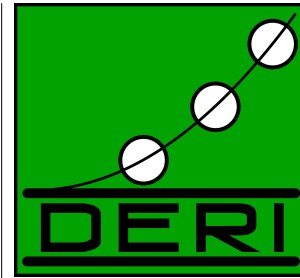


DERI – DIGITAL ENTERPRISE RESEARCH INSTITUTE



CONTINUOUS QUERY
OPTIMIZATION AND EVALUATION
OVER UNIFIED LINKED STREAM
DATA AND LINKED OPEN DATA

Danh Le-Phuoc Josiane Xavier Parreira
Michael Hausenblas Manfred Hauswirth

DERI TECHNICAL REPORT 2010-09-27

JANUARY 2010; SEPTEMBER 2010

DERI – DIGITAL ENTERPRISE RESEARCH INSTITUTE

DERI Galway
IDA Business Park
Lower Dangan
Galway, Ireland
<http://www.deri.ie/>

DERI TECHNICAL REPORT

DERI TECHNICAL REPORT 2010-09-27, JANUARY 2010; SEPTEMBER 2010

CONTINUOUS QUERY OPTIMIZATION AND EVALUATION OVER UNIFIED LINKED STREAM DATA AND LINKED OPEN DATA

Danh Le-Phuoc¹

Josiane Xavier Parreira

Michael Hausenblas

Manfred Hauswirth

Abstract. In this report we address the problem of scalable query processing over Linked Stream Data integrated with Linked Open 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 will enable the easy integration of sensor data with the quickly growing amount of Linked Open Data and facilitate the use of the large body of existing software along with a wide range of novel applications. However, the highly dynamic nature of sensor data requires new approaches for data management and processing which are not supported by existing systems. To remedy this, we present our *Continuous Query Evaluation over Linked Streams* (CQELS) approach which provides a scalable query processing model for unified Linked Stream Data and Linked Open Data. Scalability in CQELS is achieved by applying state-of-the-art techniques for efficient data storage and query pre-processing, combined with a new adaptive cost-based query optimization algorithm for dynamic data sources, such as sensor streams. In traditional Database Management Systems

(DBMS), query optimizers use pre-computed selectivity values for the data to decide on the best execution plan, whereas with continuous query over stream data the data – and consequently its selectivity values – varies over time. This means that the optimal execution plan itself can vary throughout the execution of the query. To overcome this problem, the CQELS query optimizer retains a subset of the possible execution plans and, at query time, updates their respective costs and chooses the least expensive one for executing the query at this given point in time. We have implemented CQELS and our experimental results show that CQELS can greatly reduce query response times while scaling to a realistically high number of parallel queries.

Keywords: Stream Data, Linked Data, unifying processing model, real-time processing, large-scale data management

¹DERI (Digital Enterprise Research Institute), National University of Ireland, Galway , IDA Business Park, Lower Dangan, Galway, Ireland. E-mail: danh.lephuoc@deri.org.

Acknowledgements: The work presented in this paper is funded by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-2).

Copyright © 2010 by the authors

Contents

1	Introduction	1
2	Related Work	2
3	Unifying Processing Model	4
3.1	Query Model	5
3.2	Query evaluation	7
4	Pre-processing	8
4.1	Data Storage	8
4.2	Pre-processing of Intermediate Query Results	9
5	Adaptive Cost-based Optimization	10
6	Performance and Scalability Evaluation	13
6.1	Setup	13
6.2	Results	14
7	Conclusions	18

1 Introduction

In the past years, sensors have become ubiquitous, for instance in mobile phones (accelerometer, compass, GPS, camera, etc.), in weather observation stations (temperature, humidity, etc.), in the health care domain (heart rate, blood pressure monitors, etc.), in devices for tracking people's and object's locations (GPS, RFID, etc.), in buildings (energy measurement, environmental conditions, etc.), cars (engine monitoring, driver monitoring, etc.), and in the Web at large, with online communities such as Twitter and Facebook delivering (typically unstructured) real-time data on various topics (RSS or Atom feeds, etc.). The raw nature of the data produced by sensors – that is, the basic readings, without any metadata attached to it – limited the use of sensor networks to specific applications domains. Typically applications are still custom-built for specific cases and are to be classified as “information stovepipes”, i.e., integration of sensor data with other data sources is still a difficult and labour-intensive task, which currently requires a lot of “hand-crafting”.

Only recently, there have been a lot of efforts to lift sensor data to a semantic level, for example, by the W3C Semantic Sensor Network Incubator Group¹, Semantic Streams [38], Semantic System S [11], and Semantic sensor web [32]. These projects make sensor data available following the Link-ed Data principles [9], a concept known as *Linked Stream Data* [31]. Linked Stream Data aims at the seamless integration of sensor data with other data sources, such as found in the Linked Open Data (LOD) cloud and enabling a range of new, “real-time” applications, while also making this data accessible to the wide range of existing software. Exposing stream data according to Linked Data standards seems to be a promising way to “understand” this information through semantic enrichment. It can be easily integrated with existing data sets using established standards specifically designed for query-processing, and generally makes information accessible in a way which furthers its re-use and integration with data sets currently unthought of.

Experiments, such as the Live Social Semantics experiment [2], have already demonstrated the potential benefits of correlating physical data from sensors and social relationships from other data sources. By combining data from Active RFID tags worn by people during a conference and their online profile, the system was able to offer services to conference attendees to enhance their social experience, including the spatial visualisation of contact data. Taking this idea a step further by involving more users and more sensors, for example, GPS location from mobile phones and social interactions seems to be a logical next step. Projects, such as NoiseTube² which re-purposes existing infrastructure (mobile phone's microphone for environmental noise measurement), have already created large semantic data sets (in Knowledge Markup Language, KML) which provide an invaluable source of information to city planners. These are just two prominent examples for two specific domains. The number of projects and the amount of data they generate is currently growing at an exponential rate.

However, the realization of such applications bears a couple of significant challenges, including:

- Data is **distributed**: data collections are stored at or streamed from different sources across the world.
- Data is **heterogeneous**: although represented in a format that can be integrated, data is still heterogeneous and this has to be considered when processing it. This applies to the instance as well as to the schema level.
- Data is produced in **real-time**: sensors can stream data at high rates. While for “traditional” sensors this is obvious, it applies to many systems on a general basis, for instance, Twitter has reached the

¹<http://www.w3.org/2005/Incubator/ssn/>

²<http://noisetube.net/>

mark of 50 million tweets per day.

- Data collections are **large**: the Linked Open Data cloud contains, at time of this writing, 20 billion triples. In particular, for example, DBLP³ has more than 170 million triples, and LinkedGeoData⁴ has more than 1 billion triples.

Current attempts in combining sensor and Linked Open Data suffer from scalability problems (for example, the Live Social Semantic experiment involved only 139 conference attendees), and therefore such solutions are not suitable for large-scale applications. Also, the real-time nature of data is often not covered and the generated data sets frequently are not available online but only offline through normal download specifically due to the lack of efficient query processing.

In this paper we address the problem of scalable query processing over Linked Stream Data integrated with Linked Open Data. We present our *Continuous Query Evaluation over Linked Streams* (CQELS) approach, an unifying processing model for scalable continuous query evaluation over combined Linked Stream Data and Linked Open Data. Scalability in CQELS is achieved by applying state-of-the-art techniques for efficient data storage and query pre-processing, as well as by deriving an adaptive cost-based query optimization algorithm for dynamic data sources, such as streams. Contrary to traditional query optimizers, where pre-computed selectivity values for the data are used to decide on the best execution plan, our CQELS query optimizer keeps a subset of the possible execution plans and, at query time, updates their respective costs and chooses the least expensive one for executing the query. We have implemented CQELS and show through experimental evaluation that our model achieves great performance in terms of query response time while scaling to a realistically high numbers of parallel queries.

