1 – Energy Data Problem Domain

Energy Management Systems (EMSs) are energy monitoring tools that collect data streams from energy meters and other related sensors. In real-world, buildings are equipped with a large number of those sensors and meter devices, producing huge quantities of data that must be integrated in real-time, to enable Energy Managers to quickly identify and respond to anomalous situations.

2 – Getting Started with ESPER

Data Stream Management Systems (DSMSs) are able to continuously process arriving data without having to persist them, speeding up the data evaluation process. Coarsely speaking data streams flows through queries, producing new data streams. Moreover, DSMSs adapt SQL query language to be more adequate to express the real-time data processing operations. EMSs are applications that by nature take advantage of DSMSs to perform data processing. In this lab guide, we will use Esper [1], a DSMS engine, to illustrate how to create and run *continuous* real-time queries on the energy data domain.

2.1 – Experimental Setup

Our experimental setup architecture, summarized in Figure 1, is as follows. To access Esper engine you will use EsperShell, a command line application that allows you to formulate and submit *continuous* queries to the engine. On the other hand, the engine will be *continuously* fed by a set of data streams made available through a *DeviceAPI*. Those streams of data come from a simulator that, resorting to a database, mimics the behaviour of a *Modbus* energy meter's network.

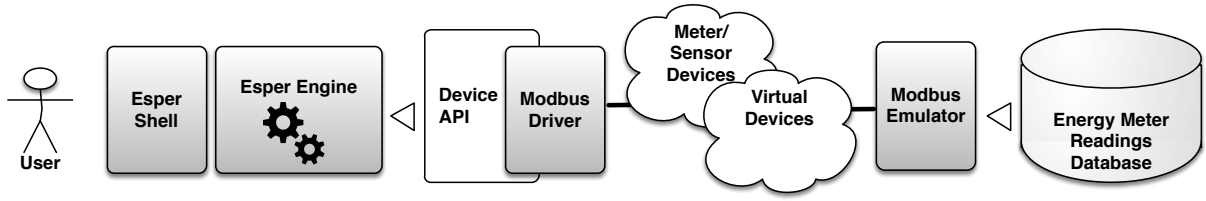

Figure 1: Experimental setup architecture. User resorts on EsperShell to install queries in Esper engine, and a (simulated) network of energy meters fed the engine by pushing data streams into it.

To set up the EsperShell environment, proceed as follows:

1. Login on Windows 7 with the user: `rnl\summmer`, with the password: `ulisboa`
   EsperShell depends on a database previously installed on each computer's lab. Therefore, the usage of those computers to proceed with this tutorial is strictly necessary.

2. Use Google Chrome web browser to download the command line application from:
   `https://ftp.rnl.tecnico.ulisboa.pt/pub/classes/ds3/esperShell.zip`

3. Unzip the `esperShell.zip` archive, to the desktop, using the password: `ulisboa`

4. Open the folder `esperShell`. Run the application by double clicking the file `startLab_x64.cmd`

5. From the five command line windows that will appear in the screen, to proceed with this lab guide, you only have to interact with three of them. We suggest that you organizes windows titled vertically as illustrated in Figure 2. With `'Input datastream monitor'` on the left, the `'Query editor'` in the center, and the `'Query output results monitor'` on the right.

