python), access_permissions, supported Query languages, and (iii) Custom metadata which provides information such as subject, category and sharing permission.

Tabular Data Store refers to table, column-family or BigTable like Data Store.

Column family Big Data store refers to a storage in logical groups of column families. The storage may be similar to columns of sparse matrix. They use a pair of row and column keys to access the column fields.

BigTable Data Store is a popular column-family based Data Store. Row key, column key and timestamp uniquely identify a value. Google BigTable, HBase and Cassandra DBs use the BigTable Data Store model.

Document Store means a NoSQL DB which stores hierarchical information in a single unit called document. Document stores data in nested hierarchies; for example in XML document object model, *JSON* formats data model or machine-readable data as one BLOB.

Tuple means an ordered list of elements. An n-tuple relates to set theory, a collection (sequence) of "n" elements. Tuples implement the records.

Collection means a well-defined collection of distinct objects in a set, the objects of a set are the elements. That also means a store within a single DB to achieve a single purpose. A collection may be analogous to a table ofRDBMS.A collection in a database also refers to storage of a number of documents. A collection may store documents which do not have the same fields. Thus, documents in the collection are schema-less. Thus, it is possible to store documents of varying structures in a collection.

Aggregate refers to collection of data sets in the key value, column family or BigTable data stores which usually require sequential processing.

Aggregation function refers to a function to find counts, sum, maximum, minimum, other statistical or mathematical function using a collection of datasets, such as column or column-family.

Sequence refers to an enumerated collection of objects, (the repetitions can be there) which contain members similar to a set. Sequence length equals the number of elements (can also be infinite). Sequence should reflect an order which matters, unlike a set.

Document refers to a container for the number of collections. The container can be a unit of storing data in a database, such as MongoDB.

Projection refers to a unary operation (single input or operand) written as *ITattri, attr2, ..., attrn* where *(attrl, attr2, ..., attm)* is a set of n attribute names. Projection returns a set obtained by selecting only the n attributes. A generalized projection includes a method using attribute values. *ITstudent_Id, sum (GPA),* sum *(SGPA)*·

Natural join is where two tables join based on all common columns. Both the tables must have the same column name and the data type.

Inner join is the default natural join. It refers to two tables that join based on common columns mentioned using the ON clause. Inner Join returns all rows from both tables if the columns match.

Node refers to a place for storing data, data block or read or write computations.

Data center in a DB refers to a collection of related nodes. Many nodes form a data center or rack.

Cluster refers to a collection of many nodes.

Keyspace means a namespace to group multiple column families, especially one per partition.

Indexing to a field means providing reference to a field in a document of collections that support the queries and operations using that index. A DB creates an index on the _id field of every collection.

This chapter describes MapReduce programming, Hive and Pig APIs in the MapReduce programming model and the HDFS data storage environment. Section 4.2 describes the MapReduce paradigm, map tasks using key-value pairs, grouping by keys and reduce tasks using partitioning and combiners in the application execution framework. Section 4.3 describes algorithms for using MapReduce. Section 4.4 describes Hive, architecture, installation and comparison with traditional databases. Section 4.5 describes HiveQL, querying the data, sorting and aggregating, scripts, joins and sub-queries. Section 4.6 introduces Pig architecture, grunt shell commands, data model, Pig Latin, developing scripts and extensibility using UDFs.

4.2 MAPREDUCE MAP TASKS, REDUCE TASKS AND MAPREDUCE EXECUTION

LO 2.4

MapReduce map tasks using the key-value store, grouping by keys, reduce tasks using combiners, and coping with node failures

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 shows MapReduce programming model.

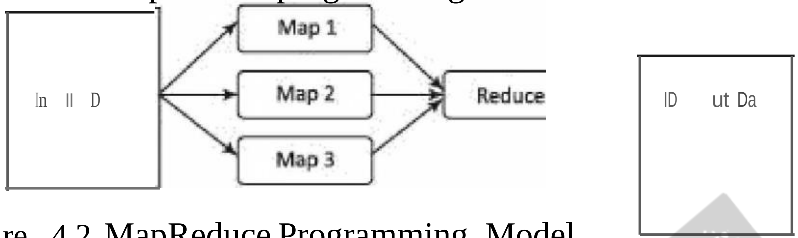


Figure 4.2 MapReduce Programming Model

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.

EXAMPLE 4.1

How can a car company quickly compute an aggregation function using the number of cars of a specific car-model sold at the company showrooms as input? Use the concept of division of an application task into a number of sub-tasks (running in parallel).

SOLUTION

The company's showrooms sell a specific model. Assume the analysis requires us to find the aggregate number N. The N computes by counting the number of cars of that model which have been sold over a specific period (Practice Exercise 3.3). N is a very large number. The application process will require a long time to count this sequentially from the sales figures.

The programming-model splits the application task into number of n sub-tasks, running in parallel. Each sub-task thus takes up and counts N/n sales entries for the car-model. Each sub-task fetches the items containing information of car sales separately. The results of all the application sub•tasks later combine at the end to send the result to the application. High volumes of data (Big Data) need the splitting and parallel processing of the tasks.

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. Following explains the concept:

EXAMPLE 4.2

How does MapReduce enable query processing quickly in Big Data problems?

SOLUTION

MapReduce provides two important functions for query processing. The distribution of task based on user's query to various nodes within the cluster is the first function. The other function is organizing and reducing the results from each node into a cohesive answer to a query.

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.