The remainder of this paper is organized as follows: We start with a review of related work in Section 2. Based on this, Section 3 introduces our unifying model for integrating Linked Stream Data and LOD, and presents the query model and evaluation used. The data storage and query pre-processing techniques are described in Section 4, and our adaptive cost-based query optimizer is presented in Section 5. The experimental results are presented and discussed in Section 6. Section 7 concludes the paper and highlights some planned future work.

2 Related Work

In the context of the Semantic Web a reasonable amount of work on storage and query processing for Linked Data is available, for instance, Sesame [12], Jena [39], RISC-3X [26], YARS2 [20], and Oracle Semantic [14]. Most of them focus on scalability in terms of size of the dataset and query complexity. They typically assume that the data changes infrequently, and scalability is obtained by carefully choosing appropriate data structures and by building complex indexes. Such systems are not suitable for applications involving stream data which per definition includes a high number of updates. The Berlin SPARQL benchmark⁵ shows that the throughput of a typical triple store 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 of 1000 to 10,000 (in certain cases even up to 100,000) updates per second. Nevertheless, some of the techniques and data design principles from triple stores are still useful for scalable processing of Linked Stream Data, for instance physical data organization [1, 12, 39], indexing schema [14, 20, 26].

³<http://dblp.l3s.de/dblp++.php>

⁴<http://linkedgeodata.org/>

⁵<http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/>

Data stream management systems (DSMSs) such as STREAM [3], Aurora [13], and TelegraphCQ [23] were built to overcome limitations of traditional database management systems in supporting streaming applications [17]. For that, they proposed new data models, new query semantics, as well as new execution models. They have shown better performance compared to traditional DBMS in the context of high volumes of updates. Even though they support join operations between streams and static metadata, to the best of our knowledge, there has not been any performance evaluation involving a large static datasets. In our scenario, the non-streamed linked datasets can contain millions, even billions of triples. Therefore DSMSs are not suitable for processing unified Linked Stream Data and Linked Open Data.

The idea of Linked Stream Data was introduced in [31], where user-friendly URIs were suggested for identifying sensors and stream data, but there were already other approaches that aimed at using RDF and SPARQL for representing and processing stream data. StreamSPARQL [10] and C-SPARQL [8] are two most recent efforts in providing query languages to process RDF stream data. Both extend the SPARQL query language [27] by introducing sliding window operators. While StreamSPARQL presents a rather simple query evaluation model without taking into account performance issues, C-SPARQL proposes an execution framework built on top of existing stream data management systems and triple stores. 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 does not take into account large static data sets which can degrade the performance of the stream processing considerably as we do in CQELS. Because both stream data management and triple storage systems are used independently as “black boxes”, C-SPARQL cannot take advantage of data storage and query pre-processing and optimization over the unified data but only separately as defined in the underlying systems. Thus, C-SPARQL may miss out on additional potential for optimization.

Query optimization has been extensively studied in traditional database management systems (DBMS), such as relational and XML systems [15, 29, 34]. Recently, also approaches to query optimization in the Semantic Web context have been presented. In [30] the authors propose a hybrid cost-based query optimization for the Semantic Web that applies traditional relational cost models for basic facts and an adaptive sampling technique for estimating the cost of inferred facts. Vidal et al. [37] address the problem of efficiently joining group patterns in SPARQL queries, where queries are partitioned into star-shaped groups, i.e., groups of triple patterns with only one join variable appearing in each triple at the same position, and a query optimization technique is applied to each group.

Whitehouse et al. [38] uses Prolog-based logic rules to represent the semantics of the sensor data stream in the context of the Microsoft SensorMap project⁶. In [11] Bouillet et al. propose the use of the Web Ontology Language (OWL) to represent sensor data streams, as well as processing elements for composing applications from input data streams. This work is applied in the IBM stream process system, called System S, a prototype of IBM InfoSphere Streams⁷. Along the same lines, the W3C Semantic Sensor Network Incubator Group⁸ brought experts from the Semantic Web and the Sensor Network community together to look into standardization of the Semantic Sensor Web [7, 32] technologies which aims at annotating sensor data with spatial, temporal, and thematic semantic metadata. Additionally, there are ongoing efforts in reasoning over stream data [36], fostering new paradigms for knowledge representation languages and frameworks for stream reasoning oriented towards software architectures.

⁶<http://atom.research.microsoft.com/sensewebv3/sensormap/>

⁷<http://www-01.ibm.com/software/data/infosphere/streams/>

⁸<http://www.w3.org/2005/Incubator/ssn/>

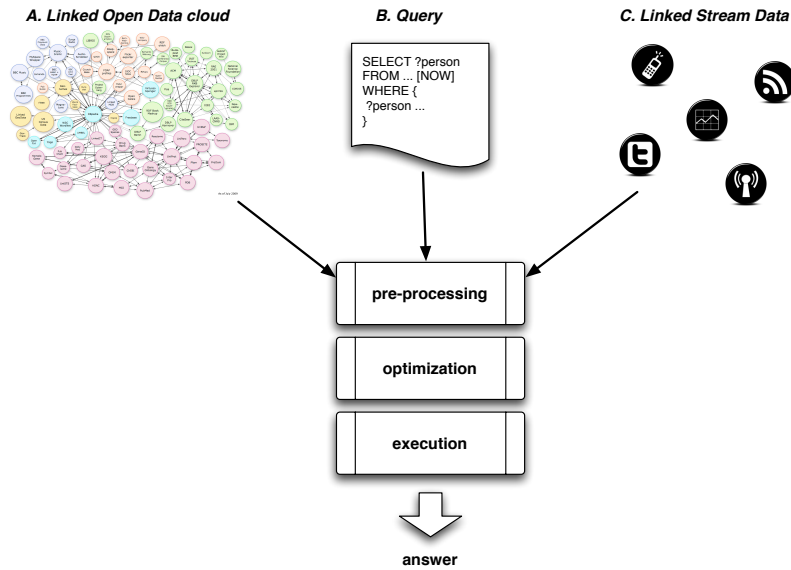


Figure 1: CQELS Processing Model.

3 Unifying Processing Model

Fig. 1 shows a conceptual view of *Continuous Query Evaluation over Linked Streams* (CQELS) which proposes an unified processing model for data coming from both sensor streams and the Linked Open Data cloud. As CQELS focuses on continuous queries – that is, queries that are registered in the system, and executed every time new data matches their criteria – we consider the queries themselves as input parameters.

CQELS aims at efficient and scalable query execution. In order to accomplish this, there is a pre-processing phase and an optimization phase, prior to the query execution. The *pre-processing phase*, described in detail in Section 4, is responsible for representing and storing the data in a more compact format, to facilitate the loading of larger amounts of data into memory at query execution time, and for storing intermediate query results. In the *optimization phase*, described in detail in Section 5, the least expensive execution plan out of a set of pre-computed plans, is chosen based on the current selectivity values, and passed on to the *execution* module, which runs this query and returns the results.

The amount of pre-computation that is done in both pre-processing and optimization phases depends on the update rate of the incoming stream. A slow update rate means that the current data is valid for a long period, and the execution of the query on this data is expected to always deliver the same results most of the time and updates are unlikely to arrive during the execution of the query. A high update rate means that query results might vary each time the query is executed and updates are likely to arrive during the query execution and would have to be considered.

