Open Access

Open Journal of Databases (OJDB)
Volume 6, Issue 1, 2019
http://www.ronpub.com/ojdb
ISSN 2199-3459

Provenance Management over Linked Data Streams

Qian Liu[A], Marcin Wylot[A], Danh Le Phuoc[A], Manfred Hauswirth[A,B]

[A]Open Distributed Systems, TU Berlin, Einsteinufer 25 10587 Berlin, Germany,
qian.liu@tu-berlin.de, mwylot@gmail.com, danh.lephuoc@tu-berlin.de
[B]Fraunhofer Institute for Open Communication Systems, Kaiserin-Augusta-Allee 31 10589 Berlin, Germany,
manfred.hauswirth@tu-berlin.de, manfred.hauswirth@fokus.fraunhofer.de

Abstract

Provenance describes how results are produced starting from data sources, curation, recovery, intermediate processing, to the final results. Provenance has been applied to solve many problems and in particular to understand how errors are propagated in large-scale environments such as Internet of Things, Smart Cities. In fact, in such environments operations on data are often performed by multiple uncoordinated parties, each potentially introducing or propagating errors. These errors cause uncertainty of the overall data analytics process that is further amplified when many data sources are combined and errors get propagated across multiple parties. The ability to properly identify how such errors influence the results is crucial to assess the quality of the results. This problem becomes even more challenging in the case of Linked Data Streams, where data is dynamic and often incomplete. In this paper, we introduce methods to compute provenance over Linked Data Streams. More specifically, we propose provenance management techniques to compute provenance of continuous queries executed over complete Linked Data streams. Unlike traditional provenance management techniques, which are applied on static data, we focus strictly on the dynamicity and heterogeneity of Linked Data streams. Specifically, in this paper we describe: i) means to deliver a dynamic provenance trace of the results to the user, ii) a system capable to execute queries over dynamic Linked Data and compute provenance of these queries, and iii) an empirical evaluation of our approach using real-world datasets.

Type of Paper and Keywords

Regular Research Paper: *provenance, Linked Data Streams, dynamic provenance, continuous query, stream data, triplestore, rdf stream*

1 Introduction

"Provenance is information about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability or trustworthiness." [28]. It is a central part of Linked Data management. Systems should be able to both maintain provenance but also interchange it using common vocabularies and data formats. Understanding where and how a piece of data is produced (its provenance) has long been recognized as an important factor in determining the quality of a data item particularly in data integration systems [34]. Thus,

This paper is accepted at the *Workshop on High-Level Declarative Stream Processing (HiDeSt 2018)* held in conjunction with the 41st German Conference on Artificial Intelligence (KI) in Berlin, Germany. The proceedings of HiDeSt@KI 2018 are published in the Open Journal of Databases (OJDB) as special issue.

it is no surprise that provenance has been of concern within the Linked Data community where a major use case is the integration of data sets published by multiple different actors [5].

Heterogeneous environments involve multiple participants and data sources producing erroneous data that are propagated and aggregated. Propagated errors are amplified due to various operations in the data flow, e.g., joins, analytics, recovery, etc. To understand and quantify these errors, we develop techniques to describe how particular pieces of data were produced and combined to deliver query results. These techniques enable to understand the impacts of error propagation on the resulting query answers. Our techniques provide detailed fine-grained provenance trace for queries executed over dynamic Linked Data Streams. It is pivotal for such techniques to work efficiently, with low costs in terms of memory consumption and query execution time. With the added provenance trace, we foresee some performance penalty. State-of-the-art research [13, 4, 37] show that, for static data, we can reduce the overhead to 20%-30% which is considered to be an acceptable cost. Our techniques achieve similarly low overhead for dynamic Linked Data Streams.

Unlike existing approaches, which are applied on static data, we address dynamic data, which are gaining momentum these days in the context of the Internet of Things Sensor Network and Smart Cities. For example, a livestock monitoring system[1], periodically collects data from multiple devices, which are either attached on the bodies of the livestock or deployed all over the grass field. Data from such sources is continuous and unbounded. It is not realistic to store all of the data and then processing them altogether. On one hand,the volume of the data can be too large, on the other hand, even all the data are stored, the time consumptions of processing them can be unimaginable. Therefore, algorithms that handling such dynamicity are more reasonable.

In this paper, we address the following research question: **How can we efficiently compute dynamic provenance trace in the context of Linked Data Streams?** The *continuous provenance polynomial*, i.e., a dynamic provenance trace of the continuous query (i.e., query executed over a long period of time over dynamic data) represents how particular pieces of data were produced and combined to deliver the results. The continuous provenance polynomial has to satisfy two requirements i) it has to be computed efficiently in a continuous fashion along with the execution of the query, ii) it has to show how the query execution process evolves over time.

[1] http://www.cattle-watch.com/

Concretely, the contributions of this paper are:

- definition and description of the provenance polynomial (Section 3);
- system architecture for processing continuous queries and tracing their provenance (Section 4);
- query execution strategies for executing continuous queries and tracing their provenance (Section 5);
- empirical evaluation of our techniques (Section 6).

2 Related Work

Provenance pertains to tracing particular pieces of data throughout the processing pipeline, and has been studied in several fields [9, 8, 33]. Data processing in distributed environments often takes place across heterogeneous systems, yielding the need to exchange provenance information, i.e., how data was combined, recombined, and processed. Various types of provenance information have been semantically formalized in the Open Provenance Model [27]. In the same context, the W3C PROV model [32] has been introduced to standardize a recommendation for the exchange of provenance over the Web. Such provenance models provide a way to describe how data is processed and propagated.

Provenance in Linked Data is often attached to a dataset descriptor [2] that is typically embedded in a Vocabulary of Interlinked Dataset (VoID) [3] file. VoID is an important aspect of Linked Data provenance as it allows to define what constitutes a dataset as well as its associated metadata. Within Linked Data, provenance is attached using either reification [16] or named graphs [7]. In fact, the support for provenance is one reason for the inclusion of named graphs in the latest version of RDF (the Resource Description Framework[2]) [38]. Provenance can also be attached to a triple as an annotation [36, 11]. Formally, these annotated data are represented by algebraic structures such as communicative semirings, which can take the form of polynomials with integer coefficients [14]. These polynomials represent how source tuples are combined through different relational algebra operators (e.g., union, joins). Theoharis et al. provide a comprehensive theoretical foundation for tracing provenance in RDF queries [35]. Provenance approaches in Linked Data have also been used to determine and propagate trust values [15].

Zimmermann et al. [40] proposed to annotate triples with temporal data and a provenance value that refers to

[2] https://www.w3.org/TR/rdf-primer/

the source of the triple. A quadruple takes the form of a statement (Subject, Predicate, Object, Annotation), i.e., a N-Quad. A similar approach is described by Udrean et al. [36] where the authors extend RDF for temporal, uncertain, and provenance annotations. The main focus of this work is to develop a theoretical model to manage such metadata information. In the same context, Nguyen et al. [31] propose to use a singleton property instead of RDF reification or named graphs to describe provenance.

The available implementations of annotated RDF approaches often do not address "how-provenance", i.e., "how" a query result was constructed [40, 36]. Moreover, these implementations have only been applied to small (around 10 million triples), static, and complete datasets focusing on inferred triples and are not aimed at reporting provenance polynomials [14] (i.e., algebraic structures representing how data is combined) for SPARQL query results.

