Stream WatDiv - A Streaming RDF Benchmark

by

Libo Gao

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2017

Abstract

Modern applications are required to process stream data which are semantically tagged. Sometimes static background data interlinked with stream data are also needed to answer the query. To meet these requirements, streaming RDF processing (SRP) engines emerged in recent years. Although most SRP engines adopt the same streaming RDF data model in which a streaming RDF triple is an RDF triple annotated with a timestamp, there is no standard query language, which means every engine has their own language syntax. In addition, these engines are quite primitive, different engines support limited and different query operation sets. What's more, they are fragile in face of complex query, high stream rate or large static dataset. This poses a lot of challenges to evaluate the SRP engines. In our work, we show that previous streaming RDF benchmarks do not have a sufficient workload to understand engine's performance. The queries in those workloads are either not executable on existing engines, or very limited in terms of number. The goal of this work is to propose a benchmark which provides diversified datasets and workloads.

In our work, we extend WatDiv to generate streaming data and streaming query, and propose a new streaming RDF benchmark, called Stream WatDiv. WatDiv is an RDF benchmark designed for diversified stress testing of RDF data management engines. It introduces a collection of query features, which is used to assess the diversity of dataset and workloads. Through proper data schema design and query generation, WatDiv shows a good coverage of values of these query features. We demonstrate the feasibility of applying the same idea in streaming RDF domain. Stream WatDiv benchmark suits contain a data generator to generate scalable streaming data and static data, a query generator to generate scalable workloads, and a testbed to monitor the engine's output. We evaluate two engines, C-SPARQL and CQELS, and measure the correctness of engine output, latency and memory consumption. The findings contain two parts.

First, we validate the result related to these two engines in previous works. (1) CQELS is more robust and efficient than C-SPARQL at processing streaming RDF queries in most cases. (2) increasing streaming rate and integrating static data will significantly degrade C-SPARQL's performance, while CQELS is marginally affected. (3) C-SPARQL is more memory-efficient than CQELS.

Second, the diversity of Stream WatDiv workloads helps detect engines' issues that are not captured before. Queries can be grouped into different types based on the query features. These types of queries can be used to evaluate a specific engine features. (1) Triple pattern count of a query influences C-SPARQL's performance. (2) Both C-SPARQL and CQELS show a significant latency increase when the query has larger result cardinality. (3)

Neither of these two engines are biased toward processing linear, star or snowflake queries. (4) CQELS is more efficient at handling queries with variously selective triple patterns, while C-SPARQL performs better for queries with equally selective triple patterns than queries with variously selective triple patterns.

Acknowledgements

First and foremost, I want to express my sincerest gratitude to my advisor, M. Tamer Özsu, who has been continuously encouraging and guiding me throughout my master program with his patience and immense knowledge. I can still clearly remember how excited and honored I was when I knew I would be Tamer's student in three years ago. Thanks to him I got this wonderful research experience. I will treasure everything that he taught me.

In addition, I would like to thank my co-supervisor, Lukasz Golab. During the last two years, every week we met and discussed the works mentioned in this thesis. I appreciate all the time and effort he spent to make this thesis come true. He also helped me form good academic habits.

I would also like to acknowledge Prof. Khuzaima Daudjee and Prof. Grant Weddell for being my thesis readers and giving valuable feedback. And thank you to Dr. Güneş Aluç, who helped me better understand WatDiv.

My sincere gratitude also goes to my wonderful labmates in Data System group: Anıl Paçacı, Brad Glasbergen, Xiao Meng, Xiaofei Zhang, Khaled Ammar, Haotian Zhang and Yan Zhang, for all the academic and life problem discussion we made, for all the days and nights we worked together and for all the unforgettable memory together. Additional thanks go to all my friends in School of Computer Science for their willingness to help out wherever needed.

Most importantly, none of these could have happened without my parents, girlfriend and the whole family. Every week they offer me their endless love and encouragement through calls. Thank you for supporting my studies in Waterloo.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

A number of modern applications like smart cities, Internet of Things and e-commerce deal with heterogeneous stream data as well as rich static data. The data may be semantically tagged and stored in various formats. In this context stream data refers to set of data items that arrive continuously, usually in real-time, ordered by a timestamp.

Although Data Stream Management Systems (DSMS) and Complex Event Processors (CEPs) have been investigated for some time, they are not sufficient to deal with the data management requirements of these applications that also have to work with heterogeneous data from multiple sources over the web. A recent approach to deal with this type of data is the Resource Description Framework (RDF), which is a World Wide Web Consortium (W3C) [1] recommendation. One main reason for RDF's popularity lies in its schema-free flexibility. It can be used to represent both structured and unstructured data, which makes it easy for users to publish and link heterogeneous datasets. Linked Open Data (LOD) project [2] is perhaps the best known approach to model web data from many sources. The LOD projects tries to publish open data sets as RDF on the web and linked these data from different data sources.

