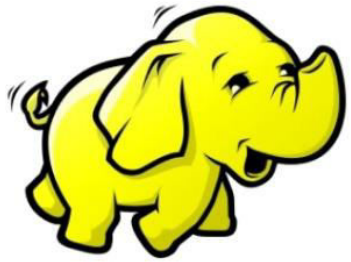
edureka!

Big Data & Hadoop



Course Topics

edureka!

✓Module 1
- ✓Understanding Big Data
- ✓Hadoop Architecture

✓Module 2
- ✓Hadoop Cluster Configuration
- ✓Data loading Techniques
- ✓Hadoop Project Environment

✓Module 3
- ✓Hadoop MapReduce framework
- ✓Programming in Map Reduce

✓Module 4
- ✓MapReduce
- ✓Combiners and Partitioners

✓Module 5
- ✓Analytics using Pig
- ✓Understanding Pig Latin

✓Module 6
- ✓Analytics using Hive
- ✓Understanding HIVE QL

✓Module 7
- ✓Advance Hive
- ✓NoSQL Databases and HBASE

✓Module 8
- ✓Advance HBASE
- ✓Zookeeper Service

✓Module 9
- ✓Advance MapReduce
- ✓Joins and Testing Examples

Module 10
- ✓Apache Oozie
- ✓Real world Datasets and Analysis
- ✓Project Discussion

Advance MapReduce Contents

- ✓COUNTERS
- ✓DISTRIBUTED CACHE
- ✓JOINS – MAP-SIDE AND REDUCE-SIDE
- ✓CUSTOM INPUT FORMAT
- ✓SEQUENCE FILE FORMAT
- ✓MRUNIT TESTING

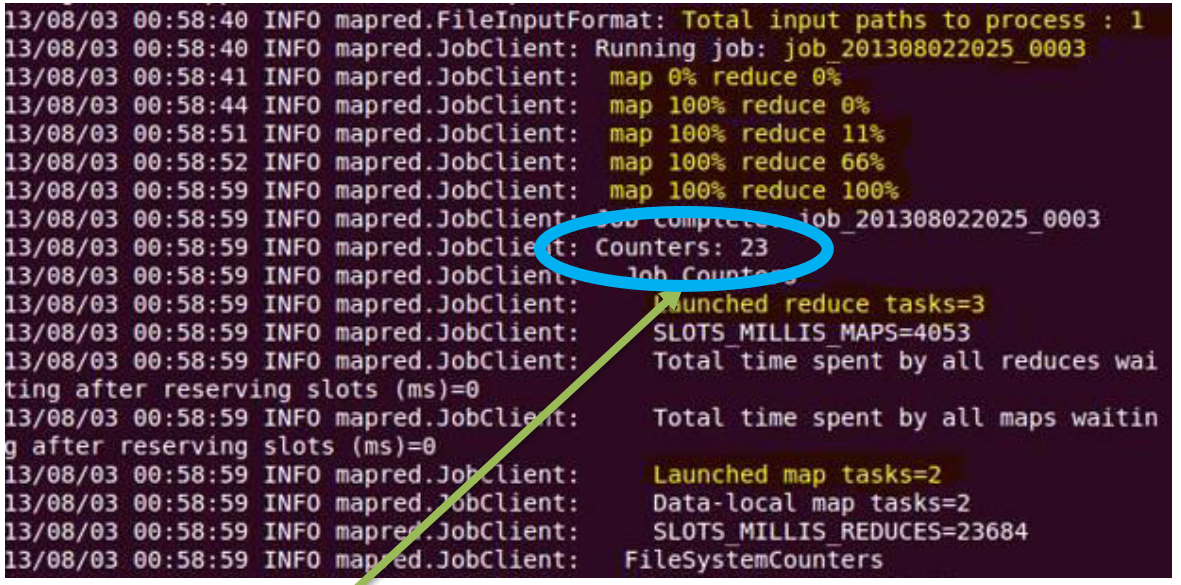
Counters

Counters are lightweight objects in Hadoop that allow you to keep track of system progress in both the map and reduce stages of processing.

You can see some of them each time you are running an hadoop job at the client console as MR dump.

By default, Hadoop defines a number of standard counters in "groups"; like JobCounters, MapReduce Framework etc.

Counters



Counters

Counters

Built In Counters In Hadoop

Hadoop provides some inbuilt counters for every job, for purposes like counting the number of records or bytes processed.