[1] http://esper.codehaus.org

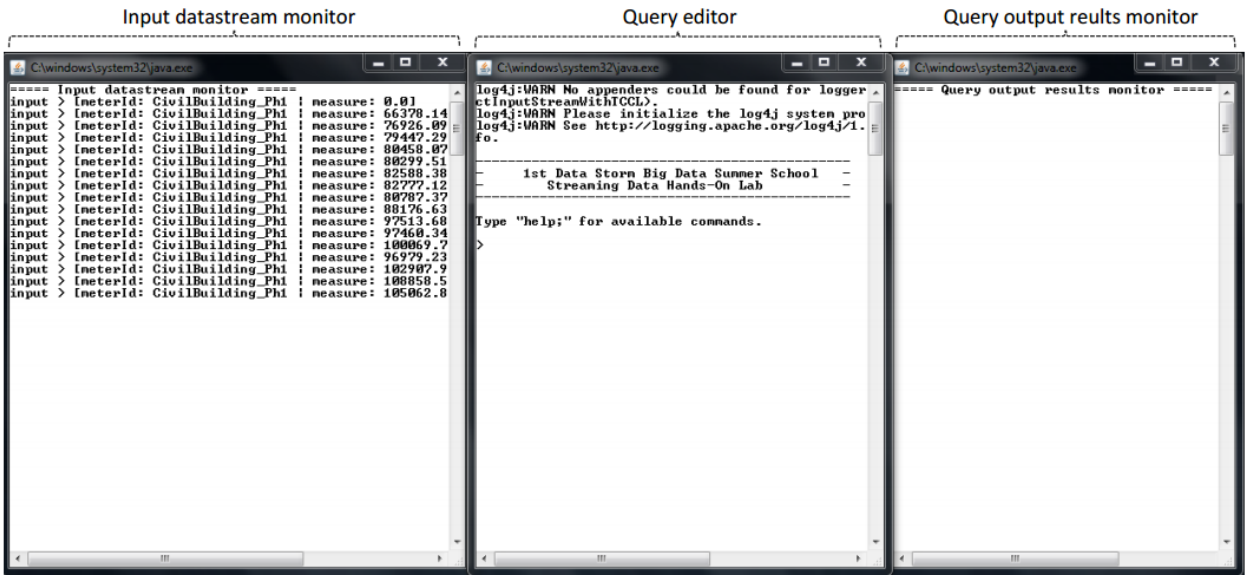

Figure 2: Suggested layout for the Esper lab. With the input data window on the left, the query editor prompt on the center, and the query output window on the right.

6. Those three windows—that compose EsperShell—should be used in the following way:

   (a) `Input datastream monitor` – Displays the data streams that are being sent to Esper engine.

   (b) `Query Editor` – Serves to compose and install new queries in the engine, and also to manage the queries that have already been installed.

   (c) `Query output results monitor` – Displays the results (data streams) produced by the installed queries.

2.2 – Configuring Data Streams

The `modbusDriverConf.json` file (located in the EsperShell folder) specifies the energy meters in which we are interested. This configuration file consists of a set of records, each one specifying a given energy meter and the role at which new measurements are taken from the devices. The follow record:

```
{
    "modbusAddr": "1",
    "meterId": "CivilBuilding_Ph1",
    "PollingTimeMillis": "3000"
}
```

states that a data stream related with the energy consumption of the meter with ID `"CivilBuilding_Ph1"` will be sent, by the modbus address `"1"`, to the query engine with the periodicity of 3 seconds. The parameter `"PollingTimeMillis"` can be used to specify the time period for each data stream event. Replacing `"3000"` by `"---"` specifies that the meter is disabled. For those changes to take effect in the system, you have to reload the file with the command `reload;` in the `'Query Editor'` window.
Before continue this tutorial, try to make some modifications to the configuration file, and reload it, the changes will be evident on the `'Input datastream monitor'`. For example, by default the file is configured to a single input. Try to change the *pooling time* of this stream, or activate two more data streams and configure them to produce events every 4 and 6 seconds, respectively.

3 – Esper Continuous Queries

For the following queries in this tutorial, let *Datastrea.Measure* be the data stream of events e, belonging to the schema of S, that represents the flow of energy consumption measures produced, through time, by a given energy meter.

$$\text{Datastream.Measure} = <e_1, e_2, \ldots, e_n>, S = (\text{meterId, measure}).$$

Where *meterId* uniquely identifies the energy meter by its location, and *measure* states in W·h the consumed energy value.

Output Control

The rest of this tutorial consists on installing queries into Esper engine and evaluating their results. To formulate and submit those queries you should use the command `install <query_statement>;` in the `'Query Editor'` window. To visualize the queries that have been installed so far, the `list;` command can be used. There is a set of another useful commands that can be used during the lab, `help;` command should be typed to find out all of them. However, you don't have to know all of them right now. During the tutorial we will stress the most important ones for each given situation.

Query 1.1 – *Which Datastream.Measure events are being consumed?*

```
SELECT *
FROM Datastream.Measure
```

The above query consumes and returns all the events with schema *Datastream.Measure*. For each consumed event that same event is returned. This is the simplest query that may be written in EPL—Esper's query language. Install this query in the Esper engine, using the command `install`, and see their result in the `'Query output result monitor'`. Please note that all commands should always terminate with `';'`.
After to successfully install this query you will notice, in the `'Query output result monitor'`, that the produced output data stream resembles with following one:

```
Query 1 OUTPUT NEW Events:
| [meterId: CivilBuilding_Ph1 | measure: 86253.81]
Query 1 OUTPUT NEW Events:
| [meterId: CivilBuilding_Ph1 | measure: 82415.97]
Query 1 OUTPUT NEW Events:
| [meterId: CivilBuilding_Ph1 | measure: 85536.26]
                              ...
```

Note that, all installed queries are identified by their unique `queryID`. Being `1` the identification of the previous one. Besides the output monitor, each query result is written in the `<queryID>_output.txt` log file, present in the folder: `esperShell/queriesOutput`. This log may be consulted to visualize all the output data stream produced so far.

