# On the need to include functional testing in RDF stream engine benchmarks

Daniele Dell'Aglio, Marco Balduini, and Emanuele Della Valle

Dipartimento di Elettronica, Informazione e Bioingegneria – Politecnico of Milano, P.za L. Da Vinci, 32. I-20133 Milano - Italy
daniele.dellaglio@mail.polimi.it, balduini@elet.polimi.it, emanuele.dellavalle@polimi.it

**Abstract.** How to improve existing RDF stream engine benchmarks with functional tests that verify result correctness before stressing the compared systems.

## 1 Introduction

Since 2008 different research groups have studied the problem of query answering over streams of RDF data [1]. Several RDF stream engines have been designed and developed, such as C-SPARQL [2], SPARQL$_{stream}$ [3] and CQELS [4]. The following step, started two years ago, has been the comparison and the evaluation of these systems. To the best of our knowledge, this effort produced two benchmarks: SRBench [5] and LSBench [6].

One common problem of those two benchmarks is that they do not consider the output of the RDF stream engines. SRBench performs only functional tests in order to determine the features supported by the RDF stream engines; LSBench does not verify the correctness of the answers: it limits the analysis to the number of outputs, but this is not enough to determine if the answer is correct. In other words, both benchmarks make two assumptions: 1) the tested systems work correctly, and 2) the tested systems have the same operational semantics. If any of the two assumptions does not hold, the benchmarks supply misleading information on the performance of the compared RDF stream engines.

In this paper, we report our experience about the analysis of three RDF stream engines: C-SPARQL, CQELS and SPARQL$_{stream}$[1]. We show that the two assumptions do not always hold, and consequently the results of the benchmark are not enough to make a comparison of the systems. We also supply guidelines for designing an improved benchmark for RDF stream engines: we propose the addition of a set of functional tests to verify the correctness of the results, in order to obtain more accurate benchmarks. In particular, we provide the following contributions:

– considerations on benchmark principles: in order to the define a benchmark for RDF stream engines it is necessary to deeply understand the operational semantics of these systems (see bold-faced text in Section 3);

[1] We used the last version available at March 14 on the public Web sites. For more information visit `http://www.streamreasoning.org/bersys2013/`

– propose guidelines to define functional tests to verify that the RDF stream engines behave correctly (see bold-faced text in Section 4);
– a discussion on the results proposed by SRBench and LSBench.

In Section 2, we briefly introduce the RDF stream engines and the existent benchmarks for evaluating their performance. Then, in Section 3, we focus on the models at the basis of the RDF stream engines, explaining why their operational semantics should be well-defined, while in 4, we discuss the need to verify that the system implements correctly the operational semantics defined by its model. Finally, Section 5 concludes with considerations about the tests and guide lines for the improvements of the RDF stream benchmarks.

## 2 Related works

In the following, we provide the background information: we first briefly describe the behaviour of a generic RDF stream engines, then we provide information about the two benchmarks for these systems.

### 2.1 RDF stream engines

In its general definition, a RDF stream engine is a system able to continuously answer a query over a RDF stream. A RDF stream is an infinite sequence of timestamped RDF triples. A timestamped RDF triple is a RDF triple annotated with a time instant (also known as *application timestamp*); we represent a timestamped RDF triple as $<s\ p\ o>$:$[\tau]$, where $<s\ p\ o>$ is a well-defined RDF triple and $\tau$ is a natural number indicating the application time.

RDF stream engines are a particular case of stream engines; in these systems the input is a generic stream, composed by a sequence of timestamped tuples [7]. To execute a query over infinite sequences of elements, stream engines usually follow a three-steps approach: 1) they transform the stream in a relation through a *stream-to-relation* (S2R) operator; 2) they execute a *relation-to-relation* transformation through a R2R operator (e.g., a query) and 3) they produce a output stream through a *relation-to-stream* (R2S) operator.