None of the approaches described above specifically targets dynamic Linked Data Streams. As they were designed for static data they do not take into account dynamicity of input streams and they do not allow to execute continuous queries over such streams. Moreover, due to the employed storage models (multiple indices and provenance annotations) their performance deteriorates in case of dynamic data. In this project, we specifically investigate the challenge in provenance management: dynamicity.

3 CONTINUOUS PROVENANCE POLYNOMIAL

Our goal is to provide a dynamic provenance trace of queries, i.e., a continuous provenance polynomial. We start with a definition of a continuous provenance polynomial for continuous queries executed over Linked Data Streams. However, in contrast to the previous work we focus on practical realization of this provenance polynomial over dynamic Linked Data.

In our system we use two operators to express the provenance polynomial:

- $\oplus$ to represent unions of elements;
- $\otimes$ to represent joins between elements.

We use $\oplus$ when a triple pattern of a query can be satisfied by multiple triples possible from multiple sources of data streams. $\otimes$ is used to express join between few triple patterns. The join operation can be performed either within one data stream or between many sources of data. To capture the dynamic nature of data we add a corresponding timestamp to each element within the provenance polynomial.

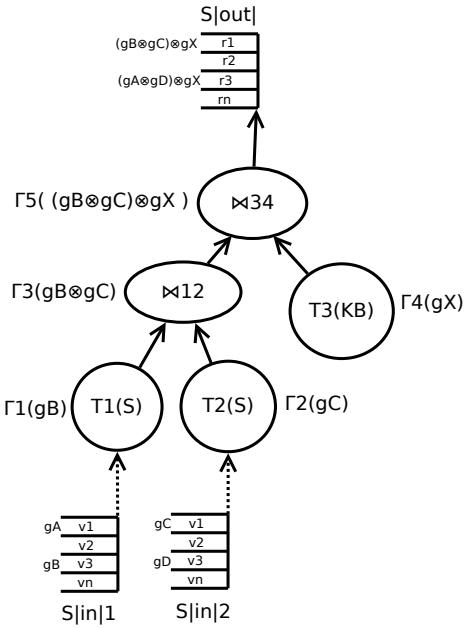


Figure 1: A provenance polynomial is computed dynamically along with the query execution. At each stage of the data propagation we have information on how the system derived the current state.

To better understand the process of computing the continuous provenance polynomial, we introduce an example describing an expected result of this task.

Let Sys be a system that consists of a knowledge base with data on drivers and companies they work for (*KB=(driver1, worksFor, company1),(driver2, worksFor, company1),(driver3, worksFor, company2), (driver14 worksFor, company1)*). Each driver is equipped with a device streaming his position (triples $v_n = (driver_n, position, location_i)$). $S|in|_n$ denotes the input stream for the driver n. The provenance information of a triple in the stream is including: device id, physical state of the device, configuration parameters, etc. The provenance of the triple is grouped under a provenance annotation expressed with a named graph g_j. We want to notify two truck drivers working for the same company when they are within a given distance, e.g., 1km. We also want to be able to trace back how the notification was generated, i.e., which pieces of data were involved to produce the notification.

Figure 1 shows a query execution flow. The figure shows how our techniques trace provenance of continuous queries in parallel to a query execution. To detect spatial locations we use a query $T_n = (?driver_n; detectedAt; ?location_i)$. Let Γ_n be the result of such a query for the n^{th} driver. We consider two data streams for two drivers hence, we obtain two

positions Γ_1 and Γ_2. Along with the intermediate results, we trace the provenance of the involved data, gB and gC, which is the first step in our provenance computation. Next, we join the two intermediate results ($\Gamma_3 = \Gamma_1 \bowtie \Gamma_2$), where $\bowtie$ stands for a natural join, to obtain information on the proximity of the two drivers. Here, we incorporate the provenance computation to keep traces this join operation, which for Γ_3 is described with the polynomial $gB \otimes gC$ ($\otimes$ denotes a provenance join operation). Subsequently, over the knowledge base KB, we evaluate a triple pattern (query) $T_3 = (?driver_n, worksFor, ?company_j)$ to retrieve information about drivers and companies they work for. In addition to the obtained result Γ_4 the KB also output a provenance trace denoted by gX. Finally, we join Γ_3 and Γ_4 in order to produce a notification. At this point we compute provenance of the notification (final result). In our example provenance of the final result is described with the following polynomial: $(gB \otimes gC) \otimes gX$.

In the following sections, we describe our novel system architecture and query execution strategies that allow us to compute this continuous provenance polynomial in parallel to query execution. Our strategies enable tracing provenance at multiple levels starting from the source of data to the final result. They provide knowledge of provenance at every stage of query execution hence, each processing node has full information on how particulate piece of data was derived.

4 ARCHITECTURE

This section provides an overview of our system and it describes how different modules cooperate. Our system adopts a fault-tolerant decentralized peer-to-peer based cluster membership service, Akka Cluster [24], which is based on gossip protocols [10, 12]. This decentralized architecture reduces the risks of single point failures and bottlenecks. The peer-to-peer gossip based nature of the cluster endows high flexibility in adding and removing nodes, i.e., the size of the cluster can be dynamically increased or decreased.

Cluster nodes play three roles: User Interface (UI), Query Execution Router, and Executor. They correspond to three modules of our system as depicted in Figure 2. The UI module is responsible for the interactions with end users, i.e., receiving SPARQL queries from a client and collecting the results for the client. After receiving a query, the UI forwards it to the Query Execution Router which dispatches the query to the Executor to run.

Each instance of the Executor module is created dynamically every time the Query Execution Router receives a SPARQL query. This instance is dispached to execute on cluster Executor node with the lowest loads based on the Adaptive Load Balancing metrics information [22] which is collected by the Query Execution Router. The load balancing metrics data is computed on each of the Executor nodes according to 1) JVM heap memory usage, 2) average system load for 1 minute, and 3) CPU usage. The run-time in-use load balancing metrics can be any combinations of 1) to 3), which is configurable.

When instances of Executor module are created and dispached onto cluster Executor nodes, they begin to execute queries with a stream based pipeline processing mechanism, Akka Stream [23]. Queries are executed over a continuous Linked Data stream from an external distributed message queue, Kafka [21].

One partition inside the queue corresponds to one source of stream data. We feed queue partitions with messages containing several RDF triples, and the size of each message can be configured. That is, we process micro batches from heterogeneous data streams.

The typical constitution of our system is one UI node with one instance of the UI module deployed, one Query Execution Router node with one instance of the Query Execution Router module deployed, and multiple Executor nodes with a number of instances of the Executor module deployed for each, i.e., the Executor nodes can be n-ary relationship with instances of the Executor module. Each Executor node can be deployed with multiple instances of the Executor module.

We implement all three modules following the actor paradigm [17]. Our rationale behind adopting the Actor Model [17] are:

1. The asynchronous nature of actor communications provides better throughput. Messages are sent to message queues of actors, therefore, the system does not have to wait for actors to accomplish their job. For example, UI does not have to wait for Query Execution Router dispatching queries to the Executors. Query Execution Router can accept new queries from the UI while Executors execute the queries.

