**Design Pattern:-**

MapReduce design patterns fill this same role in a smaller space of problems and solutions.

They provide a general framework for solving your data computation issues,

without being specific to the problem domain.

MapReduce is a new technology with a

fast adoption rate and there are new developers joining the community every day

Map‐Reduce design patterns also provide a common language for teams working together

on MapReduce problems. Suggesting to someone that they should use a “reduce-side

join” instead of a “map-side replicated join” is more concise than explaining the lowlevel mechanics of each.

MapReduce and Hadoop Refresher:-

Hadoop MapReduce jobs are divided into a set of map tasks and reduce tasks that run

in a distributed fashion on a cluster of computers.

Each task works on the small subset of the data it has been assigned so that the load is spread across the cluster.

The map tasks generally load, parse, transform, and filter data.

Each reduce task is responsible for handling a subset of the map task output. Intermediate data is then copied from

mapper tasks by the reducer tasks in order to group and aggregate the data.

It is incredible what a wide range of problems can be solved with such a straightforward paradigm,

from simple numerical aggregations to complex join operations and Cartesian products.

The input to a MapReduce job is a set of files in the data store that are spread out over

the Hadoop Distributed File System (HDFS). In Hadoop, these files are split with an input

format, which defines how to separate a file into input splits. An input split is a byteoriented

view of a chunk of the file to be loaded by a map task.

Each map task in Hadoop is broken into the following phases:-

--> record reader,

-->mapper,

-->combiner,

-->partitioner

The output of the map tasks, called the intermediate keys and

values, are sent to the reducers.

The reduce tasks are broken into the following phases:

-->shuffle,

--> sort,

--> reducer,

--> output format

The nodes in which the map tasks run are

optimally on the nodes in which the data rests. This way, the data typically does not

have to move over the network and can be computed on the local machine.

-->Record Reader:-

The record reader translates an input split generated by input format into records.

The purpose of the record reader is to parse the data into records, but not parse the

record itself.

It passes the data to the mapper in the form of a key/value pair.

-->Map

In the mapper, user-provided code is executed on each key/value pair from the

record reader to produce zero or more new key/value pairs, called the intermediate

pairs.

-->Combiner

The combiner, an optional localized reducer, can group data in the map phase. It

takes the intermediate keys from the mapper and applies a user-provided method

to aggregate values in the small scope of that one mapper.

EX:\_

you can produce an

intermediate count and then sum those intermediate counts for the final result.

Sending (hello world, 3) instead of (hello world, 1) (hello world, 1)(hello world, 1) (3 times)

Combiner provide extreme performance gains with no downside.

-->partitioner

The partitioner takes the intermediate key/value pairs from the mapper (or combiner

if it is being used) and splits them up into shards, one shard per reducer.

ex:- in word count program , we can split data into 3 partation. 1- a-z 2- 0-9 3- special chaterter

no of partation =no of reducer // 3 reducer need and it will provide 3 output.

By default, the partitioner interrogates the object for its hash code, which is typically

an md5sum. Then, the partitioner performs a modulus operation by the number

of reducers: key.hashCode() % (number of reducers).

-->shuffle and sort

The reduce task starts with the shuffle and sort step. This step takes the output files

written by all of the partitioners and downloads them to the local machine in which

the reducer is running. These individual data pieces are then sorted by key into one

larger data list. The purpose of this sort is to group equivalent keys together so that

their values can be iterated over easily in the reduce task. This phase is not customizable

and the framework handles everything automatically. The only control

a developer has is how the keys are sorted and grouped by specifying a custom

Comparator object.

-->reduce

The reducer takes the grouped data as input and runs a reduce function once per

key grouping. The function is passed the key and an iterator over all of the values

associated with that key. A wide range of processing can happen in this function,

as we’ll see in many of our patterns. The data can be aggregated, filtered, and combined

in a number of ways. Once the reduce function is done,

it sends zero or more key/value pair to the final step, the output format.

Like the map function, the re

duce function will change from job to job since it is a core piece of logic in the

solution.

-->output format

The output format translates the final key/value pair from the reduce function and

writes it out to a file by a record writer. By default, it will separate the key and value with a tab

and separate records with a newline character.

word count ex ---------------------

public static class WordCountMapper

extends Mapper<Object, Text, Text, IntWritable> {

private final static IntWritable one = new IntWritable(1);

private Text word = new Text();

public void map(Object key, Text value, Context context)

throws IOException, InterruptedException {

// Parse the input string into a nice map

Map<String, String> parsed = MRDPUtils.transformXmlToMap(value.toString());

// Grab the "Text" field, since that is what we are counting over

String txt = parsed.get("Text");

// .get will return null if the key is not there

if (txt == null) {

// skip this record

return;

}

// Unescape the HTML because the data is escaped.

txt = StringEscapeUtils.unescapeHtml(txt.toLowerCase());

// Remove some annoying punctuation

txt = txt.replaceAll("'", ""); // remove single quotes (e.g., can't)

txt = txt.replaceAll("[^a-zA-Z]", " "); // replace the rest with a space

// Tokenize the string by splitting it up on whitespace into

// something we can iterate over,

// then send the tokens away

StringTokenizer itr = new StringTokenizer(txt);

while (itr.hasMoreTokens()) {

word.set(itr.nextToken());

context.write(word, one);

}

}

}

-----------------------------------------------

public static class IntSumReducer

extends Reducer<Text, IntWritable, Text, IntWritable> {

private IntWritable result = new IntWritable();

public void reduce(Text key, Iterable<IntWritable> values,

Context context) throws IOException, InterruptedException {

int sum = 0;

for (IntWritable val : values) {

sum += val.get();

}

result.set(sum);

context.write(key, result);

}

}

----------------------------------

usage of job.setCom

binerClass. In some cases, the combiner simply cannot be used due to the nature of

the reducer. In other cases, the combiner class will be different from the reducer class.

The combiner is very effective in the “Word Count” program and is quite simple to

activate.

Next is the mapper code that parses and prepares the text. Once some of the punctuation

and random text is cleaned up, the text string is split up into a list of words

in mapper class ..

The first function, MRDPUtils.transformXmlToMap, is a helper function to parse a line

of Stack Overflow data in a generic manner. You’ll see it used in a number of our examples.

It basically takes a line of the StackOverflow XML (which has a very predictable

format) and matches up the XML attributes with the values into a Map.