A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data*

Danh Le-Phuoc[1], Minh Dao-Tran[2], Josiane Xavier Parreira[1], and Manfred Hauswirth[1]

[1] Digital Enterprise Research Institute,National University of Ireland, Galway
{danh.lephuoc,josiane.parreira,manfred.hauswirth}@deri.org
[2] Institut für Informationssysteme, Technische Universität Wien
dao@kr.tuwien.ac.at

Abstract. In this paper we address the problem of scalable, native and adaptive query processing over Linked Stream Data integrated with Linked Data. Linked Stream Data consists of data generated by stream sources, e.g., sensors, enriched with semantic descriptions, following the standards proposed for Linked Data. This enables the integration of stream data with Linked Data collections and facilitates a wide range of novel applications. Currently available systems use a "black box" approach which delegates the processing to other engines such as stream/event processing engines and SPARQL query processors by translating to their provided languages. As the experimental results described in this paper show, the need for query translation and data transformation, as well as the lack of full control over the query execution, pose major drawbacks in terms of efficiency. To remedy these drawbacks, we present CQELS (*Continuous Query Evaluation over Linked Streams*), a native and adaptive query processor for unified query processing over Linked Stream Data and Linked Data. In contrast to the existing systems, CQELS uses a "white box" approach and implements the required query operators natively to avoid the overhead and limitations of closed system regimes. CQELS provides a flexible query execution framework with the query processor dynamically adapting to the changes in the input data. During query execution, it continuously reorders operators according to some heuristics to achieve improved query execution in terms of delay and complexity. Moreover, external disk access on large Linked Data collections is reduced with the use of data encoding and caching of intermediate query results. To demonstrate the efficiency of our approach, we present extensive experimental performance evaluations in terms of query execution time, under varied query types, dataset sizes, and number of parallel queries. These results show that CQELS outperforms related approaches by orders of magnitude.

Keywords: Linked Streams, RDF Streams, Linked Data, stream processing, dynamic query planning, query optimisation

1 Introduction

Sensing devices have become ubiquitous. Mobile phones (accelerometer, compass, GPS, camera, etc.), weather observation stations (temperature, humidity, etc.), patient monitoring systems (heart rate, blood pressure, etc.), location tracking systems (GPS, RFID, etc.), buildings management systems (energy consumption, environmental conditions, etc.), and cars (engine monitoring, driver monitoring, etc.) continuously produce information streams.

* This research has been supported by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-II), by the Irish Research Council for Science, Engineering and Technology (IRCSET), by the European Commission under contract number FP7-2007-2-224053 (CONET), by Marie Curie action IRSES under Grant No. 24761 (Net2), and by the Austrian Science Fund (FWF) project P20841.

Also on the Web, services like Twitter, Facebook and blogs, deliver streams of (typically unstructured) real-time data on various topics. The heterogeneous nature of such diverse streams makes their use and integration with other data sources a difficult and labor-intensive task, which currently requires a lot of "hand-crafting."

To address some of the problems, there have been efforts to lift stream data to a semantic level, e.g., by the W3C Semantic Sensor Network Incubator Group[3] and [12, 32, 37]. The goal is to make stream data available according to the Linked Data principles [10] – a concept that is known as *Linked Stream Data* [31]. This would allow an easy and seamless integration, not only among heterogenous sensor data, but also between sensor and Linked Data collections, enabling a new range of "real-time" applications.

However, one distinguishing aspect of streams that the Linked Data principles do not consider is their temporal nature. Usually, Linked Data is considered to change infrequently. Data is first crawled and stored in a centralised repository before further processing. Updates on a dataset are usually limited to a small fraction of the dataset and occur infrequently, or the whole dataset is replaced by a new version entirely. Query processing, as in traditional relational databases, is *pull* based and *one-time*, i.e., the data is read from the disk, the query is executed against it once, and the output is a set of results for that point in time. In contrast, in Linked Stream Data, new data items are produced continuously, the data is often valid only during a time window, and it is continually *pushed* to the query processor. Queries are continuous, i.e., they are registered once and then are evaluated continuously over time against the changing dataset. The results of a continuous query are updated as new data appears. Therefore, current Linked Data query processing engines are not suitable for handling Linked Stream Data. It is interesting to notice that in recent years, there has been work that points out the dynamics of Linked Data collections [35]. Although at a much slower pace compared to streams, it has been observed that centralised approaches will not be suitable if *freshness* of the results is important, i.e., the query results are consistent with the actual "live" data under certain guarantees, and thus an element of "live" query execution will be needed [34]. Though this differs from stream data, some of our findings may also be applicable to this area.

Despite its increasing relevance, there is currently no native query engine that supports unified query processing over Linked Stream and Linked Data inputs. Available systems, such as C-SPARQL [9], SPARQLstream [14] and EP-SPARQL [3], use a "black box" approach which delegates the processing to other engines such as stream/event processing engines and SPARQL query processors by translating to their provided languages. This dependency introduces the overhead of query translation and data transformation. Queries first need to be translated to the language used in the underlying systems. The data also needs to be transformed to feed into the system. For instance, in C-SPARQL and SPARQLstream, the data is stored in relational tables and relational streams before any further processing, and EP-SPARQL uses logic facts. This strategy also does not allow full control over the execution plan nor over the implementation of the query engine's elements. Consequently, the possibilities for query optimisations are very limited.

To remedy these drawbacks, we present CQELS (*Continuous Query Evaluation over Linked Streams*), a native and adaptive query processing engine for querying over unified Linked Stream Data and Linked Data. In contrast to the existing systems, CQELS uses a "white box" approach. It defines its own native processing model, which is implemented in the query engine. CQELS provides a flexible query execution framework with the query processor dynamically adapting to changes in the input data. During query execution, it continuously reorders operators according to some heuristics to achieve improved query execution in terms of delay and complexity. External disk access on large Linked Data collections is reduced with the use of data encoding and caching of intermediate query results, and faster data access is obtained with indexing techniques. To demonstrate the efficiency of our approach, we

[3] `http://www.w3.org/2005/Incubator/ssn/`

present extensive experimental performance evaluations in terms of query execution time, under varied query types, dataset sizes, and number of parallel queries. Results show that CQELS performs consistently well, and in most cases outperforms related approaches by orders of magnitude.

The remainder of this paper is organised as follows: Section 2 discusses our contribution in relation to relational database management systems, data stream management systems, Linked Data processing, and Linked Stream Data processing. Our processing model is described in Section 3, and the query engine is discussed in Section 4. Section 5 presents an experimental evaluation of our approach, and Section 6 provides our conclusions and a brief discussion about ongoing work and next steps.

2 Related Work

RDF stores. A fair amount of work on storage and query processing for Linked Data is available, including Sesame [13], Jena [38], RISC-3X [28], YARS2 [23], and Oracle Database Semantic Technologies [16]. Most of them focus on scalability in dataset size and query complexity. Based on traditional database management systems (DBMSs), they typically assume that data changes infrequently, and efficiency and scalability are achieved by carefully choosing appropriate data storage and indexing optimised for read access, whereas stream data is characterised by high numbers and frequencies of updates. The Berlin SPARQL benchmark[4] shows that the throughput of a typical triple store currently is less than 200 queries per second, while in stream applications continuous queries need to be processed every time there is a new update in the data, which can occur at rates up to 100,000 updates per second. Nevertheless, some of the techniques and design principles of triple stores are still useful for scalable processing of Linked Stream Data, for instance some of the physical data organisations [1, 13, 38] and indexing schemas [16, 23, 28].

