Year: 2015

Running out of Bindings? Integrating Facts and Events in Linked Data Stream Processing

Gao, Shen ; Scharrenbach, Thomas ; Kietz, Jörg-Uwe ; Bernstein, Abraham

Abstract: Processing streams of linked data has gained increased importance over the past years. In many cases the streams contain events generated by sensors such as traffic control systems or news releases. As a reaction to this increased need, a number of languages and systems were developed that are aimed at processing linked data streams. These systems/languages follow one of two pertinent traditions: either they perform complex event processing or stream reasoning. However, both kinds of systems only support simulating system states as a sequence of events. This paper proposes to model a new kind of data – Facts. Facts are temporal states stored in systems that combine events. Essentially, they trade space complexity for time complexity and reduce the intermediate variable bindings compared to other approaches. They also have the advantage of keeping queries relatively simple. In our evaluation, we compile queries for typical sensor-based use-cases in TEF-SPARQL, our SPARQL extension supporting Facts, C-SPARQL, and EP-SPARQL to the well-established Event Processing Language (EPL) running on the Esper complex event processing engine. Compared to simulate Facts, we show that modeling Facts directly only creates less than 1% of intermediate bindings and improves the throughput by up to 4 times.

Originally published at:
Gao, Shen; Scharrenbach, Thomas; Kietz, Jörg-Uwe; Bernstein, Abraham (2015). Running out of Bindings? Integrating Facts and Events in Linked Data Stream Processing. In: 4th International Workshop on Ordering and Reasoning, Bethlehem, PA, USA, 11 October 2015 - 12 October 2015, s.n..

Running out of Bindings? Integrating Facts and Events in Linked Data Stream Processing

Shen Gao, Thomas Scharrenbach, Jörg-Uwe Kietz, and Abraham Bernstein

University of Zurich*
{shengao,scharrenbach,juk,bernstein}@ifi.uzh.ch

Abstract. Processing streams of linked data has gained increased importance over the past years. In many cases the streams contain events generated by sensors such as traffic control systems or news releases. As a reaction to this increased need, a number of languages and systems were developed that are aimed at processing linked data streams. These systems/languages follow one of two pertinent traditions: either they perform complex event processing or stream reasoning. However, both kinds of systems only support simulating system states as a sequence of events.

This paper proposes to model a new kind of data – Facts. Facts are temporal states stored in systems that combine events. Essentially, they trade space complexity for time complexity and reduce the intermediate variable bindings compared to other approaches. They also have the advantage of keeping queries relatively simple. In our evaluation, we compile queries for typical sensor-based use-cases in TEF-SPARQL, our SPARQL extension supporting Facts, C-SPARQL, and EP-SPARQL to the well-established Event Processing Language (EPL) running on the Esper complex event processing engine. Compared to simulate Facts, we show that modeling Facts directly only creates less than 1% of intermediate bindings and improves the throughput by up to 4 times.

1 Introduction

Data on the web is no longer purely static, but increasingly dynamic. Traffic information is continuously updated, live streams of weather data are published, and news releases are, oftentimes, even semantically tagged. To address this dynamic nature, a number of languages and systems were developed to process streams of linked data rather than static datasets. When processing Linked Data streams we have to handle three kinds of data: first, classic static *background knowledge*, i.e., things we consider to be true, which are not expected to change their value during the lifetime of the system. Second, *events* in a stream, i.e. observations of things that happened with a clearly defined time range during which they happened. While the first two are well established, in this paper we discuss a third kind of data, i.e., *facts*. It models the *current* states of a system [5] making stateful stream processing more efficient.

Current systems/languages follow one of two pertinent traditions: either they perform complex event processing or stream reasoning. Today's *Linked Data complex*

* The research leading to these results has received funding from the European Union Seventh Framework Programme FP7/2007-2011 under grant agreement no 296126.

event processing systems, such as EP-SPARQL [1], were designed to perform graph pattern matching on streams of events. It focuses more on detecting the temporal order of RDF triples, such as the SEQ operator. EP-SPARQL's capabilities of handling system states, however, are limited, and states have to be "simulated" by a sequence of ordered events. For *stream reasoning systems* such as C-SPARQL, CQELS, and SPARQL$_{stream}$ the situation is the opposite: they were designed to perform graph pattern matching and aggregation on streams of facts. They perform graph pattern matching on a context that is oftentimes defined as sliding or tumbling time-windows. However, there is a significant overhead of retaining facts associated with such window operation. Furthermore, handling two and more facts at a time will make the queries extremely complicated and create even more overhead.