1) Map Reduce Task Counters - '*MAP_INPUT_RECORDS*' which is used to count the number of input records read by each map task. The output is aggregated over all the tasks in a particular job.

2) Job Counters - By the JobTracker to collect statistics about the entire job. Example of this counter is '*TOTAL_LAUNCHED_MAPS*' which is used to count the number of map tasks that were launched over the course of a job.

Counters

Custom Java Counters

Map Reduce allows users to specify their own counters for performing their own counting operation. A custom counter is defined by a Java enum, that groups related counters. The syntax for defining a Custom Counter is:-

enum MyCounter {
MISSING,
TOTAL
}

where the counter MISSING is used to maintain the count of the missing records and the counter TOTAL is used to maintain the count of the total number of records . MyCounter represents the group of these two counters. A user can define any number of enums to group related counters.

Note:- Counters are global. The counter values are shared by all the map and reduce tasks across the MapReduce framework and aggregated at the end of the job across all the tasks.

Counters

First you have to define an Enum which represents a group of counters.

public enum MyCounters { Good,Bad }

From within the map method of our mapper, we can access the counter and increment it when we observe Good or Bad records.

context.getCounter(MyCounters.Good).increment(1);

Counters

Demo: Counters

Distributed Cache

Application may require some dependency/configuration/property files

A distributed spell-check application would require every Mapper to read in a copy of the dictionary.

Suppose for the processing of employee's salary we need data on their grades which is available with another file .

Distributed Cache

Hadoop has the concept of a distributed cache which all slaves (nodes) have access to.

When we want to distribute some common data across all task trackers we go for distributed cache.

The distributed cache can contain small data files needed for initialization or libraries of code that may need to be accessed on all nodes in the cluster.

DistributedCache is a facility provided by the Map-Reduce framework to cache files (text, archives, jars etc.) needed by applications.

Distributed Cache

Input data file

Tejas	Saravana	IT	up	270000
Daniel	Hauge	HR	ma	290000
Ajay	Bhosle	FIN	up	270000
Swetha	Reddy	IT	wb	129000
Martin	Alejandro	FIN	ma	220000
Anitha	Joshi	HR	bi	175000
Vivek	Rana	IT	bi	270000
Fahad	Jafri	FIN	ma	250000

.....

To be cached

up	Uttar_Pradesh
ma	Maharashtra
bi	Bihar
wb	WestBengal

Suppose for the processing of employee's records we need data on their detail location which is available with another file .

Distributed Cache

Use the `DistributedCache.addCacheFile()` method to add names of files which should be sent to all nodes on the system.

The file names are specified as `URI` objects; unless qualified otherwise, they assume that the file is present on the HDFS in the path indicated.

When you want to retrieve files from the distributed cache (e.g., when the mapper is in its `configure()` step), use the `DistributedCache.getLocalCacheFiles()` method to retrieve the list of paths local to the current node for the cached files.

These are copies of all cached files, placed in the **local** file system of each worker machine. (They will be in a subdirectory of `mapred.local.dir`.)