2. The Location Transparency [39, 25] provides flexibility to change the physical setup of a cluster. There are two levels of identifications for each actor, the Logical ID and the Physical ID, where Logical ID only conveys relative locations between different actors, whereas, Physical ID adds information about machines, protocols, etc. To identify different actors, we only need to use their Logical ID. Hence, the identifications of actors are independent from their concrete physical deployments. In practice, it means that all modules

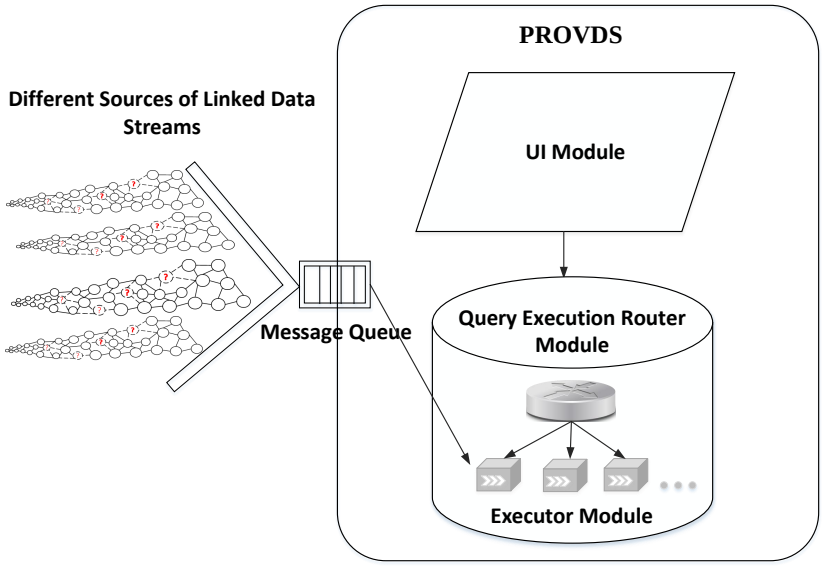


Figure 2: The generic architecture of our continuous provenance polynomial system

can be transparently re-deployed within the cluster, which enables dynamic growing and shrinking of the physical infrastructure.

5 Query Execution

In this section we describe the implementation details of the continuous query provenance polynomial which is related to the Query Execution Router module and Executor module as shown in Figure 3. The Query Execution Router module is responsible to instantiate and deploy the Executors and accordingly forwards the received query strings. The Executor module implements a query execution pipeline. The query execution pipeline enables continuous execution of SPARQL queries over the Linked Data Stream to return the query results, along with a provenance polynomial, i.e., a description what and how pieces of data were combined to derive the results. We will elaborate this query execution pipeline in the following section.

5.1 Query Execution Pipeline

Our techniques leverage Akka Stream [23] to implement the query execution pipeline so that it can handle time sliced Linked Data (Linked Data Streams).

Figure 4 depicts the different transformations over incoming Linked Data stream that our system performs in order to execute a query. We distinguish 3 different transformations throughout the pipeline:

1. **Combining Transformation**. We feed our system with multiple external Linked Data streams (Figure 3) via a distributed message queue [21], as explained in Section 4. One partition of the queue processes messages from one stream. In this transformation, triples (formatted with JSONLD [26]) from different streams are combined into one single flow of Linked Data. Besides, the interval and duration of the time slicing are configured in this step after the combination. For example, before the execution of the pipeline, we set the interval to 5 seconds and make the execution to last 10 minutes, which means every 5 seconds the pipeline executor fetches one message of triples from each external stream (from the distributed message queue) and accordingly combines these messages of triples into one flow of Linked Data. Afterwards, this single flow of Linked Data is passed through to the next two steps of the pipeline. The pipeline executor repeats this process until it expires the pre-set 10 minutes, i.e., it consumes 120 messages through the sequence of pipeline transformations.

Very often, the query execution and provenance tracing in the next two steps can not complete on time when the time interval set in the first step is due. This happens especially when time interval is set small whereas volume of data to be processed inside the message is large. However, it does not exist an universally applicable time interval configuration in practice, since the changing volume of the message data. Therefore, in our system, we adopt a back pressure [1] handling

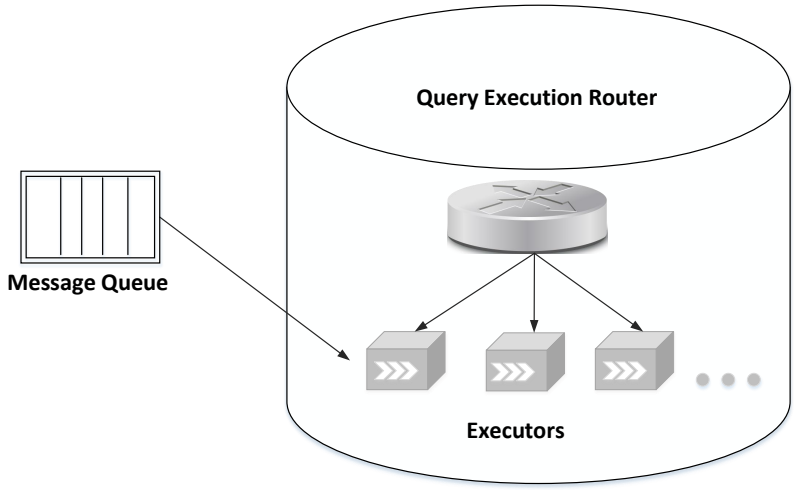


Figure 3: Query execution router and executor modules

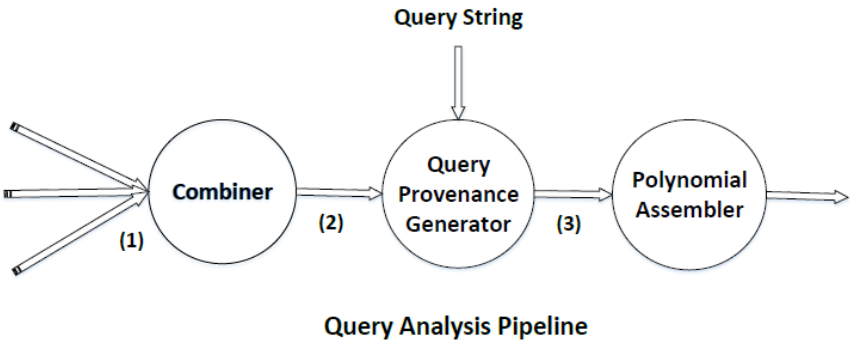


Figure 4: Query execution pipeline at executor nodes

strategy, in which slow consumer (each of the next two steps of the pipeline) signals fast producer (one step before current executing one) to require more messages of data when they finish current executions. Besides, the fast producer buffers new messages in a bounded manner until more demand is signaled by the slow consumer. This refers to a pull-based back pressure handling strategy, which applies when generation speed of the producer overpasses processing speed of the consumer during the pipeline execution. This signaling-based methodology is also applied to other situations, such as slow-producer-fast-consumer, speed of the producer and consumer matches without a delay of producer to emit new messages whenever they are available.

2. **Query Execution Transformation** implements the core algorithm to continuously execute queries and to trace provenance. We adopt Apache Jena ARQ [18] to facilitate the query execution and query provenance computation. A SPARQL query in ARQ undergoes several stages of processing [19] as below:

   (a) Parsing SPARQL query string to a Query object

   (b) Translating the Query object to a SPARQL algebra S-Expression (SSE) [20]

   (c) Establishing in-memory SPARQL algebra parsing tree (the query plan) from the SSE expression

   (d) Evaluating the query plan (traversing the in-memory SPARQL algebra parsing tree and accordingly matching the relating query pattern with the streamed Linked Data graphs)

   Our query provenance tracing algorithm extends the algorithm in the last processing stage (stage

(d)), which is further demonstrated in Algorithm 1. Algorithm 1 returns the results of the query and its provenance. This provenance is later used as input for the next transformation to assemble the provenance polynomial.