Consider the following running example: suppose we have one stream S that contains events of IPTV users switching from one channel to another such as:

$$\langle Alice, join, ChannelA[1,1]\rangle$$
$$\langle Bob, join, ChannelB[3,3]\rangle$$

By using a time-based tumbling window to cache the events arrived in the last 5 seconds we want to determine the average time users stay in each channel and repeat this query every 5 seconds. In both stream reasoning and complex event processing systems, a tumbling window normally "forgets" the data in the previous window when a new one starts. However, for this query, we cannot simply ignore the data of the previous window. We need to retain information about users who have not left a channel by either copying each binding to a new window until users leave the channel or sending "refresh" or "keep-alive" events representing the current system state.

Inspired by this example this paper proposes *Facts*, which convert insights computed from events and store them in a temporal table to save the cost of maintaining events between windows. Modeling both facts and events in the same formalism allows reducing the cost (in terms of time) of maintaining states in-memory. We show that it takes $O(n^2)$ time to maintain events in systems without Facts, whilst employing Facts reduces time complexity to the linear $O(n)$ time.

To fairly investigate the trade-offs involved, we implemented and compared the performance of well-established state-aware complex event processing system Esper queries that are written in three different ways, each of them is inspired by the following three systems, respectively: 1) EP-SPARQL, a stateless complex event processing system; 2) C-SPARQL, a stateless stream reasoning system; 3)TEF-SPARQL [5], our Fact/event integrated language. We establish empirically that storing Facts can significantly reduce the number of intermediate variable bindings produced.

In the following, we first introduce the concepts of events and Facts in detail, which is followed by a discussion of our three use-cases. Then we present the implementation of the use-cases and their evaluation followed by a discussion of our findings/limitations. We close with an overview over related work and some general conclusions.

2 Streams, Events, and Facts: Definitions and Implementation

Most current RDF stream processing (RSP) systems rely on events or using events to simulate states/facts. The basic *proposition of this paper is that these two concepts*

Events and Facts need to be integrated to allow for sufficient expressiveness whilst curbing an explosion of bindings for certain queries. To state this proposition, this section first defines the relevant concepts such as 'events' and 'Facts' in the context of RSP, illustrating how to implement Facts, and then presents our main hypotheses.

2.1 Concept definitions

Background knowledge are immutable statements that we consider as true. There are no explicit valid time intervals associated with them. They are not consumed via streams but are available to a RSP engine via SPARQL queries.

Streams are the input supplied to the RSP engines. Each data element in streams consists of a partially ordered unbound collection of RDF triples $\langle s, p, o \rangle$. A data element is associated with a time interval $[t_s, t_e]$, where t_s and t_e represent the starting and ending time for the data element. Therefore, a data elements is usually represented as:[1]

$$\langle s, p, o \rangle^* [t_s, t_e], \text{ where } * \text{ represents one or more RDF triples.}$$

Events are statements about things that happened, conceptually. For example "Alice joins ChannelA" is an event. Events are typically modeled by RDF triples in the streams. Once happened, an event never becomes invalid, but it may be irrelevant after a while. For example, a query is counting the number of events in the past 5 seconds. Events have no states but they may change the states of a system.

Facts are statements that a system *currently* considers true. Similar to Background knowledge, they represent the states of a system with the difference that Facts start being valid at a certain point in time (or $-\infty$), and they may be considered *invalid* after a certain period of time. E.g., "Alice is inChannelA" is a Fact and we know that this Fact has a well-defined start and end timestamp. Note that the end time may be ∞, indicating that the system believes this Fact to be true forever. Furthermore, even invalid Facts may still be part of future results. For example, computing the average number of viewers in the last hour may need to rely on Facts that ended in the last hour. While such invalid Facts may still be contained in the stream processing operators' caches as they remain immutable. In particular, the end timestamp of a Fact invalid can no longer change ensuring monotonicity.

Events vs. Facts Events can trigger Facts and vice versa. Systems can derive new states from events. For example, Alice joining channel B can trigger the Fact that Alice watches channel B. Some events can also make Facts invalid, e.g., Alice leaving channel B invalidates the Fact that Alice watches channel B. Facts, in turn, can also trigger events, for example, by maintaining each viewer's viewership, the system can generate the event for counting how many users for each channel every 5 minutes.

2.2 Implementing Facts

Simulating Facts by Events means maintaining the system's states on the stream for systems that cannot represent states explicitly. In the viewership use-case, e.g., one approach is to insert the users that have been watching a certain channel in the last matching context as a *refresh event* back into the stream. As a result, for each new context we have to produce a new binding for every channel.

[1] Note that we make no assumptions on how the triples are consumed by the system.