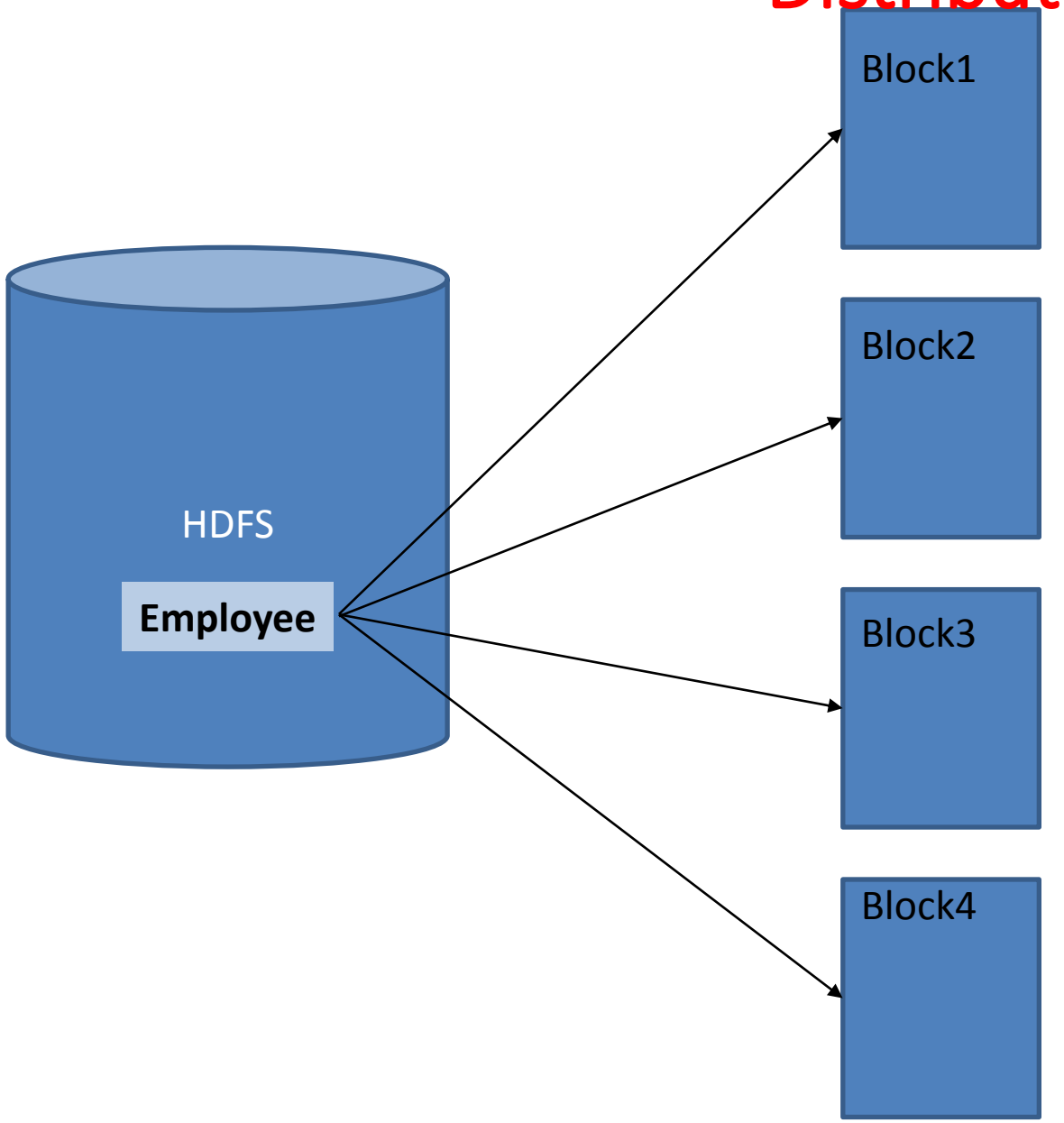
In Driver class

```
try{
   DistributedCache.addCacheFile(new URI("/state.dat"), job.getConfiguration());
   }catch(Exception e){
      System.out.println(e);
   }
```

