# MapReduce and Hadoop: Advanced topics

## Multiple inputs



## Multiple inputs

- In some applications data are read from two or more datasets
  - Datasets could have different formats
- Hadoop allows reading data from **multiple inputs** (multiple datasets) with different **formats**
  - One different mapper for each input dataset must be specified
  - However, the key-value **pairs emitted** by the mappers must be **consistent in terms of data types**



## Multiple inputs

- Example of a use case
  - Input data collected from different sensors
  - All sensors measure the same "measure"
  - But sensors developed by different vendors use a different data format to store the gathered data/measurements



## Multiple inputs

- In the driver
  - Use the addInputPath method of the MultipleInputs class multiple times to
    - Add one input path at a time
    - Specify the input format class for each input path
    - Specify the Mapper class associated with each input path



## Multiple inputs

- E.g.,

```
MultipleInputs.addInputPath(job, new Path(args[1]),
TextInputFormat.class, Mapper1.class);

MultipleInputs.addInputPath(job, new Path(args[2]),
TextInputFormat.class, Mapper2.class);
```

  - Specify two input paths (args[1] and args[2])
  - The data of both paths are read by using the TextInputFormat class
  - Mapper1 is the class used to manage the input key-value pairs associated with the first path
  - Mapper2 is the class used to manage the input key-value pairs associated with the second path

# Multiple outputs



## Multiple outputs

- In some applications it could be useful to store the output key-value pairs of a MapReduce application in different files
  - Each file contains a specific subset of the emitted key-value pairs (based on some rules)
    - Usually this approach is useful for splitting and filtering operations
  - Each file name has a prefix that is used to specify the "content" of the file
- All the files are stored in one single output directory
  - i.e., there are no multiple output directories, but only multiple output files with different prefixes



## Multiple outputs

- Hadoop allows specifying the prefix of the output files
  - The standard prefix is "part-" (see the content of the output directory of some of the previous applications)
  - The MultipleOutputs class is used to specify the prefixes of the output files
    - One different prefix for each "type" of output file
    - There will be one output file of each type for each reducer (for each mapper for map-only jobs)



## Multiple outputs - Driver

- Use the method **MultipleOutputs.addNamedOutput** multiple times in the Driver to specify the prefixes of the output files
- The method has 4 parameter
  - The job object
  - The "name/prefix" of MultipleOutputs
  - The OutputFormat class
  - The key output data type class
  - The value output data type class
- Call this method one time for each "output file type"



## Multiple outputs - Driver

- E.g.,
  MultipleOutputs.addNamedOutput(job, "hightemp", TextOutputFormat.class, Text.class, NullWritable.class);

  MultipleOutputs.addNamedOutput(job, "normaltemp", TextOutputFormat.class, Text.class, NullWritable.class);
- This example defines two types of output files
  - The first type of output files while have the prefix "hightemp"
  - The second type of output files while have the prefix "normaltemp"



## Multiple outputs – Map-only example

- Define a private MultipleOutputs variable in the mapper if the job is a map-only job (in the reducer otherwise)
  - E.g.,
    - private MultipleOutputs<Text, NullWritable> mos = null;
- Create an instance of the MultipleOutputs class in the setup method of the mapper (or in the reducer)
  - E.g.,
    - mos = new MultipleOutputs<Text, NullWritable>(context);

## Multiple outputs – Map-only example

- Use the write method of the MultipleOutputs object in the map method (or in the reduce method) to write the key-value pairs in the file of interest
  - E.g.,
    - mos.write("hightemp", key, value);
      - This example writes the current key-value pair in a file with the prefix "hightemp-"
    - mos.write("normaltemp", key, value);
      - This example writes the current key-value pair in a file with the prefix "normaltemp-"

13

## Multiple outputs – Map-only example

- Close the MultipleOutputs object in the cleanup method of the mapper (or of the reducer)
  - E.g.,
    - mos.close();

14

## Distributed cache

15

## Distributed cache

- Some applications need to share and cache (small) read-only files to perform efficiently their task
- These files should be accessible by all nodes of the cluster in an efficient way
  - Hence a copy of the shared/cached files should be available in all nodes used to run the application
- **DistributedCache** is a facility provided by the Hadoop-based MapReduce framework to cache files
  - E.g., text, archives, jars needed by applications

16

## Distributed cache

- In the Driver of the application, the set of shared/cached files are specified
  - By using the job.**addCacheFile**(path) method
- During the initialization of the job, Hadoop creates a "local copy" of the shared/cached files in all nodes that are used to execute some tasks (mappers or reducers) of the job (i.e., of the running application)
- The shared/cache file is read by the mapper (or the reducer), usually in its setup method
  - Since the shared/cached file is available **locally** in the node, its content can be read efficiently

17

## Distributed cache

- The **efficiency** of the distributed cache depends on the **number of** multiple **mappers** (or reducers) running on the **same node**
  - For each node a local copy of the file is copied during the initialization of the job
  - The **local** copy of the **file** is **used by all mappers** (reducers) running on the **same node** (server)
- **Without the distributed cache**, each mapper (reducer) should read, in the setup method, the shared HDFS file
  - Hence, **more time** is needed because reading **data from HDFS** is more inefficient than reading data from the local file system of the node running the mappers (reducers)

18

# Distributed cache

Structure



# Distributed cache: driver

```
public int run(String[] args) throws Exception {

   .....

  // Add the shared/cached HDFS file in the
  // distributed cache
  job.addCacheFile(new Path("hdfs
  path").toUri());

  ......
}
```



# Distributed cache: mapper/reducer

```
protected void setup(Context context) throws IOException,
   InterruptedException {

        ........
        String line;

        // Retrieve the paths of the local copies of
        // the distributed files
        URI[] urisCachedFiles = context.getCacheFiles();
```



# Distributed cache: mapper/reducer

```
        // Read the content of the cached file and process it
        // in this example the content of the first shared file is opened
        BufferedReader file = new BufferedReader(new FileReader(
                new File(urisCachedFiles[0].getPath())));


        // Iterate over the lines of the file
        while ((line = file.readLine()) != null) {
                // process the current line
                .....
        }

        file.close();
}
```