One of the most used families of stream-to-relation operators is *sliding windows*: they allow to extract a continue portion of the stream. For the sake of brevity, we report only the definition of time-based sliding window. A time-based sliding window has an associate time range $[t_o, t_c)$ (known as *scope*) and it contains all the tuples of the stream having application time stamp in the scope. The scope is regularly updated to consider different blocks of the stream. The scope and its update are defined by two parameters, size and slide: the size ($\omega$) defines the width of the window ($\omega = t_c - t_o$); the slide ($\beta$) defines the update rate and it is the time distance between two consecutive blocks of the window. A particular case of time-based sliding window is the one where the size equals to the slide. The name of this window is *time-based tumbling window*: it partitions the stream, ensuring that each timestamped tuple of the stream is in one and only one window block.

As relation-to-relation operators, those systems usually use well known transformations, such as algebras: RDF stream engines usually consider SPARQL and its algebra. It is worth to note that the relations are time-dependent relations and they changes over time (when the scope is updated).

Finally, relation-to-stream operators are used when the output of the query engine should be a new stream. [7] defines three operators: Rstream (it streams the whole output relation), Istream (it streams only new tuple added to the relation) and Dstream (it streams only the tuple that are deleted from the relation).

All the three systems we consider in this work implements time-based sliding windows and they all support (at least partially) SPARQL 1.1. As relation-to-stream operators, C-SPARQL implements Rstream, CQELS implements Istream, while SPARQL$_{stream}$ implements all the three operators.

### 2.2 Benchmarks for RDF stream engines

SRBench [5] proposes a suite of test queries and defines metrics to evaluate the performance of the system. This benchmark contains 17 queries to gather the properties of the RDF stream engines. The queries vary to ensure that several features of the target system are tested: queries involving single or multiple input streams, queries over stream-only data sources or over mixed stream and static data source, etc. In [5] the authors applied the benchmark on the existent RDF stream engines, and explained the differences in term of supported functionalities. Time and memory performance tests, and scalability tests are not targeted in the actual version of SRBench.

LSBench [6] proposes three tests to evaluate the RDF stream engines. The first one is a functional test to verify the operators and the functionalities supported by the engines: it is a test similar to the one proposed by SRBench. The second test is a correctness test: its goal is to verify if the tested RDF stream engine produces the correct output. Actually this analyses only the number of produced answers, assuming that the contents of the output are correct. Finally, the third test is a maximum input throughput test: it has the goal evaluate the maximum throughput of the RDF stream engines. This test is done increasing the rate of data in the stream and verifying the number of the answers. For each test a set of 12 queries is provided; similarly to SRBench, the queries vary to take into account different features of the engines (single and multiple streams, presence of static data, etc).

## 3 Compare RDF stream processors knowing their operational semantics

The first point, we want to focus on is related to the task of determining the correct answer of a RDF stream engine. Given an input data streams, an input query and the operational semantics of the engine, it should be possible to determine which is the expected answer of a RDF stream engine. The specifications of the RDF stream engines are usually available in the scientific articles and in the documentation available on their Web sites.

Let's consider for example SPARQL engines: for SPARQL query answering engines, a test suite[2] defined by W3C is available to verify the correctness of an

[2] Cf. `http://www.w3.org/2009/sparql/docs/tests/`

implementation through a set of SPARQL queries. Each query is associated to its expected result. In that case, the SPARQL algebra is enough to explain how to determine which should be the output of a SPARQL engine implementation given an input and a query.

In the case of RDF stream engines, this process is more complex: the inputs (the stream and the query) and the SPARQL algebra are not enough to determine one correct answer. We set up the following experiment: we consider an input stream $\mathbb{S}_1$ and an input query $Q_1$ defining a tumbling window $W$ with size and slide of ten seconds. We registered the query in different (consecutive) time instants obtaining different outputs (details of the experiment can be found in Appendix A.1). It is worth noting that all the different results are correct.

This behaviour can be explained through an extended model of stream engine, SECRET [8]. SECRET is a framework to support the task of integrating stream processes. The authors defined a model to explain the different behaviours of the S2R operators of the stream engines. In particular, the authors propose four parameters to define the behaviour of the window operator: the time range of the active window (the *scope*, as defined above); the subset of the stream included of the active window (the *content*); the conditions until which the input can be added to the active window (the *tick*); and finally the conditions until which the window content can be processed by the query engine (the *report*). One important concept define by SECRET is $t_0$. It represents the application time instant on which the first window starts.

