Conclusions

The use of the Internet as a major source of information created new challenges for computer science and let to significant innovation in areas such as databases, information retrieval and semantic technologies. Currently, we are facing another major change in the way information is provided. Traditionally information used to be mostly static with changes being the exception rather than the rule. Nowadays, more and more dynamic information, which used to be hidden inside dedicated systems, is getting available to decision makers. Data Streams - unbounded sequences of time-varying data elements - are pervasive. They occur in a variety of modern applications spanning from sensor networks, which are used as “nervous system” of large scale reactive applications, to social media, which are increasingly adopted to distribute and present information in real-time. They form a “continuous” flow of information with the recent information being more relevant as it describes the current state of a dynamic system.

Continuous processing of homogenous data stream and events has been largely investigated in the database community since the late ’90s. Specialised Data Stream Management Systems (DSMS) [Garofalakis et al., 2007] and Complex Event Processors (CEP) [Luckham, 2001] are available on the market (e.g., StreamBase\(^1\), recently bought by TIBCO, and Esper\(^2\)) and features of DSMS/CEP are appearing also in major database products, such as Oracle CEP\(^3\), Microsoft StreamInsight\(^4\) and IBM

\(^1\)http://www.streambase.com/
\(^2\)http://esper.codehaus.org/
\(^3\)http://www.oracle.com/technetwork/middleware/complex-event-processing/
8. CONCLUSIONS

InfosphereStream\textsuperscript{1}.

In 2009, the position paper [j.1, 2009]\textsuperscript{2}, which I wrote together with Stefano Ceri, Frank van Harmelen and Dieter Fensel, called the Semantic Web community to investigate languages, tools and methodologies for representing, managing and reasoning on heterogenous data streams and complex events in presence of expressive domain models. The Semantic Web community was still focusing on rather static data. In existing work on logical reasoning, the knowledge base was always assumed to be static (or slowly evolving). The work on changing beliefs [Gardenfors, 1992] on the basis of new observations was proposing solutions that were far too complex to be applicable to gigantic data streams of the kind motivating this thesis. [j.1, 2009] proposes to name this new research topic \textit{Stream Reasoning}.

This final chapter returns to the research question formulated in Chapter 1 and presents the answers based on my publications. It is worth to note that all our papers have appeared over the past seven years. All publications were innovative at the time of publishing, but, in the meantime, the state of the art has also progressed. For this reason, this chapter also takes a look at the current state of the art showing where consensus is emerging and discussing open issues.

The remainder of the chapter is organised as follows. Section 8.1 focuses on sub-question SQ.1 and discusses why extending the Semantic Web towards Stream Reasoning is possible. Section 8.2 addresses sub-question SQ.2 showing that optimising stream reasoning algorithms to provide reactive answers is possible. Section 8.3 points to sub-question SQ.3 reporting on how the combination of Deductive and Inductive Stream Reasoning allows to cope with the noisy and incomplete nature of data streams. Finally, Section 8.4, wraps up, discusses open issues and casts some light on future research directions.

\textsuperscript{1}http://www.ibm.com/software/data/infosphere/streams/

\textsuperscript{2}In order to make it easier to spot the papers that form this thesis among all those referenced in this final chapter, I cite them using the following pattern: \texttt{[(publication type), (progressive number), (year)]} where \texttt{p} stays for published in a journal, \texttt{c} for conference, \texttt{w} for workshop, and \texttt{o} for other venues (e.g., poster, book chapter, etc.). The numbers and the years follow the numbering used in Chapter 1.
8.1 Extending the Semantic Web towards Stream Reasoning

Chapter 1 introduced the sub-question:

SQ.1 Is it possible to (syntactically and semantically) extend the Semantic Web stack in order to represent heterogenous data streams, continuous queries, and continuous reasoning tasks?

This section positively answers this question. Section 8.1.1 presents my original proposal of RDF stream, which extends RDF to represent data streams, and it compares such a proposal with alternative ones emerged in parallel or after. Section 8.1.2 and Section 8.1.3 depict my suggestion (namely C-SPARQL) to extend the syntax and the semantics of SPARQL to support respectively continuous queries and continuous reasoning tasks and they compare it with alternative ones. Finally, Section 8.1.4 focuses on my implementation experiences (namely the C-SPARQL Engine and the Streaming Linked Data Framework) and those of others.

8.1.1 Extending RDF to Represent Data Streams

The DSMS community defines data streams as unbounded sequences of time-varying data elements \( \langle s, \tau \rangle \), where \( s \) is a tuple belonging to the schema of \( S \) and \( \tau \in \mathbb{Z}^+ \) is the timestamp of the data element [Babcock et al., 2002]. Normally, the sequence of timestamps is assumed to be non-decreasing\(^1\) to exclude out of order and to allow for asserting contemporaneity (or simultaneity) of data elements in the data stream (i.e., tuples with the same timestamp).

A classical example of data stream is illustrated in the Linear Road Benchmark [Arasu et al., 2004] which simulates a toll system for the motor vehicle expressways of a large metropolitan area. The position of the vehicles is represented with a time-varying data element of the form:

\[
\langle \text{VehicleID}, \text{Speed}, \text{ExpressWayID}, \text{Lane}, \text{Direction}, \text{Position} \rangle
\]

\(^{1}\)If \( \tau_i \) is the timestamp of a tuple \( t_i \) in the stream, for each tuple \( t_i \), which follows \( t_1 \), the timestamp \( \tau_i \) of \( t_i \) is greater or equal to \( \tau_1 \).
8. CONCLUSIONS

Inspired by this definition, in [o.1, 2009], Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Michael Grossniklaus and I propose the notion of an RDF Stream as an unbound sequence of time-varying triples \(<t, \tau>\) where \(t\) is an RDF triple and \(\tau \in \mathbb{Z}^+\) is a non-decreasing timestamp.

The novelty of RDF streams w.r.t. data streams is not in its definition, but in what it enables. When the information flow is a graph evolving over time, RDF streams are more adequate data model than (relational) data stream. For instance, micro-posts are small graphs part of a larger (social) graph. A micro-post is a short text posted by a user from a given location, containing zero or more hashtags, including zero or more links, referring to zero or more users, and potentially retweeting another tweet. Squeezing microposts into tuples is less natural than representing them in RDF.

Consider for instance, the tweet “Four more years. http://t.co/bAJE6Vom”\(^1\) posted by Barack Obama on 2012 Nov 7 at 4:16am that was replied by many users (e.g., Alicia Keys at 6.34pm), it was retweeted 772,301 times and it is in the favourites of 293,687 Twitter users. This information flow is hard to represent in a single relational data stream with a fix schema, but it can be represented as an RDF stream\(^2\) as follows:

\[
\begin{align*}
\text{triple} & \quad \text{ts} \\
\text{t:266031293945503744 a sioc:Post .} & \quad [\tau_1] \\
\text{t:266031293945503744 sioc:has_creator t:BarackObama .} & \quad [\tau_1] \\
\text{t:266031293945503744 sioc:content "Four more years." .} & \quad [\tau_1] \\
\text{t:266031293945503744 sioc:links_to <http://t.co/bAJE6Vom> .} & \quad [\tau_1] \\
\text{t:somebody t:favourites t:266031293945503744 .} & \quad [\tau_2] \\
\text{t:somebodyelse t:favourites t:266031293945503744 .} & \quad [\tau_3] \\
\text{t:266247220247031808 a sioc:Post .} & \quad [\tau_n] \\
\text{t:266247220247031808 sioc:has_creator t:aliciakeys .} & \quad [\tau_n] \\
\text{t:266247220247031808 sioc:content "@BarackObama WE did it!!!" .} & \quad [\tau_n] \\
\text{t:266247220247031808 sioc:reply_of t:266031293945503744 .} & \quad [\tau_n]
\end{align*}
\]

The same RDF stream can accommodate a variety of data elements, whereas a data stream allows only tuples corresponding to a defined relation to be streamed.

It is also worth to note that the term RDF stream denotes a data model. It does not imply that data has to be physically represented as time-varying triples. It also

\(^1\)https://twitter.com/BarackObama/statuses/266031293945503744
\(^2\)The example of RDF stream uses the SIOC vocabulary [Breslin et al., 2006] and an hypothetical twitter namespace t.
allows to represent non-RDF data stream as virtual RDF streams, just as virtual RDF graphs can represent non-RDF databases.