RDF represents data as a collection of *triples* of the form <subject, predicate, object>. Subject represents resource in the web; object also represents resource in the web or a literal value; predicate holds the relationship between the subject and the object. For example, an

[1] W3C is an international community to develop world wide web standards.

[2] https://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData

RDF data can be used to describe the statement “The book is in language English” as a triple: the subject denoting “the book”, the predicate denoting “language” and the object denoting “English”.

SPARQL is a standard query language in the family of W3C recommendation to manipulate and retrieve data in RDF repositories. With increasing growth of RDF data sizes, lots of works focus on efficiently executing SPARQL queries on large amounts of data. As an example of the rapid growth of RDF data is the LOD project whose content grew from around 200 data sources containing 25 billion triples in 2010 to more than 3000 data sources containing more than 84 billion triples in 2016, which is an increase of more than three times in about 6 years.

There has been an increasing number of RDF data management systems that have been developed over the past decade, which we review shortly. These systems incorporate functionality to execute SPARQL queries over RDF data storage.

A reasonable approach to support the applications mentioned above is to develop systems that combine stream processing and RDF processing capabilities. In recent years, these systems have started to appear under the name Streaming RDF Processing (SRP) Systems (or engines). Concurrent with these system developments is an interest in testing the performance of these systems. A number of benchmarks have been developed for SRP systems, such as SRBench [39], LSBench [25], CSRBench [19], CityBench [4] and YABench [23]. However, the workloads of these benchmarks are not sufficient enough to understand different engine’s performance. The queries in those workloads are either not executable on existing engines, or very limited in terms of number. We analyze the workloads in previous works in Section 2.3. To address these shortcomings, in this thesis we develop a new SRP benchmark, and demonstrate its use by evaluating two SRP engines: C-SPARQL [12, 9, 8, 11, 10] and CQELS [24].

1.2 Approach

The SRP engines consume and process the RDF stream, which is a sequence of RDF triples annotated with timestamps. The expressive power of the RDF data model makes the RDF stream capable of describing heterogeneous stream data. In developing a benchmark, we foresee the following requirements:

- Language independence: Currently there’s no standard of SRP query language. Different engines have their own query languages, which are extended from SPARQL query language. These languages share many common operations, but also support some operations specific to each system [39]. It is likely that this situation will continue in the near-term until a convergence occurs. Therefore, the benchmark should contain workloads independent of a given engine's query language, so it can be used to evaluate measurement criterion across a number of engines. In addition, as the query language evolves, the workload needs to be easily extended to cover new query language features.

- Diversified workloads and datasets: Real world applications need to handle more and more complex workloads and heterogeneous data, the benchmark should be able to capture all kinds of possible issues of the SRP engines, so users do not need to spend more cost to fix them after the engines are deployed.

- Ease of scaling up/out: The SRP engines are expected to have more powerful processing ability as their development matures. The benchmark should not only focus on evaluating the power of SRP engines in the current stage. It must have a scalable dataset and workload so as to evaluate engine's capability in different development stages. A scalable dataset means users can define the size of the static dataset and the length of RDF stream. A scalable workload means it can be both scaled up and scaled out. Scaling up the workload refers to increasing the query complexity. Scaling out the workload refers to generating a larger set of queries.

- Flexibility of customizing the experiment setting: The benchmark should be capable of both generally measuring the engines' performance and specifically testing engines' features, such as handling large static dataset and large stream rate.

- Reproducibility: No matter how many times reruning the experiment, once the workloads, datasets and experiment settings are fixed, the result should be consistent.

Our approach to developing a SRP benchmark is to start with an RDF engine benchmark and extend it with streaming data functionality. There are a number of RDF system benchmarks such as LUBM [22], BSBM [14], DBSB [27] and SP2Bench [34]. However, these benchmarks do not have as diverse dataset and workload as real data and applications. Consequently, engine issues may be undetected by using these benchmarks. WatDiv [5] has been developed to address these concerns. WatDiv is an RDF triple store benchmark. It introduces some query features used to assess the diversity of dataset and workload, and illustrates how these query features are related to the engine issues. Through proper dataset design and workload generation, the workloads demonstrate a better coverage across the values of these query features than the other benchmarks. Its diversified workloads and dataset can be used to stressfully test the RDF data management systems. The workloads it uses are limited to a basic fragment of SPARQL, which is also compatible in streaming RDF query language. WatDiv allows users to flexibly scale out/up the workloads and datasets. The RDF benchmarks and WatDiv are discussed in more detail in Chapter 2.