3. **Provenance Polynomial Assembling Transformation**. This transformation transforms the $queryProvenanceMap$ from last step to generate a provenance polynomial. It converts the pairs inside the map to the sequence of provenance information connected by the two polynomial operators, ($\otimes$ and $\oplus$).

Our system repeats executing this query pipeline over the Linked Data Stream to generate consecutive query provenance polynomials. In the following section, we will present our different implementation strategies for the query execution pipeline.

5.2 Implementation Strategies

We implement the query execution pipeline with both synchronous and asynchronous mechanisms, as shown in figure 5. Although asynchronism, message of previous time slice will never be processed later than message of current time slice, i.e., our asynchronous implementations still preserve the time sequence of messages of Linked Data graphs during the execution of the pipeline. In the pipeline, we choose 2 pointcuts to integrate asynchronism. The first pointcut is at all the three steps of the pipeline. The asynchronous strategy enables these steps to execute in parallel with multiple threads. The second pointcut is at step (2) (Algorithm 1) of the pipeline, where we implement the asynchronous strategy of query pattern matching for all the Basic Graph Patterns (BGPs). We implement both the synchronous and the asynchronous strategies at the two pointcuts respectively, which means we totally bring 4 implementation strategies into effect. And next, we will further interpret the 4 different implementation strategies as followed.

CoreSyncQESync This strategy is the vanilla version for the pipeline implementation. It leverages a single thread to undertake the execution of the synchronous implementation strategy for both pointcuts. During the execution of the pipeline, all the three steps of transformations have to wait for each other, i.e., step (1) can not start combining new arriving messages of Linked Data graphs until step (3) accomplishes generating polynomials for current message. Besides, Step (2) executes all the query patterns (from the SPARQL algebra parsing tree of Algorithm 1) in sequence. Here "Core" stands for the pipeline implementation, "QE, Query Execution", is the algorithm implementation for Basic Graph Pattern (BGP) matching in step (2) Algorithm 1.

In this sequential implementation strategy, the SPARQL query execution and the provenance tracing strictly follow the time sequence of the arriving messages of Linked Data graphs. Besides, it starts processing the new arriving message only after it finishes processing the old one.

CoreSyncQEAsync This implementation strategy differs from the CoreSyncQESync strategy only with the asynchronous implementation of the BGP matching algorithm. In this strategy, BGP matchings are executed in parallel with multiple threads. If BGP includes many Triple patterns, all the Triple patterns are simultaneously evaluated as well by many threads. For example, the evaluations of the left and right sub-trees of a JOIN query pattern, each sub-tree is a GRAPH-BGP, the evaluation for each of them in this implementation strategy is asynchronous. However, the execution of the whole pipeline is still step-by-step.

CoreAsyncQESync This implementation strategy implements the non-blocking executions for the whole pipeline. However, BGP matchings in this strategy is still executed in sequence. During the execution process of the whole pipeline, when the new message arrives, step (1) can directly start processing with a new thread and not necessarily wait for finish of step (3).

CoreAsyncQEAsync It implements both the pipeline and the BGP matchings in asynchronism, i.e., both the executions of all the three steps of the pipeline and the evaluations of the BGPs are executed in parallel without waiting.

In the following section, we will further evaluate these 4 different implementation strategies with experiments.

6 Experimental Evaluation

To empirically evaluate our system we have conducted 4 experiments. Firstly, we compare four implementations of our system and differ, then we compare the best of our implementations with two state-of-the-art triplestores: Virtuoso[3] and Blazegraph[4]. Finally, we evaluate scalability of our system based on different dataset sizes and we break down the pipeline executions to find the areas for improvements.

The datasets, queries, scripts, all results, and the source code are available on the project website.[5].

Hardware Platform: All experiments were executed on 6 Dell PowerEdge R810 servers with Xeon E7-2830 processors (2 CPUs/server, 8 cores/CPU), 64GB of

[3] https://virtuoso.openlinksw.com/
[4] https://www.blazegraph.com/
[5] http://www.ods.tu-berlin.de/PROVDS

Algorithm 1: Query provenance tracing algorithm

Input: $combinedDataset = \{graph_1, graph_2, graph_3, ...graph_n\}$: The set of Linked Data graphs after combination, in which $graph_1, graph_2, graph_3, ...graph_n$ are Linked Data graphs from distinct sources

Input: $parsingTree = \{queryPattern_1, queryPattern_2, queryPattern_3, ...$ $queryPattern_n\}$: The SPARQL algebra parsing tree (query plan) established from the SSE expression; set of query patterns

Output: queryResults, queryProvenanceMap

```
1  begin
2      queryResults ⟵ NULL
3      queryProvenanceMap ⟵ ∅
4      queryPattern ⟵ queryPattern_1
                          /* traverse the parsing tree from bottom to top */
5      while queryPattern still inside ParsingTree do
6          gt = graph_1
               /* matching current query pattern with all the Linked Data graphs
              inside combinedDataset */
7          while gt still inside combinedDataset do
8              if queryPattern is NOT JOIN or UNION then
9                  if match(queryPattern, gt) is NOT EMPTY then
10                     queryProvenanceMap.add("time stamp of current combinedDataset+
                       queryPattern" : [matched triples and source of gt])
11                 end
12             end
13             else
14                 if match(queryPattern, gt) is NOT EMPTY then
15                     leftPattern = queryPattern.left()
16                     rightPattern = queryPattern.right()
17                     leftResults = queryProvenanceMap.lastFind(leftPattern).value()
18                     rightResults = queryProvenanceMap.lastFind(rightPattern).value()
19                     queryProvenanceMap.add("time stamp of current combinedDataset +
                       queryPattern" : [matched triples, leftPattern, leftResults, rightPattern,
                       rightResults and source of gt])
20                 end
21             end
22             gt = next graph in combinedDataset
23         end
24         if queryPattern is the TOP of ParsingTree then
25             queryResults = queryProvenanceMap.find(queryPattern).value()
26         end
27         queryPattern = next query pattern in ParsingTree
28     end
29 end
```

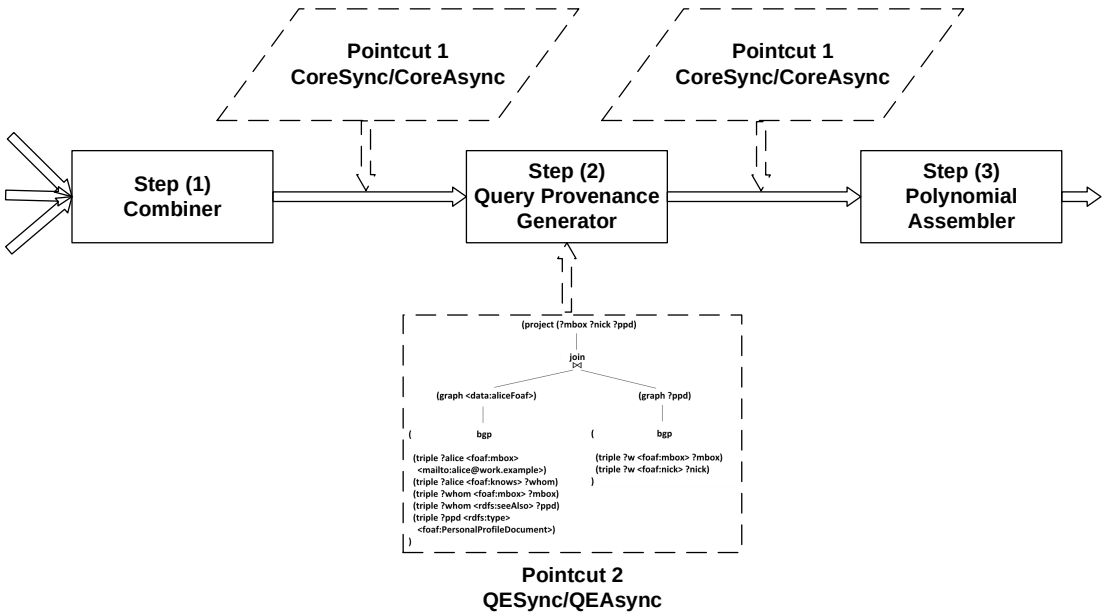


Figure 5: Implementation strategies for 2 pointcuts of the query execution pipeline (Core - pipeline implementation; QE - query execution, implementation for algorithm of basic graph pattern (BGP) matching)

DDR3 RAM, 4TB 2.5" SATA SSD per server, running Ubuntu Server 16.04.3 LTS on each server.