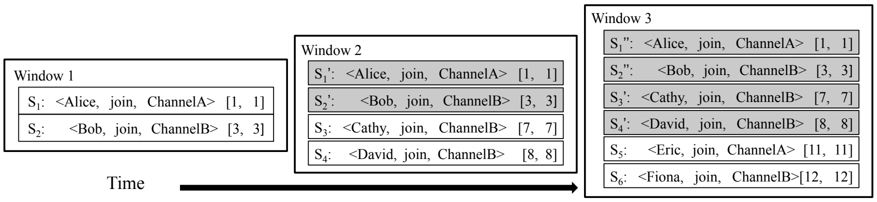
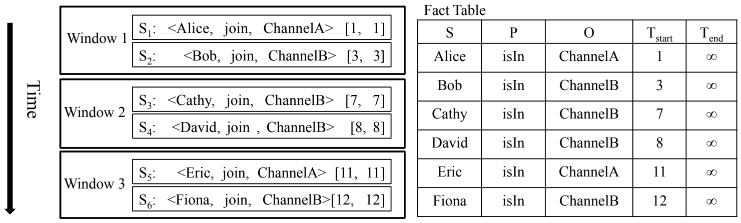

Window 1
S1: <Alice, join, ChannelA> [1, 1]
S2: <Bob, join, ChannelB> [3, 3]
Window 2
S1': <Alice, join, ChannelA> [1, 1]
S2': <Bob, join, ChannelB> [3, 3]
S3: <Cathy, join, ChannelB> [7, 7]
S4: <David, join, ChannelB> [8, 8]
Window 3
S1'': <Alice, join, ChannelA> [1, 1]
S2'': <Bob, join, ChannelB> [3, 3]
S3': <Cathy, join, ChannelB> [7, 7]
S4': <David, join, ChannelB> [8, 8]
S5: <Eric, join, ChannelA> [11, 11]
S6: <Fiona, join, ChannelB>[12, 12]
Time

Fig. 1: Retaining Events between windows


Time
Window 1
S1: <Alice, join, ChannelA> [1, 1]
S2: <Bob, join, ChannelB> [3, 3]
Window 2
S3: <Cathy, join, ChannelB> [7, 7]
S4: <David, join , ChannelB> [8, 8]
Window 3
S5: <Eric, join, ChannelA> [11, 11]
S6: <Fiona, join, ChannelB>[12, 12]

Fact Table

S	P	O	T_{start}	T_{end}
Alice	isIn	ChannelA	1	∞
Bob	isIn	ChannelB	3	∞
Cathy	isIn	ChannelB	7	∞
David	isIn	ChannelB	8	∞
Eric	isIn	ChannelA	11	∞
Fiona	isIn	ChannelB	12	∞

Fig. 2: Storing Events as Facts in a table

Generally, for each stream Fact we have to produce a fresh binding for every matching context. To avoid stream Fact duplication, the fresh Fact binding must only occur once. For tumbling time windows, we can easily fulfill this requirement by inserting the fresh binding as an event into the direct successor context of the matching context. Assuming we have a tumbling time window of $5seconds$ and the stream Fact would be valid at time $[1, 5]$. Then we would insert the refresh event to the next window. As shown in Figure 1, we have to create intermediate events, such as creating and copying $\langle$`:Alice :isIn :ChannelA`$\rangle[1, 1]$ (the gray triples) from Window 1 to Window 2. However, as long as *Alice* does not leave the channel, we have to retain such refresh events by copying them between windows. In a traditional tumbling window implementation, when a new window starts, it will trash all the events in the previous window. However, in our case, we have to examine every refresh event and copy those that are still valid. Therefore, in Figure 1 Window 3, we have not only the refresh events of $S3'$ and $S4'$, but also the $S1''$ and $S2''$ that are copied from Window 2. Simulating Fact by events will retain the events in the previous window. However, it has the shortcoming of creating excessive intermediate bindings.

Storing Facts. Instead of simulating Facts, we can directly store Facts in a table. As shown in Figure 2, we have the Fact $\langle Alice, is, ChannelA[1, \infty]\rangle$, when event S_1 enters the system. Note that we leave the ending time of the Fact as ∞, which means the Fact is currently valid. When a future event of $\langle Alice, quits, ChannelA[20, 20]\rangle$ comes, we invalid the Fact to be $S\ :\ \langle Alice, watches, ChannelA[1, 20]\rangle$. In this way, the fact table will record the temporal state of the system by only processing each event only once.