The `OUTPUT` keyword may be used to control query's output:

Query 1.2 – *What is the meterId value of all Datastream.Measure events consumed so far? Update the result set every 10 seconds.*

```
SELECT meterId
FROM Datastream.Measure
OURTPUT ALL EVERY 10 SECONDS
```

The `OUTPUT` clause is optional in Esper and is used to control or stabilize the rate at which events are output and to suppress output events. EPL provides several different ways to control output rate. When omitted, the default output behaviour is: `OUTPUT ALL EVERY 1 EVENTS`, meaning that the query's evaluation result should be produced for each consumed event.
To know more about Esper's output control capabilities you can take a look at:
`http://esper.codehaus.org/esper-5.0.0/doc/reference/en-US/html/epl_clauses.html#epl-output-rate`

By this time you should have installed two queries. Use the `list;` command to see which queries are currently running at the engine. With the command `disable 1;` you can tell to Esper engine to stop the evaluation of the query with `1` as their `queryID`. Command `enable 1;` will restart the query evaluation process. Beside those, you also can use `drop 1;` to permanently remove from engine the query identified by `1`, or use `dropall;` to definitely drop all the installed queries so far.

You can try to solve the following query by yourself:

Query 1.3 – *Sample the data stream returning just the First event every 5 received events.*

(you may check the answer at the end of tutorial...)

Windows

For a given stream, `win` operators serve to instruct the engine to only evaluate a finite part of the stream, instead of the entire stream. This part is called a time window, and just the events within it will be evaluated. Note that, the window is only relevant for those queries composed by aggregation operators, those that can only be evaluated over a finite set of events (e.g. `AVG`, `MAX`, `ORDER BY`). For example:

Query 2.1 – *What is the maximum, minimum, and average energy consumption value for the past 8 minutes? Update the result every 5 seconds.*

```
SELECT max(measure), min(measure), avg(measure)
FROM Datastream.Measure.win:time(8 minutes)
OUTPUT ALL EVERY 5 SECONDS
```

Esper provides a rich set of window operators, however the most useful are: `time`, `time_batch`, `length`, and `length_batch`. The difference between those window operators lies on how the boundaries, that specify which events belong to the window, are defined. You can know more about them at:
`http://esper.codehaus.org/esper-5.0.0/doc/reference/en-US/html/processingmodel.html#processingmodel_time_window`
Before installing the above query, you should adjust the configuration file, `modbusDriverConf.json`, in order to have just one data stream being pushed into Esper's engine. It is worth nothing that, in a real scenario, time windows can be configured with larger ranges, such as 8 hours, instead of 8 minutes.

To improve the interaction with the `'Query Editor'`, you can elaborate a script file with commands that you want to execute in the EsperShell. To proceed this way, you have to create a `<scriptfilename>.txt`, in the folder `esperShell/script`. Then you can run the script through the command `run <scriptfilename>.txt` in the `'Query Editor'`. This is particularly useful to install new queries. If you make some mistake on the query elaboration, you can directly fix the bug in the script file and reload it. Avoiding the need to rewrite all the query (as you have to do when writing directly to the `'Query editor'`).

Query 2.2 – *Which is the InformaticsBuildingI average energy consumption value for the last 10 event measures?*

```
SELECT avg(measure)
FROM Datastream.Measure.win:length(10)
WHERE meterId = 'InformaticsBuildingI'
```

Note that, to evaluate this query you have to properly configure the `modbusDriverConf.json` file in order to have a data stream available for the energy meter located at 'InformaticsBuildingI'.

Now try to solve the queries:

Query 2.3 – *For every 10 seconds, how many Datastream.Measure events have been received?*

Query 2.4 – *For every 10 received Datastream.Measure events sort them in descending order by their energy consumption measure.*

To properly evaluate this query you should configure `modbusDriverConf.json` file in order to have 10 energy meters producing measures at the same rate.

Filters and Where clauses

Data streams filters allow filtering events out of a given stream *before* they enter the data window, by specifying under which conditions the event can enter the window.