Moreover, this extension is at the logical level and it does not impose any specific syntax. In [w.2, 2010], Davide Francesco Barbieri and I propose the Streaming Linked Data format to publish RDF streams following the Linked Data principles [Bizer et al., 2009]. This format uses two types of RDF named graphs: instantaneous Graphs (iGraphs) to group all the triples with the same timestamp, and streaming Graphs (sGraphs) to represent a portion of an RDF stream as a list of iGraphs. The sGraph for the example above can be serialised in the Streaming Linked Data format as follows:

```rml
:sgraph sld:lastUpdate "⌧n"^^xsd:dataTime .
:sgraph sld:expires "⌧n+1"^^xsd:dataTime .
:sgraph rdfs:seeAlso :iGraph⌧1 .
:iGraph⌧1 sld:receivedAt "⌧1"^^xsd:dataTime .
:sgraph rdfs:seeAlso :iGraph⌧2 .
:iGraph⌧2 sld:receivedAt "⌧2"^^xsd:dataTime .
:sgraph rdfs:seeAlso :iGraph⌧3 .
:iGraph⌧3 sld:receivedAt "⌧3"^^xsd:dataTime .
...
:sgraph rdfs:seeAlso :iGraph⌧n .
:iGraph⌧n sld:receivedAt "⌧n"^^xsd:dataTime .
```

The triples in the iGraph identified by :iGraph⌧i are those timestamped with ⌧i in the example above. For instance the content of the iGraph identified by :iGraph⌧1 is

```rml
t:266031293945503744 a sioc:Post .
t:266031293945503744 sioc:has_creator t:BarackObama .
t:266031293945503744 sioc:content "Four more years." .
t:266031293945503744 sioc:links_to <http://t.co/bAJE6Vom> .
```

It is worth to note that my Streaming Linked Data format not only provides a syntax for RDF streams, but it also addresses the problem of a stream processing engine to decide whether all the time-varying data elements with the same timestamp have been received. In DSMS, without the introduction of punctuation [Tucker & Maier, 2002], this problem is semi-decidable because, the engine knows that it has received all the data elements with the same timestamp as soon as the timestamp increases, but when the flow is not continuous the engine may wait for an indefinitely long time before the timestamp increases. The Streaming Linked Data format, which proposes to timestamp graphs instead of triples, introduces a form of punctuation.
8. CONCLUSIONS

The notion of RDF Stream as time-varying timestamped triples was adopted in several other works [Calbimonte et al., 2010; Komazec et al., 2012; Le-Phuoc et al., 2012; Rinne et al., 2012] while the idea of time-varying timestamped graphs is still only used in the Streaming Linked Data framework.

The two notions above use point in time semantics\(^1\), thus they assume each data element (i.e., either a triple or a graph) to be annotated with one timestamp. An alternative approach is to adopt interval base semantics. In this case each data element is annotated with two timestamps that respectively denote the start and the end of a time period. In the area of Stream Reasoning, only [Anicic et al., 2012] adopts interval-base time semantics.

The main open issue w.r.t. RDF streams is the lack of a common syntax and protocol. Each of the existing work proposes a proprietary extension of one of the existing RDF syntax and hardcodes in the implementation the protocol for transmitting time-varing triples (or graphs) across the Web. This is becoming a major bottle neck for adoption by practitioners and for comparative research in the field. A community action is required; discussions are undergoing in the RDF Stream Processing community group\(^2\) recently established at W3C (shortly, W3C RSP group). The current consensus is for modelling an RDF stream as an unbound sequence of time-varing RDF graphs \(<g, \tau>\) (where where \(g\) is an RDF graph and \(\tau \in \mathbb{Z}^+\) is a non-decreasing timestamp put by the consumer of the RDF stream) optionally annotated with properties that carry more timestamps, e.g., the generatedAtTime and invalidatedAtTime properties of the provenance ontology [Belhajjame et al., 2013].

Wrapping up, this section provides evidence that it is possible to extend RDF to represent data streams. The first two rows of Table 8.1 illustrate my two proposals, whereas the other rows (but the last one) report on the choices of alternative approaches available in the current state-of-the-art. The last row present the current consensus reached within the W3C RSP group as of July 2015.

\(^1\) Interested readers may refer to [Jensen & Snodgrass, 1996] for a broad discussion on semantics of time-varying information.

\(^2\) http://www.w3.org/community/rsp/
8.1 Extending the Semantic Web towards Stream Reasoning

Table 8.1: A wrap up of the alternative proposals for extending RDF to represent data streams. The first two rows illustrate my two ones.

<table>
<thead>
<tr>
<th>What</th>
<th>Data element</th>
<th>Time model</th>
<th># of timestamps</th>
</tr>
</thead>
<tbody>
<tr>
<td>[o.1, 2009]</td>
<td>triple</td>
<td>point in time</td>
<td>1</td>
</tr>
<tr>
<td>[w2, 2010]</td>
<td>graph</td>
<td>point in time</td>
<td>1</td>
</tr>
<tr>
<td>[Calbimonte et al., 2010]</td>
<td>triple</td>
<td>point in time</td>
<td>1</td>
</tr>
<tr>
<td>[Le-Phuoc et al., 2012]</td>
<td>triple</td>
<td>point in time</td>
<td>1</td>
</tr>
<tr>
<td>[Rinne et al., 2012]</td>
<td>triple</td>
<td>point in time</td>
<td>1 (implicit)</td>
</tr>
<tr>
<td>[Komazec et al., 2012]</td>
<td>triple</td>
<td>point in time</td>
<td>1</td>
</tr>
<tr>
<td>[Anicic et al., 2012]</td>
<td>triple</td>
<td>interval-bases</td>
<td>2</td>
</tr>
<tr>
<td>W3C RSP group</td>
<td>graph</td>
<td>flexible</td>
<td>1 (also implicit) or 2</td>
</tr>
</tbody>
</table>

8.1.2 Extending SPARQL Syntax and Semantics to Support Continuous Queries

The previous section shows how to extend the Semantic Web stack to represent data stream introducing the RDF stream data model. This section provides a positive answer to the part of sub-question SQ.1 that investigates the possibility to extending SPARQL syntax and semantics to support continuous queries and continuous reasoning tasks.

In [o.1, 2009;j.2, 2010;j.3, 2010], my co-authors and I propose Continuous SPARQL (C-SPARQL) as a way to express SPARQL queries on multiple RDF streams as well as static information stored in RDF graphs. [o.1, 2009] contains the first syntactic sketch of C-SPARQL based on the CQL [Arasu et al., 2006]. In particular, C-SPARQL inherits the notion of registered queries and windows through which RDF stream are observed. [j.2, 2010] formalises the semantics of C-SPARQL and [j.3, 2010] updates C-SPARQL to SPARQL 1.1 syntax and semantics. Specifically, [j.3, 2010] records the decision for adopting SPARQL 1.1 aggregates instead of those proposed in [o.1, 2009;j.2, 2010].

An example of C-SPARQL query is illustrated in Listing 8.1. It monitors an RDF stream of user opinions (identified by http://ex.org/opinions), it knows who follows whom (the information is in the RDF graph identified by http://ex.org/followers) and it reports every 5 minutes on a output RDF stream who are the opinion makers in the last 30 minutes. A user is an opinion maker about a topic, if she stated an opinion
(e.g., :alice likes :wonderland) and at least 3 of her followers did the same after her (e.g., :bob, :carolin and :david :likes :wonderland) within 30 minutes.

```
REGISTER STREAM OpinionMakers AS
CONSTRUCT { ?opinionMaker sd:about ?topic }
FROM STREAM <http://ex.org/opinions> [RANGE 30m STEP 5m]
FROM <http://ex.org/followers>
WHERE {
  FILTER ( cs:timestamp(?follower ?opinion ?resource) >
           cs:timestamp(?opinionMaker ?opinion ?resource) )
}
HAVING ( COUNT(DISTINCT ?follower) > 3 )
```

**Listing 8.1:** An example of C-SPARQL query

Line 1 contains the registration clause that starts with the keyword REGISTER and ends with the keyword AS. The REGISTER clause tells the engine to start the continuous execution of the query, which follows the AS clause. The STREAM clause requires the resulting triples to be output as time-varing triples on an RDF stream. The text OpinionMakers is used both to identify the query and to build the identifier of the RDF stream it generates.

Line 2 defines the form of the output using a SPARQL CONSTRUCT clause. Line 3 uses the clause FROM STREAM to introduce in the dataset of the query the stream http://ex.org/opinions. In particular, the clause [RANGE 30m STEP 5m] defines a window open on the stream that considers only the last 30 minutes and slides every 5 minutes. The computation of the query is triggered every time the window slides. Line 4 introduces in the default graph the named graph http://ex.org/followers using the clause FROM of SPARQL.