Complexity analysis: When simulating Facts with Events a worst-case situation implies that we need recursively copy *all* the events from previous window to the current one. When assuming the total input has n events and the average number of events in each window is ω that results in the following number of events to be copied:

$$T(n) = T(n-\omega) + T(\omega) = T(n-2\omega) + 2T(\omega) = \sum_{i=1}^{n/w} T(\omega) = \frac{1}{2}(1+\frac{n}{w})\frac{n}{w}T(\omega)$$

It will take quadratic running time $O(\frac{n^2}{\omega})$, since we need to process each data in the previous window whenever a new one starts. The worst-case space complexity for storing all events is $O(n)$. Note that the running time $O(\frac{n^2}{\omega})$ also depends on the size of a window. As will be shown in the experiments, longer window length reduces the overhead. In an extreme case, we can define a separate window with an infinite length to hold bindings. However, this case has already employed the idea of storing events as Facts.

If we store Facts in a temporal table, it is obvious that we need $O(n)$ time to process all the events, as we only process each event once. The space complexity remains the same: $O(n)$ to store all the facts.

Given these definitions we can now give our proposition of integrating events and facts as the following *Hypotheses*:

H1 Simulating Facts adds a significant overhead to RSP systems compared to modeling Facts in the language and the execution model explicitly.

H2 The ability of modeling Facts in the language leads to simpler queries.

We have shown that storing Facts can reduce the time complexity, compared to simulate facts. In the following, we will use three use-cases to empirically demonstrate the differences between the two approaches.

3 Use-Cases

This section will introduce three use-cases to support our claims. The use-cases are taken from the EU FP7 project ViSTA-TV.[2] We will show that they generalize in a straightforward manner. Our use-cases consider two input streams. The first stream contains events of IPTV users switching from one channel to another. The second stream provides information about the shows currently broadcast on each channel. IPTV providers are interested in the number of users currently watching a channel (Sec. 3.1), the average time of active users that stay with a channel (Sec. 3.2) and a TV show (Sec. 3.3).

3.1 Use-case 1: Viewership per IPTV Channel

The first use-case has a stream of input events that state when a user joins or quits a channel, and we need compute the number of current users per channel. In general, this use-case refers to computing an aggregate over a set of facts. We do not have to store all single facts (i.e., when a user join and quit a channel) but restrict ourselves to store the current value of the aggregate. Note that the aggregate is a stream fact itself. Since facts are immutable we have to create a new stream fact for each matching context (e.g., time window). Therefore, for this use-case, we expect stateful systems to perform equally well as stateless systems.

TEF-SPARQL will first construct a fact for each user that stores the time of joining a specific channel.

```
CONSTRUCT FACT UserFact  {?user :watching ?channel .}
(UNION  (SINCE   ?user :join ?channel)
(UNTIL  ?user :quit ?channel))
```

To compute the number of users per channel, we need to aggregate facts over channels:

[2] http://vista-tv.eu

```
SELECT ?channel AS CHANNEL, ?user as USER
(AGGREGATE ?channel, COUNT ?user
WHERE (CURRENT ?user :watching ?channel)
GROUP BY ?channel   EVERY  "P10S"^^xsd:Duration)
```

As long as every user can watch only one channel at a time, `COUNT ?user` is correct, otherwise `COUNT DISTINCT ?user` has to be used.

C-SPARQL first computes the number of “join” and “quit” events per channel and outputs these to another stream.

```
REGISTER STREAM UserJoinQuitCounts AS
CONSTRUCT { ?cnl uc:joinCount ?joinCount ; uc:quitCount ?quitCount . }
FROM STREAM <http://vista-tv.eu/userEvents/> [RANGE 10s TUMBLING]
WHERE { { SELECT ( COUNT(?userJoin) as ?joinCount ) ?cnl
WHERE { ?userJoin ue:join ?cnl } GROUP BY ?cnl }
    { SELECT ( COUNT(?userQuit) as ?quitCount ) ?cnl
    WHERE { ?userQuit ue:quit ?cnl } GROUP BY ?cnl } }
```

The second query performs the counting of user per channel. Since the information of join counts, quit counts and current users is available on the input stream, C-SPARQL can compute a new number of total current users and the new number of current users is inserted again into the source stream. Note that the new stream of counts is the overhead of maintaining facts. At the end of each window, it has to copy all of them to the new window. This overhead will be amplified in the next two use-cases.

```
REGISTER STREAM UserChannelCountsQuery AS
CONSTRUCT {?cnl uc:currentCount ?newCount . }
FROM STREAM <http://vista-tv.eu/userChannelCounts/> [RANGE 10s TUMBLING]
WHERE { { SELECT ( ?currentCount + ?joinCount - ?quitCount as ?newCount ) ?cnl
WHERE  {  ?cnl uc:joinCount ?joinCount ; uc:quitCount ?quitCount ;
uc:currentCount ?currentCount . } } }
```