It becomes possible to determine which should be the correct answer given the inputs by extending operational semantics of a stream engine with the additional window parameters defined by SECRET. **RDF stream benchmark designers should understand the differences in operational semantics of the systems they want to target. In particular, different behaviours of the S2R operators affect the outputs and the performance of those systems, and the explanation of the benchmarks results without considering this fact can be misleading.**

## 4 Adherence of RDF stream engines to their operational semantics

In the previous sections, we explained that in order to benchmark RDF stream engines, their operational semantics should be complete and comparable. The last point we consider is the adherence of the RDF stream engines to their operational semantics: even if the specification is fine, if the implementation is wrong, the results of the benchmarking would be misleading. To verify it we exploit the correctness of the output. It may appear obvious, but, let us restate that given a RDF stream engine, an input query and a dataset, the result is correct if it conforms to the one defined by the system specification, better if in the form of a formal semantics.

The specifications of the RDF stream engines describe not only the operational semantics of the systems, but also the requirements at the basis of their design, the features they should implement and the use case scenarios with suites of functional and non-functional tests. It can happen that the test suite is badly defined – it is too strictly related to the use-case to elicitate all the requirements. Thus, the implemented RDF stream engine does not work correctly in general cases.

When designing tests to check that the RDF stream engines correctly implements the operational semantics, it is important to take into account both the whole RDF stream engine answering process and the single sub-processes that compose it. In fact, the query answering process would work in the right way if the three transformations that composes it are correct.

Before talking about the tests on the S2R operator (the main focus of this work), we explain how tests for the R2R and the R2S operators could be build. Regarding the R2R operator, an initial work is done by the two existent benchmarks. In their analysis these work check which are the limit of these operators, verifying which constructs are supported. The next step should be the verification of the correctness of the result of the R2R operators; the goal can be achieved exploiting the existent work on SPARQL query answering engines. **The input data should be placed in one window (so the size has to be properly defined) and the output must be the same of the one defined by the SPARQL tests.**

The correct implementation of the R2S operator can be verified with the following test: let's consider as input stream a generic stream $\mathbb{S}$. The input query $Q$ uses as S2R operator a time-based sliding window $W$ with size $\omega > 1$ and slide $\beta = 1$; $Q$ uses as the R2R operator the identity transformation. It is easy to verify that Rstream works correctly if the system outputs the current content of $W$; Istream works correctly if the system outputs the new triples that enter $W$; finally Dstream works correctly if the system outputs the triples that exit the $W$.

**The tests on S2R operators aim to verify if the windows work properly. Tests should verify that the window contains the correct elements and that its update is performed in the right way.** To check the content of the window, if the tested RDF stream engine supports the Rstream S2R operator, it is easy to verify the behaviour of the window: it is enough to use a identity transformation as R2R operator (i.e. copy content of the input relation in the output relation) and the outputs would be the content of the window. If the RDF stream engine does not support Rstream, the test described above is not enough. If the system has the Istream operator and the test query uses the identity transformation, the output would allow to check only if the tuples are correctly inserted in the window, but not if they are correctly removed (a similar consideration is possible for systems supporting Dstream operator only). This problem can be fixed preparing the test with a different SPARQL query, exploiting the aggregate functions or the join operator. In Appendix A.2 we present an example of test of this kind.

The results of these tests on the systems we considered allowed us to verify that C-SPARQL and SPARQL$_{stream}$ correctly implements the window, while CQELS does not remove the elements in the window. In other words, CQELS ignores the window size and processes the whole data contained in $\mathbb{S}$. It is not in the scope of this paper to report on the potential improvement in terms of input throughput of such a CQELS shortcoming; we intend to further inspect this behaviour checking memory allocation in the long run. A performing RDF

Stream engine should not only maximize input throughput, but also minimize and keep as stable as possible memory allocation.