Query 3.1 – *Which is the average energy consumption difference between the Mathematics and Physics buildings , in the last 10 minutes? Update the result for each received new event, but just after the first 30 seconds.*

```
SELECT avg(Math.measure) - avg(Physics.measure) AS delta
FROM Datastram.Measure(meterId='MathematicsBuilding').win:time(10 minutes) AS Math,
     Datastram.Measure(meterId='PhysicsBuilding').win:time(10 minutes) AS Physics
OUTPUT AFTER 30 SECONDS ALL EVERY 1 EVENTS
```

On the other hand, `where` clause will eliminate result rows at a later processing stage, after events have been processed into data window. You can learn more about this at:
`http://esper.codehaus.org/esper-5.0.0/doc/reference/en-US/html/processingmodel.html#processingmodel_filter`
Again, before installing the previous query fo not forget to configure the `modbusDriverConf.json` properly.

Try to solve this query:

Query 3.2 – *Which meters in the past 5 minutes had an energy consumption measure above 1500 W·h?*

Aggregating and Grouping

Aggregation and Grouping operators will be evaluated under the sub-part of the stream delimited by a given window. If there is no window specification, then the engine will evaluate those results according to *all* the stream seen so far. The `snapshot` keyword is often used with the aggregation to output *all* current aggregated results contained in the window. For example:

Query 4.1 – *How many Datastream.Measure events have been sent, by each meter, in last minute? Update the result set for each received event.*

```
SELECT meterId, count(*)
FROM Datastream.Measure.win:time(1 minute)
GROUP BY meterId
OUTPUT SNAPSHOT EVERY 1 EVENTS
```

Now try it yourself:

Query 4.2 – *Which meters in the last 8 minutes reported a energy consumption 25% above the average of all meters on the same 8 minutes?*

Joins

Two or more data streams can be part of the from-clause and thus both (all) streams determine the resulting events. Again, window operators play an important role on delimiting the parts of each stream that can be joined. Joins require the specification of a windows operator for each stream, otherwise the query will not be accepted for the query engine.

Query 5.1 – *In last minute, which percentage of total energy consumed in the Civil Engineering Building was consumed by HVAC? The HVAC consumption is given by the energy meter* `CivilBuilding_Ph3`.

```
SELECT avg(CivilHVAC.measure)/
       (avg(Civil1.measure)+avg(Civil2.measure)+avg(CivilHVAC.measure))  AS HVAC_ratio,
FROM Datastream.Measure(meterId='CivilBuilding_Ph1').win:time(1 minutes) AS Civil1,
     Datastream.Measure(meterId='CivilBuilding_Ph2').win:time(1 minutes) AS Civil2,
     Datastream.Measure(meterId='CivilBuilding_Ph3').win:time(1 minutes) AS CivilHVAC
```

Try to solve this query:

Query 5.2 – *For the Mathematics Building, which is the variation between the energy that is being consumed now and the average energy consumption of the last 8 minutes?*

4 – Solutions

Query 1.3 – *Sample the data stream returning just the First event every 5 received events.*

```
SELECT *
FROM Datastream.Measure
OURTPUT FIRST EVERY 5 EVENTS
```

Note that, `FIRST` could be replaced by `LAST` in order to return the last event instead of the first one.

Query 2.3 – *For every 10 seconds, how many Datastream.Measure events have been received?*

```
SELECT COUNT(*)
FROM Datastream.Measure.win:time_batch(10 seconds)
```

Query 2.4 – *For every 10 received Datastream.Measure events sort them in descending order by their energy consumption measure.*

```
SELECT meterId, measure
FROM Datastream.Measure.win:length_batch(10)
ORDER BY measure desc
```

Query 3.2 – *Which meters in the past 5 minutes had an energy consumption measure above 1500 W·h?*

```
SELECT DISTINCT meterId
FROM Datastream.Measure.win:time(5 minutes)
WHERE measure > 1500
OUTPUT SNAPSHOT EVERY 1 EVENTS
```

`SNAPSHOT` keyword outputs the current state of a windows based query.

Query 4.2 – *Which meters in the last 8 minutes reported a energy consumption 25% above the average of all meters on the same 8 minutes?*

```
SELECT meterId
FROM Datastream.Measure.win:time(8 minutes)
GROUP BY meterId
HAVING avg(measure)*1.25 > (SELECT avg(measure)
                            FROM Datastream.Measure.win:time(8 minutes))
```

Query 5.2 – *For the Mathematics Building, which is the variation between the energy that is being consumed now and the average energy consumption of the last 8 minutes?*

```
SELECT (avg(Now.measure)/avg(Past.measure)) - 1 AS Variance
FROM Datastream.Measure(meterId='MathematicsBuilding').win:time(10 seconds) AS Now,
     Datastream.Measure(meterId='MathematicsBuilding').win:time(8 minutes) AS Past
```