There are two approaches to the development of a SRP benchmark. One can either start from a stream data benchmark such as Linear Road [7] and extend it with RDF processing capability, or start with an RDF benchmark and extend it to deal with stream data. In this thesis, we take the second approach; in particular we extend WatDiv to handle streaming RDF data (the resulting benchmark is called Stream WatDiv[3]). RDF processing requirements are far more complicated as the original WatDiv development has shown, and starting from that is expected to result in a more feature-rich benchmark.

1.3 Novelty

Various SRP benchmarks like SRBench [39], LSBench [25], CSRBench [19], CityBench [4] and YABench [23] have been proposed, focusing on different evaluation goals. Benchmarking in a specific domain certainly can (1) help users understand the strengths and weaknesses of solutions in that domain, and (2) also foster the development of that domain. However, none of the existing streaming RDF benchmarks fully achieve these two goals. By using these benchmarks, users do not get a complete picture of scenarios where one engine outperforms the other; developers may get lost in supporting rich query operations.

By contrast Stream WatDiv focuses on only basic fragment of the query language, namely basic graph pattern (BGP) matching, but covers as many different types of queries as possible to detect unexpected engine behaviors. We utilize the query features defined in WatDiv to differentiate different types of queries. Since most SRP engines support this basic fragment of the query language, the workloads can be used to evaluate performance across engines. In addition, if users find the engine has problems with handling a certain type of query, they can analyze the features of these queries and generate more similar queries to locate the engine issue. For example, if a query causes the engine to run out of memory, users can analyze whether the result cardinality of the query is too big to fit into the memory or improper query execution plan results in huge intermediate result. Based on the analysis, developers of the engines can find the appropriate solutions.

[3] Description and source code can be found on http://dsg.uwaterloo.ca/watdiv/stream-watdiv

1.4 RDF Overview

RDF was originally proposed to describe web objects in Semantic Web. However, the usage of RDF as a general data model is now far beyond the Semantic Web. For example, companies such as BBC transform and publish the content of web pages in RDF data, and enable the SPARQL queries across pages [13]. Biologists record their experiment result in RDF and publish them as linked datasets (bio2rdf.org, dev.isb-sib.ch/projects/uniprot-rdf). Both Yahoo and Google promote the use of RDF to optimize their search engines.

In this section we define RDF model and SPARQL language precisely[4], talk about streaming extension of RDF data and SPARQL language, and discuss the approaches that have been adopted for managing RDF data.

Definition 1.4.1. (**RDF triple**) Let I, B and L denote the set of all IRIs, blank nodes and literals respectively. IRIs (International Resource Identifier) denote resources in the web. Literals are used for values such as integers, strings, etc. Blank nodes represent resources that are not explicitly named. Then an RDF triple (subject, predicate, object) $\in (I \cup B) \times (I) \times (I \cup B \cup L)$.

A set of triples can be represented as a directed, labelled graph with the subjects and objects as vertices, and the predicates as edges in the graph.

Definition 1.4.2. (**RDF graph**) RDF graph is a labelled directed graph. It can be denoted as $G = (V, E, L_e)$. V denotes the vertices in the graph involving subjects and objects. E denotes the the edges in the graph. Each edge in the graph represents an RDF triple. L_e denotes a set of edge labels. It consists of all the predicates appeared in the RDF data.

Definition 1.4.3. (**RDF triple pattern**) Let I, B, L and V denote the set of all IRIs, blank nodes, literals and variables respectively. Then an RDF triple pattern (s, p, o) $\in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$.

Definition 1.4.4. (**SPARQL query**) The SPARQL query expression can be defined recursively:

1. If P is an RDF triple pattern, then P is a SPARQL expression.

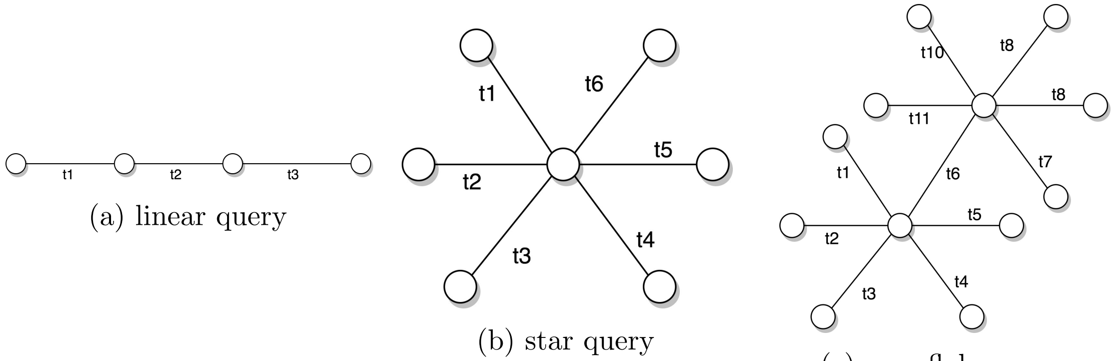
[4] More details about RDF and SPARQL can be found in RDF prime (https://www.w3.org/TR/rdf11-primer/), SPARQL 1.1 recommendation (https://www.w3.org/TR/sparql11-overview/) and other W3C documents.

2. If P is a SPARQL expression, P FILTER R is also a SPARQL expression where R is a SPARQL built-in filter condition.

3. if P_1 and P_2 are both SPARQL expressions, then P_1 UNION P_2, P_1 OPTIONAL P_2, P_1 AND P_2 are all SPARQL expressions.

A set of RDF triple patterns is called Basic Graph Pattern (BGP). SPARQL queries that only involve BGPs are SPARQL BGP queries. SPARQL queries can also be represented as query graphs. In our work, we only consider the SPARQL BGP queries. With different combination of triple patterns, the query graph can be in linear, star, snowflake or even more complex shapes (see Figure 1.1).

Definition 1.4.5. (**SPARQL BGP query graph**) If a SPARQL query only contains BGPs, it is a SPARQL BGP query. A SPARQL BGP query can be represented as $G^Q = (V^Q, G^Q, E^Q)$. Let V_{Var} be the set of variables appeared in the query, then $V^Q \subset V \cup V_{Var}$ denotes the vertices in the graph involving subjects and objects and variables. $G^Q \subset V^Q \times V^Q$ denotes the edges in the graph. E^Q denotes a set of edge labels. It is formed by all the predicates appeared in the RDF data. The predicate itself can also be a variable.



(a) linear query

(b) star query

(c) snowflake query

Figure 1.1: Let t_i denotes a triple pattern, the shape of a query graph can be linear, star or snowflake.

One way to think about SPARQL semantics is the following. Each triple pattern is matched to the RDF data and candidate triples that may satisfy it are collected in a candidate list. These candidate lists are then (pair-wise) joined on the unbounded variables to generate the final result. The joins can be subject-subject, subject-object, or object-object[5].

[5] Although it is possible to have variables on the predicates, these are not much discussed in literature.

Although there is no standard model of streaming RDF data, most previous works apply the same extension to the RDF data model. Each RDF triple is attached with a timestamp. Timestamps are monotonically non-decreasing, they determine the partial order of triples in the stream. They are not required to be unique, because multiple streaming RDF triples can have the same timestamp if they appear at the same time in the stream.

Definition 1.4.6. (**RDF stream**) A streaming RDF triple is a timestamp-annotated RDF triple, which can be represent as $< s, p, o, \tau >$. The timestamp τ is monotonically non-decreasing and discrete. An RDF stream is an unbounded ordered sequence of streaming RDF triples.

Most streaming RDF query languages extend SPARQL with time-based window operations. Time-based window operations decompose an RDF stream into a sequence of finite subsets. Continuous query will be evaluated over each subset in turn. Time windows that are progressively advanced by the sliding step δ (δ is smaller than the window length ω) are called sliding windows. If the sliding step equals the window length ω, a series of non-overlapping windows are generated, we call them tumbling windows.

Definition 1.4.7. (**Sliding Window**) Let the first window starts at τ_0, ω be the window length, δ be the sliding step and $\delta < \omega$, then the content of the ith sliding window over the stream S can be denoted as $W^i_{\omega,\delta}(S) = \{< s, p, o, \tau > \mid < s, p, o, \tau > \in S, \tau_0 + (i-1) * \delta \leq \tau \leq \tau_0 + (i-1) * \delta + \omega\}$.

Definition 1.4.8. (**Tumbling Window**) Let the first window starts at τ_0, ω be the window length, then the content of the ith tumbling window over the stream S can be denoted as $W^i_{\omega}(S) = \{< s, p, o, \tau > \mid < s, p, o, \tau > \in S, \tau_0 + (i-1) * \omega \leq \tau \leq \tau_0 + i * \omega\}$.