In the following, we first describe the underlying query model and the physical operators used to execute the queries. We focus on the query operators needed for continuous query processing, namely the ones concerned with the temporal aspect of the queries, and reuse operators defined in SPARQL [27] and CQL [3]. Then, we show which physical operators are needed, how the query model is translated into these physical operators, and how they are executed.

3.1 Query Model

Our query model can be formalized by using similar notations as in RDF [18], SPARQL [27] and RDF temporal [19]. They define the following three pair-wise disjoint infinite sets I , B , and L as the domains of Information Resource Identifiers (IRIs), blank nodes and literals, respectively. Moreover,

$$IL=I \cup L, IB=I \cup B \text{ and } IBL=I \cup B \cup L$$

are unions of two or more of the domain sets. They also denote a triple

$$(s,p,o) \in IB \times I \times IBL$$

as an *RDF triple*, and an *RDF graph* is a set of RDF triples. A *temporal triple* is an RDF triple with a temporal label t , and it is represented as $(s, p, o) : [t]$, $t \in \mathbb{N}$. We use $t \in \mathbb{N}$ to indicate a *logical* timestamp to facilitate ordered logical clocks for local and among distributed data sources as done by classic time-synchronization approaches [22]. The issue of distributed time synchronization is beyond the scope of this paper and we assume that there are existing approaches, e.g., [24, 16], for this to be in place.

A *temporal graph* is a set of temporal triples. A *RDF stream*, G^T , is a temporal graph with a potentially infinite number of temporal triples, where T is a discrete, ordered time domain. For a temporal graph G^T , we define its snapshot at time t as a RDF graph.

$$G^T(t) = \{(s, p, o) | (s, p, o) : [t] \in G^T\}$$

Similar to [3] we have sliding window operators defined over infinite RDF streams that output a finite RDF graph that is then processed by the graph pattern operators defined later. A sliding window operator, ω , over G^T will produce a finite subset of triples as follows:

$$\omega(G^T) = \cup_{t \in \omega} G^T(t)$$

Let V be an infinite set of variables disjoint from the domain sets. We extend the list of graph patterns given in [27] by adding a pattern for aggregation queries. Therefore, we have a total of four graph patterns, which are:

1. A tuple from $(IL \cup V) \times (I \cup V) \times (IL \cup V)$ is a graph pattern (a *triple pattern*).
2. If P_1 and P_2 are graph patterns, then $(P_1 \text{ AND } P_2)$, $(P_1 \text{ OR } P_2)$ and $(P_1 \text{ UNION } P_2)$ are also graphs patterns.
3. If P is a graph pattern and R is a SPARQL built-in condition, then the expression $(P \text{ FILTER } R)$ is a graph pattern.
4. If P is a graph pattern and A is a aggregation over P , then the expression $(P \text{ AGG } A)$ is a graph pattern.

The semantics of the operators on graph patterns are defined via the concept of mapping. A mapping μ from V to T is a partial function

$$\mu: V \longrightarrow IBL.$$

For a given triple pattern τ , the triple obtained by replacing variables within τ according to μ is denoted as $\mu(\tau)$. The domain of μ , $\text{dom}(\mu)$, is the subset of V where μ is defined. Two mappings μ_1 and μ_2 are compatible if the following holds:

$$\mu_1(x) = \mu_2(x) \quad \forall x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2).$$

Let Ω_1 and Ω_2 be two mapping sets. We define the join, union, difference and left outer-join operators over Ω_1 and Ω_2 as

$$\begin{aligned}
\Omega_1 \bowtie \Omega_2 &= \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible}\} \\
\Omega_1 \cup \Omega_2 &= \{\mu \mid \mu_1 \in \Omega_1 \vee \mu_2 \in \Omega_2\} \\
\Omega_1 \setminus \Omega_2 &= \{\mu \in \Omega_1 \mid \forall \mu' \in \Omega_2, \mu \text{ and } \mu' \text{ are not compatible}\} \\
\Omega_1 \sqsupset \Omega_2 &= (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)
\end{aligned}$$

In addition, we also employ relational aggregation operators [15] to define aggregation operators on mapping sets. We denote $\text{AGG}(\Omega)_A$ as the set of mappings generated from applying the aggregation function A over Ω . Finally, we define $[[\cdot]]_G$ as the function which takes a graph pattern as input and returns a set of mappings.

Our query model can now be represented as a recursive definition of the operators described so far as follows:

1. $[[\tau]]_G = \{\mu \mid \text{dom}(\mu) = \text{var}(\tau) \wedge \mu(\tau) \in G\}$, where $\text{var}(\tau)$ is the set of variables occurring in triple pattern τ .
2. $[[P_1 \text{ AND } P_2]]_G = [[P_1]]_G \bowtie [[P_2]]_G$
3. $[[P_1 \text{ OPT } P_2]]_G = [[P_1]]_G \sqsupset [[P_2]]_G$
4. $[[P_1 \text{ UNION } P_2]]_G = [[P_1]]_G \cup [[P_2]]_G$
5. $[[P \text{ FILTER } R]]_G = \{\mu \in [[P]]_G \mid \mu \models R\}$, where $\mu \models R$ if μ satisfies condition R .
6. $[[P \text{ AGG } A]]_G = \text{AGG}([[[P]]_G)_A$
7. $[[P]]_{\omega(G^T)} = \bigcup_{t \in \omega} [[P]]_{G^T(t)}$

With the seven rules above we can now extend SPARQL to support continuous queries. In this paper, we extend SPARQL 1.1⁹ with keywords for sliding window operators, similar to [8]. We provide four example queries to illustrate this:

- Q1:** Peoples' location in the last 30 seconds, including locations and peoples' names.
- Q2:** Notify two people who are co-authors of a paper if they are in the same location (within the last 30 seconds), including the names of the location and both people.
- Q3:** Notify two people who have a citation relationship, i.e., cite each other, if they are in the same location (within the last 30 seconds), including the names of the location and both people.
- Q4:** Notify a person about the author who, among all people at a location location, he cites the most.

Figure 2 shows how to express these four queries using our query model, where <http://deri.org/locstreams> is the URI of the stream source with peoples' locations, <http://deri.org/localization/> is the graph that contains meta-data about locations (e.g, name, description, etc.), the default graph contains information about authors and their publications, and *RANGE* and *NOW* are window operators.