Data stream management. Data stream management systems (DSMSs) such as STREAM [4], Aurora [15], and TelegraphCQ [26] were built to overcome limitations of traditional database management systems in supporting streaming applications [20]. The STREAM system proposes CQL [4] (Continuous Query Language) which extends standard SQL syntax with new constructs for temporal semantics and defines a mapping between streams and relations. The query engine consists of three components: operators, that handle the input and output streams, queues, that connect input operators to output operators, and synopses, that store the intermediate states needed by continuous query plans. In the Aurora/Borealis project [15] users can compose stream relationships and construct queries in a graphical representation which is then used as input for the query planner. TelegraphCQ introduces StreaQuel as a language, which follows a different path and tries to isolate temporal semantics from the query language through external definitions in a C-like syntax. TelegraphCQ also uses a technique called *Eddies* [6], which continuously reorders operators in a query plan as it runs, adapting to changes in the input data. DSMSs perform better compared to traditional DBMSs in the context of high volumes of updates. Even though DSMSs can not directly process Linked Stream Data, such processing is still possible by translating the queries and mapping the data to fit into the data storage. This is currently done by available systems that process Linked Stream Data. The CQELS query engine, on the other hand, can directly process Linked Stream Data, yielding consistently better performance, as we will demonstrate later on in the paper.

Streams and semantics. Semantic Streams [37] was among the first systems to propose semantic processing of streams. It uses Prolog-based logic rules to allow users to pose declarative queries over semantic interpretations of sensor data. Semantic System S [12] proposes the use of the Web Ontology Language (OWL) to represent sensor data streams,

[4] http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/

as well as processing elements for composing applications from input data streams. The Semantic Sensor Web project [8, 32] also focuses on interoperability between different sensor sources, as well as providing contextual information about the data. It does so by annotating sensor data with spatial, temporal, and thematic semantic metadata. Research like the one carried by W3C Semantic Sensor Network Incubator Group[5] aims at the integration of stream data with Linked Data sources by following the Linked Data principles for representing the data. In parallel, the concept of *Linked Stream Data* was introduced [31], in which URIs were suggested for identifying sensors and stream data.

In contrast to these approaches, our work focuses on the efficient processing of Linked Stream Data integrated with other Linked Data sources. Existing work with this focus comprises Streaming SPARQL [11], C-SPARQL [9], SPARQLstream [14], and EP-SPARQL [3] as the main approaches. They all extend SPARQL with sliding window operators for RDF stream processing. Streaming SPARQL simply extends SPARQL to support window operators without taking into account performance issues regarding the choice of the data structures and the sharing of computing states for continuous execution. Continuous SPARQL (C-SPARQL) proposes an execution framework built of top of existing stream data management systems and triple stores. These systems are used independently as "black boxes." In C-SPARQL, continuous queries are divided into static and dynamic parts. The framework orchestrator loads bindings of the static parts into relations, and the continuous queries are executed by processing the stream data against these relations. C-SPARQL is not designed for large static data sets, which can degrade the performance of the stream processing considerably.

Along the same lines, SPARQLstream also translates its SPARQLstream language to another relational stream language based on mapping rules. Event Processing SPARQL (EP-SPARQL), a language to describe event processing and stream reasoning, can be translated to ETALIS [3], a Prolog-based complex event processing framework. First, RDF-based data elements are transformed into logic facts, and then EP-SPARQL queries are translated into Prolog rules. In contrast to these systems, CQELS is based on a unified "white box" approach which implements the required query operators for the triple-based data model natively, both for streams and static data. This native approach enables better performance and can dynamically adapt to changes in the input data.

3 Processing Model

The adaptive processing model of CQELS captures all the aspects of both data modelling and query processing over Linked Stream Data and Linked Data in one single theoretical framework. It defines two types of data sources, RDF streams and RDF datasets, and three classes of operators for processing these types of data sources. Operators used in a query are organised in a data flow according to defined query semantics, and the adaptive processing model provides functions to reorder the query operators to create equivalent, more efficient data flows. The details of the processing model are described in the following.

3.1 Definitions

In continuous query processing over dynamic data, the temporal nature of the data is crucial and needs to be captured in the data representation. This applies to both types of data sources, since updates in Linked Data collections are also possible. We define RDF streams to represent Linked Stream Data, and we model Linked Data by generalising the standard definition of RDF datasets to include the temporal aspect.

[5] http://www.w3.org/2005/Incubator/ssn/

Similar to RDF temporal [22], C-SPARQL, and SPARQLstream, we represent temporal aspects of the data as a timestamp label. We use $t \in \mathbb{N}$ to indicate a *logical* timestamp to facilitate ordered logical clocks for local and distributed data sources as done by classic time-synchronisation approaches [24]. The issues of distributed time synchronization and flexible time management are beyond the scope of this paper. We refer the reader to [19, 27, 33] for more details.

Let I, B, and L be *RDF nodes* which are pair-wise disjoint infinite sets of Information Resource Identifiers (IRIs), blank nodes and literals, and $\mathit{IL} = I \cup L$, $\mathit{IB} = I \cup B$ and $\mathit{IBL} = I \cup B \cup L$ be the respective unions. Thereby,

1. A triple $(s, p, o) \in \mathit{IB} \times I \times \mathit{IBL}$ is an *RDF triple*.
2. An *RDF dataset at timestamp* t, denoted by $G(t)$, is a set of *RDF triples* valid at time t. An *RDF dataset* is a sequence $G = [G(t)], t \in \mathbb{N}$, ordered by t. When it holds that $G(t) = G(t+1)$ for all $t \geq 0$, we call G a *static RDF dataset* and denote $G^s = G(t)$.
3. An *RDF stream* S is a bag of elements $\langle (s,p,o) : [t] \rangle$, where (s,p,o) is an *RDF triple* and t is a timestamp. $S^{\leq t}$ denotes the bag of elements in S with timestamps $\leq t$, i.e., $\{\langle (s,p,o) : [t'] \rangle \in S \mid t' \leq t\}$.

Let V be an infinite set of variables disjoint from IBL. A *mapping* is a partial function $\mu \colon V \to \mathit{IBL}$. The domain of μ, $\mathit{dom}(\mu)$, is the subset of V where μ is defined. Two mappings μ_1 and μ_2 are *compatible* if $\forall x \in \mathit{dom}(\mu_1) \cap \mathit{dom}(\mu_2), \mu_1(x) = \mu_2(x)$.

A tuple from $(\mathit{IB} \cup V) \times (I \cup V) \times (\mathit{IBL} \cup V)$ is a *triple pattern*. For a given triple pattern P, the set of variables occurring in P is denoted as $var(P)$ and the triple obtained by replacing elements in $var(P)$ according to μ is denoted as $\mu(P)$. A *graph template* $\mathbb{T}$ is a set of triple patterns.