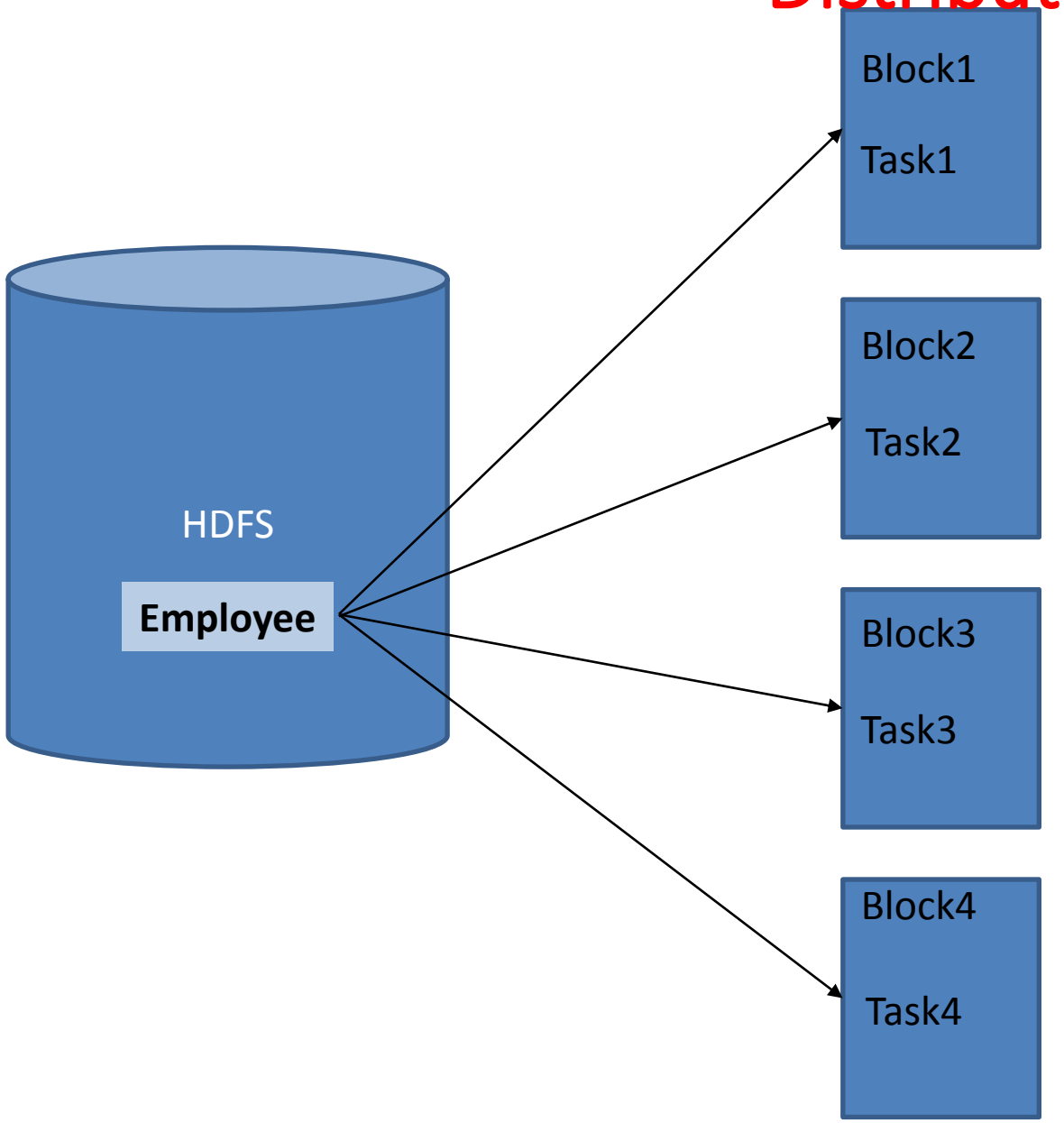
In Mapper

```
Path[] files = DistributedCache.getLocalCacheFiles(context.getConfiguration());
```