⁹<http://www.w3.org/TR/sparql11-query/>

Q1

```

SELECT ?name ?locName
FROM NAMED <http://deri.org/locstreams> [RANGE 30 seconds] as ?ls30
WHERE {
  GRAPH ?ls30 {?person loc:at ?loc}.
  GRAPH <http://deri.org/localization/>{?loc loc:name ?locName}
  ?person foaf:name ?name.
}

```

Q2

```

SELECT ?author1Name ?author2Name ?locName
FROM NAMED <http://deri.org/locstreams> [NOW] as ?lsnow
FROM NAMED <http://deri.org/locstreams> [RANGE 30 seconds] as ?ls30
WHERE {
  GRAPH ?lsnow {?person1 loc:at ?loc}.
  GRAPH ?ls30 {?person2 loc:at ?loc}.
  GRAPH <http://deri.org/localization/>{?loc loc:name ?locName}
  ?person1 dc:creator ?paper.?person2 dc:creator ?paper.?person1 foaf:name ?author1Name.?person2 foaf:name ?author2Name.
}

```

Q3

```

SELECT ?author1Name ?author2Name ?locName
FROM NAMED <http://deri.org/locstreams> [NOW] as ?lsnow
FROM NAMED <http://deri.org/locstreams> [RANGE 30 seconds] as ?ls30
WHERE {
  GRAPH ?lsnow {?person1 loc:at ?loc}.
  GRAPH ?ls30 {?person2 loc:at ?loc}.
  GRAPH <http://deri.org/localization/>{?loc loc:name ?locName}
  ?p1 dc:creator ?person1 .?p2 dc:creator ?person2
  {
    {?p1 dcterms:references ?p2} UNION {?p2 dcterms:references ?p1}
  }
  ?person1 foaf:name ?author1Name.?person2 foaf:name ?author2Name.
}

```

Q4

```

SELECT ?person1 ?person2 max(?numPub)
FROM NAMED <http://deri.org/locstreams> [NOW] as ?lsnow
FROM NAMED <http://deri.org/locstreams> [RANGE 30 seconds] as ?ls30
WHERE {
  GRAPH ?lsnow {?person1 loc:at ?loc}.
  GRAPH ?ls30 {?person2 loc:at ?loc}.
  { SELECT ?person1 ?person2 (count(?p2) AS ?numPub)
    WHERE {
      ?p1 dc:creator ?person1 .?p2 dc:creator ?person2. ?p1 dcterms:references ?p2
    }
  }
  GROUP BY ?person1 ?person2
}
GROUP BY ?person2

```

Figure 2: Sample queries in CQELS's query model.

3.2 Query evaluation

When a query arrives in the system, it is first translated into the corresponding query graph model and then the query graph model is represented as a sequence of pre-defined physical operators. In the CQELS model a query can be expressed as a sequence of the physical operators listed in Table 1.

The stream access operator is responsible for feeding the data into the sliding window buffer, which is in turn controlled by the sliding window operator, that also defines the size and duration of the window. The details of how the sliding window is implemented can be found in [3, 13]. The “mapping scan” operator

Operator	Description
Stream access	Receives data from stream
Sliding window	Builds a mapping set from snapshots of stream data
Mapping scan	Accesses the pre-computed mapping set of a static graph pattern
Join	Joins two sets of mappings
Left outer join	Left outer-join of two sets of mappings
Projection	Projects a set of mappings on a subset of its variables
Filter	Filters the mappings that satisfy a given condition
Aggregation	Builds aggregated mappings from a mapping set

Table 1: Physical operators of CQELS.

provides the interface to iterate over the pre-computed data. The remaining operators, “join”, “left outer join”, “projection”, “filter” and “aggregation”, are implemented similarly to traditional database management systems.

To execute queries we use the adaptive query evaluation approach proposed in [6], where a routing policy which is built based on the query graph model [21] controls the flow of the input mappings through the various operators. This approach enables our query optimizer, discussed Section 5, to dynamically change the execution order at runtime.

4 Pre-processing

In order to accomplish efficient and scalable query execution in CQELS, there is a pre-processing phase and an optimization phase, prior to the query execution, In the pre-processing phase we (i) parse the data and store it in a more compact format, thus supporting faster access, and (ii) pre-compute intermediate query results that are likely to be reused in multiple executions of the query, thus reducing query latency.

4.1 Data Storage

When processing the data, disk reads/writes are generally the most expensive operations. In the context of very large linked data sets, it is highly likely that the data being processed will not fit in the machine’s main memory and intermediate results need to be stored on the disk and be loaded back into memory later for further processing. Minimizing such read/write operations is therefore one of the main goals of the design of our model. To give a concrete example, suppose that we want to process the following graph pattern

$$?person\ loc:at\ ?loc.\ ?person\ foaf:name\ ?name$$

which joins data from two sources, a basic and common operation in query processing. We also assume that the triple pattern

$$?person\ loc:at\ ?loc$$

comes from a sensor data source that is continuously streaming data and

$$?person\ foaf:name\ ?name$$

comes from a linked data source, where $?person$ is a URI and $?name$ is a text string. The straightforward approach, which is used in current systems such as C-SPARQL, is to load results of these two patterns into a relational stream $(?person, ?loc)$ and into a static relation $(?person\ ?name)$, and execute the query

continuously. Unfortunately, not only the relation (*?person ?name*) contains millions of records but also the size of each record is big. Therefore, such a simple join would generate a large number of read/write operations.

To address this problem we employ *dictionary encoding*, a method which is commonly used by triple stores [1, 14, 12]. Node values such as URIs, blank nodes and literal string values are mapped to integer identifiers. By using this approach, the encoded version of the static relations like the one mentioned above is considerably smaller than the original one, allowing more data to fit into memory. Moreover, dictionary encoding also allows faster processing, since integer comparisons are cheaper.

In the case of the more dynamic data sources, with new records entering the system at high rates, keeping a dictionary might be costly. In particular, if the dictionary cannot be kept in memory, updates will involve read/write operations, which brings us back to the original problem. Therefore, we have to determine whether or not to build dictionaries. In the CQELS model we encode data coming from LOD collections, which are expected to be less frequently updated, and we skip numbers, since they do not provide a significant gain in space.

4.2 Pre-processing of Intermediate Query Results

In continuous query processing, every time a new record that is relevant to the query is inserted in the system, the query has to be re-executed to deliver the new result. For instance, in query **Q2**, whenever a new person enters the location we have to execute the operation that checks if that person has co-authored a paper someone in that same location, and if this is indeed the case, return the new result to the relevant users. During query execution, part of it simply computes the list of co-authors of the user, which is independent of the event of a new person entering the location. For speeding up query processing, our approach identifies sub-queries whose results will not vary during the duration of the continuous query, and pre-computes these results, which can be reused every time the query needs to be executed. In addition, we also build indexes for faster access to the pre-computed partial results.

Detecting the update rate of a collection is an open problem. For the remainder of this paper we will assume that sub-queries involving only LOD collections are quite likely to produce the same results throughout the lifetime of a query. This is a reasonable assumption since a recent study has shown that the majority of datasets in the LOD cloud are in fact rather static [35]. Therefore, for every query registered in the system we pre-compute the graph patterns derived from the LOD sources. For example, in the case of the query mentioned above, the following graph pattern can be pre-computed:

$$?person1 \text{ dc:creator } ?paper. ?person2 \text{ dc:creator } ?paper,$$

which is a self-join on the triple pattern $\{?person \text{ dc:creator } ?paper\}$.

For faster access during query execution, the result of this triple pattern can be stored in a two column relation [1]. For the indexes, we keep B+-trees on the keys that are used in the join operators. Additionally, only two variable bindings, $\langle ?person1, ?person2 \rangle$, of the above graph pattern are used in other parts of the query, thus we can also pre-compute the self-join of $\langle ?person, ?paper \rangle$ and store it in a indexed table as a *frequent path*, similar to [26].

For the more dynamic data, such as Linked Stream Data, the costs of maintaining such indexes might be prohibitively high to justify their use. We are aware that LOD collections can eventually change, but it is reasonable to assume that changes occur at a much slower pace, and the cost of re-computing the intermediate results and re-building the indexes are acceptable. Dataset dynamics characteristics, for example,

annotations with the Dataset Dynamics vocabulary¹⁰ could be exploited to perform an incremental update on pre-computed data, which we plan to explore as future work.

