**Testing Note**

We have started testing on the 22nd of April because we had the deadline of D2 deliverable document on the 19th. To do this test, we had to integrate 4 components and designed the testbed.

We determined the number of test sensors (100, 500, 1000, 5000 and 10000) by the number of users multiplied by 4 capabilities (25 users \*4, 125 users \*4, 250 users \* 4, 1250 users \* 4 and 2500 users \*4).

**GROUND PREPARATION FOR TESTING:**

Before testing, we had to do the following preparation.

* Developed small program to register 10000 sensors and insert additional triples for each user to OpenIoT
* Developed a script to make up relevant sensor values for each sensor. Previous values were just timestamps, now using more appropriate values
* Mocked up the component to send user ID (mock up OpenMeetings component ) to send user ID to CQELS query component sequentially. This creates different threads for each user and send user ID to CQELS query component
* Researched how to send users concurrently
* Determined the time measurement points (t1,t2,t3…..m1,m2,m3...) inside our code to take the memory footprint and system time.
* Ran pre-test to process log file to extract result

Analysed log and figured out the result pattern, then created a bash script to extract particular data.

**TESTING PLAN AND CURRENT RESULTS:**

**Performance:**

**Query Generation Test**

***Test Plan***

Firstly we observed **the average of each CQELS query generation and registration time** with increasing number of queries (100, 500, 1000, 5000 and 10000)

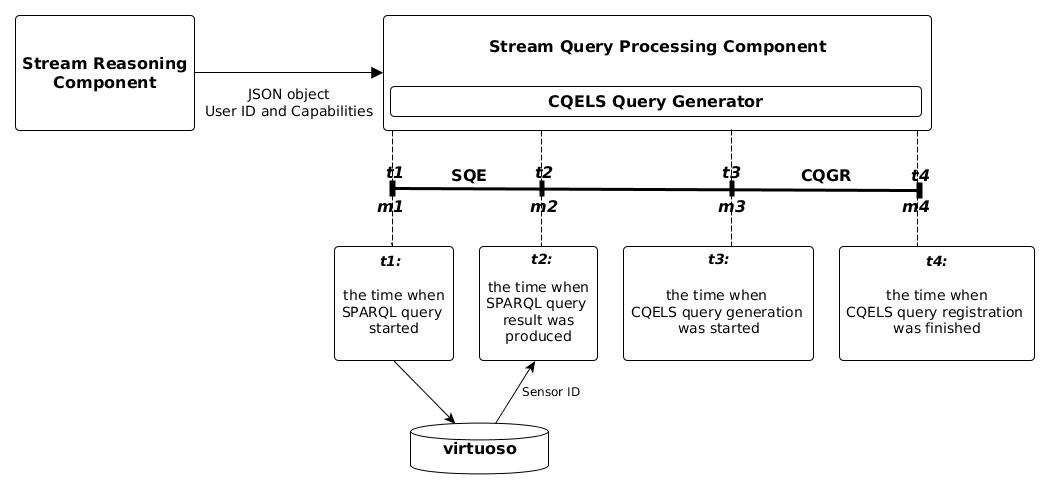
The user IDs and capabilities were sent to the CQELS query component by the program which we created .

The SPARQL query was executed in the CQELS query generation and registration in order to retrieve the sensor ID based on the user ID which was sent from the Stream Reasoning Component, hence the SPARQL query execution time was included in the measurement points to examine the average time for the CQELS query generation and registration. Figure 1 depicts four measurement points where:

* *t1* is the time when SPARQL query started to retrieve sensor ID.
* *t2* is the time when SPARQL query result was produced, hence *t2* - *t1* is the *SPARQL Query Execution Time (SQE).*
* *t3 is the time when* *CQELS query generation was started.*
* *t4* is the time when CQELS query registration to CQELS engine was finished, hence *t4* – *t3* is the *CQELS Query Generation and Registration Time* (CQGR*). (t4 – t3)* is the total time of the *SPARQL Query Execution Time* and *CQELS Query Generation and Registration Time.*

We also measured **the average memory consumption of each CQELS query generation and registration** at the following measurement points as shown in Figure 1

* *m1* is the memory use when *SPARQL* query generation was started.
* *m2* is the memory use when *SPARQL* query was finished, hence *m2 – m1* is the total memory used for the SPARQL query generation.
* *m3* is the memory use when *CQELS* query generation was started.
* *m4* is the memory use when CQELS query registration to CQELS engine was finished, hence *m4 – m3* is the total memory used for the CQELS query generation and registration.

Figure 1 Different Points for CQELS Query Generation and Registration Time Measurements 

***Test Result:***

During query generation performance test, we faced with a limitation of the number of threads which can be created by the program which we created for sending users. This program creates different threads for each user . Always we got out of memory error after sending 1030 users.

We increased Java Heap size and changed the some server setting but it didn't work, so we changed the program not using threads to send users.