EP-SPARQL. The number of users currently watching a channel A is a stateful information. Instead of copying the aggregate count event as C-SPARQL, EP-SPARQL has to model this by sending the incremented count on a stream whenever we consume an observation that a user joins channel A. In turn, we have to send the decremented count whenever we consume a quit event for channel A.

```
CONSTRUCT  {  _:xxx a :CountEvent ; :channel ?channel ; :count ?ncount . }
WHERE { ?count_event a :CountEvent ; :channel ?channel ; :count ?count .
  SEQ { ?user_event a :UserEvent ; :channel ?channel ; :action :join . }
  <ensure consecutive sequence>
  BIND(?ncount as ?count +1) }
```

Beside the additional count event overhead, there is no direct way in EP-SPARQL to ensure that 2 events form a direct consecutive sequence.

```
<ensure consecutive sequence>:
EQUALSOPTIONAL {  {?count_event a :CountEvent . }
  SEQ {  {?other_event a :CountEvent ; :channel ?channel . }
  UNION {?other_event a :UserEvent ; :channel ?channel .} }
  SEQ {?user_event a :UserEvent . }
} FILTER ( !bound(?other_event) )
```

We must repeat this query for quit events with BIND(?ncount as ?count - 1). Therefore, for each incoming event, it has the overhead of creating a new count event.

3.2 Use-Case 2: Average Viewing Time per Channel

This use-case contains a stream of events that state whether a user joins or quits a channel, and we want to compute the average time all active users (i.e., those currently watching a channel) have been watching that channel in the last time period. The computation shall be performed for each channel.

In general, this use-case refers to storing many instances of a kind of fact (i.e., one fact per user session). For a stateful system we only have to update each fact when it ends (i.e., the user quits the session); for other systems, we still have the overhead of retaining events between windows. For this use-case we expect stateful systems to outperform stateless systems.

TEF-SPARQL has queries that are very similar to use-case 1 with the addition that we need access to the start and end time of the UserFact from use-case 1. `?user :watching ?channel [?SC,?EC]` will bind `?SC` to the time point when this fact became true and `?EC` to the current time for the ones that are still true, which is enforced by `CURRENT`.

```
SELECT  ?channel AS Channel, ?Count AS CurrentUsers,
   ?TotalTime/?Count AS AverageDuration ,
   (AGGREGATE ?channel, SUM DURATION(?SC,?EC) AS ?TotalTime, COUNT ?user AS ?Count
   WHERE (CURRENT ?user :watching ?channel [?SC,?EC])
   GROUP BY ?channel
   EVERY "PT1M"^^xsd:Duration)
```

C-SPARQL queries work in the same way as it did in use-case 1. However, here we need to store the current state of a user-session on a separate stream along with its join time. This allows us computing the current watching time per session. Compared to a single aggregate count event in user-case 1, the overhead of storing states for each user is much higher. Furthermore, as we will show in the experiment section, the overhead depends on the size of window. When window size is small, we need maintain events more frequently, which causes even higher cost. For the sake of brevity we omit providing the actual EPL implementation of C-SPARQL queries in the paper. Please note that all queries are available online.[3]

EP-SPARQL For EP-SPARQL, we see two theoretical approaches to handle this use-case, however both fail in practice given the amount of data that needs to be handled. The first approach will use a window large enough to cache all the “join” events and then count the users and sum their duration for every join event of a user to a channel that have not got a “quit” event afterwards. However users may stay in channels for a very long time (i.e., we had cases where people stayed online in the same channel for several days). With millions of events per day and a window of several days long, the overhead is forbiddingly high. The second approach would be similar to the C-SPARQL solution above, which involves an extremely high overhead. For this reason, we omitted presenting the queries in EP-SPARQL for use-case 2.

3.3 Use-Case 3: Joining User Facts and Program Facts

In use-case 3, we are given a stream of events that describes the start and end of a session for any user watching a channel and a stream of events determines the start

[3] The queries are available at: https://goo.gl/igcYi9

and end of each program per channel. We want to compute the number of users per program and how long they watched this program on average. In general, this use-case corresponds to matching two types of facts. Both may be valid at the same time and influence each other during each other's lifetime.