5 Adaptive Cost-based Optimization

In traditional database management systems, the query optimizer typically computes the optimal query plan in the query compiling phase. It uses the distribution of the data at compile-time to define the order of the physical query operators. In contrast, for stream data, this optimizing technique does not yield satisfying results, as the distribution of the data changes during run-time.

In order to illustrate this, we revisit query **Q2** with the example data shown in Figure 3. **Q2** creates two sliding windows, [RANGE 30s] and [NOW], on the location stream *S*. The data from these two windows is shown in Figures 3(a) and 3(b). Further, Figure 3(c) depicts the output mapping set of the *co-author* static graph pattern

?person1 dc:creator ?paper.?person2 dc:creator ?paper.

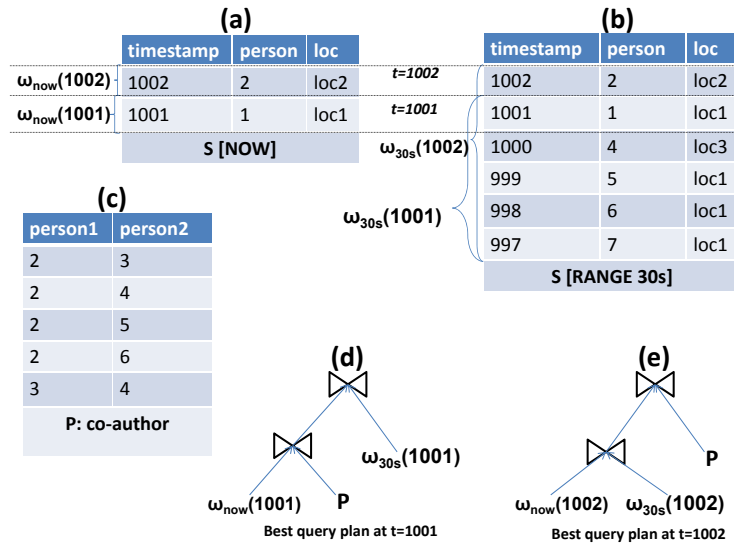


Figure 3: Examples of dynamically optimal query plans.

At time $t=1001$, the best order to execute the query is shown in Figure 3(d), since person ‘1’ does not author any paper. However, at $t=1002$, this same plan will lead to many intermediate results that will not contribute to the final answer. In this case, a better way to execute the query is shown in Figure 3(e). Hence, standard query optimizers are not suitable for stream data processing and we need an algorithm that is able to suggest different query plans at query run time to reflect updates of the data in the data streams. In the following, we introduce our adaptive cost-based optimizing algorithm to find such an optimal query plan at run-time.

¹⁰<http://purl.org/NET/dady>

A typical way to optimize a query is to try to reduce the number of intermediate results produced by physical operators in the query plan by looking at the selectivity and the cardinality of the input data of a physical operator [15, 33]. In the case that the input comes from a data stream these two values are constantly changing, so our algorithm has to adapt to these changes.

Let $card(\Omega)$ be the cardinality of a mapping set Ω , and $\Omega_{x=c} = \{\mu | \mu \in \Omega \wedge \mu(x) = c\}$. The selectivity of the variable x bound to the value c on the mapping set Ω is given by:

$$sel_{x=c}(\Omega) = \frac{card(\Omega_{x=c})}{card(\Omega)}.$$

For each registered query, every time a mapping arrives or leaves the sliding window buffer, the operators consuming that data are notified. Upon this notification, the corresponding cardinality and selectivity values are updated: The values $card(\Omega)$ and $card(\Omega_{x=c})$ are increased or decreased by one, respectively, and consequently, $sel_{x=c}(\Omega)$ is recomputed from these new values.

Each of the physical operators listed in Section 3.2 can be implemented in different ways, each having an associated cost function. Let \otimes be an implementation of a physical operator. Its cost is given by a function $\Phi_{\otimes}(\Omega_1, \Omega_2)$, which depends on the cardinality of the binding variables. The *join* operator, for example, can be implemented as a nested loop join or as an index join with a B-Tree index on the second input. (Ω_2)¹¹ The costs for these two implementations are given below:

- $\Phi_{\otimes}(\Omega_1, \Omega_2) = card(\Omega_1) \times card(\Omega_2)$, if \otimes is a nested loop join
- $\Phi_{\otimes}(\Omega_1, \Omega_2) = card(\Omega_1) \times log_2(card(\Omega_2))$, if \otimes is an index-join with a B-Tree index on Ω_2

We use dynamic programming for searching the optimal query plan similar to System R [5]: Let P_{\emptyset} be the null graph pattern where $[[P_{\emptyset}]] = \emptyset$ (empty mapping set). Let P_1 be a graph pattern and P_2 a graph pattern or a null graph pattern. An evaluation of a graph pattern P can be transformed as

$$[[P]] = [[P_1]] \otimes [[P_2]].$$

The set of transformation rules for a graph pattern P is given by [28] as

$$\mathfrak{S}(P) = \{(P_1, \otimes, P_2) | [[P]] = [[P_1]] \otimes [[P_2]]\}.$$

The optimizing function Ψ is then defined recursively as:

1. $\Psi(P_{\emptyset}) = 0$
2. $\Psi(P) = 0$: P is a static graph pattern
3. $\Psi(P) = 0$: P is a dynamic triple pattern
4. $\Psi(P) = \min\{\psi_{\otimes}(P_1, P_2) | (P_1, \otimes, P_2) \in \mathfrak{S}(P)\}$
 where $\psi_{\otimes}(P_1, P_2) = \Phi_{\otimes}([[P_1]], [[P_2]]) + \Psi(P_1) + \Psi(P_2)$

To avoid recursive re-computation of $\Psi(P)$, we use a bottom-up dynamic programming approach [25] to maintain all $\Psi(P)$ and their associated sub-optimal query plans. Figure 4 shows the pseudo-code of CQELS's adaptive cost-based optimizing algorithm.

As soon as a new mapping arrives ($\mu, +$) or an existing mapping expires ($\mu, -$), both denoted in short as (μ, \pm), it triggers the `updateMapping` function to update the selectivity and cardinality values. For each new update, it calls the function `propagateUpdatePhysicalCost` to update the cost of executing a physical operator $\Phi_{\otimes}([[P_1]], [[P_2]])$ via the function `physicalCost([[P_1]], \otimes, [[P_2]])`. Then, the newly computed $\Phi_{\otimes}([[P_1]], [[P_2]])$ value is passed to the function `triggerUpdateOptimalPlan` to check if it is necessary to update the current $\Psi(P)$ value and optimal plan, respectively denoted as $\Psi(P).value$ and $\Psi(P).plan$.