Datasets: We used two different datasets for our experiments: the Billion Triples Challenge (BTC)[6] and the Web Data Commons (WDC)[7][29]. Both datasets are collections of RDF data gathered from the Web. They represent two very different kinds of RDF data. The Billion Triple Challenge dataset was created based on datasets provided by Falcon-S, Sindice, Swoogle, SWSE, and Watson using the MultiCrawler/SWSE framework. The Web Data Commons project extracts all Microformat, Microdata and RDFa data from the Common Crawl Web corpus and provides the extracted data for download in the form of RDF-quads or CSV-tables for common entity types (e.g., products, organizations, locations, etc.).

We consider 100 million triples for each dataset (11GB). To prepare the data, we firstly transformed the datasets to be stored from NQuads [6] to triples of JSONLD, stored the sources information with folders. We split the datasets into multiple files distributed among folders. Each file corresponds to one message and each folder corresponds to one data source streaming messages.

Workload Queries: We take two distinct sets of workload queries into consideration. For BTC, we use 8 existing queries originally proposed in [30]. Based on the queries used for BTC dataset, we constructed 7 new queries for WDC dataset, encompassing different kinds of typical query patterns for RDF, such as star-queries of different sizes and up to 5 joins, object-object joins, object-subject joins, and triangular joins. We also included UNION and OPTIONAL clauses in 2 of the 7 workload queries. We set the time interval for continuous queries to one second.

Experimental Methodology: In the following, we report the average execution time of the 10 query runs. We measured execution time of the query execution pipeline, total execution time of all the 3 steps, and execution time of each of the 3 steps of the the pipeline.

6.1 Experiment 1

We implemented four versions of our system to explore the differences between synchronous and asynchronous executions at different stages in our query execution pipeline. In this experiment scenario, the query execution time includes all 3 steps of the query execution pipeline.

Figure 6 and 7 show query execution time of different strategies for different workload queries. The asynchronous execution of the SPARQL algebra parsing tree has higher impact on the performance than the pipeline strategy. For all the workload queries over both BTC and WDC datasets, the

[6] https://km.aifb.kit.edu/projects/btc-2012/
[7] http://webdatacommons.org/

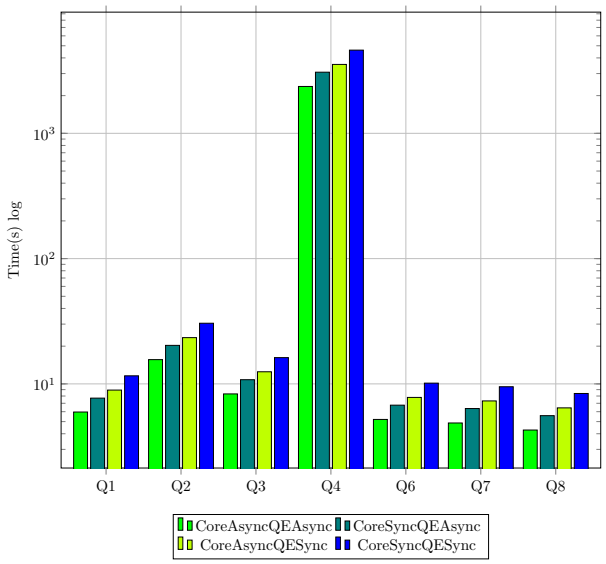


Figure 6: Experiment 1: Query execution time for our four strategies and BTC dataset, logarithmic scale, Query 5 timed out

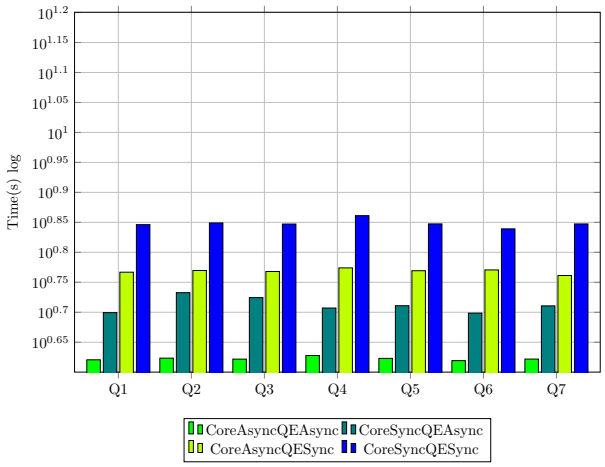


Figure 7: Experiment 1: Query execution time for our four strategies and WDC dataset, logarithmic scale

CoreAsyncQEAsync and CoreSyncQEAsync strategies performs 1.4 to 1.6 times faster than the corresponding CoreAsyncQESync and CoreSyncQESync strategies. By contrast, the asynchronous pipeline execution strategies, CoreAsyncQESync and CoreAsyncQEAsync, are 1.2 to 1.4 times more efficient than the corresponding CoreSyncQESync and CoreSyncQEAsync strategies. The reason for this is that the execution of the SPARQL algebra parsing tree accounts for a largest proportion of the execution time of the Query Execution Transformation which is the most expensive one out of our three pipeline steps. In consequence, the fully asynchronous strategy, i.e., CoreAsyncQEAsync performs the best and the CoreSyncQESync strategy is the slowest.

6.2 Experiment 2

In this scenario we compare our methods with Virtuoso and Blazegraph. To exhibit the exact cost of tracing provenance we evaluate two variants of out techniques with and without tracing provenance. Both of them base on CoreAsyncQEAsync implementation strategy, since it is the fastest one. Neither Virtuoso nor Blazegraph is a stream processing system, i.e., both are standard triplestores. Therefore, we simulate continuous query processing for these systems. We load all the data and we repeatedly execute a SPARQL query with the same time interval as for our system, i.e., 1 second. In order to focus this experiment on the query execution process, for Virtuoso and Blazegraph we count only time of the query execution without transferring and displaying the results. For our system we also count only the actual query execution, i.e., Query Execution Transformation (see Section 5).

As shown in Figure 8 and 9, for both datasets, our technique without provenance tracing (CPP-QueryOnly) is as fast at the baselines. More importantly, our system with the provenance tracing (CPP-PROV) introduces only 15% overhead which is significantly less than the overhead introduces by other state-of-the art provenance enabled systems (see Section 1).

6.3 Experiment 3

The goal of this experiment scenario is to evaluate how our system behaves with increasing data size. We begin with 10 million triples and we finish with the full dataset (100 million triples). In this experiment scenario we use both BTC and WDC datasets, the query execution time includes all 3 steps of the query execution pipeline. Due to the space limitations below we present only selected queries, all results are available online[8].