Lines 5 opens the WHERE clause of SPARQL that includes the triple patterns to match (Lines 6-8) and the conditions to keep a solution mapping (Lines 9-10). In particular, Lines 6-8 look for users that express the same opinion on the same topic and follow each others. Lines 9-10 use the *timestamp function*\(^1\) of C-SPARQL, which

\(^1\)It is worth to note that the timestamp function illustrated in Listing 8.1 is the one implemented in the current version of the C-SPARQL Engine. It is different from the one proposed in [a.1, 2009; j.2, 2010; j.3, 2010]. This one accepts as parameters a triple pattern, whereas the original one accepts a variable. The original proposal offers an handier syntax, but the current one better suites the use
8.1 Extending the Semantic Web towards Stream Reasoning

allows to access the timestamp of the triples that match a given triple pattern, to verify that the opinion maker expresses her opinion before her followers. Finally, Line 11 uses the SPARQL 1.1 `HAVING` clause to include in the list of solution mappings to stream out only opinion makers that influenced at least three of their followers.

It is worth to note that the `REGISTER STREAM` clause requires the `CONSTRUCT` or `DESCRIBE` query forms because it has to stream on an RDF stream. However, C-SPARQL also supports the registration of instantaneous queries, i.e., SPARQL queries that are periodically run against the RDF streams and graphs. This second case can be declared using the `REGISTER QUERY` clause that allows also `SELECT` and `ASK` query forms.


After the publication of C-SPARQL several alternatives emerged. The most notable are: SPARQL Stream [Calbimonte et al., 2010], CQELS language [Le-Phuoc et al., 2012] and EP-SPARQL [Anicic et al., 2011a].

SPARQL Stream is similar to C-SPARQL. The main differences are in the lack of the timestamp function and the possibility (also present in CQL [Arasu et al., 2003b]) to asks only for delta-answers. C-SPARQL outputs the answer to the continuous query periodically: the entire answer is repeated even if in two subsequent answers the result does not change. In CQL terminology, it supports only the Rstream operator among the possible relation-to-stream ones. In SPARQL Stream, it is possible to ask the system not to repeat answers, but only to stream the differences between two subsequent answers. In CQL terminology, SPARQL Stream supports also the Istream (i.e., the results in the current answer not present in the previous one) and Dstream (i.e., the results present in the the previous answer and not in the new one) operators.

CQELS language does not include a timestamp function, focuses on Istream only, and offers a different syntax for the WINDOW close. C-SPARQL proposes to add RDF Streams as a new type of dataset using the `FROM STREAM` clause, whereas CQELS assumes the stream to be already registered in the system and uses the `STREAM` clause as a variation of the `GRAPH` one in the `WHERE` clause. Listing 8.2 illustrates how to write the C-SPARQL query in 8.1 in CQELS language.

cases the C-SPARQL Engine was deployed into.
8. CONCLUSIONS

```
CONSTRUCT { ?opinionMaker sd:about ?topic }
WHERE {
  GRAPH <http://ex.org/followers> {
  }
  STREAM <http://ex.org/opinions> [RANGE 30m SLIDE 5m] {
    FILTER (?followerTime > ?opinionMakerTime )
  }
} HAVING ( COUNT(DISTINCT ?follower) > 3 )
```

Listing 8.2: A CQELS query equivalent to the C-SPARQL one presented in Listing 8.1

Line 6 illustrates how the syntax of CQELS associates a Basic Graph Pattern to a window open on a stream using the `STREAM` and the window `[RANGE 30m SLIDE 5m]` clauses.

It is worth to note that the lack of the timestamp function requires to change the data model of the information flowing in the RDF stream `http://ex.org/opinions` to represent the timestamp explicitly in the data. For instance, the timestamped triple, which is adequate to get the intended answer of the C-SPARQL query in Listing 8.1,

```:alice :likes :wonderland . [τi]```

in order to get the intended answer from the CQEL query in Listing 8.2, has to be represented as

```
:alice :likes _:a1 . [τi]
_:a1 :what :wonderland . [τi]
_:a1 :when "τi"^^xsd:dateTime . [τi]
```

where `_:a1` indicates a blank node, so to make the timestamp matchable using the triple patterns at Lines 7 and 8.

It is worth to note that the timestamp function of C-SPARQL allows for a more compact data representation and it makes query writing handier, but this comes at the cost of an extra data structure to manage the timestamps in the C-SPARQL Engine.

C-SPARQL, SPARQL Stream, and CQELS language are all inspired by DSMS. A different alternative is offered by EP-SPARQL [Anicic et al., 2011a] which is inspired by CEP. It does not offer window clauses. Instead, it contains language constructs to
8.1 Extending the Semantic Web towards Stream Reasoning

detect temporal sequences in graph pattern matching (e.g., \texttt{SEQ}) and functions to access the length of the time interval associated to the graph patterns (i.e., \texttt{getDURATION()}).

Listing 8.3 illustrates how to write the C-SPARQL query in 8.1 in EP-SPARQL. Line 6 illustrates how to use the \texttt{SEQ} clause to declare that the triple pattern \texttt{?opinionMaker ?opinion ?topic} has to be matched before the pattern \texttt{?follower ?opinion ?topic}. This in C-SPARQL is obtained using the \texttt{timestamp} function. Line 7 illustrates how to use the \texttt{getDURATION()} function to impose that the duration of the time interval determined by the \texttt{SEQ} clause is less than 30 minutes. This in C-SPARQL is declared using the window clause.

\begin{verbatim}
CONSTRUCT { ?opinionMakers sd:about ?topic }
WHERE {
FILTER (getDURATION() < "P30m"^^xsd:duration)
} HAVING (COUNT(DISTINCT ?follower) > 3 )
\end{verbatim}

Listing 8.3: A EP-SPARQL query equivalent to the C-SPARQL one presented in Listing 8.1

Wrapping up, this section provides evidence that it is possible to extend SPARQL syntax and semantics to support continuous queries. Table 8.2 illustrates the differences among my proposal (i.e., C-SPARQL) and alternative approaches available in the current state-of-the-art.

Table 8.2: A wrap up of the alternative proposals for extending SPARQL syntax and semantics to support continuous queries. The first row illustrate my one.

<table>
<thead>
<tr>
<th>What</th>
<th>Window operators</th>
<th>Boolean operators</th>
<th>Time-aware operators</th>
<th>Relation-to-stream operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-SPARQL</td>
<td>✓</td>
<td>✓</td>
<td>SPARQL 1.1</td>
<td>timestamp function only</td>
</tr>
<tr>
<td>\texttt{SPARQL}_{\texttt{stream}}</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>Rstream only</td>
</tr>
<tr>
<td>CQELS language</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>Istream only</td>
</tr>
<tr>
<td>EP-SPARQL</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
8. CONCLUSIONS

As for RDF streams, the lack of a common syntax and semantics is becoming a major bottle neck for adoptions and comparative research. The undergoing discussion in W3C RSP group may lead to a community proposal for an RDF Stream Processing language (namely RSP-QL) that encompasses features of C-SPARQL, SPARQL Stream [Calbimonte et al., 2010], and CQELS language. At this stage of the discussion, it is unclear whether it is advisable to also include EP-SPARQL features in such a language. Further user-centric investigation may be required to understand what is an adequate language for RDF Stream Processing.

8.1.3 Extending SPARQL Syntax and Semantics to Support Continuous Reasoning Tasks

The positive answer to this part of sub-question SQ.1 is straight forward: when information flow at high rates and the knowledge captured in OWL ontologies evolves slowly, C-SPARQL under OWL entailment regime [Glimm et al., 2013] can supports the encoding of continuous reasoning tasks without requiring to change the OWL semantics. The successful deployment of stream reasoning techniques for social media analytics [j.4, 2010;w.5, 2011;j.5 2012;j.7, 2014] supports such positive answer. The rest of the section wraps up and exemplify the content of [c.2, 2010;o.1, 2014].

Consider, for instance, a social media analytics query that monitors every 10 minutes the last 40 minutes of discussions on a social media stream to determine the consensus generated by the users posts. Let us assume that the monitored social media RDF stream contains the following triples:

```
:alice :posts "If I had a world of my own, everything would be nonsense." . [10:00]
:bob :agreesWith :alice . [10:00]
:carol :agreesWith :bob . [10:10]
:david :agreesWith :alice . [10:20]
:carol :agreesWith :david . [10:30]
```

where the property :agreesWith is declared to be transitive (i.e., :agreesWith ref:type owl:TransitiveProperty).

Listing 8.4 shows the C-SPARQL query that, under OWL2RL entailment regime (see Section 4 of [Calvanese et al., 2009]), continuously computes the consensus on such a stream.

260
8.1 Extending the Semantic Web towards Stream Reasoning