¹¹These are just two examples. Other implementations are also possible.

```

void updateMapping( $\mu, \pm$ ){
  for( $P : \text{var}(P) \cap \text{dom}(\mu) \neq \emptyset$ ){
     $\text{card}([[P]]) = \text{card}([[P]]) \pm 1$ 
    for( $x \in \text{dom}(\mu)$ ) {
       $\text{card}([[P]]_{x=\mu(x)}) = \text{card}([[P]]_{x=\mu(x)}) \pm 1$ 
       $\text{sel}_{x=\mu(x)}([[P]]) = \frac{\text{card}([[P]]_{x=\mu(x)})}{\text{card}([[P]])}$ 
      propagateUpdatePhysicalCost( $P$ )
    }
  }
}

void propagateUpdatePhysicalCost( $P_u$ ){
  for( $(P_1, \otimes, P_2) : (P_1, \otimes, P_2) \in \mathfrak{S}(P) \wedge (P_1 = P_u \vee P_2 = P_u)$ ){
     $\Phi_{\otimes}([[P_1]], [[P_2]]) = \text{physicalCost}([[P_1]], \otimes, [[P_2]])$ 
    triggerUpdateOptimalPlan( $P, P_1, \otimes, P_2$ )
  }
}

void triggerUpdateOptimalPlan( $P, P_1, \otimes, P_2$ ){
  if( $\Phi_{\otimes}([[P_1]], [[P_2]]) + \Psi(P_1).value + \Psi(P_2).value < \Psi(P).value$ ){
     $\Psi(P).value = \Phi_{\otimes}([[P_1]], [[P_2]]) + \Psi(P_1).value + \Psi(P_2).value$ 
     $\Psi(P).plan = (\Psi(P_1).plan, \otimes, \Psi(P_2).plan)$ 
    if ( $P$  is not root graph pattern)
      propagateSubOptimalUpdate( $\Psi(P)$ )
  }
}

void propagateSubOptimalUpdate( $\Psi(P_{sub})$ ){
  for( $(P_1, \otimes, P_2) : (P_1, \otimes, P_2) \in \mathfrak{S}(P) \wedge (P_1 = P_{sub} \vee P_2 = P_{sub})$ ){
    triggerUpdateOptimalPlan( $P, P_1, \otimes, P_2$ )
  }
}

```

Figure 4: Adaptive cost-based optimization algorithm.

6 Performance and Scalability Evaluation

In order to evaluate our CQELS model, we first built a system to be our baseline, which we named System X, for stream data and LOD processing. System X is implemented along the same lines as C-SPARQL, following the guidelines given by the Linear Road benchmark [4]. We then integrated our CQELS model on top of the base system and measured the performance gain by applying our pre-processing and optimization approaches. We had to provide our own implementation of a baseline system since it was not possible to have access to any implementation of the other related systems. However, this does not pose a problem for the evaluation of our model, since CQELS is designed to be integrated with any system implementation, and the same performance gains as described below are to be expected for other baselines as well.

System X makes use of a triple storage and a DSMS, which are independent of each other, acting as two black boxes. To process a query, System X first splits the query into a static part and a dynamic part. In the query registration phase, the query engine translates the static part into a SPARQL query and then directs it to the triple store to load the binding results into a relational database. In this evaluation, we use MySQL on the same physical computer as the query engine. For the DSMS, we use the ESPER open source implementation¹². ESPER provides a continuous query language to correlate data from the streams with data stored in relational tables, which were generated from the static part. It further supports the use of a least recently used (LRU) cache to store intermediate results from relational tables.

We then integrated our CQELS model, by adding the pre-processing phase and the query optimizer to dynamically change the query plan. For the pre-processing we reused most of the building blocks from ESPER, with some straightforward modifications. We encoded the stream data using the encoding dictionary (cf. section 4.1) before feeding it to the data processing component and modified the projection and join operations of the ESPER system to make use of the indexes in the pre-computed data. From our observation, even for simple queries (like query **Q1**), the large size of the DBLP dataset significantly affects the performance of System X. To speed it up, we also allow System X to use the indexes on the join attributes that were pre-computed before query execution. Moreover, for a fair comparison and to highlight the benefits of the efficient data storage of the input data and our adaptive query optimizing algorithm, CQELS uses the same physical query operators as System X, such as the data flow control and the sliding window operators.

6.1 Setup

We performed the evaluation of the CQELS model using the DBLP dataset provided by L3S¹³. The dataset was obtained in April 2010. It contains 170 million triples about authors and their publications. As dynamic stream data we used an RFID-based tracking data set similar to the Live Social Semantic application [2]. The latter is provided by the Open Beacon community¹⁴ from active RFID tags. We have simulated streams of location data from the data log recorded during the 24C3 conference on July 2007¹⁵. After registering the queries described in Section 3.1, we streamed the locations associated with the URIs of the authors in the DBLP dataset. The experiments were executed on a standard PC with 2 AMD Opteron 250 processors, Ubuntu 9.10/x86_64, 4GB memory, 2 x 1TB storage, and JVM 1.6 with 1GB heap size.

¹²<http://esper.codehaus.org/>

¹³<http://www.l3s.de/web/page25g.do?kcond12g.att1=1029>

¹⁴<http://www.openbeacon.org/>

¹⁵<http://people.openbeacon.org/meri/openbeacon/sputnik/data/24c3/>

We measured the performance of three implementations in terms of query execution time and scalability: (1) System X as the baseline, (2) CQELS only with our pre-processing approach (without query optimization), and (3) CQELS using both our pre-processing and adaptive cost-based optimization algorithm.

6.2 Results

In the first experiment, we measured the performance of the three implementations for the four queries described in Section 3.1. These four queries cover all physical operators introduced in Section 3.2: **Q1** is used to evaluate the simple operation of joining stream data and static triple patterns. This query is also a special case for the query optimizer, since it has only one possible query plan. **Q2** contains typical multi-way joins, and **Q3** includes a UNION with joins between static and stream relations which are commonly used to show the benefits of choosing a good execution plan by reordering join orders. The fourth query, **Q4**, is a more complicated query with an aggregation, a very important operator for stream processing.

We registered each query in the system once for 30 minutes and once for 60 minutes. After registering a query, we streamed the location data at rates of 1000 to 5000 updates per second. For each of these settings we measured the execution time, i.e., the time it takes to deliver the query results. The execution times were then averaged. Results are shown in Figure 5, in a logarithmic scale. In addition, for the CQELS model, we also report on the time spent in the optimizing phase, i.e., the time needed for updating the cardinality of the input data, re-computing the costs of the query plans, and deciding on the optimal plan (shaded areas in Figure 5).

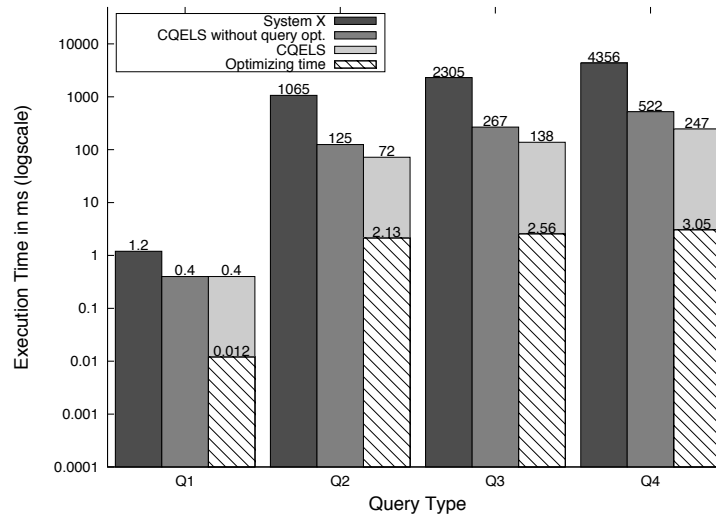


Figure 5: Execution time (in ms) for four different query.

We can see that the pre-processing module of CQELS alone already outperforms System X by a large margin. For **Q1**, the pre-processing phase improves the execution time by 300% on average, compared to System X. As we mentioned earlier, this query is a particular special case for the query optimizer, since there is only one execution plan, therefore the query optimizer does not lead do any further improvement.

For **Q2**, **Q3**, and **Q4**, the query execution with the pre-processing is over 8 times faster. In particular, for **Q3**, the execution time is about 11.5% in average of the original time needed by the baseline. With the addition of the optimization algorithm the execution time for these three queries is roughly cut by half.

