1 – Energy Data Problem Domain

Energy Management Systems (EMSs) are energy monitoring tools that collect multiple data streams from energy meters and sensors, and integrate this information to identify savings opportunities and misuse situations.

Real-time decision making applications, that require (near) real-time integration of huge quantities of persisted sensor and meter data, results in Big Data challenges specially in cloud scenarios.

Let *EnergyMeterEvent* be a table of events e with schema S.

$$\text{EnergyMeterEvent} = <e_1, e_2, \ldots, e_n>,\ S = (\text{deviceId, measure, timestamp}).$$

Where `deviceId` is a string which represents the sensor identification, `measure` is a long integer representing the event value in Wh, and the `timestamp` is a long integer in seconds. In this tutorial, consider the present time as being the maximum `timestamp`, which has the value of 1388534373.

2 – Setting up the Eclipse environment

We suggest that you use the Eclipse IDE for this lab. The required libraries, configuration files, source files, and examples are available for download at:

`https://ftp.rnl.tecnico.ulisboa.pt/pub/classes/ds3/javaforbigdatalab.zip`.

In order to setup the Eclipse environment for our lab, proceed as follows:

1. Open Eclipse
2. Import the archive `javaforbigdatalab.zip` to Eclipse by choosing:
   `File -> Import -> General -> Existing Projects into Workspace`.
3. Select option `Select archive file`
4. Browse the file `javaforbigdatalab.zip`
5. Click `Finish`

The project will then appear on eclipse.

Later on, for testing purposes, you can run locally the Hadoop job by executing the desired main class on Eclipse.

3 – Hadoop and MongoDB

Traditional, DBMSs typically used to handle EMSs data, are not conceived to properly cope with the requirements of streaming and processing such massive amounts of meter and sensor date. They must first persist all data on the database, to then evaluate it.

MongoDB and Hadoop are a powerful combination and can be used together to deliver complex analytics and data processing capabilities. MongoDB can be used as a file system, taking advantage of load balancing and data replication features over multiple machines for storing files. Hadoop, a MapReduce implementation, can be used for batch processing of data and aggregation operations. The aggregation framework enables users to obtain the kind of results for which the SQL GROUP BY clause is used.

MapReduce is a framework for processing parallelizable problems across huge datasets, using a large number of computers (nodes), collectively referred to as a cluster (if all nodes are on the same local network and use similar hardware) or a grid (if the nodes are shared across geographically and administratively distributed systems, and use more heterogenous hardware).

MapReduce works in two different tasks. The "Map" step: The master node takes the input, divides it into smaller sub-problems, and distributes them to worker nodes. The worker node processes the smaller problem, and passes the answer back to its master node. The "Reduce" step: The master node then collects the answers to all the sub-problems and combines them in some way to form the output, i.e., the answer to the problem it was originally trying to solve.

Computation processing may apply to data stored either on a filesystem (unstructured) or in a database (structured). MapReduce can take advantage of the locality of data, processing it near the storage assets in order to reduce the distance over which date must be transmitted.

3.1 – MongoDB setup instructions

These instructions will start the MongoDB server and will load the database for this tutorial. Please do not close the terminal or you will close the MongoDB server.

In order to initialize the MongoDB server, proceed as follows:

1. Open a terminal console in your computer
2. Type `mongodb-summer.sh`
3. Press enter

Later to access the database using the terminal type `mongo energy` and you will enter in a Mongo console for a database called `energy`. There you can use `show collections` to view the collections (tables) and `db.{collection name}.find()` to view the collection content. If you want to repeat an exercise we advise you to delete the output collection for that exercise using `db.{collection name}.drop()`.

4 – Your first Map-Reduce query

Now we will explain how to execute the following query over MongoDB using Hadoop.

Query 4.1 – *Which are the EnergyMeterEvents that were consumed in the last 8 hours?*

Hint: In this database, the last 8 hours means a `timestamp` greater then 1388505573 .