```sql
REGISTER QUERY Consensus AS
SELECT ?person ?content (COUNT(?anotherPerson) AS ?consensus )
FROM STREAM <http://ex.org/discussions> [RANGE 60m STEP 10m]
WHERE {
  FILTER(?anotherPerson != ?person)
} GROUP BY ?person ?content
```

**Listing 8.4:** An example of C-SPARQL that under OWL2RL entailment regime continuously computes the consensus around the opinions expressed by social media users. Note: agreesWith is a transitive property.

Line 3 tells the engine to use a window of 40 minutes sliding every 10 minutes on such an RDF stream. Line 5 matches the content of the microposts of the people active on the social network. Line 6 looks for people that agrees with those that made the posts. Given that :agreesWith is transitive, a person can support another one also indirectly, e.g., Carol can support Alice indirectly via Bob. Line 7 groups the solution mappings by person and content so to count (Line 2) the number of direct and indirect supports to the contents posted by the people.

The results of the query together with the content of the window are illustrated in Figure 8.1. The first column reports the instants at which the query is evaluated. The second and the third columns show the triple that, respectively, enter into and exit from the window. The fourth column depicts the explicit content of the window. The fifth column illustrates the inferred content of the window. Finally, the last three columns on the right report the answers to the query splitting the contribution of the explicit triple from the one of the implicit ones.

The fact that, at least in the chosen setting, it is not necessary to extend OWL semantics, does not mean that existing reasoning algorithms for OWL can provide reactive answers to the users. As illustrated in Figure 8.1, the query in Listing 8.4 requires to retract facts when they exit the window. As known from the database literature, where this problem is known under the name *incremental view maintenance*, decrementing a view is twice as expensive as incrementing it [Ceri & Widom, 1991]. As reported in Section 8.2, while answering sub-question SQ.2, naïve approaches do not adequate performances and new reasoning algorithms are needed.
8. CONCLUSIONS

Figure 8.1: An example of continuous execution of the C-SPARQL query under OWL2RL entailment regime in Listing 8.4.

<table>
<thead>
<tr>
<th>Time</th>
<th>Enter window</th>
<th>Exit window</th>
<th>Explicitly in window</th>
<th>Inferred in window</th>
<th>Consensus</th>
<th># Exp.</th>
<th># Inf.</th>
<th>SUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:00</td>
<td>A←B</td>
<td>A ← B</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10:10</td>
<td>B←C</td>
<td>A ← B ← C</td>
<td>A ← C</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>10:20</td>
<td>A←D</td>
<td>D ← A ← B</td>
<td>A ← C</td>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>10:30</td>
<td>D←C</td>
<td>B ← D ← A</td>
<td>A ← C</td>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>10:40</td>
<td>A←B</td>
<td>D ← A ← B</td>
<td>A ← C</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>10:50</td>
<td>B←C</td>
<td>D ← C ← A</td>
<td>A ← C</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>11:00</td>
<td>A←D</td>
<td>D ← C</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Note: the RDF statement bob agreesWith alice is graphically represented as A←B

The positive answer provided to sub-question SQ.1 is a solid step-stone for future researches. At this stage of my research I can see two possible research questions related to the extension of OWL semantics:

- Is it possible to extend OWL semantics or the semantics of continuous SPARQL queries under OWL entailment regime to consider notions of oblivion more flexible than the one offered by windows?

In the example illustrated in Figure 8.1, triples exiting the window are treated as updates that retract what exits from the window. However, those triples may still be true when they fall out of the window. A part from the practical reason to run in finite memory, why shall the reasoner forget them?

- Is it possible to extend OWL semantics so to perform continuous reasoning tasks when not only the data, but also the knowledge captured in OWL ontologies is rapidly evolving?

Consider, for instance, a social media analytics scenario where inductive reasoning is used to extract an ontology of the topics under discussion. This ontology would
still evolve at a slower pace than the data it is computed from, but it may change fast enough to require to rethink also OWL semantics.

8.1.4 Extending SPARQL Engines to Support Continuous Queries and Continuous Reasoning Tasks

The three previous sections introduce a data model (RDF stream) to represent heterogeneous data streams within the Semantic Web stack and a language (C-SPARQL) to continuously query such a data model under different entailment regimes (including OWL one). To complete the positive answer of SQ.1, this section reports on my implementation experience with the C-SPARQL Engine\(^1\) and the Streaming Linked Data (SLD) framework \cite{SRLD} built around it. I used them to run the experiments reported in \cite{M10,M11,M12,M13,M14} and Marco Balduini and I deployed them in the real-world in socially listening to large events \cite{SRLD} such as London Olympics Games 2012 and the 2013 and 2014 editions of Milano Design Week. Last but not least, I’d like to highlight that in 2015 a spin-off of Politecnico di Milano (namely Fluxedo) was created to commercialise applications of continuous social media analytics built on the SLD framework.

The remainder of the section is organised around two topics: the engines and the middlewares. Section 8.1.4.1 presents the C-SPARQL Engine and discusses alternative architectural approaches considered in other engines. The Section 8.1.4.2 presentes the SLD framework and discusses other middlewares for RDF stream management.

8.1.4.1 Engines

It is worth to note that, as envisioned in \cite{SPARQL}, two ways exists towards the realisation of Stream Reasoners: 1) an evolutionary approach that combines existing solutions; and 2) a revolutionary approach that proposes a new processing paradigm. The evolutionary approach allows for rapid prototyping, but it treats the subsystems, which it relies on, as black-boxes, thus it prevents to fully optimise query processing. On the contrary, the more expensive revolutionary approach permits it.

For the C-SPARQL Engine, Davide F. Barbieri, Stefano Ceri, Daniele Braga and I, chose the evolutionary approach. The architecture, illustrated in Figure 8.2.(a), consists

\(^1\)http://streamreasoning.org/download/csparqlreadytogo
8. CONCLUSIONS

of two subsystems: a DSMS/CEP and a SPARQL Engine that operates under different entailment regimes. The DSMS/CEP performs the windowing operation on the RDF streams. It delivers the content of the window as an RDF graph to the SPARQL Engine which evaluates the SPARQL portion of the registered query and streams out the result. Reasoning capabilities of the chosen SPARQL Engine can also be activated. The C-SPARQL Engine allows to plug-in an EPL\(^1\) compliant DSMS/CEP and any SPARQL Engine. The binaries of the engines uses Esper\(^2\) and Apache Jena-ARQ\(^3\). Different entailment regimes can be activated by changing the file of rules used by the engine.

Figure 8.2 illustrates the architecture of some of the approaches alternative to the C-SPARQL Engine. The evolutionary approach is adopted by Morph-Stream\(^4\) (the engine that interprets SPARQL\(_{\text{stream}}\)) and Streaming Knowledge Bases [Walavalkar et al., 2008], while the revolutionary approach is taken by CQELS\(^5\) (the engine that interprets CQELS language), ETALIS\(^6\) (the engine that interprets EP-SPARQL), Spark-wave\(^7\) [Komazec et al., 2012], the Stream Reasoning via Truth Maintenance Systems presented in [Ren & Pan, 2011] and INSTANS\(^8\) [Rinne et al., 2012].

Morph-Stream implements a typical OBDA architecture (Figure 8.2.(b)). It rewrites continuous SPARQL\(_{\text{stream}}\) queries (expressed on top of virtual RDF streams) down into continuous queries expressed on top of relational data streams. It extends the R2RML\(^9\) language to map relational DB to RDF to handle the characteristics of continuous query languages. Experiments were run using the DSMS TinyDB [Madden et al., 2005], the CEP Esper, and the sensor networks middleware GSN [Aberer et al., 2007].

The architectural choice of ETALIS and CQELS (see Figures 8.2.(c) and8.2.(d)), instead, allows them to handle continuous query processing in a uniform framework.

\(^1\)The Event Processing Language (EPL) is a joint proposal of EsperTech Inc. and Oracle Corporation. The documentation of EPL is available online on EsperTech Inc. website (http://esper.codehaus.org/esper-4.2.0/doc/reference/en/html/epl_clauses.html) and on Oracle Corporation website (http://docs.oracle.com/cd/E13157_01/wlevs/docs30/epl_guide/)

\(^2\)http://esper.codehaus.org/

\(^3\)http://jena.apache.org/documentation/query/

\(^4\)https://github.com/jpcik/morph-streams

\(^5\)https://code.google.com/p/cqels/

\(^6\)https://code.google.com/p/etalis/

\(^7\)http://sparkwave.sti2.at/gettingstarted.html

\(^8\)https://github.com/aaltodsg/instans

\(^9\)http://www.w3.org/TR/r2rml/
8.1 Extending the Semantic Web towards Stream Reasoning

<table>
<thead>
<tr>
<th>Evolutionary Architectural Approach</th>
<th>Revolutionary Architectural Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) C-SPARQL Engine</td>
<td>(c) CQELS</td>
</tr>
<tr>
<td>![C-SPARQL Engine Diagram]</td>
<td>![CQELS-QL Engine Diagram]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(b) Morph-Stream</th>
<th>(d) ETALIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Morph-Stream Diagram]</td>
<td>![ETALIS Engine Diagram]</td>
</tr>
</tbody>
</table>

**Legenda:**
- □□□□□ RDF stream
- □□□□□ relational data stream

**Figure 8.2:** A comparison of the architecture of the C-SPARQL engine, the SPARQL\textsubscript{stream} engine, the CQELS engine and ETALIS.
Table 8.3: A wrap up of the alternative proposals for extending SPARQL Engine to support continuous queries and continuous reasoning tasks. The first row illustrates my one.

<table>
<thead>
<tr>
<th>What</th>
<th>Supported Language</th>
<th>Supported Entailment Regime</th>
<th>Supports Temporal Operators</th>
<th>Architectural Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-SPARQL Engine</td>
<td>C-SPARQL</td>
<td>OWL2RL subset</td>
<td>Timestamp function only</td>
<td>Evolutionary</td>
</tr>
<tr>
<td>Morph-Stream Streaming Knowledge Bases</td>
<td>SPARQL_stream</td>