Figure 10 and Figure 11 show the execution time of query 1 to 8 (for BTC, in which query 5 reached time-out) and query 1 to 7 (for WDC) respectively, for different implementations of our system and over changing volume of consumed datasets. The overall conclusions in terms of comparing our four versions of the systems are similar as for Experiment 1. As expected, the fully asynchronous version of our system (CoreAsyncQEAsync) outperforms the other versions. Moreover, we can also say that all versions of our

[8] http://www.ods.tu-berlin.de/PROVDS

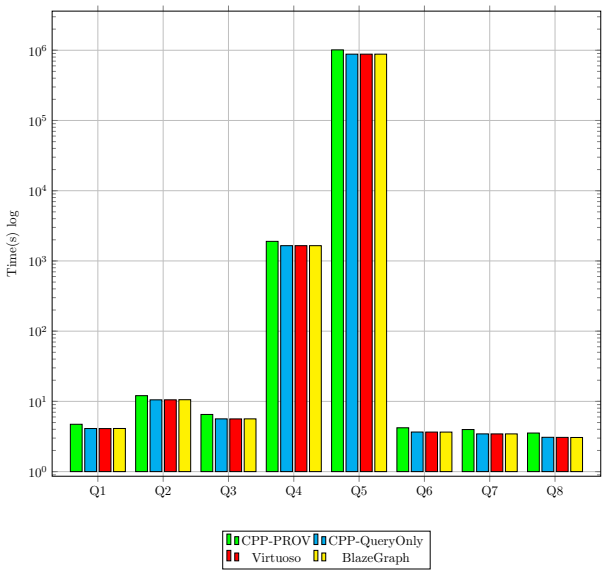


Figure 8: Experiment 2: Query execution time for Virtuoso, Blazegraph, and our CoreAsyncQEAsync with and without tracing provenance, BTC dataset, logarithmic scale

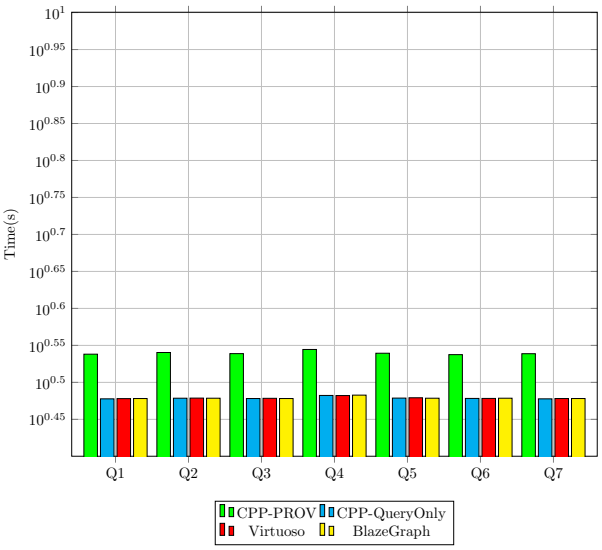


Figure 9: Experiment 2: Query execution time for Virtuoso, Blazegraph, and our CoreAsyncQEAsync with and without tracing provenance, WDC dataset, logarithmic scale

system, hence the entire architecture, scale linearly with the data size.

6.4 Experiment 4

This experiment scenario breaks down the query execution process into four transformations (Section 5) and it measures time that the system spends on each of the transformations, for our fully asynchronous implementation, i.e., CoreAsyncQEAsync. In this experiment scenario we use both datasets, however, the execution time of the query 5 for BTC took more than 10 days, so we aborted it.

Table 1 and 2 show that a large proportion of query execution time (80%) is spent on the Query Execution Transformation (Section 5), since this transformation is the most complex from the algorithmic point-of-view. It includes graph pattern matching algorithm, parsing tree traversal algorithm, and provenance computation algorithm.

7 Conclusions

This research probes into a stream based query provenance tracing system to generate continuous provenance polynomial for the SPARQL queries over Linked Data Streams. It adopts an asynchronous stream processing framework to introduce asynchronous mechanism into our system. We implement an asynchronous version of SPARQL algebra parsing tree traversing algorithm, however, it still includes many iteration based algorithms. It is valuable in future to implement a complete stream transformations (e.g., map, flatmap) driven SPARQL query execution and analysis engine to take full advantages of the asynchronous query execution. However, it unavoidably brings more complexities to handle concurrency issues, which possibly needs more advanced concurrent data structure to process the dynamic Linked Data. Furthermore, currently our system only considers complete Linked Data Streams. In future, we are going to integrate incompleteness into our system with the capabilities to recover missing data.

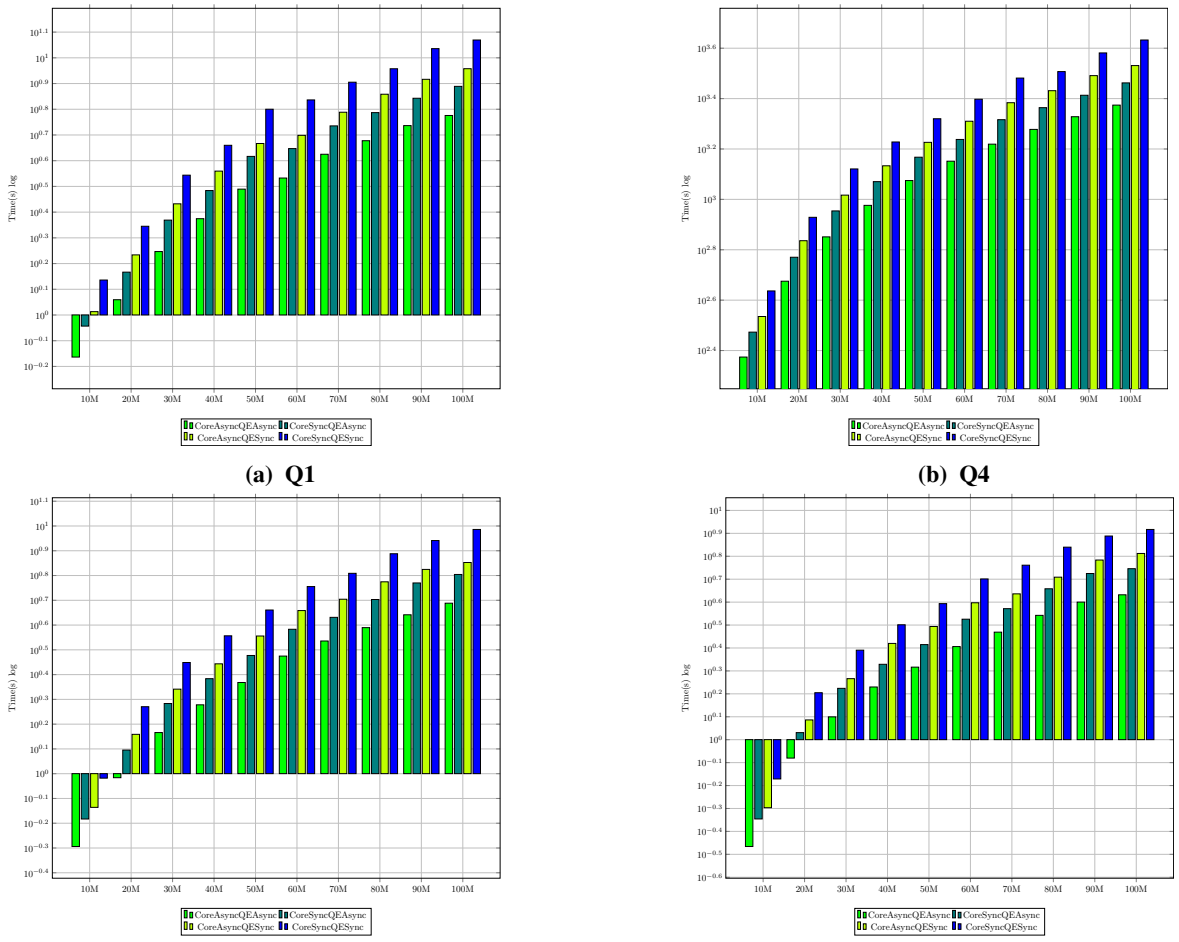
Acknowledgements

This research is part of the PROVDS project, which is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 323223507.