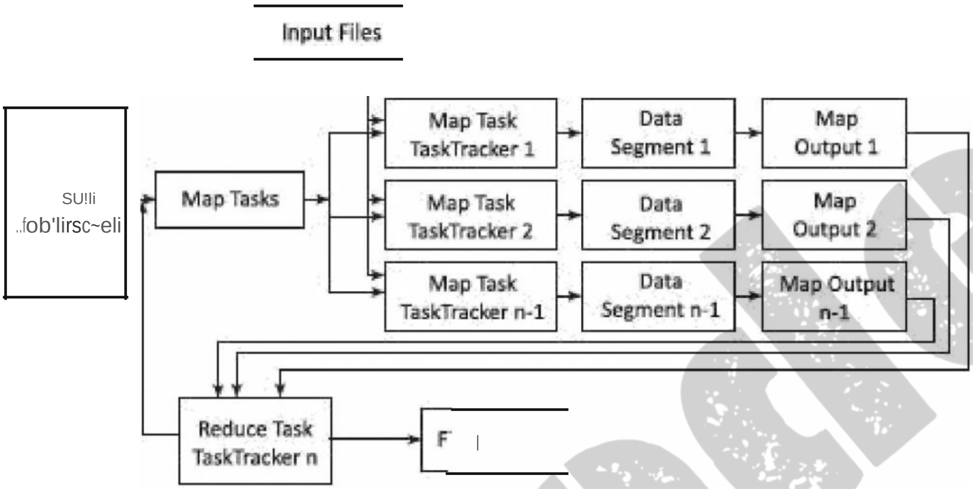


Figure 4.3 MapReduce process on client submitting a job

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, possibly of different types (Section 2.4.2). Example 2.3 explained the process of converting input files into key-values.

4.2.1 Map-Tasks

Map task means a task that implements a mapl), which runs user application codes for each key-value pair **(kl, vl)**. Key **kl** is a set of keys. Key **kl** maps to a group of data values (Section 3.3.1). Values vl 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 (kz, 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().

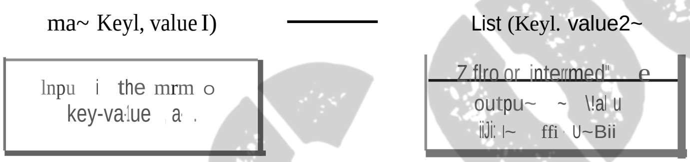


Figure 4.4 Logical view of functioning of map()

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 claeB SampleMapper extende Mapper~kl, Vl, k2, v2>
{
    void map  (kl key, V1 value, Context context) throwB IOException,
    InterruptedException
   {.. }
```

Individual Mappers do not communicate with each other.

Number of Maps The number of maps depends on the size of the input files, i.e., the total number of blocks of the input files. Thus, if the input files are of 1TB in size and the block size is 128 MB, there will be 8192 maps. The number of map task Nmap can be explicitly set by using *setNumMapTasks(int).* Suggested number is nearly 10-100 maps per node. Nmap can be set even higher.

4.2.2 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 (Section 3.3.1).

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 TextlnputFormat 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.

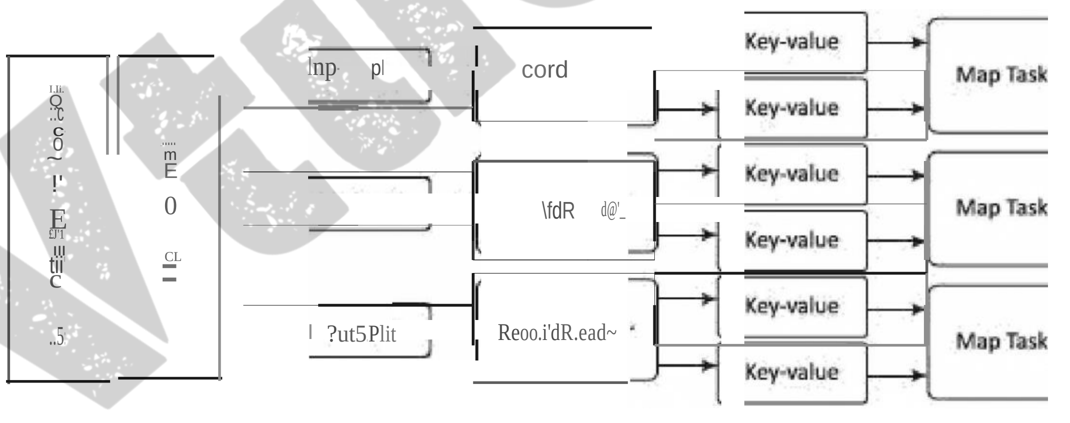


Figure 4.5 Key-value pairing in MapReduce

4.2.3 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 (kz) collect and group together, creating one group for each key. So, the Shuffle output format will be a List of <k2, List (vz)», 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.

4.2.4 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.

4.2.5 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.

4.2.6 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 (kz, 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 (ks, v3)}

Sample Code for Reducer Class