3.2 Operators

Our processing model takes as input RDF datasets and RDF streams containing possibly infinite numbers of RDF triples, applies a query Q and continuously produces outputs.

In processing Q, snapshots of the input at discrete times t, i.e., finite amounts of data, are used in the evaluation of the query. This requires dedicated operators to (i) take snapshots of the input and filter its valid part w.r.t. some condition, (ii) operate on the finite, intermediate data, and (iii) convert the final results back into a stream. The required operators are called *window*, *relational*, and *streaming operators*.

Window Operators. These operators extract triples from an RDF stream or dataset that match a given triple pattern and are valid within a given time window. Similar to SPARQL, we define a triple matching pattern operator on an RDF dataset at timestamp t as

$$[\![P, t]\!]_G = \{\mu \mid \mathit{dom}(\mu) = var(P) \wedge \mu(P) \in G(t)\}.$$

A window operator $[\![P, t]\!]^{\omega}_{S}$ is then defined by extending the operator above as follows.

$$[\![P, t]\!]^{\omega}_{S} = \{\mu \mid \mathit{dom}(\mu) = var(P) \wedge \langle \mu(P) : [t'] \rangle \in S \wedge t' \in \omega(t)\}.$$

where $\omega(t) \colon \mathbb{N} \to 2^{\mathbb{N}}$ is a function mapping a timestamp to a (possibly infinite) set of timestamps. This gives us the flexibility to choose between different window modes [5]. For example, a time-based sliding window of size T can be expressed as $\omega_{RANGE}(t) = \{t' \mid t' \leq t \wedge t' \geq \max(0, t - T)\}$, and a window that extracts only events happening at the current time corresponds to $\omega_{NOW}(t) = \{t\}$. Moreover, we can similarly define *triple-based windows* that return the latest N triples ordered by the timestamps.

We define a *result set* $\boldsymbol{\Gamma}$ as a function from $\mathbb{N} \cup \{-1\}$ to finite but unbounded bags of mappings, where $\boldsymbol{\Gamma}(-1) = \emptyset$. A *discrete result set* $\Omega = \boldsymbol{\Gamma}(t)$, $t \geq 0$, denotes the bag of mappings at time t. Discrete result sets are the input of relational operators described below.

Relational Operators. Our processing model supports the operators found in traditional relational database management systems [18]. Similar to the semantics of SPARQL [29], the operators work on the mappings from discrete result sets. As an example, given two discrete result sets, Ω_1 and Ω_2, the join and union operators are defined as

$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible } \}$
$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \vee \mu \in \Omega_2\}$.

Streaming Operators. Similarly to the relation-to-stream operator of CQL [5], we define an operator, based on some patterns, to generate RDF streams from result sets. From a graph template $\mathbb{T}$, that provides a set of triple patterns, and a result set $\boldsymbol{\Gamma}$, a streaming operator $\mathbb{C}$ is defined as

$$\mathbb{C}(\mathbb{T}, \boldsymbol{\Gamma}) = \bigcup_{t \geq 0} \{\langle \mu(P) : [f(t)] \rangle \mid \mu \in \boldsymbol{\Gamma}(t) \setminus \boldsymbol{\Gamma}(t-1) \wedge P \in \mathbb{T}\},$$

where $f \colon \mathbb{N} \to \mathbb{N}$ is a function mapping t to a new timestamp to indicate when we want to stream out the result. In the simplest case, f is the identity function, indicating that triples are streamed out immediately.

Query Semantics. Operators of a query are organised in a *data flow*. A data flow D is a directed tree of operators, whose root node is either a relational or a streaming operator, while leaves and intermediate nodes are window and relational operators, respectively.

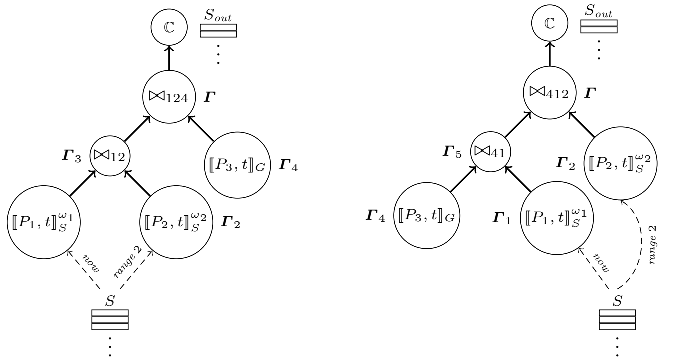
Suppose the inputs to the leaves of D are RDF streams $S_1, \ldots, S_n$ $(n \geq 1)$ and RDF datasets $G_1, \ldots, G_m$ $(m \geq 0)$. The *query semantics* of D is then defined as follows: If the root of D is a streaming (resp., relational) operator, producing a stream S (resp., result set $\boldsymbol{\Gamma}$), then the result of D at time t is $S^{\leq t}$ (resp., $\boldsymbol{\Gamma}(t)$), which is produced by recursively applying the operators comprising D to $S_1^{\leq t}, \ldots, S_n^{\leq t}$ and $G_1, \ldots, G_m$. Next we introduce the "localisation scenario" to illustrate the query semantics of our processing model. This scenario will also be used in following sections of the paper.

Localisation scenario: Consider a group of people wearing devices that constantly stream their locations in a building, i.e., in which room they currently are, and assume we have information about the direct connectivity between the rooms, given by a static RDF dataset G with triples of the form $P_3 = (?loc_1, conn, ?loc_2)$, where $G^S = \{(r_1, conn, r_2), (r_1, conn, r_3),$ $(r_2, conn, r_1)$, $(r_3, conn, r_1)\}$. Also assume that people's locations are provided in a single stream S with triples of form $(?person, detectedAt, ?loc)$. We are interested in answering the following continuous query: "Notify two people when they can reach each other from two different and directly connected rooms."

Figure 1a depicts a possible data flow D_1 for the query in the localisation scenario. It suggests to extract two windows from stream S using the functions $\omega_1 = \omega_{NOW}$ and $\omega_2 = \omega_{RANGE}$. The former looks at the latest detected person, and the latter monitors people during the last T logical clock ticks by which we can assume that they are still in the same room. For the example, we assume $T = 2$. Let $\boldsymbol{\Gamma}_1$ and $\boldsymbol{\Gamma}_2$ be the outputs of the window operators. We use the triple patterns $P_i = (?person_i, detectedAt, ?loc_i)$ for $i = 1, 2$ at the window operators; hence, mappings in $\boldsymbol{\Gamma}_i$ are of the form $\{?person_i \mapsto pid, ?loc_i \mapsto lid\}$.