4.1 – Implementing a Map

Create a class named `EnergyMapper` that extends a Mapper class. This class will define the map phase for the Hadoop job through the implementation of a method named `Map`. This method receives a `mongoid`, a `BSONObject` and a `Context`. A `mongoid` is a unique identifier at a MongoDB collection for the record. A `BSONObject` contains the record fields and its fields can be accessed using the method `get("field name")`. The `Context` is used to write an object to be passed to a `Reducer` function, for this you should use the `write(key, object)` method. Objects that should be in the same reducer need to have the same key. Moreover, you should pass an object containing valuable information for the `Reducer`.

In this example we filter the events at the Map function. As a map key we use the `mongoid` which is an unique identifier for each record. The content from the map function is an `EneryEvent` class which stores the `id`, `timestamp` and `measure`.

Solution:

```
public void map(final Object pKey, final BSONObject pValue, final Context pContext)
                throws IOException, InterruptedException {

        //Get a record into variables
        final String mongo_id = pKey.toString();
        final String id = (String) pValue.get("deviceId");
        long measure = ((Number) pValue.get("measure")).longValue();
        long timestamp = ((Number) pValue.get("timestamp")).longValue();

        // Filter the records in the last 8 hours
        if(timestamp > 1388505573){
        pContext.write(new Text(mongo_id), new EnergyEvent(id, timestamp, measure));
        }
}
```

4.2 – Implementing a Reducer

Create a class named `EnergyReducer` that extends a Reducer class. This class will define the reduce phase for the Hadoop job through the implementation of a method named `Reduce`. This method receives a key from the mapper, a collection of Objects with the same key and a `Context`. Similar to the Map method, we can write to the Context using the `write(key, object)` method, this time it will write into a MongoDB collection (or table). In order to write to a MongoDB collection you should use the `BasicBSONObject`, that represents a record, inside of a `BSONWritable` that covers the object transfer. After that you only need to specify an unique `key` for the record. You can set fields into a `BasicBSONObject` using the `put("field name",value)` method.

Solution:

```
public void reduce(final Text pKey, final Iterable<EnergyEvent> pValues,
                final Context pContext)
                                throws IOException, InterruptedException {

        //Gets the event
        Iterator<EnergyEvent> it = pValues.iterator();
        EnergyEvent event = it.next();

        // In this example is supposed to only have one event per reducer
        if(it.hasNext())
                System.out.println("Error");

        //Creates a record object to be written in a MongoDB collection
        BasicBSONObject output = new BasicBSONObject();

        // Adds the field deviceId with value event.getId() to the record
        output.put("deviceId", event.getId());
```

```
        output.put("measure", event.getMeasure());
        output.put("timestamp", event.getTimestamp());

        BSONWritable out = new BSONWritable(output);
        pContext.write(pKey, out);
}
```

4.3 – Implementing a configuration (main class)

At this point you have your Map and Reduce classes created, now you just have to create a configuration class that tells Hadoop all the important information to execute the job. The required configuration method is presented in the example bellow:

```
public class EnergyXMLConfig extends MongoTool {
    private static final Log LOG = LogFactory.getLog(EnergyXMLConfig.class);

    public EnergyXMLConfig() {
        this(new Configuration());
    }

    public EnergyXMLConfig(final Configuration conf) {
        MongoConfig config = new MongoConfig(conf);
        setConf(conf);

        config.setInputFormat(MongoInputFormat.class);

        config.setMapper(EnergyMapper.class);
        config.setMapperOutputKey(Text.class);
        config.setMapperOutputValue(EnergyEvent.class);

        config.setReducer(EnergyReducer.class);
        config.setOutputKey(Text.class);
        config.setOutputValue(BSONWritable.class);
        config.setOutputFormat(MongoOutputFormat.class);

        // Input collection location
        config.setInputURI("mongodb://labXcY:27017/energy.EnergyMeterEvent");
        // Output collection location
        config.setOutputURI("mongodb://labXcY:27017/energy.query41");

        config.setSplitSize(4);

    }

    public static void main(final String[] pArgs) throws Exception {
        System.exit(ToolRunner.run(new EnergyXMLConfig(), pArgs));
    }
}
```

For the configuration classes in this tutorial, you will mostly have to change the input and output collection locations. The process of setting these locations is detailed in Section 4.3.1, where you obtain your MongoDB location, and in Section 4.3.2, where you change the location in the configuration file.

