Linked Stream Data Processing Engines: Facts and Figures*

Danh Le-Phuoc[1], Minh Dao-Tran[2], Minh-Duc Pham[3],
Peter Boncz[3], Thomas Eiter[2], and Michael Fink[2]

[1] Digital Enterprise Research Institute, National University of Ireland, Galway
danh.lephuoc@deri.org
[2] Institut für Informationssysteme, Technische Universität Wien
{dao,eiter,fink}@kr.tuwien.ac.at
[3] Centrum Wiskunde & Informatica, Amsterdam
{p.minh.duc,p.boncz}@cwi.nl

Abstract. Linked Stream Data, i.e., the RDF data model extended for representing stream data generated from sensors social network applications, is gaining popularity. This has motivated considerable work on developing corresponding data models associated with processing engines. However, current implemented engines have not been thoroughly evaluated to assess their capabilities. For reasonable systematic evaluations, in this work we propose a novel, customizable evaluation framework and a corresponding methodology for realistic data generation, system testing, and result analysis. Based on this evaluation environment, extensive experiments have been conducted in order to compare the state-of-the-art LSD engines wrt. qualitative and quantitative properties, taking into account the underlying principles of stream processing. Consequently, we provide a detailed analysis of the experimental outcomes that reveal useful findings for improving current and future engines.

1 Introduction

Linked Stream Data [18] (LSD), that is, the RDF data model extended for representing stream data generated from sensors and social network applications, is gaining popularity with systems such as Semantic System S [8], Semantic Sensor Web [19] and BOTTARI [10]. Several platforms have been proposed as processing engines for LSD, including Streaming SPARQL [7], C-SPARQL [5], EP-SPARQL [2] on top of ETALIS, SPARQL$_{stream}$ [9], and CQELS [14]. By extending SPARQL to allow continuous queries over LSD and Linked Data, these engines bring a whole new range of interesting applications that integrate stream data with the Linked Data Cloud.

All above platforms except Streaming SPARQL provide publicly accessible implementations. As processing LSD has gained considerable interest, it is desirable to have a comparative view of those implementations by evaluating them on common criteria in

* This research has been supported by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-II), by Marie Curie action IRSES under Grant No. 24761 (Net2), by the Austrian Science Fund (FWF) project P20841, and by the European Commission under contract number FP720117287661 (GAMBAS) and FP72007257943 (LOD2).

an *open benchmarking framework*. Such a framework is also valuable in positioning new engines against existing ones, and in serving as an evaluation environment for application developers to choose appropriate engines by judging them on criteria of interest.

Unfortunately, no benchmarking systems for LSD processing exist. Close to one is Linear Road Benchmark [4], which however is designed for relational stream processing systems and thus not suitable to evaluate graph-based queries on LSD processing engines. Furthermore, [14] provides a simplistic comparison of CQELS, CSPARQL, and ETALIS, which is based on a fixed dataset with a simple data schema, simple query patterns, and just considers average query execution time as the single aspect to measure the performance. With further experience and studies on theoretical/technical foundations of LSD processing engines [15], we observed that the following evaluation-related characteristics of these engines are critically important.

– The difference in semantics has to be respected, as the engines introduce their own languages based on SPARQL and similar features from Continuous Query Language [3];
– The execution mechanisms are also different. CSPARQL uses *periodical execution*, i.e., the system is scheduled to execute periodically (time-driven) independent of the arrival of data and its incoming rate. On the other hand, CQELS and ETALIS follow the *eager execution* strategy, i.e., the execution is triggered as soon as data is fed to the system (data-driven). Based on opposite philosophies, the two strategies have a large impact on the difference of output results.
– For a single engine, any change in the running environment and experiment parameters can lead to different outputs for a single test.

All these characteristics make a meaningful comparison of stream engines a nontrivial task. To address this problem, we propose methods and a framework to facilitate such meaningful comparisons of LSD processing engines wrt. various aspects. Our major contribution is a framework coming with several customizable tools for simulating realistic data, running engines, and analyzing the output. Exploiting this framework, we carry out an extensive set of experiments on existing engines and report the findings.

The outline of the paper is as follows. Section 2 describes our evaluation framework, including the data generator for customizable data from realistic scenarios. In Section 3 we present experimentation methodology and results, including design and strategies (Section 3.1), as well as several testing aspects and outcomes ranging from functionality, correctness, to performance (Sections 3.2–3.4). Section 4 reports general findings and provides a discussion, while Section 5 considers related benchmarking systems. Finally, Section 6 concludes the paper with an outlook to future work.