Compared to System X, the execution time of query **Q3** with our CQELS model is only 6% of the original time. We can also see that the costs of having an adaptive cost-based query optimizer are low (please keep in mind the logarithmic scale used in Figure 5), and it leads to a higher improvement. This confirms the benefits of the approaches we have suggested.

To measure scalability, we tested how our CQELS model handles the simultaneous execution of multiple queries. We modified our four queries as follows:

- **Q1'**: Notify an author where he or she is located (i.e., return the location's name).
- **Q2'**: Notify an author when someone cited by him or her is in the same location, returning that persons' names.
- **Q3'**: Notify an author when 'interested authors' are in the same location, where 'interested authors' include his co-authors and people he or she cited.
- **Q4'**: Notify an author about the name of the place where the highest number of authors cited by him or her are located.

Figure 6 shows these queries. For each of these queries we varied the number of concurrent query instances in the system from 10 to 5000, each instance corresponding to a different person issuing the query. Figure 7 shows the average execution times of the multiple query instances (on a logarithmic scale). In addition, Tables 2 and 3 show the values obtained. The numbers for **Q3'** and **Q4'** were omitted since they are similar to those of **Q2'**.

No. Queries	System X	CQELS	No. Queries	System X	CQELS w/o opt.	CQELS
10	4.8	1.2	10	302	116	64
100	12.3	1.9	100	778	215	105
200	18.8	3.8	200	892	274	129
300	34.6	4.2	300	956	358	153
500	51.3	7.4	500	1105	484	225
1000	90.4	14.5	1000	3486	872	420
2000	235.3	25.6	1200	5120	961	456
3000	685.3	41.3	2000	-	1523	699
4000	1356	54.9	3000	-	2368	1070
5000	3567	70.1	4000	-	2974	1403
			5000	-	3688	1551

Table 2: Execution times (ms) for **Q1'**.Table 3: Execution times (ms) for **Q2'**.

We can see that our approach scales well with the number of queries, and that the performance of System X degrades quickly. For **Q1**, the average execution time is over 50 times faster with our CQELS model, for 5000 queries running in the system. For the remaining three queries we again observed a significant reduction in the execution times through the CQELS model, with the optimization phase delivering on average a performance twice as fast as having only the pre-processing step in place. Moreover, System X was not able to handle more than 1200 queries, resulting in a heap overflow error.

Q1'

```

SELECT ?name ?locName
FROM NAMED <http://deri.org/locstreams> [RANGE 30 seconds] as ?ls30
WHERE {
  GRAPH ?ls30 {?person loc:at ?loc}.
  GRAPH <http://deri.org/localization/> {?loc loc:name ?locName}
  ?person foaf:name "$Author Name$".
}

```

Q2'

```

SELECT ?citedAuthorName
FROM NAMED <http://deri.org/locstreams> [NOW] as ?lsnow
FROM NAMED <http://deri.org/locstreams> [RANGE 30 seconds] as ?ls30
WHERE {
  GRAPH ?lsnow {<:anAuthor> loc:at ?loc}.
  GRAPH ?ls30 {?citedAuthor loc:at ?loc}.
  ?citedAuthor foaf:name ?citedAuthorName.
  ?p1 dc:creator ?anAuthor. ?anAuthor foaf:name "$Author Name$". ?p2 dc:creator ?citedAuthor. ?p1 dcterms:references ?p2.
}

```

Q3'

```

SELECT ?interestedAuthorName ?locName
FROM NAMED <http://deri.org/locstreams> [NOW] as ?lsnow
FROM NAMED <http://deri.org/locstreams> [RANGE 30 seconds] as ?ls30
WHERE {
  GRAPH ?lsnow {?person1 loc:at ?loc}.
  GRAPH ?ls30 {?person2 loc:at ?loc}.
  GRAPH <http://deri.org/localization/>{?loc loc:name ?locName}
  ?anAuthor foaf:name "$Author Name$".
  ?interestedAuthor foaf:name ?interestedAuthorName.
  {
    {?p1 dcterms:references ?p2. ?p1 dc:creator ?anAuthor. ?p2 dc:creator ?interestedAuthor}
  UNION
  {?p dc:creator ?anAuthor. ?p dc:creator ?interestedAuthor}
}
}

```

Q4'

```

SELECT ?locName max(count(?citedAuthor))
FROM NAMED <http://deri.org/locstreams> [RANGE 15 seconds] as ?ls15
WHERE {
  GRAPH ?ls15 {?citedAuthor loc:at ?loc}.
  GRAPH <http://deri.org/localization/>{?loc loc:name ?locName}
  ?p1 dc:creator ?anAuthor. ?anAuthor foaf:name "$Author Name$". ?p2 dc:creator ?citedAuthor. ?p1 dcterms:references ?p2.
}
GROUP BY ?loc

```

Figure 6: Modified queries for scalability tests.

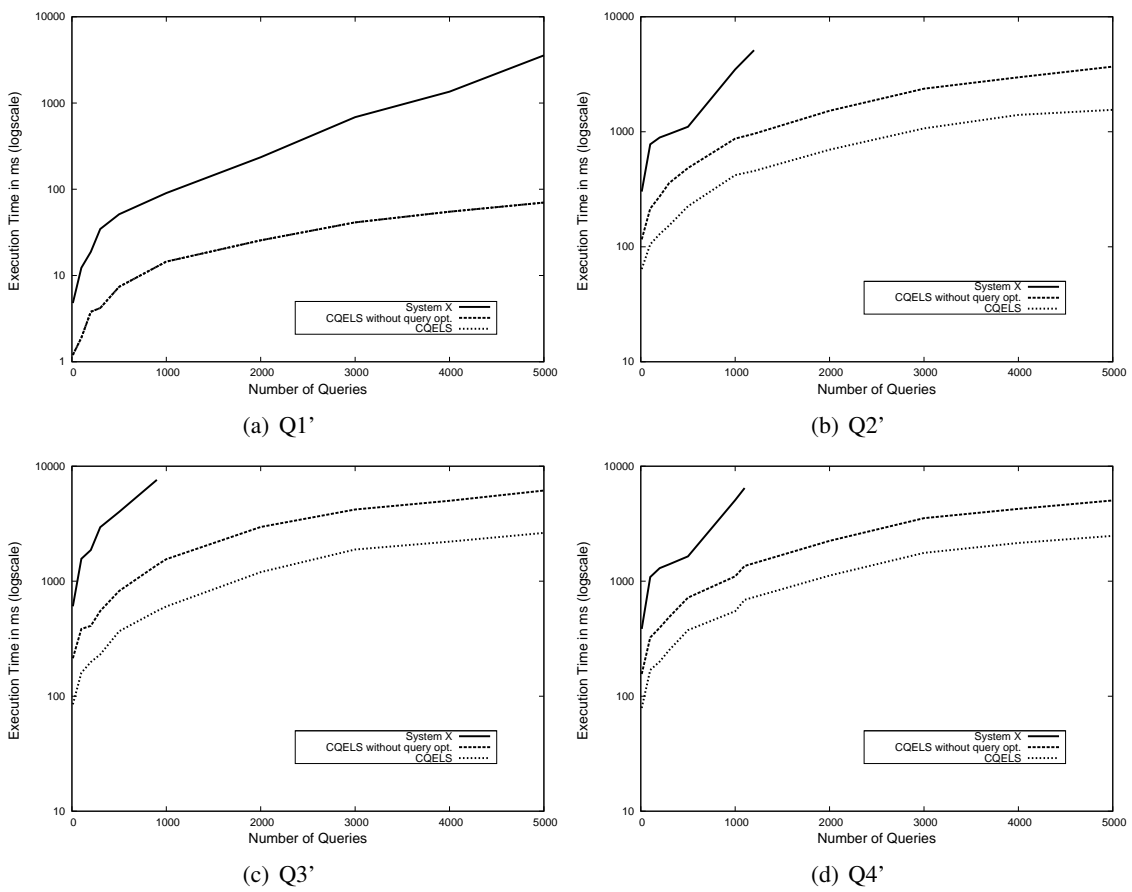


Figure 7: Execution times (in ms) for multiple query instances.