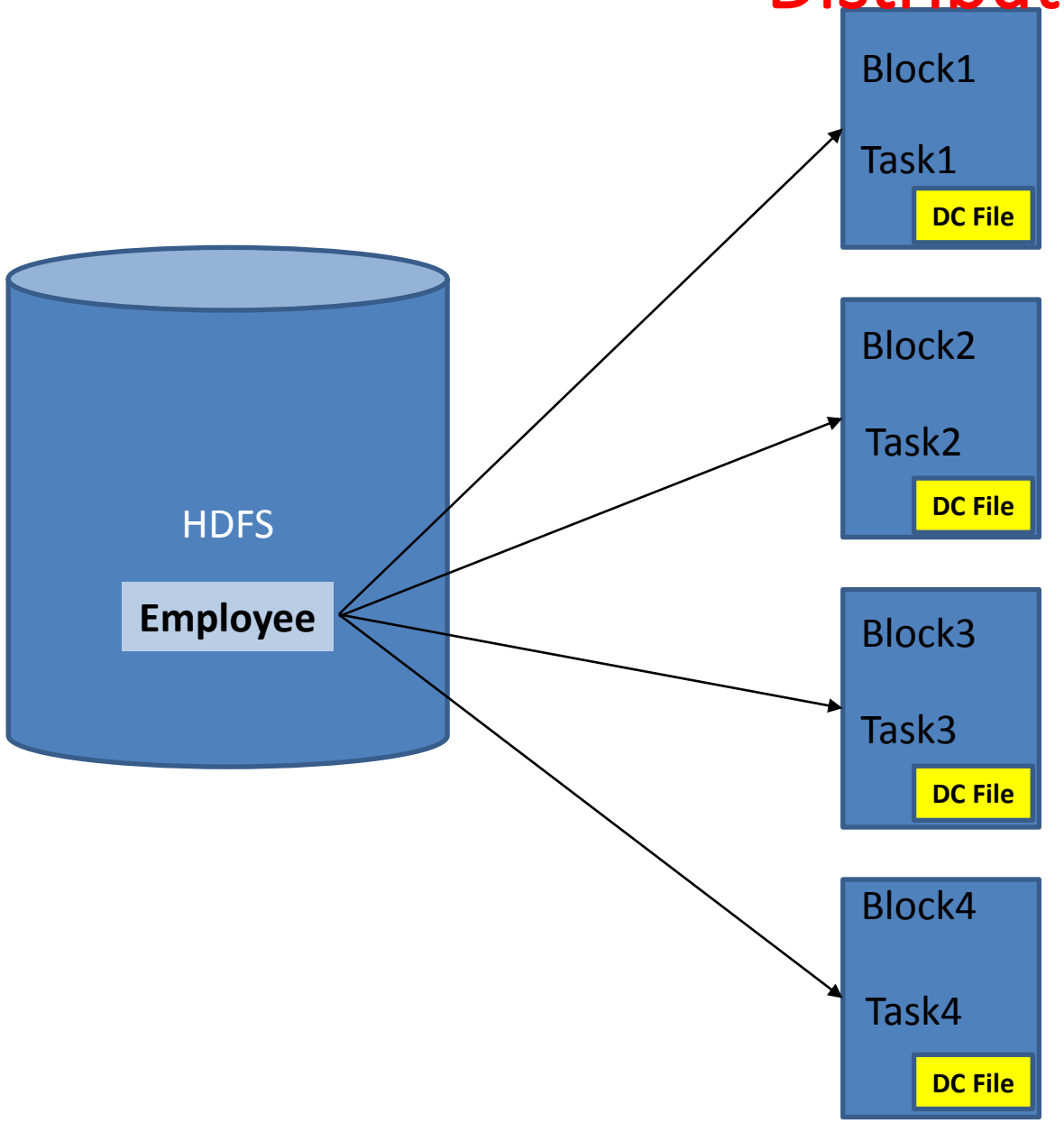
Distributed Cache



Distributed Cache



Distributed Cache



Distributed Cache

Demo: Distributed Cache

joins

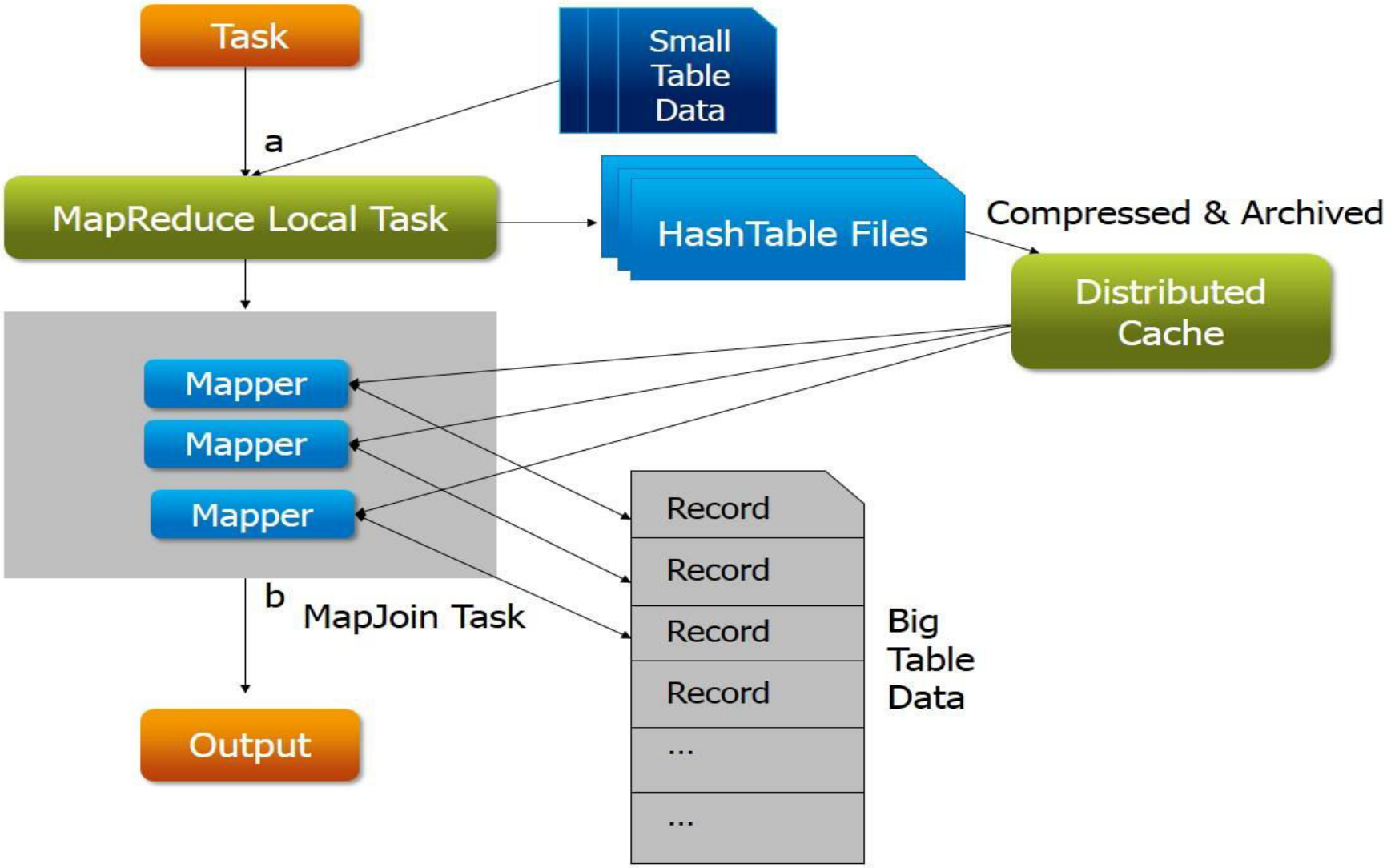
MapReduce can perform joins between large datasets, but writing the code to do joins from scratch is fairly involved.