The join $\bowtie_{12}$ of discrete result sets from $\boldsymbol{\Gamma}_1$ and $\boldsymbol{\Gamma}_2$ in Figure 1a gives us the output result set in $\boldsymbol{\Gamma}_3$ to check the reachability based on the latest detected person. After joining elements of $\boldsymbol{\Gamma}_3$ with those of $\boldsymbol{\Gamma}_4$ (the direct connectivity between locations provided by G) via $\bowtie_{124}$, we have the result set $\boldsymbol{\Gamma}$ to answer the query. To return this result in terms of a stream S_{out}, the operator $\mathbb{C}$ is used at the root of D_1.

Table 1 shows the input/output of D_1 as time progresses. To reduce space consumption, we use abbreviations as follows: dA for $detectedAt$, $?p$ for $?person$, and $?\ell$ for $?loc$.



(a) D_1 (b) D_2

Fig. 1: Possible data flows for the query in the localisation scenario.

t	S	$\boldsymbol{\Gamma}_1$	$\boldsymbol{\Gamma}_2$	S_{out}
0	$\langle(m_0, dA, r_1) : [0]\rangle$	$\{?p_1 \mapsto m_0, ?\ell_1 \mapsto r_1\}$	$\{?p_2 \mapsto m_0, ?\ell_2 \mapsto r_1\}$	$\emptyset$
1	$\langle(m_0, dA, r_1) : [0]\rangle$		$\{?p_2 \mapsto m_0, ?\ell_2 \mapsto r_1\}$	
	$\langle(m_1, dA, r_2) : [1]\rangle$	$\{?p_1 \mapsto m_1, ?\ell_1 \mapsto r_2\}$	$\{?p_2 \mapsto m_1, ?\ell_2 \mapsto r_2\}$	$\langle(m_0, reaches, m_1) : [1]\rangle$
2	$\langle(m_0, dA, r_1) : [0]\rangle$			
	$\langle(m_1, dA, r_2) : [1]\rangle$		$\{?p_2 \mapsto m_1, ?\ell_2 \mapsto r_2\}$	$\langle(m_0, reaches, m_1) : [1]\rangle$
	$\langle(m_2, dA, r_1) : [2]\rangle$	$\{?p_1 \mapsto m_2, ?\ell_1 \mapsto r_1\}$	$\{?p_2 \mapsto m_2, ?\ell_2 \mapsto r_1\}$	$\langle(m_1, reaches, m_2) : [2]\rangle$
3	$\langle(m_0, dA, r_1) : [0]\rangle$			
	$\langle(m_1, dA, r_2) : [1]\rangle$			$\langle(m_0, reaches, m_1) : [1]\rangle$
	$\langle(m_2, dA, r_1) : [2]\rangle$		$\{?p_2 \mapsto m_2, ?\ell_2 \mapsto r_1\}$	$\langle(m_1, reaches, m_2) : [2]\rangle$
	$\langle(m_3, dA, r_2) : [3]\rangle$	$\{?p_1 \mapsto m_3, ?\ell_1 \mapsto r_2\}$	$\{?p_2 \mapsto m_3, ?\ell_2 \mapsto r_2\}$	$\langle(m_2, reaches, m_3) : [3]\rangle$
⋮	⋮	⋮	⋮	⋮

Table 1: Input and output of D_1 as time progresses.

3.3 Adaptation Strategies

A data flow contains inner relational operators which can be reordered to create new equivalent data flows. For instance, Figures 1a and 1b show two equivalent data flows for the query in the localisation scenario. With respect to each alternative, an operator might have a different next/parent operator. For example, $[\![P_1, t]\!]_S^{\omega_1}$ has $\bowtie_{12}$ as its parent in D_1 while in D_2, its parent is $\bowtie_{41}$.

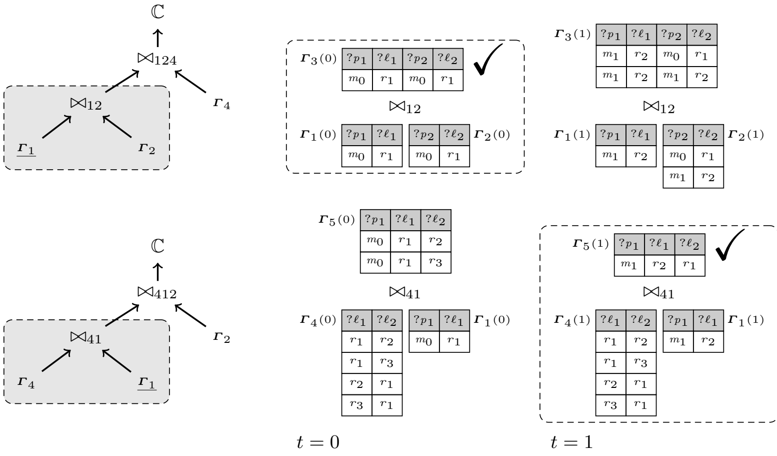
In a stream processing environment, due to updates in the input data, during the query lifetime the engine constantly attempts to determine the data flow that currently provides the most efficient query execution. We propose an adaptive query processing mechanism similar to Eddies [6], which continuously routes the outputs of an operator to the next operator on the data flow. The routing policy will dynamically tell the operator what is the next operator it should forward data to, as shown in Algorithm 1.

Function $\mathsf{route}(\mathit{routingEntry}, \mathbb{O}, t)$ is used to recursively apply the operator $\mathbb{O}$ on a mapping or timestamped triple *routingEntry* and to route the output mappings to the next operator. It uses the following primitives:

- $\mathsf{compute}(\mathit{routingEntry}, \mathbb{O}, t)$: apply $\mathbb{O}$, a window, relational, or streaming operator, to *routingEntry*, a timestamped triple or a mapping, at timestamp t, and return a discrete result set.
- $\mathsf{findNextOp}(\mathbb{O}, t)$: find the next operator to route the output mapping to, at timestamp t, based on a given routing policy.

Algorithm 1: $\mathsf{route}(routingEntry, \mathbb{O}, t)$

```
Input: $routingEntry$ : timestamped triple/mapping, $\mathbb{O}$ : operator, $t$ : timestamp
$\Omega := \mathsf{compute}(routingEntry, \mathbb{O}, t)$
if $\mathbb{O}$ is not root then
    $nextOp := \mathsf{findNextOp}(\mathbb{O}, t)$
    for $\mu \in \Omega$ do $\mathsf{route}(\mu, nextOp, t)$
else deliver $\Omega$
```



ℂ
⋈124
⋈12
Γ1
Γ2
Γ4
Γ3(0) ?p1 ?ℓ1 ?p2 ?ℓ2 / m0 r1 m0 r1 ✓
⋈12
Γ1(0) ?p1 ?ℓ1 / m0 r1
Γ2(0) ?p2 ?ℓ2 / m0 r1
Γ3(1) ?p1 ?ℓ1 ?p2 ?ℓ2 / m1 r2 m0 r1 / m1 r2 m1 r2
⋈12
Γ1(1) ?p1 ?ℓ1 / m1 r2
Γ2(1) ?p2 ?ℓ2 / m0 r1 / m1 r2
ℂ
⋈412
⋈41
Γ4
Γ1
Γ2
Γ5(0) ?p1 ?ℓ1 ?ℓ2 / m0 r1 r2 / m0 r1 r3
⋈41
Γ4(0) ?ℓ1 ?ℓ2 / r1 r2 / r1 r3 / r2 r1 / r3 r1
Γ1(0) ?p1 ?ℓ1 / m0 r1
Γ5(1) ?p1 ?ℓ1 ?ℓ2 / m1 r2 r1 ✓
⋈41
Γ4(1) ?ℓ1 ?ℓ2 / r1 r2 / r1 r3 / r2 r1 / r3 r1
Γ1(1) ?p1 ?ℓ1 / m1 r2
t = 0
t = 1