```
public class ExarrpleReducer extenda Reducer<k2, v2, k3, v3>


    void reduce (k2 key, Iterable<v2~ values, Context context) throws
    IOBxception, InterruptedBxception
    { … }
```

4.2.7 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.

Example 2.3 described sales data of the five types of chocolates in a large number of ACVMs (Automatic Chocolate Vending Machines). The example gave answers to the following: (i) how hourly data log for each flavor sales on large number of ACVMs save using HDFS, (ii) how the sample of data-collect in a file each for 0-1,1-2, ... 12-13,13-14, 15-16, ... for up to 23-24 specific hour sales, (iii) what will be output streams of map tasks which are the input streams to the reduce() tasks, and (iv) what will be the Reducer outputs.

Let us explore another 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 [Chapter 1 Example 1.6(ii)].

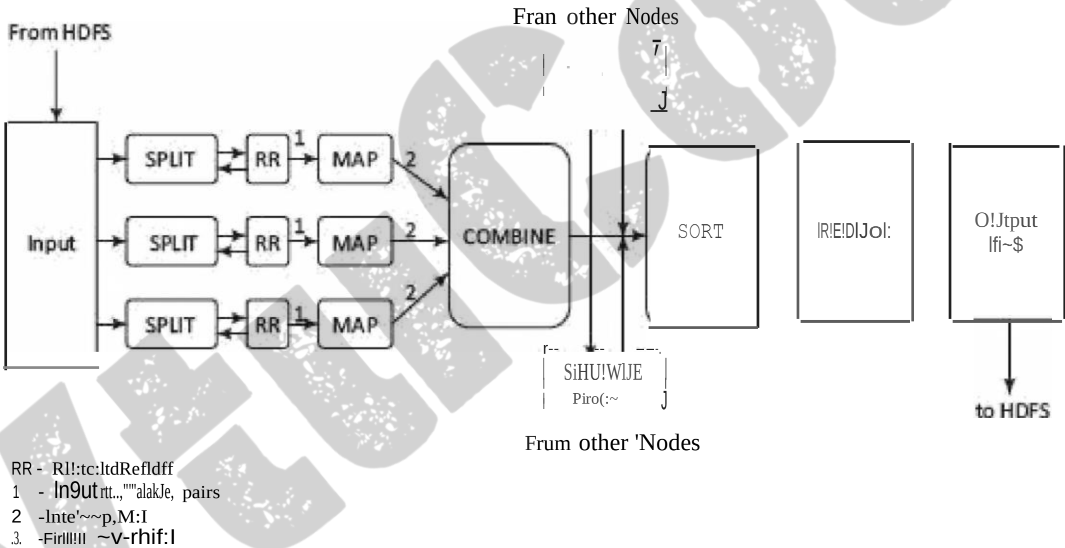


Figure 4.6 MapReduce execution steps

EXAMPLE 4.1

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.

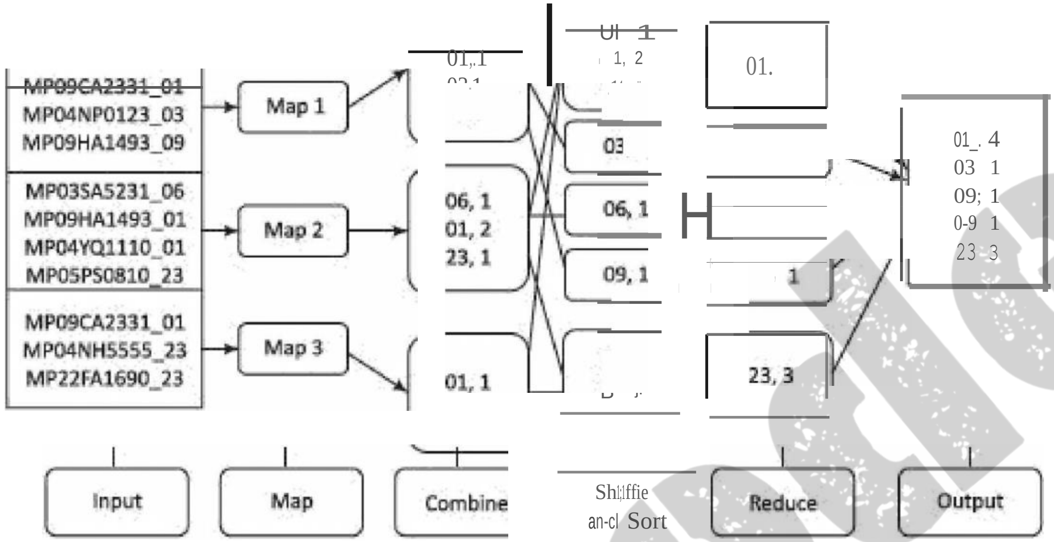


Figure 4.7 MapReduce processing steps in ACPAMS application

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.

The following example explains how the ACPAMS company receives alerts/messages.

EXAMPLE 4.4

Describe the MapReduce processing steps to illustrate how the ACPAMS receives alerts/messages.

SOLUTION

The ACPAMS Company can receive the alerts/messages every hour from several sensors of the automotive components installed in the number of cars. A server maps the keys for filling fuel, changing of the coolant, etc. It requires a lot of time to scan the hourly maintenance log sequentially because there are a million cars registered for the ACP AMS service. Each car is equipped with nearly 50 sensor-based components sending alerts/message every minute. By contrast, the MapReduce programmer can split the application task among multiple subtasks, say one hundred sub• tasks, and each sub-task processes the data of a selected set of a Service users.

The results of all the sub-tasks aggregate and produce the final result for hourly maintenance requirement of each component of the cars registered at ACPAMS Company. Finally, the aggregated hourly results appear from the hourly log of transactions files at Hadoop data nodes. The whole system maintains transparency, without knowing the presence of a distributed parallel system processing data of a hundred million records. Table 4.1 gives examples of assigning Ids to alerts/messages.

Table 4.1 Examples of Alert/Messageid (say, total 50 Ids, i.e., one for each sensor component)

Alert/ MessageId	Details
O]	Fuel fiD-io reqmst
02	Filter requrrement
03	AUgnment aimd balrun.cimg lieqlllirnm t
04	Enrg.ioe iiL lless
0.5	Coolant <Jrumge reque: t

Assumption: Let 60 files for each hour create every day. The files are file_l, file_2, ..., file_60. Each file saves as a key-value pair at the large number of the company's machines. The log contains information in the following format:

maintenance service Id:(<CarRegistrationNumber>_<alert/rnesssageld-)

Thus, every line entry becomes the key, and the value will be the instance number.

Sample data of one of such file out of 60 files (file_lO) saved as hour• maintenance-service log for the maintenance service during 15:00-16:00will be as follows:

MP09CA2331_01

MP04NP0123_03

MP09HA1493_01

MP03SA5231_06

MP09CA2331_04

MP09CC4614_01

(i) The input files are using *NLineInputFormat* input format.

(ii) Map tasks will map the input streams of key values at files, file_l, file_2, ... file_59, file_60 every hour.

(iii) The map function *extracts* the alert/message Id from each line of the input files. They are the values after underscore in each line.

(iv) The resulting 1 million x 50 key-value pairs (since there are 50 sensors assumed per car) map each hour with keys for RegistrationNumber _N (N = 1 to 50). The output stream from Mapper will be as follows:

(01, 1), (03, 1), (01, 1), .., .., ...,

The (key, value) contains (alert/message Id, the instance number)

Mapper in ACPAMS

	Set of ata	MP 9CAZ331_0l MP04NPO 3_03 MP09HAM9.3_09 MF 3 A.523 J_06 MP09HAJ49J_ l MF 4YQHW_Ol MPOS:PS S t _...] MP 9CA...JJ_OI MP04NH5555_ ~ MP2 :F.4.1690_ .1
utput	C nvert iraro another set of dam Ke~ Value K r –Allerl!M sage]d -alu - Nl1m Ji of instaace	(01,1), (03,1), (09,1), (: 01 J . '(H 1 (23,1), (01,1), (23,1), (23,1

Combiner in ACPAMS

Input	Key-Value Pain {ootput of Mapper)	(Ol,1)~ (03,l)~(09,1). (06~l),(Ol,1)~ (01.l)~ (23, 1), (01.1), (23, u, (23, l)
Output	Key-Value Pam {Group-By-key at Olde itseH)	(Ol,l)~(03,l), {09,1), (06,l), (01,2), (23,l), (01,1)~ {23,2)

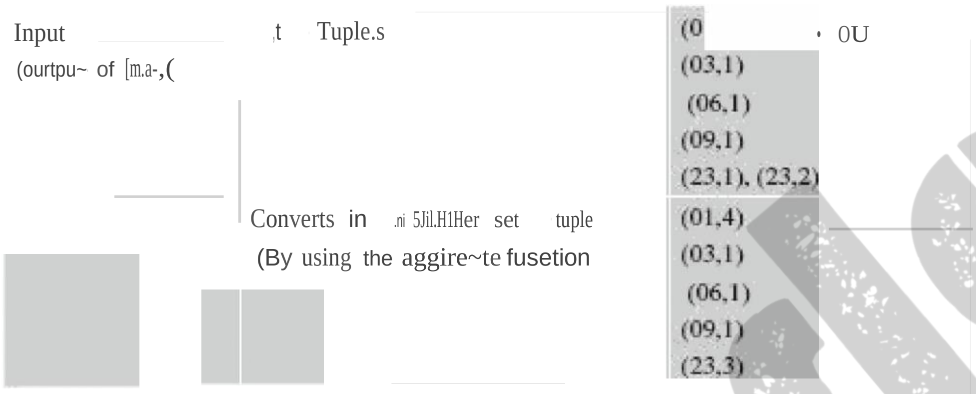
Shuffle in ACPAMS

	Key-Va!lue Pa.Lli: output of Combiner	(01,1), (03,1), (09,1), (06J , en, .. 23.1 .. (01,1), (23,2)
utpin	Key-Vah1 Pai~ 5 (moved to oth lr nod s ased om !ke r vailue	(01,1), (01,2), (01,1) (03,1) (09,1) (06,1) (23,1), (23,2)

Sort in ACPAMS

		(01,1), (01,2), (01,1) (03,1) (09,1) (06,1) (23,1), (23,2)
Output	Key-Value Pairs (Sort as per key)	(01,1), (01,2), (01,1) (03,1) (06,1) (09,1) (23,1), (23,2)

Reducer - Takes the output from Map (after the sort phase) as an input and combines the data tuples into a smaller set of tuples. Following are some examples of Reduce function in Alert/Message Count.



The output of the reducel), which is the final result of MapReduce Job provides a number of alert or messages at an hourly basis for the complaint raised by each component of the cars registered at the ACPAMS Company. Then the analyst decides which components need maintenance on high to low priority. The report of ACAMPS helps the company in improving the manufacturing of its car components.

The following example gives pseudocodes for an algorithm:

EXAMPLE 4.5

Write pseduocodes 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 E file f do {
               t = Substring (i, 2, After_}
               Bmit (term t, count 1,}}}

class Reducer {
      method Reduce (term t, counts [cl, cz, .....]} {
             Bum~ 0
             for all count c E counts [cl, cz, ....] do {
             sum ~ sum+ c}
             Bmit (term t, count Bum}}}
```