For more configuration options you can access the following web page:

`https://github.com/mongodb/mongo-hadoop/blob/master/CONFIG.md`.

4.3.1 – Determining your MongoDB location

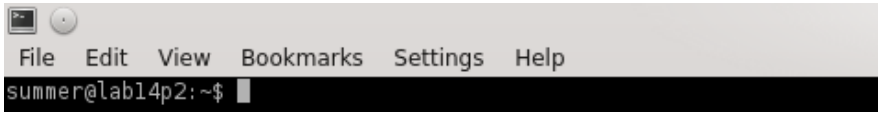

Figure 1: Terminal window example for obtaining the MongoDB location.

In this tutorial you will also need to know your `mongodb location`, to find the location of your MongoDB server, proceed as follows:

1. Open a terminal console in your computer
2. Note on prompt, labXpY
3. Your MongoDB location will then be labX**c**Y:27017

In other words, the letter **p** changes to the letter **c**, and the port number 27017 is added. In the example of Figure 1, the MongoDB location is lab14c2:27017.

4.3.2 – Updating the configuration files

At the configuration class, set `"mongodb://{mongodb location}/energy.{collection name}"` as argument into `config.setInputURI()` and `config.setOutputURI()` to define respectively the input and output MongoDB collection (or table).

For instance in `pt.ulisboa.tecnico.hadoop.energy.query41.EnergyXMLConfig` class presented in the material at computer lab14c2 then the output and input collections are set as:
`config.setInputURI("mongodb://lab14c2:27017/energy.EnergyMeterEvent")` and
`config.setOutputURI("mongodb://lab14c2:27017/energy.query41")`.

4.4 – Important Hadoop notes

Please make sure that you have the same classes on the configuration files as you have on the Map and Reduce classes. In Figure 2 the circles and the arrows indicate which class names should be the same in the project. Inconsistencies among these names are not catch by the compiler and so they make part of the most common errors.

Also note that the `MapOutputKey` and `MapOutputValue` should only be set to classes that implements the `WritableComparable` interface.