Fig. 2: Dynamically choose the next operator after $\boldsymbol{\Gamma}_1$ at timestamps 0 and 1

The routing policy decides the order in which the operators are executed at runtime. There are many ways to implement a routing policy. However, choosing the optimal order on every execution is not trivial. We are investigating mechanisms for dynamic cost-based optimisation. Preliminary findings are reported in [25]. A possible solution, common to DBMSs, is a cost-based strategy: the routing policy computes an estimated "cost" to each possible data flow, and chooses the one with the smallest cost. While the definition of cost is not fixed, it is usually measured by estimating the number of output mappings the operator will produce.

The following example illustrates how the adaptation strategies work as a whole.

Example 1. Consider again the query in the localisation scenario at timestamps 0 and 1, and assume the routing policy implemented is the cost-based strategy mentioned above. Figure 2 illustrates the decision of which operator to choose next after extracting the latest triple at $\boldsymbol{\Gamma}_1$. In this figure, two simplified versions of D_1 and D_2 are on the left. On the right hand side, we show the input/output of the join operators $\bowtie_{12}$ and $\bowtie_{41}$. At timestamp 0, $|\boldsymbol{\Gamma}_1(0)| = |\boldsymbol{\Gamma}_2(0)| = 1$ as the first triple is streamed into the system. It is preferable at this point to use D_1, i.e., to join $\boldsymbol{\Gamma}_1(0)$ with $\boldsymbol{\Gamma}_2(0)$ using $\bowtie_{12}$ because the intermediate result $\boldsymbol{\Gamma}_3(0)$ has size 1. If we follow D_2 then joining $\boldsymbol{\Gamma}_1(0)$ with $\boldsymbol{\Gamma}_4(0)$ using $\bowtie_{41}$ yields $\boldsymbol{\Gamma}_5(0)$ with size 2. However, at $t = 1$, D_2 is preferred because $|\boldsymbol{\Gamma}_3(1)| = 2$ and $|\boldsymbol{\Gamma}_5(1)| = 1$.

4 CQELS's Query Engine

The CQELS query engine implements the model introduced in Section 3. Continuous queries can be registered using our CQELS language, an extension of the declarative SPARQL 1.1 language, which is described next. We then explain the details of the engine. We show how data is encoded for memory savings, how caching and indexing are used for faster data access, and how operators and the routing policy are implemented. Before moving onto the query language, we first need to introduce our second scenario, the “conference scenario,” which is also used in the evaluation section.

Conference scenario: This scenario is based on the Live Social Semantics experiment presented in [2]. We extend the localisation scenario by considering that people are now authors of research papers and they are attending a conference. These authors have their publication information stored in a DBLP dataset. To enhance the conference experience, each participant would have access to the following services, which can all be modelled as continuous queries:

- (Q1) Inform a participant about the name and description of the location he just entered,
- (Q2) Notify two people when they can reach each other from two different and directly connected (from now on called *nearby*) locations,
- (Q3) Notify an author of his co-authors who have been in his current location during the last 5 seconds,
- (Q4) Notify an author of the editors that edit a paper of his and have been in a nearby location in the last 15 seconds,
- (Q5) Count the number of co-authors appearing in nearby locations in the last 30 seconds, grouped by location.

4.1 CQELS Language

Based on our query semantics, we introduce a declarative query language called CQELS by extending the SPARQL 1.1 grammar[6] using the EBNF notation. We add a query pattern to apply window operators on RDF Streams into the *GraphPatternNotTriples* pattern.

```
GraphPatternNotTriples ::= GroupOrUnionGraphPattern | OptionalGraphPattern
                           | MinusGraphPattern | GraphGraphPattern | StreamGraphPattern
                           | ServiceGraphPattern | Filter | Bind
```

Assuming that each stream is identified by an IRI as identification, the **StreamGraphPattern** pattern is defined as follows.

```
StreamGraphPattern ::= 'STREAM' '[' Window ']' VarOrIRIref '{' TriplesTemplate '}'
            Window ::= Range | Triple | 'NOW' | 'ALL'
             Range ::= 'RANGE' Duration ('SLIDE' Duration)?
            Triple ::= 'TRIPLES' INTEGER
          Duration ::= (INTEGER 'd' | 'h' | 'm' | 's' | 'ms' | 'ns')+
```

where *VarOrIRIRef* and *TripleTemplate* are patterns for the *variable/IRI* and *triple template* of SPARQL 1.1, respectively. *Range* corresponds to a time-based window while *Triple* corresponds to a triple-based window. The keyword *SLIDE* is used for specifying the sliding parameter of a time-based window, whose time interval is specified by *Duration*. More details of the syntax are available at `http://code.google.com/p/cqels/`.

Given the CQELS language defined above, we can represent the five queries from the conference scenario as follows, where $Name$ is replaced by a constant when instantiating the query.[7]

```
SELECT ?locName ?locDesc
FROM NAMED <http://deri.org/floorplan/>
WHERE {
  STREAM <http://deri.org/streams/rfid> [NOW] {?person lv:detectedAt ?loc}
  GRAPH  <http://deri.org/floorplan/>  {?loc lv:name ?locName. ?loc lv:desc ?locDesc}
  ?person foaf:name ``$Name$''. }
```

Query Q1

[6] `http://www.w3.org/TR/sparql11-query/#grammar`

[7] For the sake of space we omit the PREFIX declarations of lv, dc, foaf, dcterms and swrc

```
CONSTRUCT {?person1 lv:reachable ?person2}
FROM NAMED <http://deri.org/floorplan/>
WHERE {
  STREAM <http://deri.org/streams/rfid> [NOW]      {?person1 lv:detectedAt ?loc1}
  STREAM <http://deri.org/streams/rfid> [RANGE 3s] {?person2 lv:detectedAt ?loc2}
  GRAPH  <http://deri.org/floorplan/>              {?loc1    lv:connected  ?loc2} }
```

Query Q2

```
SELECT ?coAuthName
FROM NAMED <http://deri.org/floorplan/>
WHERE {
  STREAM <http://deri.org/streams/rfid> [TRIPLES 1] {?auth   lv:detectedAt ?loc}
  STREAM <http://deri.org/streams/rfid> [RANGE 5s]  {?coAuth lv:detectedAt ?loc}
  { ?paper dc:creator ?auth.     ?paper  dc:creator ?coAuth.
    ?auth  foaf:name ``$Name$''. ?coAuth foaf:name  ?coAuthorName}
  FILTER (?auth != ?coAuth) }
```