<td>×</td>
<td>×</td>
<td>Evolutionary</td>
</tr>
<tr>
<td>CQELS</td>
<td>CQELS</td>
<td>×</td>
<td>×</td>
<td>Revolutionary</td>
</tr>
<tr>
<td>ETALIS</td>
<td>EP-SPARQL</td>
<td>SubClassOf only</td>
<td>✓</td>
<td>Revolutionary</td>
</tr>
<tr>
<td>INSTANS</td>
<td>SPARQL</td>
<td>×</td>
<td>✓</td>
<td>Revolutionary</td>
</tr>
<tr>
<td>Sparkwave [Ren &amp; Pan, 2011]</td>
<td>C-SPARQL</td>
<td>RDFS subset</td>
<td>×</td>
<td>Revolutionary</td>
</tr>
<tr>
<td>[Ren &amp; Pan, 2011]</td>
<td>SPARQL</td>
<td>OWL2DL</td>
<td>×</td>
<td>Revolutionary</td>
</tr>
</tbody>
</table>

ETALIS grounds in Logic Programming the basic mechanism for Event Processing and Stream Reasoning, whereas CQELS implements the windowing operators natively and provides a dynamically adaptable query execution framework where the query processor maximises the input throughput by continuously reordering operators in the query execution plan.

Wrapping up, this section provides evidence that it is possible to extend SPARQL engines to support continuous queries and continuous reasoning tasks. Table 8.3 illustrates the differences among my proposals and alternative approaches available in the current state-of-the-art.

The main issue is the limited possibility to run comparative evaluations of those engines. As noted in [Le-Phuoc et al., 2012] different engines return different results for the same queries on the same RDF streams. Obtaining the same behaviour from the different RDF stream processing engines is, indeed, difficult and, in some cases, even
8.1 Extending the Semantic Web towards Stream Reasoning

impossible (see [w.7, 2013] and [c.4, 2013]) because the operational semantics of those engines are different. As discovered in the DSMS community, knowing which data is input in a DSMS engine and the semantics of the continuous query language is not sufficient to tell the correct answer of a DSMS [Botan et al., 2010]. The formal operational semantics of the engine is also needed. My contribution to solve this issue is a model that describes a complete semantics for a RDF stream processing query language. The initial sketch of this model presented in [c.4, 2013] makes explicit a number of hidden parameters that cannot be controlled from the continuous query languages, but are hard coded in the engines. Thanks to this model it was possible to extend SRbench [Zhang et al., 2012] with an oracle that checks the correctness of the results streamed out by the engines. Further investigations are required to establish a shared benchmark in the stream reasoning community. Some initial discussion are undergoing in the W3C RDF Stream Processing community group.

8.1.4.2 Middlewares

In order to ease the task of deploying the C-SPARQL Engine in real-world applications, Marco Balduini and I, designed and developed the SLD Framework. As explained in Figure 8.3, the SLD framework offers: a set of adapters that transcode data streams in RDF (e.g., a stream of micro-posts as an RDF stream using the SIOC vocabulary [Breslin et al., 2006], or a stream of weather sensor observation using the Semantic Sensor Network vocabulary [Compton et al., 2012]), a publish/subscribe bus to internally transmit RDF streams, facilities to record and replay RDF streams, an extendable component to decorate an RDF stream (e.g., adding sentiment annotations to micro-posts), a wrapper for the C-SPARQL Engine that allows to create networks of C-SPARQL queries, and a linked data server to publish results following the Streaming Linked Data Format [w.2, 2011].

![Figure 8.3: The architecture of the Streaming Linked Data framework.](image-url)
8. CONCLUSIONS

Two alternative approaches to SLD framework are documented in the state-of-the-art: the Linked Stream Middleware [Le Phuoc et al., 2012] and a semantically enabled service architecture for mashups over streaming and stored data [Gray et al., 2011].

The three approaches fulfil similar requirements for the end user. They offer extensible means for real-time data collection, for publishing and querying collected information as Linked Data, and for visualising data and query results. They differ in the approach. The SLD framework and the Linked Stream Middleware take both a data driven approach, but they address in a different way the non-functional requirements; while the SLD framework is an in-memory solution for stream processing of RDF streams with limited support for static information, the Linked Stream Middleware is a cloud-based infrastructure to integrate time-dependent data with other Linked Data sources. The middleware described in [Gray et al., 2011], instead, takes a service oriented approach, thus it also includes service discovery and service composition among its features.

The future real-world deployments of Stream Reasoning solutions will foster the appearance of middlewares of this kind. Future research in this direction shall include comparative evaluation both on the technical side (e.g., comparing throughput, memory usage, scalability, etc.) and on the user side (e.g., analysing adequacy of the query language, of the visualisations, etc.).

8.2 Optimising Stream Reasoning algorithms to provide reactive answers

Chapter 1 also introduced the sub-question:

SQ.2 Is it possible to optimise continuous querying and continuous reasoning tasks so to provide reactive answers to large number of concurrent users?

This section positively answers such a question. Section 8.2.1 presented the intuition I had with Heiner Stuckenschmidt, Stefano Ceri, and Frank van Harmelen, about the possibility to cascade reasoning techniques [w.1, 2010] so to tame the trade-off between the complexity of the reasoning method and the frequency of the data stream the reasoner has to handle. Section 8.2.2 and Section 8.2.3 show how to positively answer
8.2 Optimising Stream Reasoning algorithms to provide reactive answers

the SQ.2 by exploiting the ordered nature of data streams and the possibility to forget old enough information.

8.2.1 The intuition

A fundamental problem of stream reasoning is the fact that many relevant reasoning methods, e.g. for description logics, are not able to deal with high frequency data streams. While they try to derive entailments of the goal predicate, newly incoming data will pile up. However, a trade-off exists between the complexity of the reasoning method and the frequency of the data stream the reasoner is able to handle.

The intuition [w.1] to solve this problem is straightforward. It stems from the observation of a similar trade-off between memory size and access time in computer systems, which is solved using a memory hierarchy. Stream Reasoning can be optimised to provide reactive answers by using a hierarchy of processing steps of increasing complexity. Figure 8.4 illustrates this idea of cascading stream reasoners for processing streaming data. Technically, this intuition is supported by the possibility to push down processing steps in the hierarchy to speed up reasoning and the possibility to complete the reasoning process at each layer by only processing the results coming up from the layer underneath. More specifically, it has been shown that description logic reasoning to some extent can be reduced to rule-based reasoning [Grosos et al., 2003] and that rule-based reasoning, in turn, can be reduced to query processing under certain conditions [Calvanese et al., 2007]. It has also been demonstrated that the part of rule-based reasoning that cannot be reduced to query processing can still be performed on the results of such processing [Stoilos & Grau, 2011] and it can be applied in real-world challenging scenarios [Stoilos, 2014].

The lower levels are designed to cope with the volume and the velocity of streaming data. Those layers play two roles: they wrap the raw data stream into a virtual RDF stream data model and they provide the possibility to query those virtual RDF streams using a continuous query language such as C-SPARQL under OWL2QL entailment regime [Calbimonte et al., 2010] applying the OBDA methods. The efficiency of the RDF stream processing technique presented in Section 8.2.2 guarantees for the possibility to realise those lower layers. Only those parts of the raw stream, which match the registered queries, are passed on to the higher levels, at which they arrive
8. CONCLUSIONS

Figure 8.4: The intuition of cascading reasoners to tame the trade-off between the complexity of the reasoning method and the frequency of the data stream the reasoner is able to handle.

with a lower frequency. On the next higher level, relatively simple but efficient reasoning methods, e.g., OWL2RL based reasoning, can be used to further process the result stream. The incremental reasoning technique illustrated in Section 8.2.3 can be employed at this level to guarantee efficiency. Only at the top of the hierarchy where the frequency of change has been reduced significantly, we can expect to be able to use expressive reasoners, e.g. for description logics [Ren & Pan, 2011] or spatio-temporal reasoning [Anicic et al., 2011a,b]. Following this intuition, only inferences that cannot be carried out on the lower layers of the hierarchy are actually carried out using more expressive reasoning methods.

8.2.2 Optimising RDF stream processing to provide reactive answers

A number of experiments were performed using my C-SPARQL Engine under simple RDF entailment regime to verify if the ordered nature of data streams and the possibility to forget old enough information allow to optimise continuous querying.

A first result, which confirms similar ones for DSMS [Krämer & Seeger, 2009], is that C-SPARQL window-based selection under simple RDF entailment regimes outperforms the SPARQL FILTER-based selection. This result, documented in [w.3, 2009] for the C-SPARQL Engine, is illustrated in Figure 8.5. The red triangles and the blue diamonds show the time required to evaluate a query (similar to the one in Listing 8.1) that uses window-based selection (respectively when triples arrive at a rate of 5 per second and 200 per second), while the green squares show the time required to evaluate
8.2 Optimising Stream Reasoning algorithms to provide reactive answers

![Response Time C-SPARQL vs. SPARQL](image)

Figure 8.5: The window-based selection of C-SPARQL Engine outperforms the standard FILTER-based selection of the jena SPARQL Engine under simple RDF entailment regimes.

the equivalent query (shown in Listing 8.5) that uses a SPARQL FILTER clause (see Lines 6-9) to perform the same selection. The window-based selection outperforms the FILTER-based one by an order of magnitude.