2 Evaluation Framework

Social networks (SNs) provide rich resources of interesting stream data, such as sequences of social discussions, photo uploading, etc. Viewed as highly-connected graphs of users, SNs is an ideal evaluation scenario to create interesting test cases on graph-based data streams. Therefore, our evaluation environment provides a data generator to realistically simulate such data of SNs. Next, the graph-based stream data schema and the data generator are described.

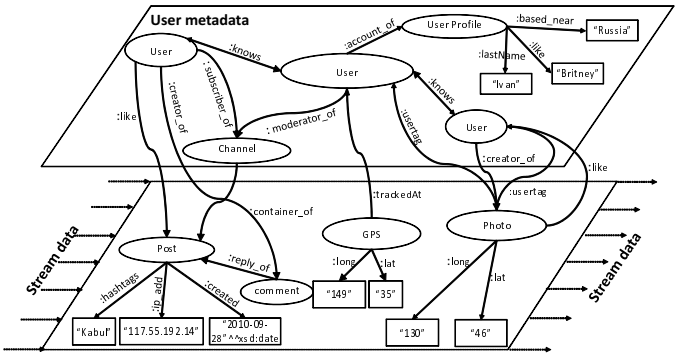

Fig. 1: Logical schema of the stream data in a social network.

2.1 Graph-based Stream Data Schema

The data schema is illustrated by a data snapshot in Figure 1. This snapshot has two layers for *stream data* and *static data* corresponding to what users continuously generate from their social network activities and user metadata including user profiles, social network relationships, etc. The details of these two layers are described below.

Stream data. that is updated or arrives frequently, is shown in the bottom layer. It contains various sources of streaming data:

– *GPS stream* (S_{gps}): inspired by the use case in Live Social Semantics [1], we assume that each user has a GPS tracking device to send updated information about her current location to the SN frequently. This information contains latitude/longitude of the user's position, e.g., the location of an event where user is attending, and the sending time.

– *Posts and comments stream* (S_{pc}): there is a huge stream of posts and comments in the SN as users start or join discussions. Similar to the availability of the "wall" for each Facebook user or the "Tweet timeline" for each Twitter, every user in our generated SN has her own forum for writing posts. People who subscribe to this forum (or "follow" the forum moderator as in Twitter) can read and reply to the posts and comments created in the forum. Each forum is used as a channel for the posting stream of a user. In this stream, we are particularly interested in the stream of "likes" (i.e., people who show their interest in a post), denoted by S_{pclike}, the stream of tags (i.e., set of words representing the content of the discussion), denoted by S_{tags}.

– *Photos stream* (S_{fo}): Uploaded photos and their associated attributes provide interesting information for discovering user habits, friend relationships, etc. In this stream, we focus on exploiting useful information from the stream of user tagged in a photo, S_{fotags}, and the stream of likes per photo, denoted by S_{folike}.

Static data. U_{data}, that is not frequently changed or updated, is shown in the upper layer. It contains user profile information (name, date of birth, relationship status, etc.), the relationships between users and the channel where they write posts, comments.

2.2 Data Generator

To the best of our knowledge, there exists no stream data generator that can realistically simulate the stream data in SNs. We propose a novel data generator for LSD,

called *Stream Social network data Generator* (S2Gen). It generates data according to the schema in Section 2.1 in consideration of the continuous query semantics [3, 7, 5, 9, 2, 14] and various realistic data distributions such as the skewed distributions of posts/comments. As window operators are primitive operators in a continuous query, the correlations of simulated data have to affect on data windows over streams. To meet this requirement, S2Gen uses the "window sliding" approach from the structure-correlated social graph generator S3G2 [16]. As such, to generate the stream data, it slides a window of users along all users in the social graph and creates social activities for each user (writing a post/comment, uploading photos, sending GPS tracking information). For creating a particular stream data, e.g., S_{pc}, S2Gen extracts all the posts/comments created for all the users, then sorts them according to their timestamps, and finally serializes these data to a file. A stream player is created in order to push the stream data from this file into a streaming engine. Similarly, S_{fo}, S_{pclike}, S_{folike}, and S_{gps} are created.

For static data, S2Gen generates the user profiles and the friendship information of all the users in order to form the static data, i.e., U_{data}. The details of this step and how to simulate the data correlations in the static data are the same as in S3G2. Note that all the generated stream data is correlated with non-stream data, e.g., user tags in the photo stream are correlated with friendship information. Various realistic situations are also simulated while generating stream data, e.g., for GPS stream data, around a specific time, the latitude and longtitude sent by those people attending the same event are close to each other and that of the event's location.