Query Q3

```
SELECT ?editorName
WHERE {
  STREAM <http://deri.org/streams/rfid> [TRIPLES 1] {?auth   lv:detectedAt ?loc1}
  STREAM <http://deri.org/streams/rfid> [RANGE 15s] {?editor lv:detectedAt ?loc2}
  GRAPH  <http://deri.org/floorplan/> {?loc1   lv:connected  ?loc2}
  ?paper      dc:creator  ?auth.   ?paper  dcterms:partOf ?proceeding.
  ?proceeding swrc:editor ?editor. ?editor foaf:name      ?editorName.
  ?auth       foaf:name  ``$Name$'' }
```

Query Q4

```
SELECT ?loc2 ?locName count(distinct ?coAuth) as ?noCoAuths
FROM NAMED <http://deri.org/floorplan/>
WHERE {
  STREAM <http://deri.org/streams/rfid> [TRIPLES 1] {?auth   lv:detectedAt ?loc1}
  STREAM <http://deri.org/streams/rfid> [RANGE 30s] {?coAuth lv:detectedAt ?loc2}
  GRAPH  <http://deri.org/floorplan/> {?loc2 lv:name ?locName. loc2 lv:connected ?loc1}
  {?paper dc:creator ?auth. ?paper dc:creator ?coAuth. ?auth foaf:name ``$Name$''}
  FILTER (?auth != ?coAuth)}
  GROUP BY ?loc2 ?locName
```

Query Q5

4.2 Data encoding

When dealing with large data collections, it is very likely that data will not fit into the machine's main memory for processing, and parts of it will have to be temporarily stored on disk. In the particular case of RDF data, with IRIs or literals stored as strings, a simple join operation on strings could generate enough data to trigger a large number of disk reads/writes. However, these are among the most expensive operations in query processing and should be avoided whenever possible. While we cannot entirely avoid disk access, we try to reduce it by encoding the data such that more triples can fit into main memory.

We apply *dictionary encoding*, a method commonly used by triple stores [1, 16, 13]. An RDF node, i.e., literal, IRI or blank node, is mapped to an integer identifier. The encoded version of an RDF node is considerably smaller than the original, allowing more data to fit into memory. Moreover, since data comparison is now done on integers rather than strings, operations like pattern matching, perhaps the most common operator in RDF streams and datasets, are considerably improved.

However, in context of RDF streams, data is often fed into the system at a high rate, and there are cases when the cost of updating a dictionary and decoding the data might significantly hinder the performance. Therefore, our engine does not encode the RDF nodes into dictionary if they can be represented in 63 bits. As such, a node identifier is presented as a 64-bit integer. The first bit is used to indicate whether the RDF node is encoded or not. If the RDF nodes does not have to be encoded, the next 5 bits represent the data type of the RDF node (e.g. integer, double or float) and the last 58 bits store its value. Otherwise, the RDF node is stored in the dictionary and its identifier is stored in the remaining 63 bits.

4.3 Caching and Indexing

While data encoding allows a smaller data representation, caching and indexing aim at providing faster access to the data. Caching is used to store intermediate results of sub-queries over RDF data sets. Indexing is applying on top of caches, as well as on output mapping sets from window operators, for faster data look-ups. Similar to data warehouses, cached data is initially kept on disk with indexes and only brought to memory when needed.

In continuous query processing, RDF datasets are expected to have a much slower update rate than RDF streams. Therefore, the output of a sub-query over an RDF dataset rarely changes during a series of updates of RDF streams. Based on this observation, as soon as a query is registered, we materialise the output of its sub-queries over the RDF datasets and store them in a cache that is available to the remaining query operators. Thereby, a possibly large portion of the query does not need to be re-executed when new stream triples arrive.

To keep the cache updated, we use triggers to notify changes in the RDF datasets. The CQELS engine has a triple store that allows the engine to load and update RDF datasets as named graphs. This triple store provides triggers that will notify the engine to update the respective cached data. For the RDF datasets that are not loaded, we manually set a timer to trigger an update. At the moment, a cache update is done by recomputing the full sub-query as a background process and replacing the old cached data by the new results as soon as they are ready. We are investigating adaptive caching [7] and materialised view maintenance [21] techniques to create more efficient cache updating mechanisms.

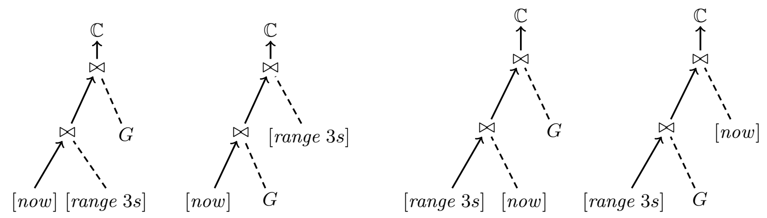
For faster lookups on the cache, indexes are built on the variables shared among the materialised sub-queries and other operator's inputs. We use similar indexing schemas as in popular triple stores [13, 16, 23, 28, 38]. Vigals et al. [36] showed that, in stream processing, building hash tables for multi-way joins can accelerate the join operation. Therefore, we also index data coming from window operators, which are the input to the relational operators. Similar to caching, there is an update overheard attached to indexes. In CQELS, the decision to create an index is as follows: cache data is always indexed. For data coming from window operators, an index is maintained as long as it can be updated faster than the window's stream rate. If this threshold is reached, the index is dropped, and the relational operators that depend on this index will be replaced by equivalent ones that can work without indexes.

4.4 Operators and Routing Policy

To recap, the CQELS processing model contains three groups of operators: window, relational and streaming operators. In the current implementation, we support two types of window operators: *triple-based window* and *sliding window*. We implement all relational operators needed to support the CQELS language. In particular, one of the join operators is a binary index join that uses indexing for faster processing. The implementation of the streaming operator is rather simple: as soon as a mapping arrives at the streaming operator, it simply binds the mapping to the graph template, then sends the output triples, tagged with the time they were created, to the output stream.

To allow adaptive query execution, our engine currently support a "cardinality-based" routing policy, based on some heuristics. For a given query, the engine keeps all possible left-deep data flows that start with a window operator. For instance, Figure 3 shows the four data flows that are maintained for the query in the localisation scenario from Section 3.

Algorithm 2 shows the findNextOp function used in the current routing policy (see Algorithm 1). It applies two simple heuristics: the first one, common in DBMSs, pushes



(a) From window now (b) From window $range\ 3s$

Fig. 3: Left-deep data flows for the query in the localisation scenario.

Algorithm 2: findNextOp($\mathbb{O}, t$)

```
Input: $\mathbb{O}$ : operator, $t$ : timestamp
$nextOp := null$
for $unaryOp \in nextUnaryOp(\mathbb{O})$ do
    if $unaryOp$ is a filter operator then return $unaryOp$ else $nextOp := unaryOp$
$mincard := +\infty$
for $binaryOp \in nextBinaryOpOnLeftDeepTree(\mathbb{O})$ do
    if $mincard > card(binaryOp.rightChildOp, t)$ then
        $mincard := card(binaryOp.rightChildOp, t)$
        $nextOp := binaryOp$
return $nextOp$
```

operators like filters closer to the data sources. The rationale here is that the earlier we prune the triples that will not make it to the final output, the better, since operators will then process fewer triples. The second looks at the cardinality of the operators' output and sorts them in increasing order of this value, which also helps in reducing the number of mappings to process.