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.

4.2.8 Coping with Node Failures

NooofBi'h.ue, ocrn rs wen ttiie TdSkTiracter fans. t.o com:im.l111ic-at2 \V11h tna Jbblira.clter fur a p:re-d@liined periodl. llilte Jbblira(~.erirestnil:s the 1aslffradceirfm cop11ligwith the IfltU:l~ faih.JHQ..

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).

Self-Assessment Exercise linked to LO 4.1

1. Show MapReduce process diagrammatically to depict a client submitting a job, the workflow ofJobTracker and TaskTracker, and TaskTrackers creating the outputs.
2. Assume an input file size of 10 TB and a data block size of 128 MB. How many map tasks are required? Assume that each node does 100 maps. How many nodes are involved in processing? How will you change the number of map tasks per node to 120 using a Java statement?
3. Explain function of Group By, partitioning and combining using one example for each.
4. How does the data convert to (key, value) pairs before passing to the Mapper? How do the InputSplit and RecodReader function?
5. How are the failures of Map TaskTracker, Reduce TaskTracker and Master JobTracker handled in MapReduce?
6. How does the execution framework handle all aspects of distributed processing after a client node submits the job to the designated node of a cluster (the JobTracker)? Explain the concept using a diagram.

4.31 COMPOSING MAPREDUCE FOR CALCULATIONS AND ALGORITHMS

LO 4.2

Composing MapReduce programs for calculations, such as counting and summing, and algorithms for relational algebraic operations, such as projections, unions, intersections, natural joins, grouping, aggregation operations and matrix multiplication

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.

4.3.1 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. 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 ls 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 groupld, and the *Reducer* calculates the total number of occurrences for each such pair.

(ii) The *Mapper* emits the values and groupld, 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.

Filteringor 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.

[Stripe means a set of arrays associated with a dataset or a set of rows that belong to a common key with each row having a number of columns.]

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".

4.3.2 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,

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

$$\text{Assume } A = \begin{bmatrix} 1 & 5 & 4 \\ 2 & 1 & 3 \\ 4 & 2 & 1 \end{bmatrix} \text{ and } B = \begin{bmatrix} 4 \\ 1 \\ 3 \end{bmatrix}. \qquad \dots (4.1)$$

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

$$\text{Hence, } C = \begin{bmatrix} 21 \\ 18 \\ 21 \end{bmatrix} \qquad \dots (4.1)$$

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 $\sigma_{ACVM\_ID <= 525}$ (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)}.

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 Π_{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,

Rl = {(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 Rl and R3 is:

Rl 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,

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

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

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

Result of Intersection operation between Rl and R3 are

Rl ∩ R3 = {(12122011,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:

Rl = {(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_l is

Rl - R3 = (12122017,Oreo, 72) and set_2 is R3 - Rl = (12122017,Oreo, 25).

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

SymmetricDifference Symmetric difference (notation is *A* fl *B* (or *A* e B)] is another relational entity. It means the set of elements in exactly one of the two relations A or B. R3 e Rl = (12122017,Oreo, 25).

The *Mapper* emits all the tuples and tag. A tag is the name of the set (say, set_l or set_2 this tuple belongs to). The *Reducer* transfers only tuples that belong to neither set_l or set_2.