TEF-SPARQL. For this use-case, we use the same UserFact from use-case 1. Additionally, we create a similar fact, called ProgramFact, that states which channel shows which show during which time.

```
CONSTRUCT FACT ProgramFact  {?channel :showing ?program .}
 (UNION  (SINCE   ?channel :start  ?program)
 (UNTIL  ?channel :finish ?program) )
```

When both the UserFact and the ProgramFact are true at the same time, we know the user is watching the show and create a Fact ViewShipFact to maintain this state.

```
CONSTRUCT FACT ViewShipFact  {?user :attends ?program .}
 {?user :watching ?channel ?channel :showing ?program}
```

Note that if the program on a channel C_1 changes from program P_1 to P_2 while a user is currently watching channel C_1, we have to emit a quit event for program P_1 and a join event for program P_2. In turn, when a user quits channel C_1 we have to emit a quit event for program P_1. Furthermore, we need to adjust the ViewShipFact to maintain such state. A user may zap through the programs and attend a show several times, to get his total time watching a show we need to aggregate all the ?user :attends ?show facts that are or were true in the window ?_ : broadcasts ?show:

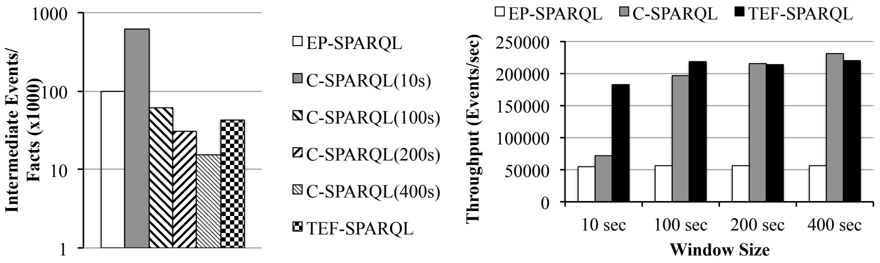
```
SELECT  ?show AS Show, ?DiffUser AS USER, ?TotalTime/?DiffUser AS AverageViewTime
 (AGGREGATE ?show, SUM DURATION(?SA,?EA) AS ?TotalTime, COUNT DISTINCT ?user AS ?DiffUser
 WHERE ?user :attends ?show [?SA,?EA]
 GROUP BY ?show  WHILE ?_  :showing ?show [?SS,?ES]
 EVERY "PT1M"^^xsd:Duration)
```

In the above query, `?user :attends ?show [?SA,?EA]` (resp. `?_ : showing ?show [?SS,?ES]`) will bind `?SA` (resp. `?SS`) to the time point when this fact became true and `?SE` (resp. `?ES`) to the time when it became out-of-scope for past facts (still in the window) and to the current time for the ones that are still true.

C-SPARQL and EP-SPARQL. This use-case requires maintaining the states of both the user sessions as well as the program sessions. Since stream reasoning systems provide no way of handling such event that a user switches twice and no program changes twice during the matching context (i.e., the window). We hence provide no queries for C-SPARQL but note that pure stream reasoning systems involve extreme difficulties with handling more than one stream fact in parallel. For EP-SPARQL, use-case 3 is even more complex than use-cases 1 and 2. For the same reasons as for use-case 2, we hence omit creating queries for EP-SPARQL. It would not give any further support or counterarguments for our claim.

4 Experimental Results

We evaluate our use-cases on TEF-SPARQL, C-SPARQL and EP-SPARQL to illustrate the differences among three stream processing models.



(a) Num. of Intermediate Events/Facts

(b) Throughput

Fig. 3: Results of EP-SPARQL, C-SPARQL and TEF-SPARQL in use-case 1

4.1 Experimental Setup

We choose the amount of intermediate variable bindings (for the refresh events simulating stream facts) as a major Key Performance Indicator (KPI), which principally reveals the processing overheads. We also report throughput as another KPI.

We model each use-case in all languages as EPL queries for the execution engine–Esper.[4] We chose Esper for three reasons: first, it is an open-source software widely adopted both by industry and academia. Second, it supports both event processing and stream reasoning. Also, we can implement facts by using so-called *named windows* or *context*. Third, the implementation of all three queries in the same execution engine reduces the bias when comparing the performance for our three use-cases.

We executed all use-cases on a real-world dataset of IPTV from the ViSTA-TV project. The data set we employed for use-case 1 and 2 contains 100,000 events of anonymous IPTV viewership logs (Log), where there are 234 different channels and 22,872 users. For use-case 3, we include an additional stream of Electronic Program Guide (EPG) events as described in Section 3.

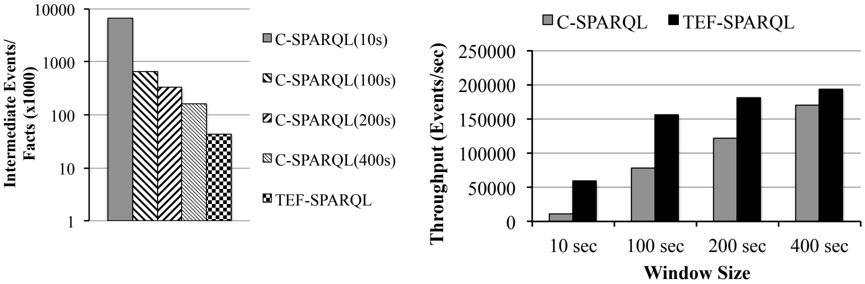
For use-cases 1 and 2, we conducted experiments for various window sizes. For the throughput measurements, we report average values over three repetitions. All experiments were conducted on a MacBookPro with a 2.7 GHz Intel Core i7 and 16GB of RAM running Mac OSX 10.9.1.