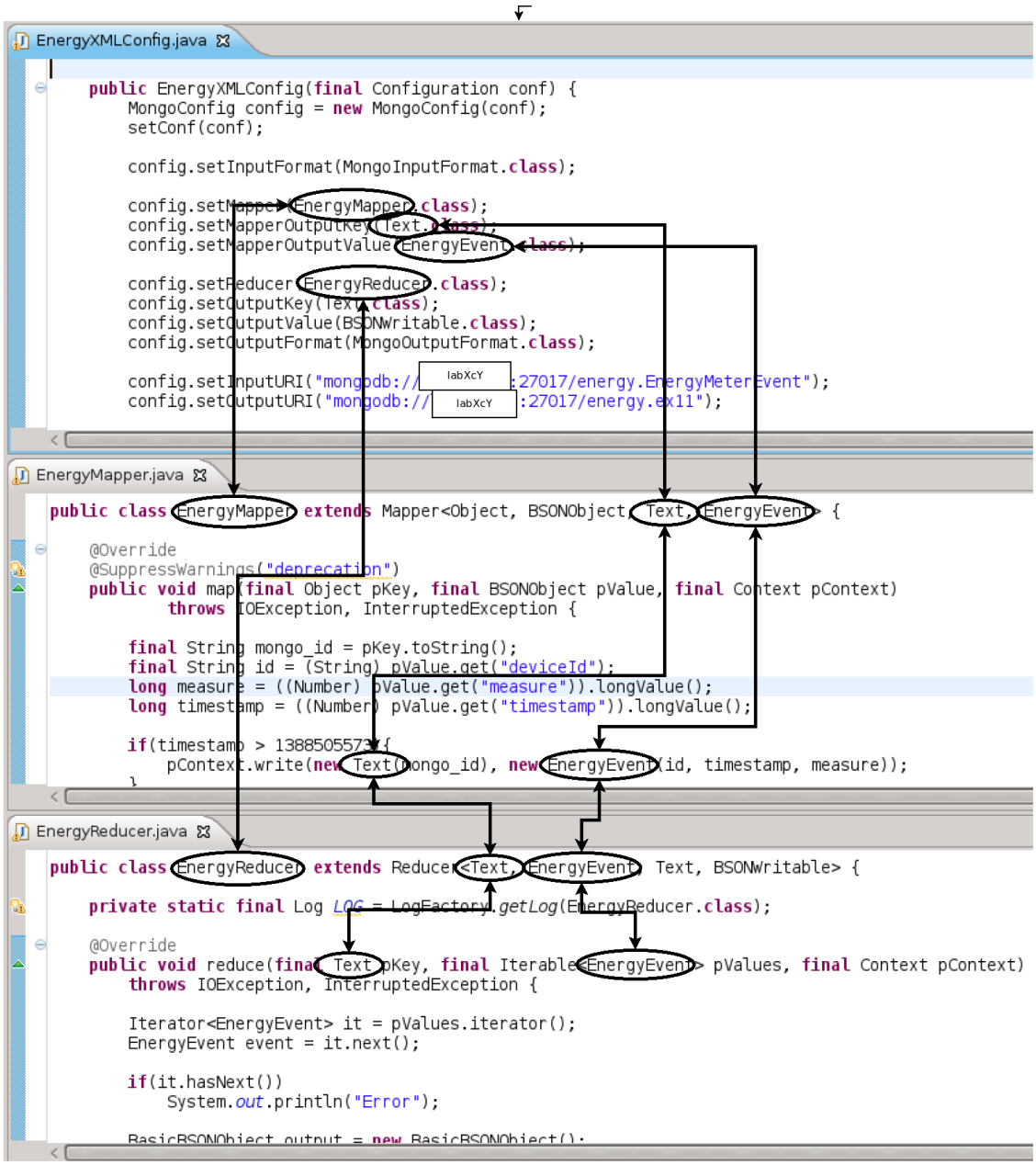

Figure 2: Eclipse environment example.

4.5 – General Program compilation instructions

In order to submit a Hadoop job, you have first to produce a `JAR` file, to produce a `JAR` file, proceed as follows:

1. In the `Project Explorer` window right click on your project name
2. Select `Export -> Java -> JAR file` (note: do not use `Runnable JAR file`).
3. In the opened window, type or browse a location and a file name after `JAR file:`
4. Click on `Finish`.

Now a .jar file should be at the location you selected.

4.6 – Hadoop job submission instructions

To submit a Hadoop job, use the .jar file created in the last section and copy it to your folder source in the cluster file system:

```
$ scp {file name}.jar {your username}@cluster:~/source/
```

After that you should access the cluster machine with the credential provided (i.e., ds3-50) and submit the job as follows:

```
$ ssh {your username}@cluster
{Type your password}
$ cd /source/
$ hadoop jar {file name}.jar {class location and name
i.e., pt.ulisboa.tecnico.query41.EnergyXMLConfig}
```

Now the job should be running and you can visualize the progress either in the terminal or accessing the web page: `http://borg.rnl.ist.utl.pt:8094/cluster`

4.7 – Collection visualization

In order to visualize the output of your Hadoop job we suggest you to use the mongo terminal and get some records from the output collection, proceed as follows:

1. Open a terminal window
2. Type `mongo energy`
3. Press enter
4. Type `db.{collection name}.find()`
5. Press enter

Now you should get some records from the collection in your terminal. Remember for query 4.1, the `{collection name}` is `query41`.

5 – Performing selections on Map phase

In the last section you learn step-by-step how to execute a query in MongoDB using a Hadoop job. Now, please try to complete the next querys by following the same process as in Section 4.

Query 5.1 – *Which are the EnergyMeterEvents that had an energy consumption measure bellow 1000 Wh?.*
Use the output collection from Query 4.1 .

6 – Performing simple aggregations on the Reduce phase

Query 6.1 – *How many EnergyMeasureEvents have been sent, by each sensor?*
Use the output collection from Query 4.1 .

In this example we aggregate all the events in a reducer step. As a map key we use the `deviceId` so we can aggregate all the events per `deviceId`. The output content from the map function is an `EneryEvent` class which stores the `id`, `timestamp` and `measure`. Since the `deviceId` is a key, we do not need to specify one into the `EnergyEvent` class. We then count how many EnergyEvents exist in each reducer.