4.3.3.5 *Natural Join*

Consider two relations Rl and R2 for tuples a, band c. Natural Join computes for Rl (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, (Rl, a)) for each tuple (a, b) of Rl, similarly emits (b, (R2, c)) for each tuple (b, c) ofR2.

The *Mapper* is mapping both with Key for b. The *Reducer* transfers all pairs consisting of one with first component Rl and the other with first component R2, say (Rl, 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 (Rl, 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 INNERJoin 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 countl), suml), avgl), 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 MatrixMultiplication

Consider matrices named A (i rows andj columns) and BG 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:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1j} \\ a_{21} & a_{22} & \cdots & a_{2j} \\ \vdots & & & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{ij} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1k} \\ b_{21} & b_{22} & \cdots & b_{2k} \\ \vdots & & & \vdots \\ b_{j1} & b_{j2} & \cdots & b_{jk} \end{bmatrix} \quad C = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1k} \\ c_{21} & c_{22} & \cdots & c_{2k} \\ \vdots & & & \vdots \\ c_{i1} & c_{i2} & \cdots & c_{ik} \end{bmatrix} \quad \ldots (4.3)$$

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

LO 1.1 ... (4.4)

First row of C

C first column element = (a11b11+ a12b21 +.... + a1j.bji). Second column element = (a11b12+ a12b22+.... + a1j·bjz).
The kth column element= (a11b1k+ a12b2k+ .... + a1j bjk).

Second row of C

C first column element= (a21b11+ a22b21 +.... + a2j"bj1). Second column element = (a21b12+ a22b22+.... + azj·bjz).
The kth column element= (a21b1k+ a22b2k+ .... + azj· bjk).

The ith row of C

C first column element = (ai1b11+ ai2b21 +.... + aij.bj1). Second column element = (ailb12+ ai2b22+ .... + aij·bj2). The kth column element= (ai1b1k+ ai2b2k+ .... + aij.bjk).

Consider two solutions of matrix multiplication.

Matrix Multiplication with Cascading of Two MapReduce Steps

Table 4.2 gives the names, attributes, relations *RM* RB and Re, tuples in A, B, C, natural Join of RA and RB, keys and values, and seven steps for multiplication of A and 8.

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

	Step description			
	Name	~		
2	Sp iify attri1but of K.ey, lblli1 pBliirs of each lenremt [row number, co1Jllfilrli ll.llllm Ji' val IJl]	, J. J		
J	relations	R = A. {I,.1, v	$R_B = A (J, K, v_b)$	$R_C = A (I, K, v_c)$

4	Coasider tlIJ)l of Jr ~ nirn.d i, j ii	(.i!. k, CjJ
:S	Find natural Jrun - R andRg = Mattix: elements n.ij bJ y ts common r0 tltil]	mp1 s .t j, k, v • vb
	Get tapl for fi111dlliag Prodhict t	f'ou.rr-component tuple '.ii.j,k ',f)(v
7	oupin and aggrregatlon of luple tt'lb atrt:rfhut I and K	-cI, Jo 5 SUM v x "'mi.

The product A.B = Natural join of tuples in the relations RA and R8 followed by grouping and aggregation. Natural Join of A (I,J, v) and B (J, K, vb), having only attribute J in common = Tuples (i, j, k, va, vb) from each tuple (i, j, va> in A and tuples 0, k, vb) in B.

1. *MapReduce tasks for Steps 5 and* 6: Five-component tuple represents the pair of matrix elements
   (aij, bjk). Requirement is product of these elements. That means four-component tuple (i,j, k, va x vb),
   from equation (4.4) for elements Cik = Sum (aij.bjk) j= 1 to J'

   (a) Mapper Function: (i) *Mapper* emits the key-value pairs 0, (A, i, aij)) for each matrix element aij, and (ii) *Mapper* emits the key-value pair 0, (B, k, bjk)) for each matrix element aij.

   (b) Reduce Function: Consider the tuples of A = (A, i, aij) for each key j, consider tuples of
   B = (B, k, bjk) for each key j. Produce a key-value pair with key equal to (i, k) and
   value= aij x bjk· A and Bare just the names, may be represented by 0101 and 1010.

2. *Next MapReduce Steps* 7: Perform <I, K> i SUM (va x vb). That means do grouping and aggregation, with I and K as the grouping attributes and the sum of vax vb as the aggregation.

   (c) The *Mapper* emits the key-value pairs (i, k, v.) for each matrix element of C inputs with key i and k, and vc from earlier task of the reducer va x

(d) *Reducer* groups (i, k, v_c) in C using [C, I, k, sum (v_c)] from aggregated values of v_c from sum $(v_{.})$. 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.

Self-Assessment Exercise linked to LO 4.2

1. How does MapReduce program implement counting, filtering and parsing?
2. How does MapReduce collate all items which have the same value?
3. How does MapReduce perform graph analysis in a network of computing nodes to build a spanning tree information at a particular node?
4. How does MapReduce program collate and process items with the same value of the function together in an ETL operation?
5. How does MapReduce program implement <grouping attributes> **S** <function-list> (R)?

4.4 ! HIVE

LO 4.3

H1w. arrchiteii:wre. inst ~~aticm. rnmparisolill otl·nveam sto.rewitlh traditional databases

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. (Section 1.6.1. 7) 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.

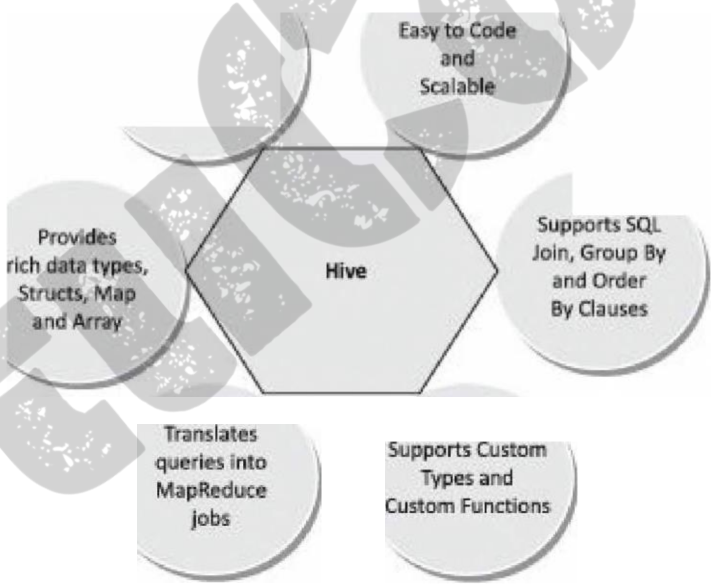


Figure 4.9 Main features of Hive

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).