References

[1] L. Akka, “Back Pressure Problems,” 2017. [Online]. Available: http://doc.akka.io/docs/akka/current/java/stream/stream-flows-and-basics.html#back-pressure-explained

[2] Alasdair J.G. Gray, Michel Dumontier, M. Scott Marshall, Joachim Baran, “Dataset descriptions: Hcls community profile,” 2014.



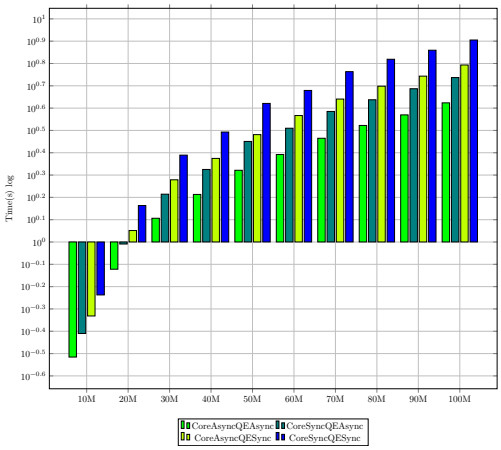
(a) Q1 (b) Q4

(c) Q7 (d) Q8

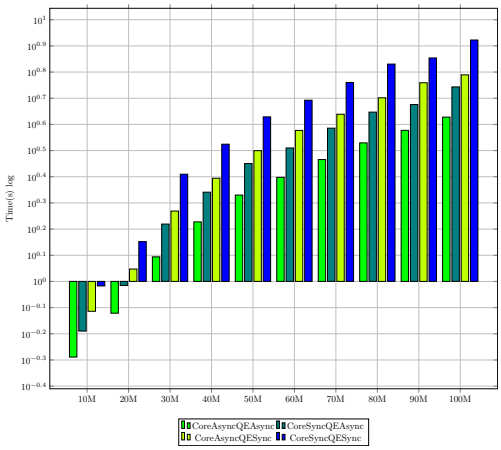
Figure 10: Experiment 3: Query execution for our strategies with increasing size of data for WDC dataset, logarithmic scale

Table 1: Experiment 4: Execution time for each transformation in the query execution pipeline for the fully asynchronous implementation of our system, i.e., (CoreAsyncQEAsync) for BTC dataset

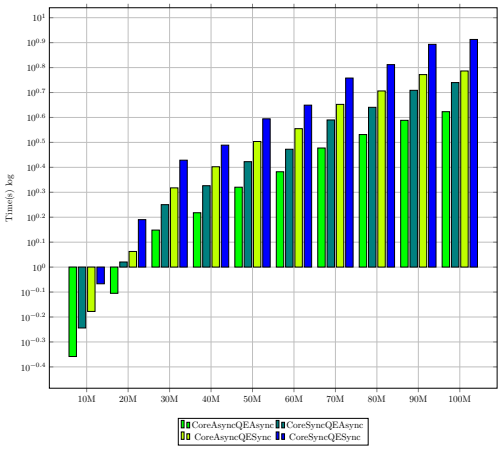
	Step (1)	Step (2)	Step (3)	Total
Q1	0.798	4.722	0.443	5.963
Q2	2.342	12.063	1.215	15.62
Q3	1.175	6.517	0.624	8.316
Q4	302.405	1900.292	163.798	2366.496
Q6	0.632	4.208	0.363	5.203
Q7	0.580	3.972	0.323	4.875
Q8	0.480	3.543	0.26	4.282



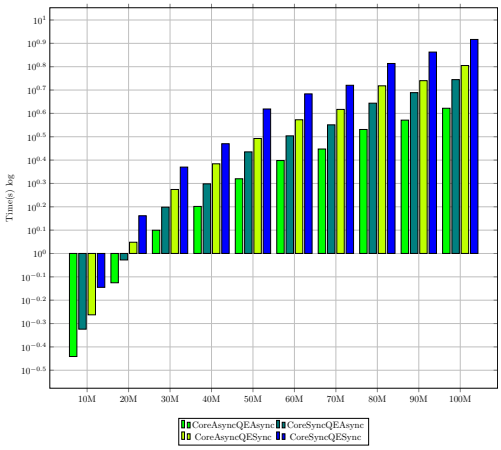
(a) Q2



(b) Q4



(c) Q5



(d) Q7

Figure 11: Experiment 3: Query execution for our strategies with increasing size of data for WDC dataset, logarithmic scale

Table 2: Experiment 4: Execution time for each transformation in the query execution pipeline for the fully asynchronous implementation of our system, i.e., (CoreAsyncQEAsync) for WDC dataset

	Step (1)	Step (2)	Step (3)	Total
Q1	0.469	3.451	0.253	4.173
Q2	0.475	3.469	0.256	4.2
Q3	0.472	3.457	0.255	4.184
Q4	0.480	3.502	0.26	4.242
Q5	0.477	3.461	0.258	4.196
Q6	0.464	3.446	0.25	4.16
Q7	0.473	3.456	0.256	4.185

[Online]. Available: https://www.w3.org/2001/sw/hcls/notes/hcls-dataset/

[3] K. Alexander and M. Hausenblas, "Describing linked datasets-on the design and usage of void, the'vocabulary of interlinked datasets," in *In Linked Data on the Web Workshop (LDOW 09), in conjunction with 18th International World Wide Web Conference (WWW 09*, 2009.

[4] B. Arab, D. Gawlick, V. Radhakrishnan, H. Guo, and B. Glavic, "A generic provenance middleware for queries, updates," in *6th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2014)*, Cologne, Jun. 2014.

[5] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data - the story so far," *International Journal on Semantic Web and Information Systems*, pp. 1–22, 2009.

[6] G. Carothers, "Rdf 1.1 n-quads," 25 February 2014. [Online]. Available: https://www.w3.org/TR/n-quads/

[7] J. J. Carroll, C. Bizer, P. Hayes, and P. Stickler, "Named graphs, provenance and trust," in *Proceedings of the 14th international conference on World Wide Web*, 2005, pp. 613–622.

[8] D. Ceolin, P. Groth, W. R. Van Hage, A. Nottamkandath, and W. Fokkink, "Trust evaluation through user reputation and provenance analysis," in *Proceedings of the 8th International Conference on Uncertainty Reasoning for the Semantic Web*, ser. URSW'12, Aachen, Germany, Germany, 2012, pp. 15–26.

[9] J. Cheney, L. Chiticariu, and W.-c. Tan, "Provenance in databases: Why, how, and where," *Foundations and Trends in Databases*, vol. 1, pp. 379–474, 01 2009.