7 Conclusions

In this paper we addressed the problem of scalable query processing over Linked Stream Data integrated with Linked Open Data. We presented our *Continuous Query Evaluation over Linked Streams* (CQELS) approach which provides a scalable query processing model for unified Linked Stream Data and Linked Open Data. Scalability in CQELS is achieved by applying techniques for efficient data storage and query pre-processing, combined with a new adaptive cost-based query optimization algorithm for dynamic data sources, such as sensor streams. The efficient data storage allows more data to fit into main memory, reducing the latency caused by disk read/write operations. The query pre-processing computes intermediate results that are likely to be valid throughout the duration of the query, avoiding unnecessary re-computations. The CQELS query optimizer retains a subset of the possible execution plans and, at query time, updates their respective costs and chooses the least expensive one for executing the query at this given point in time.

Our experimental results show that CQELS can reduce query response times by orders of magnitude compared to related systems while scaling to a realistically high number of parallel queries. The paper also shows how CQELS can enable the easy integration of sensor data with the quickly growing amount of Linked Open Data, facilitating the use of the large body of existing software along with a wide range of novel applications.

As future work we plan to research on how available metadata about the LOD cloud, such as provided by void (Vocabulary of Interlinked Dataset) descriptions¹⁶ along with dataset dynamics characteristics, for example, annotations with the Dataset Dynamics vocabulary¹⁷) could be exploited to determine the update rates of datasets. We also plan to look at adaptive caching mechanisms which can work with our query optimizer. In the context of sensor data, we will pursue a “share nothing” approach which will enable the use of Grid and Cloud computing approaches to scale CQELS to very large numbers of sensors.

References

- [1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *33rd International Conference on Very Large Data Bases*, pages 411–422. VLDB Endowment, 2007.
- [2] H. Alani, M. Szomszor, C. Cattuto, W. V. den Broeck, G. Correndo, and A. Barrat. Live social semantics. In *The Semantic Web - ISWC 2009, 8th International Semantic Web Conference, Proceedings*, pages 698–714, 2009.
- [3] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.
- [4] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: A stream data management benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB)*, pages 480–491, 2004.
- [5] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. Gray, P. P. Griffiths, W. F. K. III, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson.

¹⁶<http://semanticweb.org/wiki/Void>

¹⁷<http://purl.org/NET/dady>

- System r: Relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, 1976.
- [6] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 261–272, 2000.
- [7] M. Balazinska, A. Deshpande, M. J. Franklin, P. B. Gibbons, J. Gray, M. H. Hansen, M. Liebhold, S. Nath, A. S. Szalay, and V. Tao. Data management in the worldwide sensor web. *IEEE Pervasive Computing*, 6(2):30–40, 2007.
- [8] D. F. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus. An execution environment for c-sparql queries. In *EDBT 2010, 13th International Conference on Extending Database Technology*, pages 441–452, 2010.
- [9] 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.
- [10] A. Bolles, M. Grawunder, and J. Jacobi. Streaming sparql - extending sparql to process data streams. In *ESWC'08: Proceedings of the 5th European semantic web conference on The semantic web*, pages 448–462, 2008.
- [11] E. Bouillet, M. Feblowitz, Z. Liu, A. Ranganathan, A. Riabov, and F. Ye. A semantics-based middleware for utilizing heterogeneous sensor networks. In *Distributed Computing in Sensor Systems, Third IEEE International Conference, DCOSS 2007, Proceedings*, pages 174–188, 2007.
- [12] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: An architecture for storing and querying rdf data and schema information. In *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential [outcome of a Dagstuhl seminar]*, pages 197–222, 2003.
- [13] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases*, pages 215–226, 2002.
- [14] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient sql-based rdf querying scheme. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 1216–1227, 2005.
- [15] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems (5th Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
- [16] C. J. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991.
- [17] L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003.
- [18] C. Gutierrez, C. Hurtado, and A. Mendelzon. Foundations of semantic web databases. In *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 95–106, New York, NY, USA, 2004. ACM.
- [19] C. Gutierrez, C. A. Hurtado, and A. A. Vaisman. Introducing time into rdf. *IEEE Transactions on Knowledge and Data Engineering*, 19(2):207–218, 2007.

- [20] A. Harth, J. Umbrich, A. Hogan, and S. Decker. Yars2: A federated repository for querying graph structured data from the web. In *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference (ISWC/ASWC)*, pages 211–224, 2007.
- [21] O. Hartig and R. Heese. The sparql query graph model for query optimization. In *The Semantic Web: Research and Applications, 4th European Semantic Web Conference, ESWC 2007, Proceedings*, pages 564–578, 2007.
- [22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
- [23] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 49–60, 2002.
- [24] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.
- [25] G. Moerkotte and T. Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 930–941, 2006.
- [26] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, 19(1):91–113, 2010.
- [27] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 34(3), 2009.
- [28] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 39–48, 1992.
- [29] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 2003.
- [30] E. Ruckhaus, E. Ruiz, and M.-E. Vidal. Query evaluation and optimization in the semantic web. *TPLP*, 8(3):393–409, 2008.
- [31] J. F. Sequeda and O. Corcho. Linked stream data: A position paper. In *2nd International Workshop on Semantic Sensor Networks (SSN)*, 2009.
- [32] A. P. Sheth, C. A. Henson, and S. S. Sahoo. Semantic sensor web. *IEEE Internet Computing*, 12(4):78–83, 2008.
- [33] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th International Conference on World Wide Web, WWW 2008*, pages 595–604, 2008.
- [34] J. D. Ullman, H. Garcia-Molina, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

- [35] J. Umbrich, M. Hausenblas, A. Hogan, A. Polleres, and S. Decker. Towards dataset dynamics: Change frequency of linked open data sources. In *Proceedings of the WWW2010 Workshop on Linked Data on the Web (LDOW2010)*, 2010.
- [36] E. D. Valle, S. Ceri, F. van Harmelen, and D. Fensel. It's a streaming world! reasoning upon rapidly changing information. *IEEE Intelligent Systems*, 24(6):83–89, 2009.
- [37] M.-E. Vidal, E. Ruckhaus, T. Lampo, A. Martínez, J. Sierra, and A. Polleres. Efficiently joining group patterns in sparql queries. In *The Semantic Web: Research and Applications, 7th Extended Semantic Web Conference, ESWC 2010, Proceedings*, pages 228–242, 2010.
- [38] K. Whitehouse, F. Zhao, and J. Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. In *Wireless Sensor Networks, Third European Workshop, EWSN 2006, Proceedings*, pages 5–20, 2006.
- [39] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient rdf storage and retrieval in jena2. In *Proceedings of SWDB'03, The first International Workshop on Semantic Web and Databases, Co-located with VLDB 2003*, pages 131–150, 2003.