Function $nextUnaryOp(\mathbb{O})$ returns the set of possible next unary operators that $\mathbb{O}$ can route data to, while $nextBinaryOpOnLeftDeepTree(\mathbb{O})$ returns the binary ones. Examples of unary operators are filters and projections, and they can be directly executed on the output produced by $\mathbb{O}$. Binary operators, such as joins and unions, have two inputs, called left and right child, due to the tree shape of the data flows. $\mathbb{O}$ will be the left child, since the data flows are all left-deep. The right child is given by the $rightChildOp$ attribute. For each binary operator, we obtain the cardinality of the right child at time t from $card(binaryOp.rightChildOp, t)$. We then route the output of $\mathbb{O}$ to the one whose cardinality function returns the smallest value.

5 Experimental Evaluation

To evaluate the performance of CQELS, we compare it against two existing systems that also offer integrated processing of Linked Streams and Linked Data – C-SPARQL [9] and ETALIS [3].[8] Note that EP-SPARQL is implemented on top of ETALIS. We first planned to express our queries in EP-SPARQL, which would then be translated into the language used in ETALIS. However, the translation from EP-SPARQL to ETALIS is currently not mature enough to handle all queries in our setup, so we decided to represent the queries directly in the ETALIS language. We also considered comparing our system against SPARQLstream [14],

[8] We would like to thank the C-SPARQL, ETALIS, and SPARQLstream teams for their support in providing their implementations and helping us to understand and correctly use their systems.

	Q_1	Q_2	Q_3	Q_4	Q_5
CQELS	0.47	3.90	0.51	0.53	21.83
C-SPARQL	332.46	99.84	331.68	395.18	322.64
ETALIS	0.06	27.47	79.95	469.23	160.83

Table 2: Average query execution time for single queries (in milliseconds).

but its current implementation does not support querying on both RDF streams and RDF dataset. Next, we describe our experimental setup, and then report and discuss the results obtained. All experiments presented in this paper are reproducible. Both systems and datasets used are available at `http://code.google.com/p/cqels/`.

5.1 Experimental Setup

We use the conference scenario introduced in Section 4. For the stream data, we use the RFID-based tracking data streams provided by the Open Beacon community.[9] The data is generated from active RFID tags, the same hardware used in the Live Social Semantics deployment [2]. The data generator from SP^{2}Bench [30] is used to create simulated DBLP datasets. We have also created a small RDF dataset, 172 triples, to represent the connectivity between the locations given in the Open Beacon dataset.

The experiments were executed on a standard workstation with 1 x Quad Core Intel Xeon E5410 2.33 GHz, 8GB memory, 2 x 500GB Enterprise SATA disks, running Ubuntu 11.04/x86_64, Java version "1.6", Java HotSpot(TM) 64-Bit Server VM, and SWI-Prolog 5.10.4. The maximum heap size on JVM instances when running CQELS and C-SPARQL was set to 4GB. For ETALIS, the global stack size is also 4GB.

We evaluate performance in terms of average query execution time. At each run, after registering the query, we stream a number of triples into the system and every time the query is re-executed we measure its processing time. We then average these values over multiple runs.

The queries used follow the templates specified in Section 4.1. They were selected in a way that cover many operators with different levels of complexity, for instance joins, filters and aggregations. One query instance is formed by replacing $Name$ in the template with a particular author's name from the DBLP dataset. We have performed the following three types of experiments:

Exp.(1) *Single query*: For each of the Q1, Q3, Q4 and Q5 templates we generate 10 different query instances. For query template Q2, since it has no constants, we create one instance only. Then we run each instance at a time and compute the average query execution time.

Exp.(2) *Varying size of the DBLP dataset*: We do the same experiment as in (1) but varying the numbers of triples of the DBLP dataset, ranging from 10^4 to 10^7 triples. We do not include Q2 in this experiment, since it does not involve the DBLP dataset.

Exp.(3) *Multiple queries*: For query templates Q1, Q3 and Q4, we register 2^M query instances at the same time, with $0 \leq M \leq 10$, and execute them in parallel.

In experiments Exp.(1) and Exp.(3), the numbers of triples from DBLP is fixed to 10^5.

5.2 Results & Analysis

Table 2 shows the results for Exp.(1). We can see that, for most of the cases, CQELS outperforms the other approaches by orders of magnitude; sometimes it is over 700 times

[9] `http://www.openbeacon.org/`

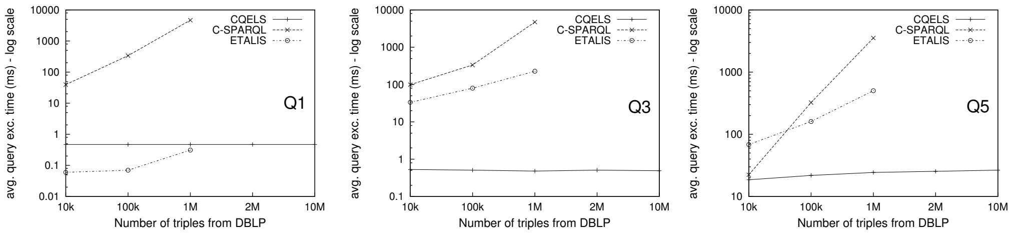


Fig. 4: Average query execution time for varying sizes of simulated DBLP dataset.

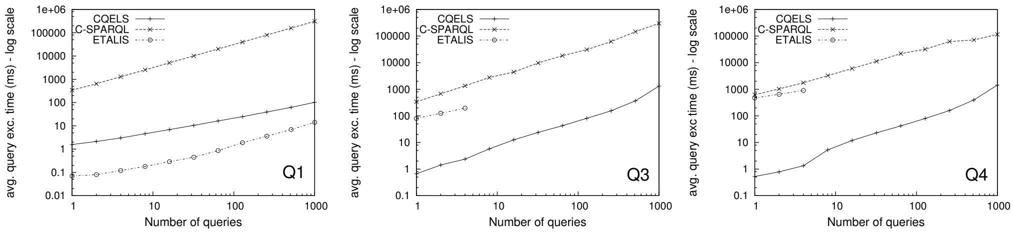


Fig. 5: Average query execution time when running multiple query instances.

faster. The only exception is query Q1, where ETALIS is considerably faster. The reason is that ETALIS supports three consumption policies, namely *recent*, *chronological*, and *unrestricted*, where *recent* is very efficient for queries containing only simple filters on the stream data. For more complex queries, the performance of ETALIS drops significantly. C-SPARQL is currently not designed to handle large datasets, which explains its poor performance in our setup. CQELS, on the other hand, is able to constantly deliver great performance, due to its combination of pre-processing and adaptive routing policy.

The results from Exp.2 are shown in Figure 4, for query templates Q1, Q3 and Q5. The results for query template Q4 are very similar to those from query template Q3, so we omit them for the sake of space.