```sparql
CONSTRUCT { ?opinionMaker sd:about ?topic }
WHERE {
  FILTER ( ?opinionMakerTime > "2009-07-20T22:17:00Z" xsd:dateTime &&
            ?opinionMakerTime < "2009-07-20T22:47:00Z" xsd:dateTime &&
            ?followerTime > "2009-07-20T22:17:00Z" xsd:dateTime &&
            ?followerTime < "2009-07-20T22:47:00Z" xsd:dateTime )
  FILTER ( ?followerTime > ?opinionMakerTime )
} HAVING ( COUNT(DISTINCT ?follower) > 3 )
```

Listing 8.5: A SPARQL query equivalent to the C-SPARQL one in Listing 8.1
8. CONCLUSIONS

Given that the FILTER-based results are better fitted by a power function, while the window-based are better fitted by linear functions, a break-even point exists after which the window-based selection performs worst than the FILTER-base one. In this experiment, it is around 360,000 triples in the window when the triples arrive at a rate of 5 per second (i.e., a tumbling window 20 hours wide, which makes no sense in the stream processing setting), and it is around 890,000 triples when they arrive at a rate of 200 per seconds (i.e., a tumbling window 1 hour and 14 minutes wide, which also makes little sense in our setting).

Intuitively the result can be understood comparing the complexity of inserting, deleting and accessing triples in a typical RDF store and the one used in the C-SPARQL Engine and in the SLD framework. A typical RDF store uses a binary tree with linked leaves to index triples and to implement fast range queries (as the one we are analysing). This data structure requires $O(\log(n))$ operations to insert, delete and access a triple, where $n$ is the number of triples, and $O(m)$ operations to extract all the $m$ elements in the range. In the C-SPARQL engine, which delegates the task to the underlying DSMS, and in the SLD framework, where windows are also implemented natively, the incoming triples are kept in a linked list of buckets of triples with the same timestamp. This data structure requires $O(1)$ operations to add bucket to the beginning of the list or to delete one from the end of the list, and $O(p)$ operations to get the $p$ buckets in the linked list. Also note that each bucket may contains multiple triples with the same time-stamp.

When, as in the experiment, $n = m$ and $p = m$, an RDF store spends an amount of time, which increases with the size $n$ of the window, in inserting and deleting triples from the binary tree, while the C-SPARQL Engine and the SLD framework spend a fix time (which is order of magnitude smaller than the time to index a triple even when the binary tree is almost empty) in those operations. The two approaches spend proximally the same time to access the data, but the C-SPARQL Engine and the SLD framework still have to dump the content of the linked list to get the rest of the SPARQL query answered. This explains why the break even point exists. It is also straight forward to see that the cost of dumping the triples in the buckets is lower than the one to perform a range query when $p \leq m$. The larger are the buckets, the cheaper it is.

A complementary way to provide positive evidence to answer SQ.2 is to measure the input throughput, i.e., the ability to consume triples as inputs. This measure is
8.2 Optimising Stream Reasoning algorithms to provide reactive answers

traditionally used in publish-subscribe systems [Fabret et al., 2001] and it is computed as follows:

\[
\text{input throughput} = \frac{\text{size input}}{\text{time to process the input}}
\]

[j.5, 2011;j.7, 2014] report on measuring the input throughput of the SLD framework, which wraps the C-SPARQL Engine adding the possibility to create networks of queries, by sending to it a recorded portion of an RDF stream containing tweets and by measuring the time required to process it, i.e., by computing, for this portion of RDF stream, all the answers to all the C-SPARQL queries in the network. To improve confidence, each experiment was repeated for 30 minutes and the average, the minimum and the maximum time required to process the portion of the RDF stream were measured. The experiments were conducted on a laptop with CPU 2.2 GHz and 4 GB RAM, which corresponds to a 80 €/month share in a cloud environment. The results are plotted in Figure 8.6, taken from [j.7, 2014]. The maximum throughput achieved is 700 tweets per second that roughly corresponds to 10,000 triples per second. It is worth to note that similar results are available for other RDF stream processing engines [Le-Phuoc et al., 2012].

![Figure 8.6: Throughput results of my C-SPARQL Engine and my Streaming Linked Data framework (source [j.7, 2014]).](image)

As already discusses in Sections 8.1.4 the main issue, at the current state of development of the field, is the limited possibility to run comparative evaluations of those
8. CONCLUSIONS

engines due to the heterogeneity in execution semantics of existing RDF stream processing engines [w.8, 2013; c.5, 2013]. Further investigations are required to establish a fair and comprehensive RDF stream processing benchmark [c.4, 2013].

8.2.3 Optimising reasoning algorithms to provide reactive answers

As explained in Section 8.2.1, a fundamental problem of stream reasoning is the fact that many relevant reasoning methods, e.g. for description logics, are not able to deal with high frequency data streams. In the previous section, I provided positive evidence for the ability to build the lower layers of the cascading stream reasoner illustrated in Figure 8.4. In this section, I do the same focusing on the layers above, the one that uses OWL2RL reasoning methods.

As already illustrated in Section 8.1.3 this kind of reasoning is hard in presence of deletions, because decrementing a materialised view in databases is twice as expensive as incrementing it [Ceri & Widom, 1991]. The state of the art algorithm to perform this task is DRed [Gupta et al., 1993] (whose theoretical foundation can be found in [Ceri & Widom, 1991]) which incrementally maintains a view (or an ontological materialisation in the case of OWL2RL reasoning) in three steps:

1. Overestimation of deletion: this step overestimates deletions by computing all direct consequences of a deletion. Consider, for instance, the graph illustrated below (taken from the row marked with 10.30 in Figure 8.1) where a triple like :bob :agreesWith :alice is represented as A ← B and agreesWith is a transitive property. If the triple marked with ✗ is deleted, then also the entailed triple A ← C is candidate for deletion.

   ![Graph Illustration](image)

2. Rederivation: this step prunes those triples candidate for deletion for which at least an alternative derivation exists. For instance, the triple A ← C can be rederived along the path A ← D ← C marked with ✓.

   ![Graph Illustration](image)

3. Insertion: this step adds the new derivations that are consequences of insertions.
8.2 Optimising Stream Reasoning algorithms to provide reactive answers

It is worth to note that DRed is designed to handle random insertions and deletions, but in a streaming setting, when a triple enters the window, given the size of the window, the reasoner knows already when it will be deleted. Consider the running example in Figure 8.1: given that the window is 40 minutes wide, when, at 10:00, the triple $A \leftarrow B$ enters the window, we know that it will exit on 10:40. Therefore, the deletions in the streaming setting are predictable.

The IMaRS algorithm [c.2, 2010;o.2, 2014], which I design with Davide Francesco Barbieri, Daniele Braga, Stefano Ceri and Michael Grossniklaus, exploits this intuition and proposes an algorithm for optimised incremental maintenance of ontological entailments on RDF streams. IMaRS annotates each triple entering a window or entailed by them with an expiration time. The algorithm consists in two steps:

1. Exact deletions: this step deletes all the triple whose expiration time is equal to now.

2. Insertion: this step adds the new entailments, which are consequences of insertions, annotating each of them with an expiration time (the minimum of those of the triples it is derived from), and when multiple derivations occur, for each of them it keeps the maximum expiration time.

Notably, in step 1, by construction, only the entailments that cannot be rederived are deleted.

Figure 8.7 shows how IMaRS incrementally maintains the materialisation required to answer the query in Listing 8.4 when triples are streamed as in the example illustrated in Figure 8.1.