4.2 Results

Use-case 1: Overhead. In use-case 1, we implemented EP-SPARQL and C-SPARQL by creating a new binding type, named “Count Event”.

As shown in Fig. 3a, EP-SPARQL produces exactly 100,000 intermediate Count Events, whose amount equals to that of input Log Events. For each incoming event, EP-SPARQL needs to produce an additional variable binding for maintaining the counter. C-SPARQL’s amount of intermediate bindings depends on the window sizes. When the window size decreases, the amount of intermediate binding also decreases roughly at a linear rate. Smaller windows produce more count events, as they are created on a per window base. The number of count events significant influences the throughout. The stateful representation of stream facts in TEF-SPARQL requires no refresh events. Instead we maintain a UserFact table. Each “joins” Log Event create an entry in that table

[4] The actual queries are available online: http://goo.gl/jSa6yY



(a) Num. of Intermediate Events/Facts

(b) Throughput

Fig. 4: Results of C-SPARQL and TEF-SPARQL in use-case 2

whereas a "quit" Log Event will delete it. From the query we can observe that TEF-SPARQL creates an entry for each pair of user and channel, while C-SPARQL only creates the Count Events for each window. For this specific use-case, when the window size is large, C-SPARQL will have less overhead than TEF-SPARQL. Therefore, we have to consider the trade-off between direct modelling or simulating Facts.

Use-case 1: Throughput. Fig. 3b reports the throughput for different models. EP-SPARQL has the lowest throughput, which results from its replication of the input stream. For each event in the stream, it performs at least as many joins as the number of events in the time-window. C-SPARQL's per-window overhead improves the performance, compared with EP-SPARQL.

TEF-SPARQL outperforms C-SPARQL by up to 4 times for small window sizes, but the effect diminishes when the window size increases. A window is essentially an in-memory store. Increasing the window size in C-SPARQL makes stored refresh events more persistent per time-unit and reduce the overhead. On the other hand, it clearly shows that the overhead of refresh events can have a negative impact on system performance with small window size.

Use-case 2. In use-case 2, simulating facts has to maintain several orders of magnitude more events than in use-case 1. We hence expect the results for use-case 1 to be replicated with an even stronger impact of overhead on system performance. We therefore omitted the performance of EP-SPARQL, which can only be worse. For C-SPARQL the overhead (Fig. 4a) is about 10 times larger than for use-case 1 while we have to maintain about 100 times more stream facts (one for each of the 220 channels for use-case 1 and one for each of the 22,000 users in use-case 2). This is reflected by the results for input throughput for use-case 2 (Fig. 4). C-SPARQL catches up with TEF-SPARQL with increasing window size, yet at a slower pace.

Use-case 3. We report results for TEF-SPARQL only, since we cannot model this use-case with C-SPARQL (Sec. 2). The input-throughput is roughly 150,000 events per second and is comparable to our previous findings. We hence conclude that also handling multiple concurrent facts is feasible.

5 Discussion

While we developed TEF-SPARQL [5] as a universal algebra for stateful RDF stream processing, we never intended to create a custom tailored execution engine. This is in contrast to other existing stream processing languages, which were designed or at least influenced by their corresponding execution model. TEF-SPARQL aims at investigating the potential usefulness of Facts.

Our study revealed two issues that current RSP languages bear: first, our proof and experiments support our hypothesis that the overhead created by simulating stream Facts can have a severe impact on input throughput. Although it does not scale with the number of stream Facts, there exists a trade-off for storing state on the stream stateful systems storing Facts in an in-memory store. The smaller the matching context (i.e., the window size), the more favorable are languages allowing for stateful processing.

Considering that stateless systems perform in-memory event processing, they bear the overhead of repeatedly producing the state information at each matching cycle. Even if they currently cannot handle more than a single Fact (see below), we believe that the necessary changes to both the language and the implementation are rather small. Regarding our claim of query simplicity, we found that the Fact simulation requires to express state in multiple queries, whereas a stateful system can present Facts in a single query. Second, as a side-effect we found that more than one stateful entities (i.e., a Fact) cannot be handled in parallel by current RSP systems. Pure stream reasoning systems cannot react to state changes within the matching context (e.g., time window) and the stateless complex event processing system EP-SPARQL would have to replicate the input streams for storing the stream Facts.