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

Limitations

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.

4.4.1 Hive Architecture

Figure 4.10 shows the Hive architecture.

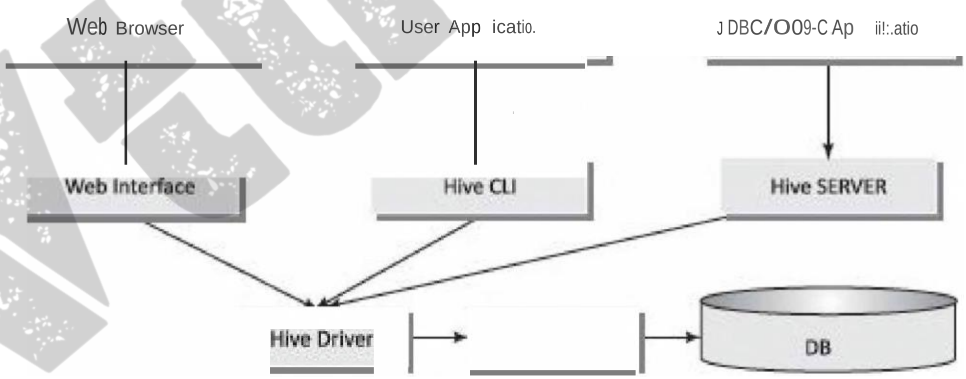


Figure4.10 Hive architecture

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:// hadoopi-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 (Iavac) and interpreter
- Hadoop
- Compatible version of Hive withJava- 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 Makejava 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 jdkl.7.0_71, export JAVA_HOME=/usr/local/jdkl.7.0_71, export PATH=$PATH: $]AV A_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.

4.4.3 Comparison with RDBMS (TraditionalDatabase)

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	Peta bytes	Tera bytes

Data per query	Peta bytes	Gigabytes
Query language	HiveQL	SQL
Support JDBC/ODBC	Limited	Full

4.4.4 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	UNIXtimestamp with optional nanosecond precision. It supports java.sql.Timestamp format "YYYY-MM-DD HH:MM:SS.fffffffff"
DATE	YYYY-MM-DDformat
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 ('keyl', 'a', 'keyz', '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,JSONand 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.

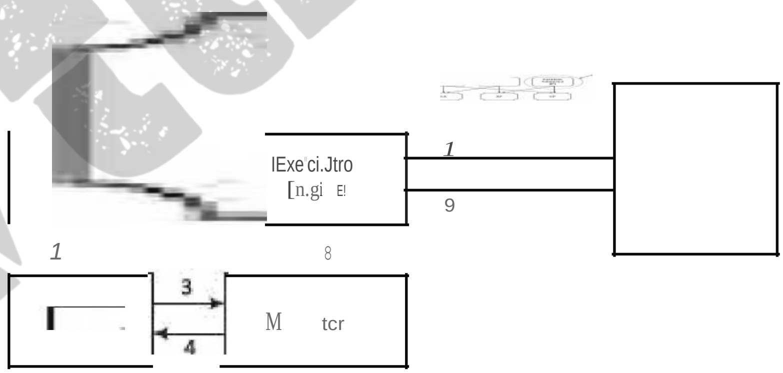


Figure 4.11 Dataflow sequences and workflow steps

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.

4.4.7 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 Return types, syntax, and descriptions of the functions

	Syntax	Description

BIG INT	round(double a)	Returns the rounded BIGINT (8 Byte integer) value of the 8 Byte double-precision floating point number a
BIG INT	floor(double a)	Returns the maximum BIGINT value that is equal to or less than the double.
BIG INT	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 strl, string str2, ...)	Returns the string resulting from concatenating strl 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	subs tr(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(l0.5) from marks; Output= 10.0

SELECT ceil(l0.5) from marks; Output= 11.0

Self-Assessment Exercise linked to LO 4.3

1. How does Hive install? What are the features of Hive? What are the components of the Hive architecture?
2. Give reasons for Hive provided with distinct integer types: TINYINT, SMALLINT, INT and BIGINT.
3. How does Hive use text, sequential, RC and ORC files?
4. How does the Hive use Collection data types: STRUCT, MAP and ARRAY?
5. How does Hive integrate with MapReduce and HDFS?
6. Give one example each of usages of round(), floorl), ceil(), rand() and upper() built-in functions in Hive.

4.5 | 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

Programme

Hi'll@QlL for CjUeryiliilg, sortiliilg, ag-g reyati ng, querj,iing scripts, ;umd hp Red um sonptrstor Joins and SlJlb-{j,l.!!leries 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 HiveQLData 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] [sdatabase name=] <table name>

[(<column name> <data type> [COMMENT <column comment=], ...)]

[COMMENT <table comment=]

[ROW FORMAT <row formats]

[STORED AS <file fornat-]

Table 4.9 gives the row formats in a Hive table.

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.narne'

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 di 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 (toyscompany)» 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?

Fltid	Data type
u tCategmy	stt ag
Pr uctlcl	1Dt
~ nr-tNariite	trlng
p ııı ~PFice	float

SOLUTION

CREATE TABLE IF NOT EXISTS toy _products (ProductCategory String,

```
ProductId int, ProductName String, ProductPrice float)
COMMENT'Toy details'
ROW FORMATDELIMITED
FIELDS TERMINATEDBY '\t'
LINES TERMINATEDBY '\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 DATABASEISCHEMA [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 (DATABASEISCHEMA) [IF EXISTS] <database name>
[RESTRICT|CASCADE];
```

The following example gives the sample usages of the commands.

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 (partcoll=vall,
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. vall is value assigned to partition column 1 (partcoll) and valz is value assigned to partition column 2 (partcolz).

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:

LO 1.3

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 DATALOCAL INPATH '/home/user/ jigsaw_puzzle_info.txt'

OVERWRITEINTO TABLEjigsaw_puzzle;

(ii) Alter the table using the following commands:

ALTERTABLE<name> RENAMETO <new name>

ALTERTABLE<name> ADD COLUMNS(<col spec> [, <col spec> ...])