## 5 Conclusion and future directions

In this work, we analyzed the existing RDF stream engines and we showed that the absence of exhaustive functional testing hides anomalous behaviours of those systems. It is worth to note that both SRBench and LSBench do not detect S2R related problems, because they assume that the answer returned by the system is correct. Additionally, we illustrated how the operational semantics of actual RDF stream engines are not able to define a unique correct answer given an input stream and a query.

Both the benchmarks for RDF stream engines take into account different dimensions in the definition of the experiments, such as: single and multiple input streams; single and multiple windows over the input stream; optional presence of static knowledge. However, they should take into account the variety of possible systems outputs, in order to relate the performance to the correctness of the results, as it happens in stream engine benchmarks such as the Linear Road Benchmark [9]. It is important to verify the correctness of the system in all those cases, in order to create an exhaustive set of functional tests.

In our future works, we intend to extend one of the RDF stream benchmark with correctness tests. This would allow the community to improve the existent systems and, potentially, to test new systems against a exhaustive set of tests.

## References

1. Della Valle, E., Ceri, S., Van Harmelen, F., Fensel, D.: It's a Streaming World! Reasoning upon Rapidly Changing Information (2009)
2. Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: C-sparql: a continuous query language for rdf data streams. Int. J. Semantic Computing **4**(1) (2010) 3–25
3. Calbimonte, J.P., Jeung, H., Corcho, O., Aberer, K.: Enabling Query Technologies for the Semantic Sensor Web. International Journal on Semantic Web and Information Systems **8**(1) (2012) 43–63
4. Le-Phuoc, D., Dao-Tran, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: International Semantic Web Conference (ISWC 2011). Volume 1380., Bonn, Germany, Springer (2011) 370–388
5. Zhang, Y., Pham, M.D., Corcho, O., Calbimonte, J.P.: SRBench: A Streaming RDF/SPARQL Benchmark. In: International Semantic Web Conference (ISWC 2012), Boston, USA (2012) 641–657
6. Le-Phuoc, D., Dao-Tran, M., Pham, M.: Linked Stream Data Processing Engines: Facts and Figures. In: International Semantic Web Conference (ISWC 2012). Volume 1380., Boston, USA, Springer (2012) 300–312
7. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language : semantic foundations. The VLDB Journal **15**(2) (2006) 121–142
8. Botan, I., Derakhshan, R., Dindar, N., Haas, L.M., Miller, R.J., Tatbul, N.: Secret: A model for analysis of the execution semantics of stream processing systems. PVLDB **3**(1) (2010) 232–243
9. Arasu, A., Cherniack, M., Galvez, E.: Linear road: A stream data management benchmark. In: International Conference on Very Large Data Bases (VLDB 2004), Toronto, Canada, Morgan Kaufmann Publishers Inc. (2004) 480–491

## A Experiments

### A.1 Experiment 1 - Multiple valid results

As scenario we use a simplified version of the scenario considered by CQELS authors in [4]: there are two connected rooms, $r_1$ and $r_2$; each room has a sensor able to detect the individuals inside, $m_1$ and $m_2$. The stream contains triples in the form:

$$<\!:\mathtt{m_i}\ \mathtt{ex{:}detectedAt}\ :\!\mathtt{r_j}\!> : [\tau]$$

indicating that the individual $m_i$ has been detected in the room $r_j$.

Let's set up the stream $\mathbb{S}_1$, depicted in Figure 1. $\mathbb{S}_1$ contains four triples describing the detections of the two individuals $m_1$ and $m_2$ in $r_1$ first (respectively triples $S_1$ and $S_2$) and in $r_2$ then (respectively triples $S_3$ and $S_4$).