Rather than writing MapReduce programs, you might consider using a higher-level framework such as Pig, Hive, in which join operations are a core part of the implementation.

Map-side joins, as the name implies, read the data streams into the mapper and uses logic within the mapper function to perform the join.

In contrast, a **reduce-side join** has the multiple data streams processed through the map stage without performing any join logic and does the joining in the reduce stage.

Joins - Using the Distributed Cache



Joins

Demo: Joins in MapReduce

http://www.edureka.in/blog/map-side-vs-join/

Files, splits and records

1.Files will be physically split and kept as blocks in hdfs.
2.Logical inputSplit will be created as a part of the job startup and the data will be sent to the mapper implementation.
3.InputFormat class will provide how to split an input file into the splits required for map processing.
4.InputFormat class will also create a RecordReader class that will generate the series of key/value pairs from a split.
5.The combination of the InputFormat and RecordReader classes therefore are all that is required to bridge between any kind of input data and the key/value pairs required by MapReduce.

Steps

1.Create Custom Key/Value Pair
2.Create Custom InputFormat
3.Create RecordReader
4.Implement Custom InputFormat in mapper
5.Define CustomInputFormat in Driver

Custom InputFormat

First define 2 custom WritableComparable classes. These classes represent the key-value pairs that are passed to the mapper, much as how TextInputFormat passes LongWritable and Text to the mapper.

To demonstrate how to create a custom WritableComparable, we shall write an implementation that represents a group of strings, called MyKey and MyValue.

Sample Data

==========

Sensor Type	timestamp	status	value1	value2
a	1386023259550	on	22	23
b	1389523259550	off	33	34
a	1386023259550	off	44	43
a	1389523259550	on	34	24

Key - Group of SensorType,timestamp,status
Value - Group of value1,value2

WritableComparable

```
public void readFields(DataInput in) throws IOException{
        value1.readFields(in);
        value2.readFields(in);
}

public void write(DataOutput out) throws IOException{
        value1.write(out);
        value2.write(out);
}
public int compareTo(Object o){
        MyValue other = (MyValue)o;
        int cmp = value1.compareTo(other.value1);
        if(cmp != 0){
                return cmp;
        }
        return value2.compareTo(other.value2);
}
```

1.Define Text instance variables and associated constructors, getters and setters .

1. write method serializes each Text object in turn to the output stream, by delegating to the Text objects themselves.

1. readFields() deserializes the bytes from the inputstream by delegating to each Text object.