For flexibility, S2Gen offers a range of parameters. Some main parameters used in following experiments are:

– *Generating period*: the period in which the social activities are generated, e.g., 10 days, one month, etc. By varying this parameter, one can create streams with different sizes for testing scalability.
– *Maximum number of posts/comments/photos for each user per week*: each of these parameters can be adjusted in order to change the amount of data that arrives in a window of time. It thus can increase/decrease the input rate (e.g., number of triples/seconds) as the stream player pushes the data according to a window of time. Besides, it also varies the total amount of generated streaming data for a fixed generating period.
– *Correlation probabilities*: there are various parameters for the data correlations between graph data and the graph structure, e.g., the probability that users will be connected if they are living in the same area. They can be customized to specify how data is skewed according to each data attribute. The tested systems need to recognize these correlation properties in order to optimize their query plan.

3 Experimentation

3.1 Evaluation Design and Strategies

The setup to evaluate an engine E with a stream query Q is as follows. Suppose that Q requires as input a non-empty set of finite streams $\mathcal{S}_Q = \{S_1, \ldots, S_m\}$, $(m \geq 1)$, and possibly static data. Let $R = \{r_1, \ldots, r_m\}$ be a set of rates (elements/sec) such that each S_i is fed into E at rate r_i. We expect as the output a sequence of elements $O(E, Q, R) = o_1, \ldots, o_n$, abbreviated by O_E when Q and R are clear from the context.

Table 1: Queries classification

	Patterns covered							S	N_P	N_S	Engines		
	F	J	A	E	N	U	T				CQ	CS	JT
Q_1	√								1	1	√	√	√
Q_2		√						√	2	1	√	√	√
Q_3		√						√	3	1	√	√	√
Q_4	√	√							4	1	√	√	√
Q_5		√						√	3	2	√	√	∅
Q_6		√						√	4	2	√	√	∅
Q_7	√	√						√	7	2	√	Ⓢ	∅
Q_8		√			√				3	2	×	Ⓢ	∅
Q_9	√	√				√		√	8	4	√	E	∅
Q_{10}			√						1	1	√	√	√
Q_{11}		√	√	√					2	2	×	√	×
Q_{12}			√				√		1	1	×	√	×

F: filter **J**: join **E**: nested query **N**: negation **T**: top k **U**: union **A**: aggregation **S**: uses static data
N_P: number of patterns, N_S: number of streams, Ⓢ: syntax error, E: error, ∅: return no answer, ×: not supported
CQ: CQELS, CS: C-SPARQL, JT: JTALIS

Our evaluation design is general enough to capture different stream engines. For examples, E can be CQELS, CSPARQL, or EP-SPARQL[4] for testing LSD engines, where the static data is a set of RDF graphs, each S_i is a Link Data stream, i.e., a sequence of RDF triples, and the output is a sequence of triples or SPARQL results.

Note that EP-SPARQL is just a wrapper of the Prolog-based stream engine ETALIS. When testing EP-SPARQL, we observed that it suffered considerably heavy loads from parsing big RDF graphs. Moreover, it does not support multi-threading to easily control the rates of input streams. Recently, JTALIS has been developed as a Java wrapper for ETALIS. It does not exhibit the above parsing problems, as it works with facts and Prolog rules. Furthermore, using Java makes it very convenient to control input rates via multi-threading. Hence we decided to evaluate JTALIS. Here the static data is a set of ground facts, each S_i as well as the output is a sequence of ground facts,[5] and queries can be formalized as sets of Prolog rules. We thus compare CQELS, CSPARQL, and JTALIS with the following plan:[6]

– *Functionality tests*: we introduce a set of stream queries with increasing complexities and check which can be successfully processed by each engine;
– *Correctness tests*: for the executable queries, we run the engines under the same conditions to see whether they agree on the outputs. In case of big disagreement, we introduce notions of *mismatch* to measure output comparability. When the engines are incomparable, we thoroughly explain the reasons to our best understanding of them.
– *Performance tests*: for queries and settings where comparability applies, we execute the engines under their maximum input rates and see how well they can handle the limit. Performance is also tested wrt. different aspects scalability, namely the volume of static data size and the number of simultaneous queries.

3.2 Functionality Tests

We use the SN scenario generated by the tool in Section 2.2. Here, the static data is U_{data}, and the incoming streams are S_{gps}, S_{pc}, S_{pclike}, S_{fo}, S_{folike} (cf. Section 2.1). The data generator considers many parameters to produce plausible input data, but for our

[4] SPARQL$_{stream}$ implementation has not supported native RDF data (confirmed by the main developer)
[5] We normalize the outputs to compare sets of facts and SPARQL result sets.
[6] All experiments are reproducible and recorded, details are available at `http://code.google.com/p/lsbench/`.

experimental purpose, we are interested in the size (number of triples/facts) of the static data and input streams, i.e., $|U_{data}|$, $|S_{pc}|$, etc.