We can see how the performance is affected when the size of the RDF dataset increases. For both ETALIS and C-SPARQL, not only does the average execution time increase with the size of the RDF dataset, but they are only able to run up to a certain number of triples. They can execute queries with a RDF dataset of 1 million triples, but at 2 million ETALIS crashes and C-SPARQL does not respond. CQELS' performance is only marginally affected by the RDF dataset's size, even for values as high as 10 million triples, and the performance gains sometimes were three orders of magnitude. This is mainly due to the cache and indexes used for storing and accessing pre-computed intermediate results. We have observed that the size of the cache, which stores the co-authors and editors of a certain author, does not increase linearly with the size of the dataset. Moreover, by using indexes on this cache, the access time of a mapping increases only logarithmically with the cache size. This behaviour shows the importance of having such cache and index structures for efficient query processing.

As a scalability test, we wanted to analyse how the systems perform with a number of queries running in parallel. Figure 5 presents the results for Exp.(3). Again, ETALIS delivers the best performance when there is no join operator on the stream data (Q1). But, for the other cases, the number of queries it can handle in parallel is very limited (less than 10). Both C-SPARQL and CQELS can scale to a large number of queries, but in C-SPARQL queries face a long execution time that exceeds 100 seconds, while in CQELS, even with 1000 queries running, the average execution time is still around one second. This scalability is mainly due to our encoding technique, which allows more efficient use of main memory, consequently reducing read/write disk operations.

In summary, our experimental evaluation shows the great performance of CQELS, both in terms of efficiency and scalability. Its query engine, with the cache, index, and routing policy, adapts well to different query complexities and it can scale with the size of the RDF datasets. Our encoding technique enhances memory usage, which is crucial when handling multiple queries. Even though ETALIS performed better for simpler queries, CQELS performs consistently well in all the experiments, and in most cases outperforms the other approaches by orders of magnitude.

6 Conclusions

This paper presented CQELS, a native and adaptive approach for integrated processing of Linked Stream Data and Linked Data. While other systems use a "black box" approach which delegates the processing to existing engines, thus suffering major efficiency drawbacks because of lack of full control over the query execution process, CQELS implements the required query operators natively, enabling improved query execution. Our query engine can adapt to changes in the input data, by applying heuristics to reorder the operators in the data flows of a query. Moreover, external disk access on large Linked Data collections is reduced with the use of data encoding, and caching/indexing enables significantly faster data access. Our experimental evaluation shows the good performance of CQELS, in terms of efficiency, latency and scalability. CQELS performs consistently well in experiments over a wide range of test cases, outperforming other approaches by orders of magnitude.

Our promising results indicate that an integrated and native approach is in fact necessary to achieve the required query execution efficiency. For future work, we plan to improve the performance of CQELS further. Query optimisation in adaptive query processing is still an open problem under active research [17]. We have already started investigating cost-based query optimisation policies [25] and we plan to look into adaptive caching [7] and materialised view maintenance [21] to enhance the efficiency of our query execution algorithms.

References

1. D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB'07*, pages 411–422, 2007.
2. H. Alani, M. Szomszor, C. Cattuto, W. V. den Broeck, G. Correndo, and A. Barrat. Live Social Semantics. In *ISWC'09*, pages 698–714, 2009.
3. D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. Ep-sparql: a unified language for event processing and stream reasoning. In *WWW '11*, pages 635–644, 2011.
4. A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.
5. A. Arasu and J. Widom. A denotational semantics for continuous queries over streams and relations. *SIGMOD Record*, 33(3):6–12, 2004.
6. R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. *SIGMOD Rec.*, 29(2):261–272, 2000.
7. S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive Caching for Continuous Queries. In *ICDE'05*, pages 118–129, 2005.
8. M. Balazinska, A. Deshpande, M. J. Franklin, P. B. Gibbons, J. Gray, M. Hansen, M. Liebhold, S. Nath, A. Szalay, and V. Tao. Data Management in the Worldwide Sensor Web. *IEEE Pervasive Computing*, 6(2):30–40, 2007.
9. D. F. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus. An execution environment for C-SPARQL queries. In *EDBT 2010*, pages 441–452, 2010.

10. C. Bizer, T. Heath, and T. Berners-Lee. Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.
11. A. Bolles, M. Grawunder, and J. Jacobi. Streaming SPARQL - Extending SPARQL to Process Data Streams. In *ESWC'08*, pages 448–462, 2008.
12. E. Bouillet, M. Feblowitz, Z. Liu, A. Ranganathan, A. Riabov, and F. Ye. A semantics-based middleware for utilizing heterogeneous sensor networks. In *DCOSS'07*, pages 174–188, 2007.
13. J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: An architecture for storing and querying rdf data and schema information. 2003.
14. J. P. Calbimonte, O. Corcho, and A. J. G. Gray. Enabling ontology-based access to streaming data sources. In *ISWC'10*, pages 96–111.
15. D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: a new class of data management applications. In *VLDB'02*, pages 215–226, 2002.
16. E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In *VLDB 2005*, pages 1216–1227, 2005.
17. A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Found. Trends databases*, 1, January 2007.
18. R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems (5th Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
19. C. J. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991.
20. L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003.
21. A. Gupta and I. S. Mumick. Materialized views. chapter Maintenance of materialized views: problems, techniques, and applications, pages 145–157. 1999.
22. C. Gutierrez, C. A. Hurtado, and A. Vaisman. Introducing Time into RDF. *IEEE Transactions on Knowledge and Data Engineering*, 19:207–218, 2007.
23. A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In *ISWC'07*, pages 211–224, 2007.
24. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
25. D. Le-Phuoc, J. X. Parreira, M. Hausenblas, and M. Hauswirth. Continuous query optimization and evaluation over unified linked stream data and linked open data. Technical report, DERI, 9 2010.
26. S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *2002 ACM SIGMOD International Conference on Management of Data*, pages 49–60, 2002.
27. F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.
28. T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010.
29. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):1–45, 2009.
30. M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Sp2bench: A sparql performance benchmark. In *ICDE'09*, pages 222–233, 2009.
31. J. F. Sequeda and O. Corcho. Linked stream data: A position paper. In *SSN'09*, 2009.
32. A. P. Sheth, C. A. Henson, and S. S. Sahoo. Semantic Sensor Web. *IEEE Internet Computing*, 12(4):78–83, 2008.
33. U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS '04*, pages 263–274.
34. H. Stuckenschmidt, R. Vdovjak, G.-J. Houben, and J. Broekstra. Index structures and algorithms for querying distributed rdf repositories. In *WWW*, pages 631–639, 2004.
35. J. Umbrich, M. Karnstedt, and S. Land. Towards understanding the changing web: Mining the dynamics of linked-data sources and entities. In *KDML (Workshop)*, 2010.
36. S. D. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB '03*.
37. K. Whitehouse, F. Zhao, and J. Liu. Semantic Streams: A Framework for Composable Semantic Interpretation of Sensor Data. In *European Workshop on Wireless Sensor Networks*, pages 5–20. EWSN, 2006.
38. K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in Jena2. pages 35–43, 2003.