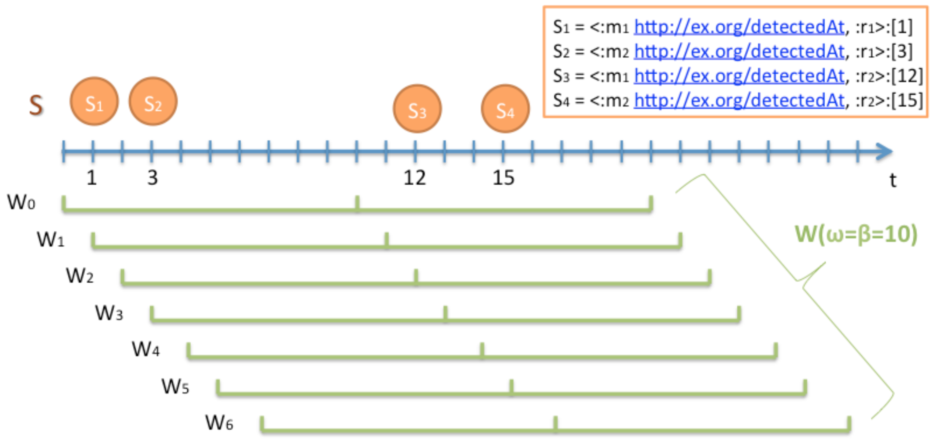
S1 = <:m1 http://ex.org/detectedAt, :r1>:[1]
S2 = <:m2 http://ex.org/detectedAt, :r1>:[3]
S3 = <:m1 http://ex.org/detectedAt, :r2>:[12]
S4 = <:m2 http://ex.org/detectedAt, :r2>:[15]
S
S1
S2
S3
S4
1
3
12
15
t
W0
W1
W2
W3
W4
W5
W6
W(ω=β=10)

**Fig. 1.** Input stream $\mathbb{S}_1$ used in Experiment 1

We want to know when the two individuals $m_1$ and $m_2$ are in the same room. As window, we consider a time-based tumbling window of 10 seconds, the query returns the room where both $m_1$ and $m_2$ are detected, or empty-set otherwise. The query, expressed in the C-SPARQL syntax, is available in Listing 1.1.

```
SELECT ?room
FROM STREAM <http://myexample.org/stream1> [RANGE 10s STEP 10s]
WHERE {
  <http://ex.org/m1> <http://ex.org/detectedAt> ?room .
  <http://ex.org/m2> <http://ex.org/detectedAt> ?room
}
```

**Listing 1.1.** C-SPARQL query used in Experiment 1

Let's now try to determine the correct answer the system should return. Looking at the stream $\mathbb{S}_1$, the expected answer is to have `http://ex.org/r1` first, and `http://ex.org/r2` then. But it is worth to note that it is not the **only** correct answer: all the results supplied in Table 1 are correct. The different results are consequence of the window operator: if it starts at $t = 0$ or $t = 1$ (windows $W_0$ and $W_1$ in Figure 1) the answer is the one presented above. If the window starts at $t > 1$ the output produced would be different: if $t = 2$ (window $W_2$) the first answer of the window would be empty, because its content at 12[3] is $S_2$ only; the second returned answer is `http://ex.org/r2`, because at 22 $W_2$ contains both $S_3$ and $S_4$.

**Table 1.** Results of Experiment 1

| **Window** | **C-SPARQL** | | | | **CQELS** | | | | **SPARQL$_{stream}$** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **First result** | | **Second result** | | **First result** | | **Second result** | | **First result** | | **Second result** | |
| | **ts** | **?room** | **ts** | **?room** | **ts** | **?room** | **ts** | **?room** | **ts** | **?room** | **ts** | **?room** |
| $W_0$ | 10 | `ex:r1` | 20 | `ex:r2` | 3 | `ex:r1` | 15 | `ex:r2` | 10 | `ex:r1` | 20 | `ex:r2` |
| $W_1$ | 11 | `ex:r1` | 21 | `ex:r2` | 3 | `ex:r1` | 15 | `ex:r2` | 11 | `ex:r1` | 21 | `ex:r2` |
| $W_2$ | 12 | – | 22 | `ex:r2` | no result | | 15 | `ex:r2` | 12 | – | 22 | `ex:r2` |
| $W_3$ | 13 | – | 23 | – | no result | | 15 | `ex:r2` | 13 | – | 23 | – |
| $W_4$ | 14 | – | 24 | – | no result | | 15 | `ex:r2` | 14 | – | 24 | – |
| $W_5$ | 15 | – | 25 | – | 15 | `ex:r2` | no result | | 15 | – | 25 | – |
| $W_6$ | 16 | `ex:r2` | no result | | 15 | `ex:r2` | no result | | 16 | `ex:r2` | 26 | – |

The results are reported in Table 1. The outputs of the SPARQL$_{stream}$ and C-SPARQL engines are similar: the only difference is the second output of $W_6$. It happens because C-SPARQL answers when the content of the window is not empty; looking at Figure 1 it easy to observe that the second block of $W_6$ is empty.