ALTERTABLE<name> DROP [COLUMN]<column name>

ALTER TABLE <name> CHANGE <column name> <new name> <new type>

ALTERTABLE<name> REPLACECOLUMNS(<col spec-], <col spec> ...])

The following query renames the table from jigsaw _puzzle to toy _tbl:

ALTERTABLEjigsaw_puzzle RENAMETO toy_tbl;

The following query renames the column name ProductCategory to ProductCat:

ALTERTABLEtoy_tbl CHANGEProductCategory ProductCat String;

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

ALTERTABLEtoy_tbl CHANGEProductPrice ProductPrice Double;

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

ALTERTABLEtoy_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 STRINGProdName 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;

4.5.3 HiveQLFor Queryingthe 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>I [DISTRIBUTE BY <column
List>] [SORT BY <column List>]]
 [LIMIT number];
```

4.5.3.1 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, tr_1, tr_2, tr_3 and tr_4 horizontally row-wise. Each sub-table has one row and eight columns. Now, convert each sub-table tr_1, trz, tr_3 and tr_4 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 'locationl'] 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;
```

The following example explains concept of add, rename and drop a partition.

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 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/filel
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
```

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.

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.

4.5.3.2 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 ith bucketed column. The hash value hi = hashing function H(Cbucket) *mod* (Nbuckets).

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_ t.b I?
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 Productld from the second bucket.

```
SELECT DISTINCT Productid FROM toy_tbl_buckets
TABLE FOR 10725(BUCKET 2 OUT OF 5 ON Price US$);
```

4.5.3.3 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 HiveQLView 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.

4.5.3.4 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 JOINToy_Airplane
    ON (Toy_Airplane.Productld= productld.id) WHERE productld=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 JOINToy_Airplane
    ON (Toy_Airplane.Productld= productld.id) WHERE productld=10725
    ) toys_ catalog SELECT prdocutMfgDate WHERE prdocutMfgDate = '2017-10-23';
```

4.5.4 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
BIG INT	count(*), count(expr)	Returns the total number of retrieved rows.
DOUBLE	surntcol), 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;

4.5.5 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.

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 gives an example of a table named *price* of ID or Product ID and ProductCost.

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);
```

4.5.6 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 WHEREclause 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>I [DISTRIBUTE BY <column List>J [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;
```

Self-Assessment Exercise linked to LO 4.4

1. **What are the results after execution of the following command?**

```
CREATE TABLE IF NOT EXISTS toy_products
 (ProductCategory String, Productid int, ProductNarne
String, ProductPrice float)

COMMENT 'Toy details'

ROW FORMAT DELIMITED

FIELDS TERMINATED BY '\t'

LINES TERMINATED BY '\n'

STORED AS TEXTFILE;
```

2. **What do the following statements mean?**

```
ALTER TABLE <puzzle info> RENAME TO
<jigsaw_puzzle info>

ALTER TABLE <jigsaw_puzzle info> ADD COLUMNS
 (<puzzle code_narne>[,<pieces> [,<puzzle_cost> [, ... ])

ALTER TABLE <jigsaw_puzzle info> DROP [COLUMN]
<puzzle cost_US$>
```

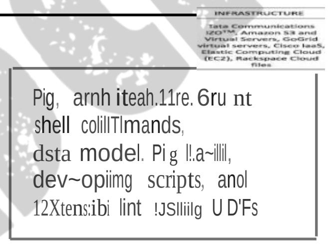
3. **How do you create partitions and buckets in a Hive database?**
4. **Consider sales table for all five car models at a large number of showrooms How are the sales figures of a specific model queried?**
5. **Explain the meaning of the following statements:**

```
SELECT [ALL | DISTINCT] <select expression>, <select
```

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

4.6 | 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.

1. Applications ofApache 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.

2. 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 MapReducejobs. 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, *]Ruby.* 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. Table 4.13 gives differences between Pig and MapReduce.

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
PerformingJoin, 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	ComplexJava implementations require exposure to Java language
Uses multi-query approach, thereby reducing the length of the codes significant! y	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

Pig and Hive codes, both create MapReduce jobs when execute. Hive in some cases, operates on HDFS in a similar way Apache Pig does. Table 4.15 gives a few significant points that set Pig apart from Hive.

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

3. 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.

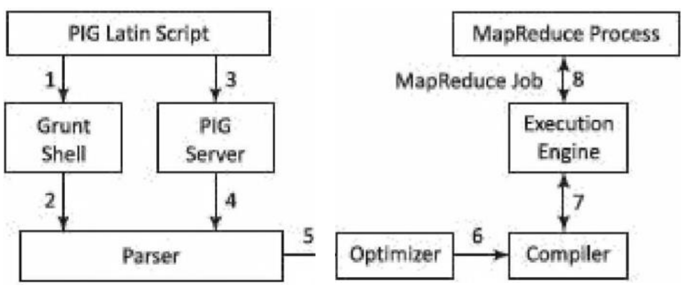


Figure 4.12 Pig architecture for scripts dataflow and processing

The three ways to execute scripts are:

1. **GruntShell:** 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 MapReducejobs execute and it outputs the final result.

4.6.1 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 ofls command:

```
grunt> sh ls
```

Grunt shell includes a set of utility commands. Included utility commands are clear, help, history, quit and set. The shell includes commands such as exec, kill and run to control the Pig from the Grunt shell.