Map Solution:

```
public void map(final Object pKey, final BSONObject pValue, final Context pContext)
                throws IOException, InterruptedException {

        //Get a record into variables
        final String mongo_id = pKey.toString();
        final String deviceId = (String) pValue.get("deviceId");
        long measure = ((Number) pValue.get("measure")).longValue();
        long timestamp = ((Number) pValue.get("timestamp")).longValue();

        // Use deviceId as key and EnergyEvent as content to Reducers
        pContext.write(new Text(deviceId), new EnergyEvent( "", timestamp, measure));
}
```

Reduce Solution:

```
public void reduce(final Text pKey, final Iterable<EnergyEvent> pValues, final Context pContext)
                throws IOException, InterruptedException {

        // Iterates over the pValues collections and count the number of events
        int count = 0;
        for (final EnergyEvent value : pValues) {
                count++;
        }

        //Create a record with a deviceId and the number of events
        BasicBSONObject output = new BasicBSONObject();
        output.put("deviceId", pKey);
        output.put("count", count);

        BSONWritable out = new BSONWritable(output);
        pContext.write(pKey, out);

}
```

Query 6.2 – *Which is the maximum, minimum, and average energy consumption value of all events?*
Use the output collection from Query 4.1 .

7 – Executing Map/Reduce tasks

Query 7.1 – *Which meters for every range of 5 minutes had an energy consumptions measure bellow 1000 Wh?*
Use the output collection from Query 4.1 .

Hint: 5 minutes means a timestamp of 300 .

In this example, we aggregate all the events in a reducer step. As a map key we use the `deviceId`, and the `timestamp` divided by 300 (meaning the 5 minutes bucket where the event belongs). We aggregate both keys into the `EneryEvent` class. The output content from the map function is a `LongWritable` containing the event `measure`. Then in the reducer we can calculate the total measured by meter for each 5 minutes and filter by just writing the ones bellow 1000000Wh.

Map Solution:

```
public void map(final Object pKey, final BSONObject pValue, final Context pContext)
                throws IOException, InterruptedException {

        //Get a record into variables
        final String mongo_id = pKey.toString();
        final String deviceId = (String) pValue.get("deviceId");
        long measure = ((Number) pValue.get("measure")).longValue();
        long timestamp = ((Number) pValue.get("timestamp")).longValue();

        // Use deviceId and timestamp/ 5 minutes as keys and
        // EnergyEvent as content to Reducers
        pContext.write(new EnergyEvent(deviceId, timestamp / 300, new Long(0)),
                        new LongWritable(measure));
}
```

Reduce Solution:

```
public void reduce(final EnergyEvent pKey, final Iterable<LongWritable> pValues,
                final Context pContext)
                throws IOException, InterruptedException {

        // Iterates over the pValues collections and sums the measures of the events

        long sum = 0;
        for (final LongWritable value : pValues) {
                sum += value.get();
        }

        // Filter Sums bellow 1000000 Wh
        if(sum < 1000000){

                BasicBSONObject output = new BasicBSONObject();
                output.put("deviceId", pKey.getId());
                output.put("sum", sum);
                output.put("5 minutes", pKey.getTimestamp());
                BSONWritable out = new BSONWritable(output);
                pContext.write(new Text( pKey.toString()), out);
        }

}
```

Query 7.2 – *Which meters had reported a total energy consumption bellow 25% of the average of all events?*
Use the output collection from Query 4.1 and average energy consumption value from Query 6.2 .

Query 7.3 – *How many EnergyMeasureEvents have been sent, by each sensor, for every range of 5 minute?*
Use the output collection from Query 4.1 .

Query 7.4 – *Which was the maximum, minimum, and average energy consumption value for every deviceId?*
Use the output collection from Query 4.1 .

Query 7.5 – *Which was the maximum, minimum, and average energy consumption value for each 5 minutes and for every deviceId?*
Use the output collection from Query 4.1 .

8 Big Data

Exercise 8.1 – *Repeat the previous querys using the full dataset.*
Use the original EnergyMeterEvent collection instead of the output collection from Query 4.1. This way you can process a big collection of data and face the Big Data problems.