We can send more users but the CQELS query component has a limitation of the number of query generations. After sending around 1060 users (1st test was 1063 users, 2nd test was 1068 users and 3rd test was 1061 users) , the CQELS query component got out of memory errors. Note that in this test, we run the Reasoning component. This also consumes threads. In case we do not run Reasoning component, the number of query generations may be increased.

**Currently the limitation of the number of query generations are 4200 queries (1060 users \* 4 capabilities).**

Figure 2 shows the result of the query generation performance test. We tested 100, 500 and 1000 query generations and executed 5 times for each number of queries. The average time of each query generation is calculated as follows:

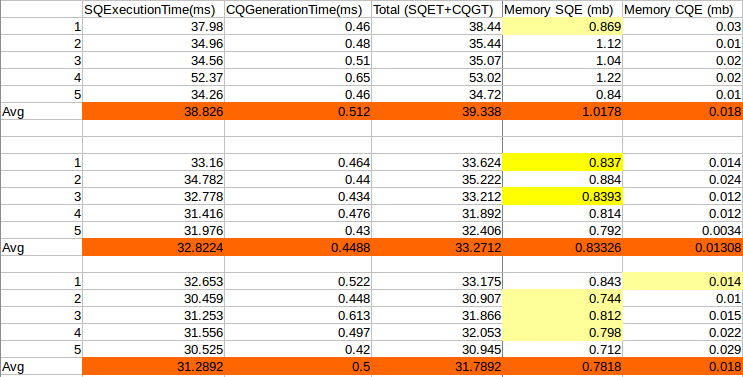


Figure 2: Average CQELS Query Execution Time with Varying Number of Sensors

**Query Execution Test**

***Test Plan***

## After generating the CQELS queries, we observed the CQELS query execution time required by the stream query processing component of our infrastructure with varying size of sensors. The number of test sensors (100, 500,1000, 5000 and 10000) were determined by the number of users multiplied by 4 capabilities (25\*4, 125\*4, 250\*4, 1250\*4 and 2500\*4).

We identified two observation points to examine the average time for the query execution. Figure 1 depicts the two measurement points where:

* *t1* is the time when sensor data stream arrived at Streamer component.
* *t2* is the time when the CQELS query was executed and the result was produced, hence *t2* -*t1* is the *CQELS Query Execution Time* (*T*).

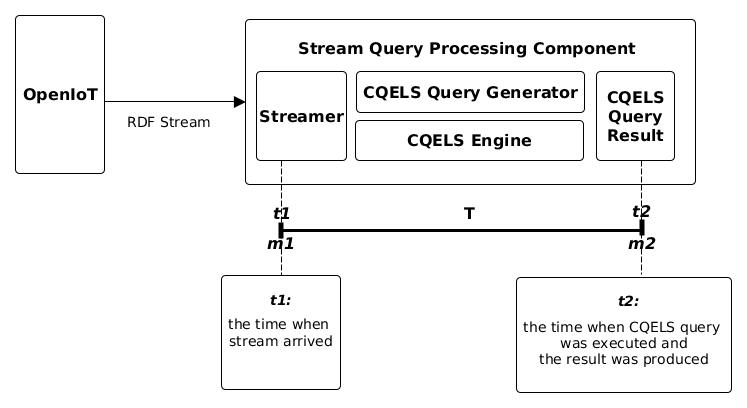


Figure 2: Different Points for CQELS Query Execution Time Measurements

We also measured average memory consumption for the CQELS query execution at the same measurement points as shown in Figure 1.

* *m1* is the memory use when sensor stream arrived at Streamer component
* *m2* is the memory use when CQELS query result was produced, hence *m2 -m1* is the memory used for the CQELS query execution.

## ***Test Result:***

Figure 3 shows the result of the query execution performance test. We tested 100, 500 and 1000 query executions and executed 5 times for each number of queries. The average time of each query execution is calculated as follows:

## *Selection_013.png*

*Figure 3 Average CQELS Query Execution Time with Varying Number of Sensors*

**Scalability:**

We researched how to send users concurrently and tried.

* + - Method 1: use multiple threads

However, users are not actually sending concurrently...

* + - Method 2: use JMeter

Studied how JMeter can send concurrent users via websocket (In the previous test, sent data via HTTP)

Studied about the “test plan” in JMeter which can control the loop of testing.

However, still don't know how to control time between 2 tests in JMeter (to run the whole test autom test automatically, not manually one by one)

Got a problem of “Out of memory” when sending 500 users concurrently (in Thu’s laptop)

**We still don't know how to send users concurrently, therefore the scalability test hasn't started yet.**

NOTE:

* We have restarted Jboss after each (Query generation + execution) test.
* Each user creates different web-socket session in query component. That's why query-generation-module can generate query concurrently for each JSON object.

Th