The engines are run on a Debian squeeze amd64 2x E5606 Intel Quad-Core Xeon 2.13GHz with 16GB RAM, running OpenJDK Runtime Environment (IcedTea6 1.8.10) OpenJDK 64-Bit Server VM, and SWI-Prolog 5.10.1.

To evaluate the engines by query expressiveness, we propose 12 queries which cover different desired features and aspects of stream queries. Table 1 reports the query patterns in detail and our observation of which queries can be successfully executed by which engines. It shows that a number of desired features are not yet satisfactorily covered. CQELS supports neither negation nor nested aggregation. CSPARQL reports syntax errors on Q_7(complicated numeric filter), Q_8(negation), and encounters runtime error on Q_9(most complicated query). Regarding JTALIS, patterns in Q_5-Q_9 are theoretically supported, but the run produces no output. Also, there is no support for explicit representations of aggregations that works with timing windows; with kind help from the JTALIS team, we encoded the simple counting aggregation in Q_{10} by recursive rules, but left out the more complicated aggregations in Q_{11}, Q_{12}.

3.3 Correctness Tests

Based on the supported functionalities from all engines, we first evaluate them on Q_1–Q_4 and Q_{10} with a static data of 1000 user profiles, and a stream of posts during one month: $|U_{data}| = 219825$ and $|S_{pc}| = 102955$, at two rates of 100 and 1000 input elements per second,[7] which are considered slow and fast, respectively. Then, we check the agreement on the results between the engines by simply comparing whether they return the same number of output elements. Here we adopt the *soundness* assumption, i.e., every output produced by the engines are correct. The results are reported in the "Output size" columns of Table 2. It turned out that:

(i) CSPARQL disagrees with the rest wrt. the total number of output triples. It returns duplicates for simple queries while for complicated ones, it misses certain outputs;
(ii) CQELS and JTALIS agree on the total number of output triples on most of the case, except for Q_4.

Explaining these mismatches, regarding (i), CSPARQL follows the periodical execution strategy, i.e., the query processor is triggered periodically no matter how fast or slow the inputs are streamed into the system. When the execution is faster than the update rate, the engine will re-operate on the same input before the next update, hence outputs replicated results between two updates. In contrast, when the execution is slower than the update rate, certain inputs are ignored between two consecutive executions. Thus, for complicated queries, one expects that CSPARQL misses certain outputs.

CQELS and JTALIS, on the other hand, follow the eager execution strategy, and trigger the computation incrementally as soon as new input arrives; or, in case data arrives during computation, they queue the input and consume it once the engine finishes with the previous input. Therefore, eager ones do not produce overlapping outputs.

[7] Each element is a triple (resp., fact) for CQELS, CSPARQL (resp. JTALIS).

For the differences between CQELS and JTALIS that lead to observation (ii), the execution speed and windows play an important role. For simple queries Q_1–Q_3,[8] the two engines perform on more or less the same speed, hence the results are identical. For the more complex query Q_4, the inputs contained in the time-based windows determine the outputs of each execution. The slower the execution rate, the less input in the windows, as more of the input is already expired when the new input is processed. Consequently, output for Q_4 produced by JTALIS (which is slower) is smaller than that produced by CQELS. To circumvent this problem, one can use triple-based windows. Unfortunately, this type of windows is not yet supported by JTALIS.

The total number of outputs is not ideal to cross-compare the engines. We next propose a more fine-grained criterion by tolerating duplication and checking for mismatch.

Comparing by mismatch. We now propose a function to compute the mismatch $mm(E_1, E_2, Q, R)$ between two output sequences $O_{E_1} = a_1, \ldots, a_{n_1}$ and $O_{E_2} = b_1, \ldots, b_{n_2}$ produced by engines E_1, E_2 running query Q at rates R, i.e., we would like to see how much of the output from E_1 is not produced by E_2.

An *output-partition* of O_{E_1} is defined as a sequence of blocks $A_1, \ldots, A_m$ where each A_i is a subsequence $a_{i1}, \ldots, a_{i\ell_i}$ of O_{E_1} such that the sequence $a_{11}, \ldots, a_{1\ell_1}, a_{21}, \ldots, a_{2\ell_2}, \ldots, a_{m1}, \ldots, a_{m\ell_m}$ is exactly O_{E_1}. There are multiple output-partitions of O_{E_1}, but we are only interested in two special ones:

– the *periodical output-partition*: each A_i is corresponding to the result of an execution of E_1, when E_1 is a periodical engine;
– the *eager output-partition*: $A_i = a_i$, i.e., a sequence of a single output element.