4.6.2 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
```

4.6.3 Pig LatinData 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.

Table 4.16 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	[Numberst]
int	Signed 32-bit integer	10
long	Signed 64-bit integer	lOL or 101
float	32-bit floating point	22.7F or 22.7f
double	64-bit floating point	3.4 or 3.4e2 or 3.4E2
char array	Char[], Character array	data analytics
byte array	BLOB (Byte array)	ff00

A simple atomic value is known as a field. For example, 'Oreo' or '10' are fields. Atomic means non-divisible. NULL denotes an unknown or non-existent value in Pig Latin.

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)

Indices of the fields access the fields in each tuple. Tuples are ordered, like $1 from above tuple will return a value 'Oreo'.

Figure 4.13 shows Pig data model with fields, Tuple and Bag.

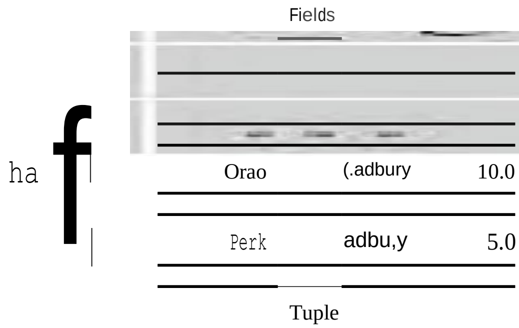


Figure 4.13 Pig Data Model with fields, Tuple and Bag

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. It is similar to a table in RDBMS, but unlike a table in RDBMS, it is not necessary that every tuple contains the same number of fields or that the fields in the same position (column) have the same type. 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,s)}) (Nestle {{Kitkat,15)})

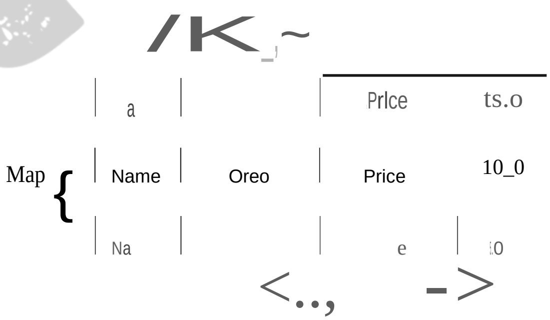


Figure 4.14 Relation and corresponding keys and their values (key-value pairs)

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#lO]

4.6.4 Pig Latinand 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 shows the order of processing Pig statements-Load, dump and store.

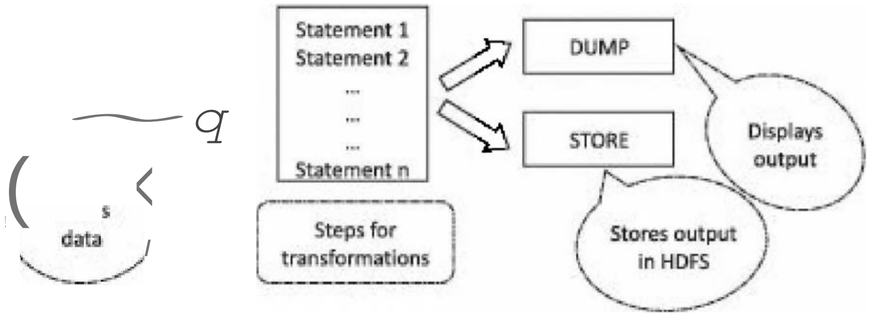


Figure 4.15 Order of processing Pig statements-Load, dump, and store

Operato1S In Pig Latin

Arithmetic Operators	+			*	/	%
Usedlmr	A.drlilioo.	Subtraction		Multiplication	Division	Remainder
C~arison Ofoerarors		=		>	≤	≥
Usedl for	EqJmalH.y	Not eqnnl	Less thilo	Greater than	Less than and equ to	Greater than and equal [O
BooJean Openrton	AND	OR	NOJ			
Used for	**Logical AND**	LogiC31 OR	Loglcal NOJ			

4.6.4.1 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 MapReducejob 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.

4.6.4.2 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.

Table 4.17 Applying transformations on columns using FOREACH operator

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);`
	`Al= foreach A generate x, y + z as yz;`
	`B = group Al by x;`
	`C = foreach B generate SUM(Al.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 (char array) 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 whichthed~asort. 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;`

DistinctDISTINCT 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)`

Pig also supports *outer joins.* Tuples which do not have a match on the other side are included, with null values being filled for the missing fields in outer joins. Outer joins can be *left*, *right* or *full.* A left outer join means tuples from the left side will be included even when they do not have a match on the right side. Similarly, a *right* outer join means tuples from the right side will be included even when they do not have a match on the left side. A full outer join implies tuples from both sides are taken even when they do not have matches.

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.

Any relational operator in Pig Latin can attach PARALLEL. However, it controls only reduce-side parallelism, so it makes sense only for operators that force a reduce phase, such as group, order, distinct,join or limit.

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)Ruby.

A UDF should extend a Filter function or Eval function and must contain a core method called *exec,* which contains a Tuple.

The UDF class extends the *Evalfunc* class which is the base for all Eval functions. All evaluation functions extend the Java class *'org.apache.pig.Evalfunc'.* It is parameterized with the return type of the UDF which is a Java String in this case.

Filter functions are Eval functions that return a Boolean value. The UDF class when extends the *Filterfunc* class can be used anywhere a Boolean expression is appropriate, including the FILTER operator or Bincond expression.

The following example gives the codes for developing a user-defined function (UDF) returning Boolean after checking the age.

EXAMPLE 4.17

Write a UDF 'IsCorrectAge' which checks if the age given is correct or not. UDF should return a Boolean value: True or False. If the Tuple is null or zero then also it should return False. Use Java. Create a JAR file and then export. Later register the JAR file. The JAR files are in the library files of Apache Pig at the time of loading.

SOLUTION

Followings are the codes for the user-defined function `IsCorrectAge`.

```
public clans I~CorrectAge extend~ FilterFunc {
@Override
   Public Boolean exec {Tuple tuple) throws IOException
   if (tuple== null || tuple.size() == 0) {
         return false,

   try
         Object object= tuple.get(O};
         if {object== null)
               return false;

         Inti=   {Integer} object;
         if  <i == as 11 i == 20 || i   21 11 i   25) {
               return true;
         else  {
               return false,

      catch (BxecException e} {
         throw new IOHxception(e);
```

Once `I sCorrectAge` create, the following command registers a JAR file into the library of JAR files:

```
register myudf.jar;
A= load 'input' as (name chararray, age int);
X = filter A by IsCorrectAge(age);
```

Self-Assessment Exercise linked to LO 4.5

1. How does Apache Pig execution engine function for faster data processing?

2. List the Grunt shell commands and the use of each command.
3. When is Hive and when is Pig used?
4. How are tuple and map used?
5. How are projections used?
6. Write the functions of GROUP, JOIN, FILTER, LIMIT, ORDER BY, PARALLEL, SORT and SPLIT.
7. How will a UDF return the difference of maximum and minimum sales from sales data values in Pig Latin?