1. compareTo() method that imposes the ordering you would expect: it sorts by the first string followed by the second

InputFormat

Next, we need to create an InputFormat to serialize the text from our input file and create the MyKey and MyValue instances. This input format extends the Hadoop FileInputFormat class and returns our own implementation of a RecordReader:

```
public class MyInputFormat extends FileInputFormat<MyKey,MyValue> {


        @Override
        public RecordReader<MyKey, MyValue> createRecordReader(InputSplit arg0,
                                TaskAttemptContext arg1) throws IOException, InterruptedException {
                return new MyRecordReader();
        }
}
```

MyRecordReader

Now, create a RecordReader to read from the input data file.

```
private MyKey key;
private MyValue value;
private LineRecordReader reader = new LineRecordReader();
public void initialize(InputSplit is, TaskAttemptContext tac)
                                        throws IOException, InterruptedException
{
                        reader.initialize(is, tac);

}
public boolean nextKeyValue() throws IOException, InterruptedException {
                boolean gotNextKeyValue = reader.nextKeyValue();
                if(gotNextKeyValue){
                                if(key==null){
                                                key = new MyKey();
                                }
                                if(value == null){
                                                value = new MyValue();
                                }
                                                Text line = reader.getCurrentValue();
                                                String[] tokens = line.toString().split("\t");
                                                key.setSensorType(new Text(tokens[0]));
                                                key.setTimestamp(new Text(tokens[1]));
                                                key.setStatus(new Text(tokens[2]));
                                                value.setValue1(new Text(tokens[3]));
                                                value.setValue2(new Text(tokens[4]));
                }
                else {
                                                key = null;
                                                value = null;
                }
                return gotNextKeyValue;
}
```

Custom InputFormat

Finally, create a simple map-only job to test the InputFormat

```
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class MyMapper extends Mapper<MyKey, MyValue, Text, Text> {

        protected void map(MyKey key, MyValue value, Context context)
            throws java.io.IOException, InterruptedException {

          String sensor = key.getSensorType().toString();

          if(sensor.toLowerCase().equals("a")){
                context.write(value.getValue1(),value.getValue2());
          }

        }
}
```

```
public class MyFile {

  public static void main(String[] args)
                throws …… {
Job job = new Job();
    job.setJarByClass(MyFile.class);
    job.setJobName("CustomTest");
    job.setNumReduceTasks(0);
    job.setMapperClass(MyMapper.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(Text.class);
    job.setInputFormatClass(MyInputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
  }
}
```

Demo: Custom Input Format

SequenceFile

✓Consist of binary Key/Value pair

✓Used in MapReduce as input/output formats

Output of Maps are stored using SequenceFile

✓The Sequence files have following advantages:

- ✓As binary files, they are intrinsically more compact than text files
- ✓They additionally support optional compression, which can also be applied at different levels, that is, compress each record or an entire split
- ✓The file can be split and processed in parallel

✓SequenceFiles have three compression options:

- ✓Uncompressed: Key-value pairs are stored uncompressed
- ✓Record compression: The value emitted from a mapper or reducer is compressed more compact than text files
- ✓Block compression: An entire block of key-value pairs is compressed

SequenceFile

✓Non splitable format

Most binary formats(compressed or encrypted) —cannot be split means that a single mapper will be used to process the entire file, causing a potentially large performance hit. In such a situation, it is preferable to either use a splitable format such as SequenceFile.

✓Too many small files

For example, a 10GB file broken up into files of size 100KB each, use a map of their own. Thus the time taken to finish the job considerably increases.

SequenceFile

Demo: Sequence files

MRUnit

MRUnit is a unit testing framework designed specifically for Hadoop.

It began as an open source offering included in Cloudera's distribution for Hadoop, and is now an Apache Incubator project.

MRUnit is based on JUnit, and allows for the unit testing of mappers, reducers, and some limited integration testing of the mapper-reducer interaction, along with combiners, custom counters, and partitioners.

It should be noted that MRUnit supports both the old (org.apache.hadoop.mapred) and new (org.apache.hadoop.mapreduce) MapReduce APIs.

Four types of tests provided by MRUnit:
1 A map test that only tests a map function (supported by the MapDriver class).
2 A reduce test that only tests a reduce function (supported by the ReduceDriver class).
3 A map and reduce test that tests both the map and reduce functions (supported by the MapReduceDriver class).
4 A pipeline test that allows a series of MapReduce functions to be exercised (supported by the TestPipelineMapReduceDriver class).

MRUnit

To write your test, you would do the following:

1. Instantiate the MapDriver class parameterized exactly as the mapper under test.

2. Add an instance of the mapper you are testing by using the withMapper call

3. The withInput call enables you to pass in a desired key and input value.

4. The expected output is specified using the withOutput call.

5. The last call, runTest, feeds the specified input values into the mapper, and compares the actual output against the expected output set in the withOutput method.