A *slice* of O_{E_2} from j, $1 \leq j \leq n$ is the sequence $O_{E_2}\lfloor j \rfloor = b_j, \ldots, b_{n_2}$. A block A_i is *covered* by a slice $O_{E_2}\lfloor j \rfloor$ iff for every $a_{ik} \in A_i$, there exists $b_t \in O_{E_2}\lfloor j \rfloor$ such that $a_{ik} = b_t$. In case of non-coverage, the *maximal remainder* of A_i wrt. $O_{E_2}\lfloor j \rfloor$ is defined by $P_i^j = A_i \Cap O_{E_2}\lfloor j \rfloor = a_{ir_1}, \ldots, a_{ir_{k_i}}$ such that for $1 \leq s \leq k_i$, $a_{ir_s} \in A_i$ and there exists $b_t \in O_{E_2}\lfloor j \rfloor$ such that $a_{ir_s} = b_t$. Intuitively, P_i^j is constructed by keeping elements in A_i that also appear in $O_{E_2}\lfloor j \rfloor$; in other words, P_i^j is the maximal sub-block of A_i which is covered by $O_{E_2}\lfloor j \rfloor$.

If P_i^j is an empty sequence, we define $match(P_i^j) = j$. Otherwise, for each element $a_{ir_s} \in P_i^j$, let $match(a_{ir_s})$ be the smallest index t, where $j \leq t \leq n_2$, such that $a_{ir_s} = b_t$, and let $match(P_i^j) = \min\{match(a_{ir_s}) \mid a_{ir_s} \in P_i^j\}$.

We now can define the *maximal remainder sequence* of O_{E_1} that is covered by O_{E_2} as $T_1, \ldots, T_m$, where $T_1 = P_1^1 = A_1 \Cap O_{E_2}\lfloor 1 \rfloor$ and $T_i = P_i^{match(T_{i-1})} = A_i \Cap O_{E_2}\lfloor match(T_{i-1}) \rfloor$ for $1 < i \leq m$. Intuitively, we progressively compute the maximal remainder of each block, starting with the slice $O_{E_2}\lfloor 1 \rfloor$ from the beginning of O_{E_2}. When finishing with one block, we move on to the next one and shift the slice to the minimal match of the last block.

The mismatch is $mm(E_1, E_2, Q, R) = \dfrac{\Sigma_{i=1}^{m}(|A_i| - |T_i|)}{\Sigma_{i=1}^{m}|A_i|} \times 100\%$

Table 2 reports the mismatches between the engines on Q_1-Q_4 and Q_{10}. For a column labeled with E_1—E_2 where $E_1 \neq E_2 \in \{CQ, CS, JT\}$, the left sub-column presents $mm(E_1, E_2, Q, R)$ and the right one shows $mm(E_2, E_1, Q, R)$, respectively.

[8] Reason for identical total number of output tuples for Q_{10} will be made clear in explaining the mismatch.

Table 2: Output Mismatch, $|U_{data}| = 219825$, $|S_{pc}| = 102955$

	Rate: 100 (input elements/sec)									Rate: 1000 (input elements/sec)								
	Output size			Mismatch (%)						Output size			Mismatch (%)					
Q	CQ	CS	JT	CQ—CS		CQ—JT		CS—JT		CQ	CS	JT	CQ—CS		CQ—JT		CS—JT	
1	68	604	68	1.47	0.00	0.00	0.00	0.00	1.47	68	662	68	1.47	0.00	0.00	0.00	0.00	1.47
2	68	124	68	1.47	0.00	0.00	0.00	0.00	1.47	68	123	68	1.47	0.00	0.00	0.00	0.00	1.47
3	533	1065	533	0.00	0.00	0.00	0.00	0.00	0.00	533	1065	533	0.00	0.00	0.00	0.00	0.00	0.00
4	11948	125910	1442	1.69	1.10	87.93	0.00	78.91	0.07	11945	127026	4462	1.54	1.12	62.65	0.00	52.79	0.02
10	28021	205986	28021	14.96	0.04	87.66	0.00	44.67	0.00	28021	209916	28021	14.70	0.04	86.30	0.00	43.25	0.00

Table 3: (Comparable) Maximum Execution Throughput

	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_{10}
CQELS	24122	8462	9828	1304	7459	3491	2326
C-SPARQL	10	1.68	1.63	10	1.72	1.71	10
JTALIS	3790	3857	1062	99	—	—	87

For simple queries Q_1-Q_3, the mismatches are very small, meaning that CSPARQL computes many duplicates but almost of all its output is covered by CQELS and JTALIS.

When the query complexity increases in Q_4 and Q_{10}, CSPARQL misses more answers of CQELS as $mm(CQ, CS, Q_4, 100) = 1.69\%$ and $mm(CQ, CS, Q_{10}, 100) = 14.96\%$. On the other hand, JTALIS produces far less output than the other two for Q_4, due to the reasons in explanation (ii) above. The big mismatches here (from 52.79% to 87.93%) result from the different execution speeds of JTALIS and the other engines.