At 10.00, when $A \leftarrow B$ enters in the window, it is annotated with the expiration time 10:40. When, at 10.10, the triple $B \leftarrow C$ enters the window, it is annotated with expiration time 10:50 and the entailed triple $A \leftarrow C$ is annotated with 10:40, i.e., the minimum expiration time of the two triples that contribute to its derivation. At 10:20 IMaRS only annotates the expiration time of $A \leftarrow D$ with 11:00. When, at 10:30 the triple $D \leftarrow C$ enters the window, the entailed triple $A \leftarrow C$ is inferred with a longer lasting expiration time, and thus the expiration time of $A \leftarrow C$ is changed from 10:40 to 11.00. So far IMaRS behaved as DRed, because no deletions have occurred, yet. When, at 10:40, $A \leftarrow B$ exists the window, IMaRS performs no action, while DRed would
8. CONCLUSIONS

Figure 8.7: How my IMaRS algorithm incrementally maintains the materialisation required to answer the query in Listing 8.4. Differently from the notation used in Figure 8.1, each triple is annotated with an expiration time. The syntax \( A \leftarrow^{10:40} B \) means that the triple \( A \leftarrow B \) expires at 10:40.

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{Time} & \text{Enter window} & \text{Exit window} & \text{Explicitly in window} & \text{Inferred in window} & \text{Consensus} \\
\hline
10:00 & A \leftarrow B & 10:40 & A \leftarrow B & & \\
10:10 & B \leftarrow C & 10:40 & A \leftarrow B & 10:50 & A \leftarrow B \\
10:20 & A \leftarrow D & 11:00 & D \leftarrow 10:40 & 10:50 & A \leftarrow B \\
10:30 & D \leftarrow C & 11:00 & D \leftarrow 10:40 & 10:50 & A \leftarrow B \\
10:40 & A \leftarrow B & 11:00 & A \leftarrow B & 11:00 & A \leftarrow B \\
10:50 & B \leftarrow C & 11:00 & A \leftarrow B & 11:00 & A \leftarrow B \\
11:00 & A \leftarrow D & 11:00 & A \leftarrow B & 11:00 & A \leftarrow B \\
\hline
\end{array}
\]

Note: the RDF statement \( \text{bob agreesWith alice} \) is graphically represented as \( A \leftarrow B \)

have candidate \( A \leftarrow C \) for deletion and, then, it would have discovered that it can be rederived via \( A \leftarrow D \leftarrow C \). Also when at 10:50 and at 11:00 \( B \leftarrow C \) and \( A \leftarrow D \) exit the window, IMaRS simply deletes the triples whose expiration time is 10:50, while DRed would have run two inference steps (i.e., overestimate deletion and rederive).

Figure 8.8, taken from [o.2, 2014], compares IMaRS with DRed and the naïve approach of rematerialising the whole content of the window each time it slides. The figure plots the time required to maintain the materialisation as a function of the percentage of deletions w.r.t the content of the window. As one can expect the naïve methods takes a time independent from the percentage of deletion. As documented in literature [Gupta et al., 1993], DRed outperforms the naïve methods of one order of magnitude for small percentages of deletions, but there is always a break-even point after which incremental maintenance takes longer than the naïve rematerialisation. In

\[1\]The incremental view maintenance, in the database setting, expects large insertion and few deletions, which are always a very small percentage w.r.t. the size of the entire database.
8.2 Optimising Stream Reasoning algorithms to provide reactive answers

the specific experimental setting of [c.3, 2013;o.2, 2014], the break-even point of DRed is around 3%. IMaRS is two order of magnitude better than naïve rematerialisation and one order of magnitude better of DRed for small percentages. It keeps being two order of magnitude better than naïve rematerialisation and becomes two order of magnitude better than DRed, when the percentage of deletions w.r.t. to the window size grows to 1%. It remains two order of magnitude better up to 5% and it reaches the break-even point around 15%.

Figure 8.8: A comparison of the time required by IMaRS, DRed and naïve rematerialisation to compute a new materialisation as a function of the percentage of deletion w.r.t the content of the window.

This provides experimental evidence that OWL2RL reasoning algorithms can be optimised to cope with high changing rate data, but it does not prove that the entire machinery is able to provide reactive answer. Figure 8.9 supplies such an evidence. It considers the experimental setting of Figure 8.8 and it compares the average time needed to answer the C-SPARQL query in Listing 8.4, when 2% of the content exits the window each time it slides, using: a) the backward reasoner offered by Jena on the window content, b) the DRed algorithm implemented using Jena rule engine and its SPARQL engine (namely DRed+SPARQL), and c) IMaRS algorithm implemented using Jena rule engine and its SPARQL engine (namely, IMaRS+SPARQL). As one can expect the backward reasoner outperforms the DRed+SPARQL, but IMaRS is so fast in incrementally maintaining the materialisation to perform even better than the backward reasoner.

Approaches alternative to IMaRS are ETALIS [Anicic et al., 2011a,b], Sparkwave [Komazec et al., 2012], Streaming Knowledge Bases [Walavalkar et al., 2008] and Stream Reasoning via Truth Maintenance Systems [Ren & Pan, 2011]. In the chosen experimental settings, they are all order of magnitude better than state of the art, but no
8. CONCLUSIONS

**Figure 8.9:** A comparison of the time required by IMaRS+SPARQL, DRed+SPARQL and Backward Reasoner (all implemented in Jena) to answer to the query in Listing 8.4, when 2% of the content exits the window each time it slides

comparative evaluation among them has been attempted so far.

ETALIS is a Complex Event Processing system that grounds event processing and stream reasoning in Logic Programming. It is based on event-driven backward chaining rules that realise event-driven inferencing as well as RDFS reasoning. IMaRS and ETALIS are largely incomparable. ETALIS focuses on back-ward temporal reasoning over RDFS, while IMaRS focuses on forward reasoning on OWL2RL. The temporal reasoning is peculiar of ETALIS and it is not present in IMaRS. This restricts the comparison to the continuous query answering task only. The evaluation of IMaRS shows that, in the chosen experimental setting (see Figure 8.9), the continuous query answering task over a materialisation maintained by IMaRS is faster than backward reasoning. However, further investigation is needed to comparatively evaluate the two approaches.

Sparkwave [Komazec et al., 2012] is a solution to perform continuous pattern matching over RDF data streams under RDFS entailment regime. It allows to express temporal constraints in the form of time windows while taking into account RDF schema entailments. Sparkwave adds to the Rete algorithm [Forgy, 1982] an additional memory structure, which computes RDFS entailments, and time-based window support. Sparkwave is very similar to IMaRS on a conceptual level. It offers an efficient implementation of the IMaRS’s maintenance program for RDFS. However, the approach proposed by
Sparkwave cannot be extended to OWL2RL (i.e., the ontological language targeted by IMaRS), because RDFS can be encoded as rules that are activated by a single triple from the stream, whereas OWL2RL can be encoded as a rule that may be activated by multiple triples from the stream (e.g., the rule that treats `owl:transitiveProperty`). Future investigation should comparatively evaluate IMaRS and Sparkwave w.r.t. RDFS entailment regime.

Streaming Knowledge Bases [Walavalkar et al., 2008] is one of the earliest stream reasoners. It uses TelegraphCQ [Chandrasekaran et al., 2003] to efficiently handle data stream, and the Jena rule engine to incrementally materialise the knowledge base. The architecture of Streaming Knowledge Bases is similar to the one of the C-SPARQL Engine. It supports RDFS and the `owl:inverseOf` construct (i.e., only rules that are activated by a single triple from the stream), therefore the discussion reported above for Sparkwave also applies to Streaming Knowledge Bases. Unfortunately, the prototype has never been made available.

IMaRS and all the works above trade expressiveness for performance. They use light-weight ontological languages and time-based windows to optimise for high throughputs. The authors of [Ren & Pan, 2011] take a different perspective; they investigate the possibility to optimise Truth Maintenance Systems so to perform expressive incremental reasoning when the knowledge base is subject to a large amount of random changes (both updates and deletes). They optimise their approach to reason with \( \mathcal{EL}++ \), the logic underpinning OWL 2 EL, and provide experimental evidence that their approach outperform re-materialisation up to 10\% of changes.

### 8.3 Coping with the noisy and incomplete nature of data streams

Having positively answered the two sub-question SQ.1 and SQ.2, this section addresses the third one introduced in Chapter 1:

SQ.3 Is it possible to cope with the noisy and incomplete nature of data streams?

This section positively answers such a question based on: 1) the assumption that known *noise reduction* techniques elaborate for DSMS systems (e.g., [Subramaniam et