A first observation on the results of CQELS is that when the system produces a result, it is always at $\tau = 3$ and $\tau = 15$. It happens because CQELS does not wait that the window closes (i.e., the size of the active window equals to $\omega$). It is a possible behaviour of the S2R operator and we think it is correct. The second consideration is related to the fact that CQELS always output `ex:r2` in each experiment. We are not able to explain if this behaviour is correct or not, because we are not able to control the $t_0$ of the window as in the other systems[4]. We tried to probe, but on the one hand its source code is not available on the Web site; and on the other hand some bugs of CQELS (see Appendix A.2) do not allow us to investigate through additional experiments.

Finally, it is worth to note that in this experiment we considered a tumbling window, but it is easy to observe that there are multiple correct results every time the query defines a time-based sliding window with slide $\beta > 1$. In general, no one of the operational semantics of the RDF stream engines we considered can determine a unique correct answer given a stream and a query.

[3] The scope of $W_2$ at 12 is [2,12).

[4] In C-SPARQL and SPARQL$_{stream}$, the window $W$ is created when the first query with $W$ in its `FROM STEAM` clause is submitted to the system (i.e., $t_0$ equals to the query registration time). We are not able to determine if it is true also for CQELS.

## A.2 Experiment 2 - Adherence tests for window

As scenario we use the same of the first experiment: there are two connected rooms, $r_1$ and $r_2$; , $r_1$ and $r_2$; each room has a sensor able to detect the individuals ($m_1$, $m_2$, $m_3$ and $m_4$) when they are inside. The stream contains triples in the form:

$$<\texttt{:m}_\texttt{i}\ \texttt{ex:detectedAt}\ \texttt{:r}_\texttt{j}> : [\tau]$$

indicating that the individual $m_i$ has been detected in the room $r_j$.

We set up the stream $\mathbb{S}_2$ depicted in Figure 2: it contains four triples $S_k$ ($k \in [1, 4]$). Every 5 seconds a triple is sent (so, $S_1$ has time stamp 0, $S_2$ has time stamp 5 and so on). We want to know in which room there are two (different) individuals that enter within 3 seconds. To do it, we defines a time-based tumbling window $W$ with size and slide $\omega = \beta = 3$ seconds. For the sake of brevity, we report here only the query in CQELS syntax (Listing 1.2); queries for the other two systems and the code to repeat the experiment are available at: `http://streamreasoning.org/benchmarks/bersys2013`.

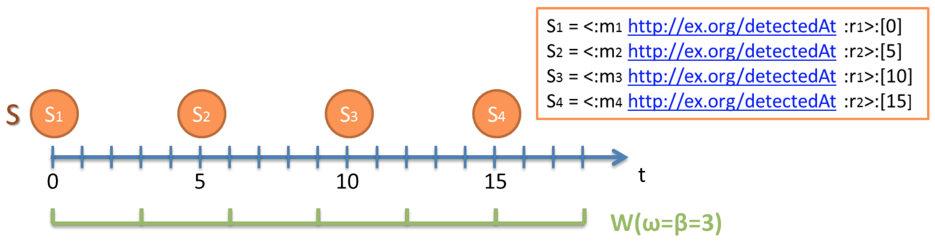
S
S1
S2
S3
S4
0
5
10
15
t
W(ω=β=3)
S1 = <:m1 http://ex.org/detectedAt :r1>:[0]
S2 = <:m2 http://ex.org/detectedAt :r2>:[5]
S3 = <:m3 http://ex.org/detectedAt :r1>:[10]
S4 = <:m4 http://ex.org/detectedAt :r2>:[15]