Interestingly, CQELS and JTALIS output the same number of tuples for Q_{10}, but the contents are very different: $mm(CQ, JT, Q_{10}, 100) = 87.66\%$ and $mm(CQ, JT, Q_{10}, 1000) = 86.30\%$. This is because Q_{10} is a simple aggregation, which gives one answer for every input; hence we have the same number of output tuples between CQELS and JTALIS, which follow the same execution strategy (eager). However, again the difference in execution speed causes the mismatch in the output contents. For all queries that JTALIS can run, $mm(JT, CQ, Q, R) = 0$ where $Q \in \{Q_1, \ldots, Q_4, Q_{10}\}$ and $R \in \{100, 1000\}$, meaning that the output of JTALIS is covered by the one of CQELS.

In concluding this section, we formalize the notion of *comparability by mismatch* as follows. Given a set of engines $\mathcal{E}$, a query Q, and rates R, the engines in $\mathcal{E}$ are said to be comparable with a prespecified mismatch tolerance ϵ, denoted by $comp(\mathcal{E}, Q, R, \epsilon)$, iff for every $E_1, E_2 \in \mathcal{E}$, it holds that $mm(E_1, E_2, Q, R) \leq \epsilon$.

3.4 Performance Tests

This section defines execution throughput, and then reports on this measure when the engines run on a basic setting as well as when different input aspects scale.

Execution Throughput. Besides comparability, one also would like to see how fast the engines are in general. We therefore conduct experiments to compare the engines wrt. performance. The most intuitive measure to show the performance of a stream processing engine is *"throughput,"* which is normally defined as the average number of input elements that a system can process in a unit of time. However, as mentioned above, systems like C-SPARQL using the periodical execution mechanism can skip data

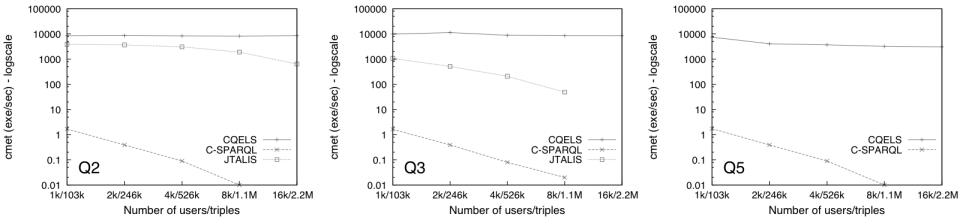

Fig. 2: Comparable max. execution throughput for varying size of static data.

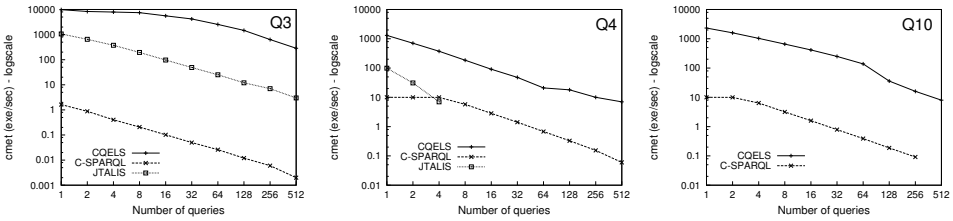

Fig. 3: Comparable max. execution throughput running multiple query instances.

coming in at high stream rate. Therefore, a maximum streaming rate is not appropriate to measure "throughput." Moreover, as shown in previous sections, there are several reasons that make the output from such engines incomparable. We thus propose *"comparable maximum execution throughput"* as a measure for the performance test.

According to this measure, we first need to make sure that the engines produce *comparable* outputs at some (slow) rate, e.g., $comp(\{CQELS, CSPARQL, JTALIS\}, Q_2, 100, 0.147)$ holds. When this is settled, we modify all periodical engines such that the input and execution rates are synchronized. Interestingly, confirmed by our tests on all executable queries on C-SPARQL, there are only marginal changes in execution rates when varying input rates. In this particular evaluation, thanks to the APIs from CSPARQL that notify when an execution finishes, we can achieve this modification by immediately streaming new inputs after receiving such notifications. Note that CSPARQL schedules the next execution right after the current one unless explicitly specified.

Then, given the size N_E of the input streamed into an engine $E \in \mathcal{E}$ and the total running time T_E that E needs to be processed by a query Q, assume that E *immediately* reads the new input once finishing with the previous one, the number of executions is N_E as E does one computation per input. When $comp(\mathcal{E}, Q, R, \epsilon)$ holds, the *comparable maximum execution throughput of* E is defined as $cmet(E, \mathcal{E}, Q, R, \epsilon) = N_E/T_E$.