8. CONCLUSIONS

...al., 2006]) can be easily used in the C-SPARQL Engine thanks to its plug-able architecture; and 2) the results obtained in [j.4, 2010] and used in [c.3, 2013;j.5, 2011;j.7, 2014]. Those results address the noisy introduced by Natural Language Processing and the incomplete nature of social media streams by combining RDF streams and Continuous-SPARQL with Machine Learning technologies, in particular relational learning ones [Getoor & Taskar, 2007].

Figure 8.10: Architecture of the Deductive and Inductive Stream Reasoner that I proposed in [j.4, 2010]

Figure 8.10, taken from [j.4, 2010], illustrates both the points. As proposed in the cascading stream reasoning conceptual architecture (see Section 8.2.1), raw data streams are first processed by a DSMS that can apply known techniques for noise reduction in the data streams. For instance, in a work on modelling Big Data Analytics applications [Ceri et al., 2013], Themis Palpanas and I applied outlier detection [Subramaniam et al., 2006] on streaming sensor observation before processing them with C-SPARQL. Cleansed data are then processed as virtual RDF streams and fed into the deductive reasoner. This reasoner copes with the part of incompleteness in the data stream that can be repaired using a deductive reasoner together with a domain ontology. When the result of the deductive reasoner can be modelled as a relation (e.g., likes) between two types of resources (e.g., person and topic), an inductive reasoner (in my experiments it is SUNS [Huang et al., 2010]) all missing values in the matrix are inductively materialised and can be interpreted as the probability of the missing fact to be true. This technique copes with incompletenesses in the data that cannot be repaired with deductive approaches and it is robust to noise.
8.4 Conclusions

It is also worth to note that in [j.4, 2010] I propose to apply the principle of time window to inductive reasoning by abstracting the results of queries registered on the deductive reasoner as matrix with different time-spans. Those matrices capture the same type of relation over two different time windows. In the case of the system shown in Figure 8.10 one matrix captures a long lasting time window (i.e., months), while the other one captures the a short time window (i.e., a week). Given that we are applying inductive materialisation to data streams, the information inductively materialised in the long-term matrix is a correct prediction if the dynamic system observed through the data stream is stable, whereas the one materialised in the short-term matrix is a correct prediction of hype effects. By comparing the two inductive materialisations, our system can understand if the likelihood of a given relation (e.g., Alice, who liked Wonderland, may also like the Middle-Earth) is stable, it is increasing or it is decreasing. In [j.4, 2010], we shown that best top-k predictions are obtained aggregating the best predictions of the two matrices.

The inductive and deductive stream reasoning framework for social media analytics presented in [j.4, 2010] was first shown to be effective on Glue social network\(^1\) and, then, thanks to the cooperation with Saltlux, on Twitter. The result of this joint effort is BOTTARI [j.5, 2011; j.7, 2014]; the winner of Semantic Web Challenge 2011.

Existing works in combining machine learning and semantic web address the problem of learning ontological classes of data by mining data instances. They can be useful in the RDF stream context, but do not attack the problem of predicting links between resources received via noisy and incomplete data streams. To the best of my knowledge, the only work comparable to [j.4, 2010] is presented in [Lécué & Pan, 2013], where the authors investigate the detection of statistical correlations in a stream of time-varying ontologies and on their future projections. The incomplete and noisy nature of data streams calls for further investigation in this field.

8.4 Conclusions

The research question that guided the investigations presented in this thesis is: *is it possible to make sense in real time of multiple, heterogeneous, gigantic and inevitably\(^1\)
8. CONCLUSIONS

noisy and incomplete data streams in order to support the decision process of extremely large numbers of concurrent user?

Such research question was inspired by the growing number of application domains where real-time inference on rapidly changing information was required. Nowadays, The emergence of Big Data, an in particular its velocity and variety dimensions, calls even more for investigating and engineering Stream Reasoning.

Summary. The collection of papers, which made up this thesis, answered this research question by showing that:

1. it is possible to (syntactically and semantically) extend the Semantic Web stack in order to represent heterogenous data streams, continuous queries, and continuous reasoning tasks (Chapter 3 wrapped up in Section 8.1);

2. it is possible to optimise continuous querying and continuous reasoning tasks so to provide reactive answers to large number of concurrent users (Chapters 4 wrapped up in Section 8.2);

3. it is possible to cope with the noisy and incomplete nature of data streams (Chapter 5 wrapped up in Section 8.3); and

4. it is useful (Chapters 6 and 7 discussed across the sections of this final chapter).

The community has picked up the notion of RDF stream proposed in [c.1, 2008]. The C-SPARQL language [o.1, 2009;j.1, 2009;j.3, 2010] was shown to be adequate to encode useful continuous queries under simple RDF, RDFS and OWL2RL entailment regimes [w.3, 2009;w.4, 2010;w.5, 2011;w.6, 2013;c.3, 2013;j.4, 2010;j.5, 2011;j.7, 2014]. Those results fostered the research of three complementary extension of SPARQL for continuous querying RDF streams in presence of static RDF graphs [Anicic et al., 2011a; Calbimonte et al., 2010; Le-Phuoc et al., 2011]. The IMaRS [c.2, 2010;o.2, 2014] algorithm showed how to take advantage of order and oblivio in performing reasoning tasks. It paved the way for other complementary approaches [Anicic et al., 2011a; Komazec et al., 2012; Ren & Pan, 2011] that explore temporal reasoning and expressive $\mathcal{EL}++$ reasoning.

Limitation. While this thesis provides experimental evidence to support [j.1, 2009] claims on Stream Reasoning, existing approaches have some important limitations that
require future work and that make Stream Reasoning an open field of research. First of all, the streams are parallel and distributed in nature, so far only [Le Phuoc et al., 2013] has reported on successful investigation on distributed and parallel RDF Stream Processing, while on parallel Stream Reasoning some work in progress was reported in [Hoeksema & Kotoulas, 2011] in 2011 and in [Liu et al., 2014] in 2014. The sub-question SQ.3 that Stream Reasoning can deal with noisy and incomplete data has been positively answered in the domain of Social Media Analytics coupling deductive stream reasoning with relational learning [j.4, 2010;j.5, 2011;j.7, 2014], while for the sensor network domain, which is more noisy, known noise reduction techniques has been used to preprocess data stream treated as virtual RDF stream processing. However, the incomplete and noisy nature of data streams calls for further investigation. The work on deductive stream reasoning is also in an intermediate state. Example of Stream Reasoning approaches were proposed, but they are fragmented: some focus on temporal reasoning, some on rule base reasoning, some on expressive Description Logics. A unified Stream Reasoning approach is highly desirable at this stage. Most likely such a unified framework would require also to elaborate a comprehensive theory of Stream Reasoning.

**Conclusions.** Looking back at the seven year of papers collected in this thesis, I believe that Stream Reasoning is indeed possible. Stream Reasoning is no longer something that needs to be proved, but rather something that needs to be improved. I believe that the long term success of Stream Reasoning requires a framework for comparative evaluation. The community needs a comprehensive and widely accepted benchmark that can be used to provide concrete evidence that Stream Reasoning is the best solution in some domain. It is of paramount importance to include in the comparison also state-of-the-art solutions like DSMS and CEP. Last but not least, data streams are only an example of ordered dataset. Now that approaches exist [c.2, 2010;o.2, 2014] [Anicic et al., 2011a; Komazec et al., 2012] that exploit the ordered nature of data streams to optimise the reasoning task, it is time to prove that other kinds of orders can be harnessed [j.6, 2013].
8. CONCLUSIONS