MRUnit

```
public void testReducer() throws Exception {
        List<IntWritable> values = new ArrayList<IntWritable>();
        values.add(new IntWritable(1));
        values.add(new IntWritable(1));
        new ReduceDriver<Text, IntWritable, Text, IntWritable>()
                .withReducer(new WordCount.Reduce())
                .withInput(new Text("cat"), values)
                .withOutput(new Text("cat"), new IntWritable(2))
                .runTest();
}
```

MRUnit

Following is a breakdown of what is happening in this code:

1. A list of IntWritable objects that are used as the input to the reducer is created.

2. A ReducerDriver is instantiated, it is parameterized exactly as the reducer under test.

3. An instance of the reducer you want to test is passed in using the withReducer call.

4. The withInput call enables you to pass in input values for a reducer.

5**.** You can specify the expected reducer output using the withOutput call.

6. Finally, you call runTest, which feeds the reducer the inputs specified, and compares the output from the reducer against the expected output.

MRUnit

MRUnit provides the MapReduceDriver class that enables you to test how a mapper and reducer are working together.

First, you parameterize the input and output types of the mapper, and the input and output types of the reducer. Because the mapper output types always match the reducer input types, you end up with three pairs of parameterized types.

Additionally, you can provide multiple inputs and specify multiple expected outputs.

MRUnit

```
public class MyMapperReducerTest {


  MapDriver<LongWritable, Text, Text, IntWritable> mapDriver;
  ReduceDriver<Text, IntWritable, Text, IntWritable> reduceDriver;
  MapReduceDriver<LongWritable, Text, Text, IntWritable, Text, IntWritable> mapReduceDriver;

public void setUp() {
    MyMapper mapper = new MyMapper();
    MyReducer reducer = new MyReducer();
    mapDriver = MapDriver.newMapDriver(mapper);
    reduceDriver = ReduceDriver.newReduceDriver(reducer);
    mapReduceDriver = MapReduceDriver.newMapReduceDriver(mapper, reducer);
  }
```

MRUnit

```
//You can run separate Mapper and Reducer Test as follows
@Test
  public void testMapper() {
    mapDriver.withInput(new LongWritable(), new Text("6"));
    mapDriver.withOutput(new Text("6"), new IntWritable(1));
    mapDriver.runTest();
  }
@Test
  public void testReducer() {
    List<IntWritable> values = new ArrayList<IntWritable>();
    values.add(new IntWritable(1));
    values.add(new IntWritable(1));
    reduceDriver.withInput(new Text("6"), values);
    reduceDriver.withOutput(new Text("6"), new IntWritable(2));
    reduceDriver.runTest();
  }
}
```

MRUnit

```
@Test
public void testMapReduce() throws Exception {
new MapReduceDriver<LongWritable, Text, Text, IntWritable, Text,IntWritable>()
.withMapper(new WordCount.Map())
.withReducer(new WordCount.Reduce())
.withConfiguration(new Configuration())
.withInput(new LongWritable(1), new Text("dog cat dog"))
.withInput(new LongWritable(2), new Text("cat mouse"))
.withOutput(new Text("cat"), new IntWritable(2))
.withOutput(new Text("dog"), new IntWritable(2))
.withOutput(new Text("mouse"), new IntWritable(1))
.runTest();
}
```

The MapReduceDriver class enables you to pass in multiple inputs that have different keys. Here, you are passing in two records — first with a LongWritable with an arbitrary value and a Text object that contains a line "dog cat dog", and second with a LongWritable with an arbitrary value and a Text object that contains the line "cat mouse".

MRUnit

PIPELINE TESTS
MRUnit supports testing a series of map and reduce functions—these are called *pipeline tests*.

You feed MRUnit one or more MapReduce functions, the inputs to the first map function, and the expected outputs of the last reduce function.

MRUnit has a few limitations:
■ The MapDriver and ReduceDriver support only a single key as input, which can make it more cumbersome to test map and reduce logic that requires multiple keys.
■ MRUnit isn't integrated with unit test frameworks that provide rich error-reporting capabilities for quicker determination of errors.
■ The pipeline tests only work with the old MapReduce API, so MapReduce code that uses the new MapReduce API can't be tested with the pipeline tests.
■ There's no support for testing data serialization, or InputFormat, RecordReader, OutputFormat, or RecordWriter classes.

End