Table 3 reports the value of $cmet$ for Q_1–Q_6 and Q_{10}. For Q_i, the comparable test is fulfilled by $comp(\{CQELS, CSPARQL, JTALIS\}, Q_i, 100, \epsilon_i)$ where $\epsilon_i = \max\{mm(E_1, E_2, Q_i, 100) \mid E_1 \neq E_2 \in \{CQ, CS, JT\}\}$ with the mismatch values taken from Table 2. It shows that CQELS and JTALIS have higher $cmet$ than CSPARQL by some orders of magnitude.

Scalability Tests. Next, we investigate how the systems behave wrt. to two aspects of scalability, namely (i) static data size, and (ii) the number of queries. Regarding the

former, we gradually increment the static data by generating test cases with increasing number of users. For the latter, we can use the same query template but augmenting the number of registered query instances. From queries generated from the same pattern, we expect better scalability effects on the engines that support multiple query optimization [12]. Figures 2 and 3 report $cmet$ when running the engines on those settings. As seen in Figure 2, C-SPARQL's $cmet$ dramatically decreases when the size of static data increases; JTALIS performs better than C-SPARQL. On the other hand, CQELS's performance only slightly degrades for complicated queries like Q5. In Figure 3, CQELS still outperforms to C-SPARQL and JTALIS but the speed of throughput deterioration is still linear like those of the counterparts. Here, the performance gains mainly come from the performance of single queries.

4 Findings and Discussion

As most of the considered systems are scientific prototypes and work in progress, unsurprisingly they do not support all features and query patterns. Moreover, the comparability tests in the Section 3.2 clearly exhibit that even for queries with almost identical meaning resp. semantics, the outputs sometimes are significantly different due to the differences in implementation. The differences in outputs are also contributed by intrinsic technical issues of handling streaming data , e.g., time management, potentially fluctuate execution environment [12, 15]. Therefore, any reasonable cross-system comparison must take comparability criteria akin to those we considered into account. In addition, comparability tests with tolerant criteria are useful for testing the correctness of a query engine if there is an equivalent baseline system, i.e, given that the baseline system has standard features that a new system should conform to.

The performance and scalability tests show that throughout C-SPARQL yields considerably lower throughput compared to JTALIS and CQELS. This provides further evidence for the argument that the recurrent execution may waste significant computing resources. Recurrent results can be useful for some applications, such as answering "return the last 10 tweets of someone." However, re-execution is unnecessary unless there are updates between consecutive executions, and thus "incremental computing" is mainly recommended in the literature [12, 15]. There, outputs are incrementally computed as a stream, and recurrences can be extracted by a sliding window. In this fashion, the output of eager incremental computing as by CQELS and JTALIS can be used to answer recurrent queries. As the incremental execution by CQELS and JTALIS outperforms the periodic computation of C-SPARQL by an order of magnitude, the conjecture is that they would also answer recurrent queries as described above more efficiently.

Comparing CQELS and JTALIS, the former performs better mainly because it uses a native and adaptive approach (cf. [14]). Note that the performance of C-SPARQL and JTALIS heavily depends on underlying systems, viz. a relational stream processing engine and Prolog engine, respectively. Their performance can thus benefit from optimizations of the (or use of the) underlying engines. Similarly, CQELS would profit from more sophisticated, optimized algorithms compared to the current one [14].

The scalability tests show that all engines have not yet been optimized for scalable processing. Apparently, C-SPARQL already exhibits performance problems on simple queries involving static data beyond 1 million triples. JTALIS, while capable of handling

these settings, struggles on more complicated queries with static datasets larger than 1 million triples. CQELS is the only system that precomputes and indexes intermediate results from sub-queries over the static data [14], and therefore scales well with increasing static data size. Clearly, this technique is not restricted to CQELS and applicable to C-SPARQL, JTALIS, and any other LSD engine to improve on scalability wrt. static data. However, CQELS does also not scale well when increasing the number of queries (sharing similar patterns and data windows). The same holds for C-SPARQL and JTALIS, which clearly testifies that none of the systems employ multiple query optimization techniques [12, 15] , e.g., to avoid redundant computations among queries sharing partial computing blocks and memory.

5 Related Work

We already mentioned characteristics of LSD engines that are critically relevant to this paper in Section 1. In-depth theoretical/technical foundations of LSD processing can be found in [15]. We next briefly review existing related benchmarking systems.