[10] Devavrat Shah, "Gossip algorithms," *Foundations and Trends in Networking, Massachusetts Institute of Technology*, vol. 3, no. 1, p. 1–125, 2009.

[11] G. Flouris, I. Fundulaki, P. Pediaditis, Y. Theoharis, and V. Christophides, "Coloring rdf triples to capture provenance," in *International Semantic Web Conference*, 2009, pp. 196–212.

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels, "Dynamo: Amazon's highly available key-value store," in *SOSP'07*, October 14–17, 2007.

[13] B. Glavic and G. Alonso, "The perm provenance management system in action," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, 2009, pp. 1055–1058.

[14] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2007, pp. 31–40.

[15] O. Hartig, ""querying trust in rdf data with tsparql"," in *The Semantic Web: Research and Applications*, L. Aroyo, P. Traverso, F. Ciravegna, P. Cimiano, T. Heath, E. Hyvönen, R. Mizoguchi, E. Oren, M. Sabou, and E. Simperl, Eds., Berlin, Heidelberg, 2009, pp. 5–20.

[16] J. Hayes, "A graph model for rdf," Technische Universitt Darmstadt/Universidad de Chile, Tech. Rep., 2004.

[17] C. Hewitt, P. Bishop, and R. Steiger, "Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence," in *Advance Papers of the Conference*, vol. 3. Stanford Research Institute, 1973, p. 235.

[18] T. A. S. F. Jena, "Arq - a sparql processor for jena," 2017. [Online]. Available: https://jena.apache.org/documentation/query/

[19] T. A. S. F. Jena, "Arq - sparql algebra," 2017. [Online]. Available: https://jena.apache.org/documentation/query/algebra.html

[20] T. A. S. F. Jena, "Sparql s-expressions(or "sparql syntax expressions")," 2017. [Online]. Available: https://jena.apache.org/documentation/notes/sse.html

[21] A. S. F. Kafka, "Apache Kafka," 2016. [Online]. Available: https://kafka.apache.org/intro

[22] A. LightBend, "Adaptive Load Balancing," 2017. [Online]. Available: http://doc.akka.io/docs/akka/current/java/cluster-metrics.html

[23] A. LightBend, "AKKA Stream," 2017. [Online]. Available: http://doc.akka.io/docs/akka/current/java/stream/stream-introduction.html

[24] A. LightBend, "Cluster Specification," 2017. [Online]. Available: http://doc.akka.io/docs/akka/current/java/common/cluster.html

[25] A. LightBend, "Location Transparency," 2017. [Online]. Available: http://doc.akka.io/docs/akka/2.5.4/scala/general/remoting.html

[26] M. L. Manu Sporny, Gregg Kellogg, "Json-ld 1.1," W3C, Community Group Report, 12 July 2017.

[27] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. V. den Bussche, "The open provenance model core specification (v1.1)," *Future Gener. Comput. Syst.*, vol. 27, no. 6, pp. 743–756, Jun. 2011.

[28] L. Moreau and P. Missier, "PROV-DM: The prov data model, W3C recommendation," 30 April 2013. [Online]. Available: http://www.w3.org/TR/prov-dm/

[29] H. Mühleisen and C. Bizer, "Web data commons-extracting structured data from two large web corpora," *LDOW*, vol. 937, pp. 133–145, 2012.

[30] T. Neumann and G. Weikum, "Scalable join processing on very large rdf graphs," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 2009, pp. 627–640.

[31] V. Nguyen, O. Bodenreider, and A. Sheth, "Don't like rdf reification?: making statements about statements using singleton property," in *Proceedings of the 23rd international conference on World wide web*, 2014, pp. 759–770.

[32] L. M. Paul Groth, "Prov-overview," 30 April 2013. [Online]. Available: https://www.w3.org/TR/prov-overview/

[33] W. Sansrimahachai, M. J. Weal, and L. Moreau, "Stream ancestor function: A mechanism for fine-grained provenance in stream processing systems," in *Proceedings of the 6th IEEE International Conference on Research Challenges in Information Science (RCIS 2012)*, Valencia, Spain, May 2012, pp. 245–256.

[34] W. C. Tan *et al.*, "Provenance in databases: past, current, and future." *IEEE Data Eng. Bull.*, vol. 30, no. 4, pp. 3–12, 2007.

[35] Y. Theoharis, I. Fundulaki, G. Karvounarakis, and V. Christophides, "On provenance of queries on semantic web data," *IEEE Internet Computing*, vol. 15, no. 1, pp. 31–39, Jan. 2011.

[36] O. Udrea, D. R. Recupero, and V. Subrahmanian, "Annotated rdf," *ACM Transactions on Computational Logic (TOCL)*, vol. 11, no. 2, p. 10, 2010.

[37] M. Wylot, P. Cudre-Mauroux, and P. Groth, "TripleProv: Efficient processing of lineage queries in a native RDF store," in *Proceedings of the 23rd international conference on World wide web*. ACM, 2014, pp. 455–466.

[38] J. Zhao, C. Bizer, Y. Gil, P. Missier, and S. Sahoo, "Provenance requirements for the next version of rdf," in *W3C Workshop RDF Next Steps*, 2010.

[39] L. P. D. G. Ziff Davis, "Location Transparency." [Online]. Available: https://www.pcmag.com/encyclopedia/term/59018/location-transparency

[40] A. Zimmermann, N. Lopes, A. Polleres, and U. Straccia, "A general framework for representing, reasoning and querying with annotated semantic web data," *Web Semant.*, vol. 11, pp. 72–95, Mar. 2012.

AUTHOR BIOGRAPHIES

Qian Liu currently is a PhD candidate supervised by Prof. Dr. Manfred Hauswirth, in Open Distributed Systems Group, TU Berlin. His research topic is regarding the real-time provenance management techniques for the recovery of incomplete data streams. His research interests are including probabilistic graph stream manipulation algorithms, missing data recovery algorithms, machine learning, deep learning, reinforcement learning. He holds two MSc. degrees, one of Computer Science and Applied Mathematics from University of Bern, the other of Management, Technology and Economics from ETH Zurich. Contact him at qian.liu@tu-berlin.de.

Marcin Wylot is Data Scientist and Machine Learning Engineer, previously he was a postdoctoral researcher at TU Berlin in the ODS group. He received his PhD at the University of Fribourg in Switzerland in 2015, with the supervision of Professor Philippe Cudré-Mauroux. He graduated the MSc in computer science at the University of Lodz in Poland in 2010, doing part of his studies at the University of Lyon in France. During his studies he was also gaining professional experience working in various industrial companies. His main research interests revolved around database systems for Semantic Web data, provenance in Linked Data, Internet of Things, and Big Data processing. Homepage: http://mwylot.net

Danh Le Phuoc is a Marie Sklodowaka-Curie Fellow at the Technical University of Berlin. His research interests include linked data and the Semantic Web for the future Internet, big data for the Internet of Everything, databases, and pervasive computing. Le Phuoc received a PhD in computer science from the National University of Ireland. Contact him at danh@danhlephuoc.info

Manfred Hauswirth is a full professor for Open Distributed Systems at the Technical University of Berlin and the managing director of Fraunhofer FOKUS. His research interests include the Internet of Everything, domain-specific Big Data and analytics, linked data streams, semantic sensor networks, and sensor networks middleware. He holds a PhD and an MSc in computer science from the Technical University of Vienna. He is a member of IEEE and ACM and an associate editor of the IEEE Transactions on Services Computing.