**Fig. 2.** Input stream $\mathbb{S}_2$ used in Experiment 2

Looking at the picture it is easy to observe that the query should return only empty answers or no answers if the engine only reports at content change: timestamped triples in $\mathbb{S}_2$ with the same room have a time difference of 10 seconds (respectively $S_1$ and $S_3$ for $r_1$, $S_2$ and $S_4$ for $r_2$). Additionally the minimum time-distance between two triples is of 5 seconds, which is greater than the window size $\omega$: the window never contains two triples, so the `WHERE` clause of the query is never satisfied.

```
SELECT ?room
WHERE {
  STREAM <http://ex.org/streams/test> [RANGE 3s SLIDE 3s] {
    ?p1 <http://ex.org/detectedAt> ?room .
    ?p2 <http://ex.org/detectedAt> ?room
  }
  FILTER (?p1 != ?p2)
}
```

**Listing 1.2.** CQELS query used in Experiment 2

We performed this test on the three RDF stream engines we are considering, and while C-SPARQL and SPARQL$_{stream}$ behaved in the correct way, CQELS returned wrong answers in $W_4$ and $W_5$, as shown in Table 2.

**Table 2.** Results of Experiment 2 – with FILTER clause

| Block | C-SPARQL | | CQELS | | SPARQL$_{stream}$ | |
|---|---|---|---|---|---|---|
| | **ts** | **?room** | **ts** | **?room** | **ts** | **?room** |
| 1 | 3 | - | no result | | 3 | - |
| 2 | 6 | - | no result | | 6 | - |
| 3 | no result | | no result | | 9 | - |
| 4 | 12 | - | 10 | `ex:r1` | 12 | - |
| 5 | no result | | 15 | `ex:r2` | 15 | - |
| 6 | 18 | - | no result | | 18 | - |

To investigate the problem we modify the query in Listing 1.2: it projects also the `?p1` and `?p2` variables and we remove the filter clause. The modified query is represented in Listing 1.3.

```
SELECT ?p1 ?p2 ?room
WHERE {
  STREAM <http://ex.org/streams/test> [RANGE 3s SLIDE 3s] {
    ?p1 <http://ex.org/detectedAt> ?room .
    ?p2 <http://ex.org/detectedAt> ?room
  }
}
```

**Listing 1.3.** CQELS query used in Experiment 2 – without FILTER clause

The new result is reported in Table 3. C-SPARQL and SPARQL$_{stream}$ returns the empty-result or mappings where `?p1` and `?p2` bind the same individual or empty result (in other words the join are performed on the same RDF triple). CQELS returns also performs the join over different RDF triples. We traced this CQELS behaviour to a possible incorrect removal ofthe triples from the window.

**Table 3.** Results of Experiment 2 – without FILTER clause

| Block | C-SPARQL | | | | CQELS | | | | SPARQL$_{stream}$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **ts** | **?p1** | **?p2** | **?room** | **ts** | **?p1** | **?p2** | **?room** | **ts** | **?p1** | **?p2** | **?room** |
| 1 | 3 | `ex:m1` | `ex:m1` | `ex:r1` | 0 | `ex:m1` | `ex:m1` | `ex:r1` | 3 | `ex:m1` | `ex:m1` | `ex:r1` |
| 2 | 6 | `ex:m2` | `ex:m2` | `ex:r2` | 5 | `ex:m2` | `ex:m2` | `ex:r2` | 6 | `ex:m2` | `ex:m2` | `ex:r2` |
| 3 | no result | | | | no result | | | | 9 | - | - | - |
| 4 | 12 | `ex:m3` | `ex:m3` | `ex:r1` | 10 | `ex:m1` `ex:m3` | `ex:m3` `ex:m3` | `ex:r1` `ex:r1` | 12 | `ex:m3` | `ex:m3` | `ex:r1` |
| 5 | no result | | | | no result | | | | 15 | - | - | - |
| 6 | 18 | `ex:m4` | `ex:m4` | `ex:r2` | 15 | `ex:m2` `ex:m4` | `ex:m4` `ex:m4` | `ex:r2` `ex:r2` | 18 | `ex:m4` | `ex:m4` | `ex:r2` |