Linear Road Benchmark [4] is the only published benchmarking system for relational data stream processing engines so far. However, it focuses on ad-hoc scenarios to evaluate outputs. On the other hand, our work treats the processing engines and queries as black boxes. Furthermore, while Linear Road Benchmark only focuses on a single quantitative metric "scale factor," our evaluation studies several aspects of the systems as shown above. Concerning stream data generator, NEXMark[9] can be used to test relational data stream systems. However, not only is its data schema quite simple but the simulated data is unrealistically random.

Triple storage benchmarks. With increasing availability of RDF triple stores,[10] a number of RDF benchmarks have been developed to evaluate the performance of these systems.[11] However, most of the popular benchmarks such as BSBM [6], LUBM [13] and SP2Bench [17] are either limited in representing real datasets or mostly relational-like [11]. On top of that, none of them focus on time-varying data and continuous query. By extending recently developed Social Network Interlligence Benchmark (SIB)[12], our evaluation framework is natively designed not only to support continuous queries over LSD but also to simulate realistic graph-based stream data.

6 Conclusion

In this work, we propose the first—to the best of our knowledge—customizable framework with a toolset for cross-evaluating Linked Stream Data (LSD) engines. Along with the framework we developed a methodology and measures to deal with conceptual and technical differences of LSD engine implementations. Powered by this environment, another main contribution in this paper is a systematic and extensive experimental analysis, revealing interesting functional facts and quantitative results for state-of-the-art LSD engines (see Section 3). Our findings from this analysis identify performance

[9] http://datalab.cs.pdx.edu/niagara/NEXMark/

[10] http://www.w3.org/wiki/SemanticWebTools#RDF_Triple_Store_Systems

[11] http://www.w3.org/wiki/RdfStoreBenchmarking

[12] http://www.w3.org/wiki/Social_Network_Intelligence_BenchMark

shortcomings of these engines that need to be addressed in further developments of these, but also by future LSD processing engines.

It is often the case that linked stream data engines are rated negatively when compared with relational stream engines. Therefore, for further work, we will provide a baseline test set [4] with corresponding relational data schema [6] to compare LSD engines with relational ones. On top that, we plan to extend the evaluation framework in order to support LSD engines that enable continuous approximation queries and feature load shedding [12]. Also, distributed LSD engines are expected to be in place soon, and evaluating them is another challenging and interesting topic of research to pursue.

References

1. H. Alani, M. Szomszor, C. Cattuto, W. V. den Broeck, G. Correndo, and A. Barrat. Live social semantics. In *ISWC*, pp. 698–714, 2009.
2. D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *WWW*, pp. 635–644, 2011.
3. A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.
4. A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: a stream data management benchmark. In *VLDB*, pp. 480–491, 2004.
5. D. F. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus. An execution environment for C-SPARQL queries. In *EDBT*, pp. 441–452. ACM, 2010.
6. C. Bizer and A. Schultz. The Berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst*, 5(2):1–24, 2009.
7. A. Bolles, M. Grawunder, and J. Jacobi. Streaming SPARQL extending SPARQL to process data streams. In *ESWC*, pp. 448–462, 2008.
8. E. Bouillet, M. Feblowitz, Z. Liu, A. Ranganathan, A. Riabov, and F. Ye. A semantics-based middleware for utilizing heterogeneous sensor networks. In *DCOSS*, pp. 174–188, 2007.
9. J.-P. Calbimonte, O. Corcho, and A. J. G. Gray. Enabling ontology-based access to streaming data sources. In *ISWC*, pp. 96–111, 2010.
10. I. Celino, D. Dell'Aglio, E. D. Valle, M. Balduini, Y. Huang, T. L. amd Seon-Ho Kim, and V. Tresp. Bottari: Location based social media analysis with semantic web. In *ISWC*, 2011.
11. S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of rdf benchmarks and real rdf datasets. In *SIGMOD*, pp. 145–156, 2011.
12. L. Golab and M. T. Özsu. Data stream management. *Synthesis Lectures on Data Management*, pp. 1–73, 2010.
13. Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, 2005.
14. D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *ISWC*, pp. 370–388, 2011.
15. D. Le-Phuoc, J. X. Parreira, and M. Hauswirth. Linked stream data processing. In *Reasoning Web. Semantic Technologies for Advanced Query Answering*, pp. 245-289 2012.
16. P. Minh Duc, P. A. Boncz, and O. Erling. S3G2: A Scalable Structure-Correlated Social Graph Generator. In *TPCTC*, Turkey, 2012.
17. M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Sp^2bench: A SPARQL performance benchmark. In *ICDE*, pp. 222–233, 2009.
18. J. F. Sequeda and O. Corcho. Linked stream data: A position paper. In *SSN*, 2009.
19. A. Sheth, C. Henson, and S. S. Sahoo. Semantic sensor web. *IEEE Internet Computing*, 2008.