We deliberately chose to limit our study to intermediate events and throughput, in order to investigate the effects of overhead. For future work we will investigate how the use-cases behave for other key-performance indicators. We also limit our study to EP-SPARQL and C-SPARQL for complex event processing and stream reasoning. When focusing on handling Facts, C-SPARQL is similar to CQELS and SPARQL_stream. As of yet, none provides the ability of Facts as a language inherent feature but each of them has to maintain Facts on the stream. The implementation in Esper would hence be unchanged. Even the ability to store data via SPARQL Update would not sufficiently implement the concept of stream Facts. Facts must be available to the execution engine in the same way as events to avoid the overhead of external data access. Otherwise, we lose the ability of holding invalid Facts in the execution engine. We want to point out that such a change would not require dire changes to the existing systems. For C-SPARQL, for example, we could add a `REGISTER FACT` statement in order to be able to distinguish between background knowledge and events (which C-SPARQL can already do) as well as Facts.

6 Related Work

This section will give an overview of RSP systems. RSP is an indispensable part of Linked Data. For example, a large amount of semantic annotated data is generated from the widely deployed sensor networks. To model such data, the RDF temporal notation [4] has been proposed to extend the RDF data representation. Various similar data representation has been proposed along with different Semantic stream processing systems. Research data set of Linked Stream Data has also been proposed by [9]. The authors introduced the SRBench as a general-purpose benchmark designed for streaming RDF/SPARQL (strRS) engines. It also copes with a broad range of use-cases typically encountered in real-world scenarios. Our work complements the current use-cases by introducing the Fact data type.

To cope with the need of processing linked stream data, various semantic stream processing engines have been proposed. As discussed, EP-SPARQL, as a CEP system, extends the ETALIS system with a flow-ready extension of SPARQL [1] [8]. C-SPARQL [2] performs query matching on subsets of the information flow, which are defined by windows. CQELS [6] "implements the required query operators natively to avoid the overhead and limitations of closed system regimes". It optimizes the execution by dynamically re-ordering operators. SPARQLStream [3] is a streaming extension to SPARQL that allows users to query relational data streams over a set of stream-to-ontology mappings. The language supports powerful windowing constructs and SRBench [9] uses it as the default engine for evaluation. Besides CEP systems and stream reasoning systems, INSTANS [7] is an incremental SPARQL query engine based on the Rete algorithm. In INSTANS, instead of using windows, it has *timed events*, which is scheduled to wake-up periodically.

None of the above systems consider the real-word use-cases of dynamic state processing. Our model uses facts to model states, which simplifies the query complexity and saves a large amount of variable bindings to improve system's efficiency.

7 Conclusion

Our paper revealed that Facts are a useful feature of RSP, even for non-complex use-cases. Not only it can reduce the running time complexity, but also simplifies modeling queries. Even the simple use-case with a single Fact, simulating Facts with events can cause a significant overhead. Situations that call for just two Facts that may influence each other cannot be handled easily by pure stream reasoning systems and/or severely hamper the processing capabilities of complex event processing systems. In future, we will extend existing systems with the capability of handling Facts and systematically investigate the performance of using Fact with use-cases of different complexity

References

1. D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *WWW*, pages 635–644, 2011.
2. D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. C-SPARQL: A Continuous Query Language for RDF Data Streams. *International Journal of Semantic Computing*, pages 3–25, 2010.
3. J.-P. Calbimonte, O. Corcho, and A. J. G. Gray. Enabling ontology-based access to streaming data sources. In *ISWC*, pages 96–111, 2010.
4. C. Gutierrez, C. A. Hurtado, and A. Vaisman. Introducing time into rdf. *IEEE Trans. on Knowl. and Data Eng.*, 19(2):207–218, 2007.
5. J.-U. Kietz, T. Scharrenbach, L. Fischer, A. Bernstein, and K. Nguyen. TEF-SPARQL: The DDIS query-language for time annotated event and fact Triple-Streams. Technical report, University of Zurich, Department of Informatics, 2013.
6. 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*, 2011.
7. M. Rinne, E. Nuutila, and S. Törmä. Instans: High-performance event processing with standard rdf and sparql. In *ISWC*, 2012.
8. R. Stühmer, Y. Verginadis, I. Alshabani, T. Morsellino, and A. Aversa. PLAY: Semantics-based Event Marketplace. In *PRO-VE*, Dresden, Germany, 2013.
9. Y. Zhang, P. M. Duc, O. Corcho, and J.-P. Calbimonte. SRBench : A Streaming RDF / SPARQL Benchmark. In *ISWC*, pages 641–657, 2012.