A streaming RDF query may not only involve streaming data but also static data. In the rest of the thesis, we denote hybrid query as the query that involves both streaming data and static data, and denote pure stream query as the query that only involves streaming data.

Existing RDF data management systems follow two main approaches [30]: relational approaches and graph-based approaches [41]. Relational approaches store and manage RDF data in tabular format. They can be further classified into four categories:

- Systems like Sesame SQL92SAIL [15] directly store the RDF data in a single relational table and transform the SPARQL queries to SQL queries. They try to exploit the well-developed relational data management techniques to process RDF data. The main issue of this approach is the large number of self-joins since all data are stored in one table.

- Another approach is to extend the above approach with indexes. Representative engines like RDF-3X [28, 29] build indexes for all six possible permutations of subject, predicate and object. The number of self-joins can be reduced because some of them can be replaced by the range queries over a particular index. The overhead of maintaining multiple big indexes, especially when datasets are dynamic, can be considerable.

- Instead of storing all data in one table, an alternative approach is to divide the data into several so-called *property tables*. Jena [6] adopts this approach. The main idea is to group the properties which tend to appear with the same subject in the same table. This facilitates subject-to-subject joins, since only a single table needs to be scanned. The main issue of this approach is that it needs a good design to group proper properties into tables. Also, this approach only optimizes the subject-to-subject joins.

- Binary table approach [1, 2] assigns a two-column table for each property containing subjects and objects. Like column-oriented database, this approach benefits from better compression and reducing I/O cost. The shortcomings are that it introduces more join operations and higher insertion cost.

All previous approaches transform the data into relational data model or column-oriented data model, while engines like gStore [40, 42] and chameleon-db [20] maintain the graph structure of RDF data. Evaluating the query can be implemented as subgraph matching using homomorphism.

1.5 Contributions

The contributions of the thesis can be summarized as follows:

- We analyze the existing RDF streaming benchmarks and demonstrate that none of the workloads in existing RDF streaming benchmarks contain sufficient queries and are diversified enough to evaluate performance across SRP engines. To address this problem, we extend WatDiv into a new streaming RDF benchmark, called Stream WatDiv. The original WatDiv data generator is extended to generate both streaming data and static data, the original WatDiv query generator is extended to generate both pure stream query and hybrid query. WatDiv defines a collection of query

[6] https://jena.apache.org

features which can be used to assess the diversity of workloads. These query features are also applicable in streaming setting to differentiate different types of streaming queries. The goal of Stream WatDiv is to provide workloads which capture as many types of queries as possible.

- We integrate our benchmark with YABench's testbed. This generates a full benchmarking environment and toolset. The testbed can be used to verify the correctness of engine output and measure the latency of query execution. Two popular SRP engines, C-SPARQL and CQELS are evaluated. Before the evaluation, we also modify and enhance C-SPARQL so it can efficiently handle large static dataset and precisely extract and process streaming data.

- The workloads and datasets of Stream WatDiv together enable the diversified evaluation across engines. Since the workloads of Stream WatDiv capture more types of queries than previous works, evaluation of C-SPARQL and CQELS with Stream WatDiv get some new findings. First, a diversified workload can help detect engine issues which are not captured by previous benchmarks. Both C-SPARQL and CQELS fail to get correct result for some queries. These failed queries are related to various engine issues. Second, queries can be grouped into different types based on the query features, these types of queries can help test specific engine features. As shown in the result, (1) performance of C-SPARQL is influenced by increasing triple pattern count in the query. (2) Both C-SPARQL and CQELS show a significant latency increase when processing queries with large result cardinality. (3) Neither of these two engines are biased toward processing linear, star or snowflake queries. (4) CQELS is efficient at handling queries with variously selective triple patterns, while C-SPARQL performs better for queries with equally selective triple patterns than queries with variously selective triple patterns.

- Another important objective of the evaluation is to validate the result reported in previous works. The result verifies that: (1) overall, CQELS shows lower latency than C-SPARQL, (2) both increasing streaming rate and integrating static data will significantly influence C-SPARQL's performance, while CQELS is marginally influenced, (3) C-SPARQL is more efficient in memory management, and consumes much less memory than CQELS.

1.6 Organization

This thesis is organized as follows. Chapter 2 introduces related works along two main background domains: RDF benchmarks and streaming benchmarks. Chapter 3 discusses the design of Stream WatDiv benchmark. Chapter 4 presents an example of using Stream WatDiv to evaluate performance across two engines: C-SPARQL and CQELS. Finally, Chapter 5 presents the conclusions and discusses the potential future work.

Chapter 2

Related Work

There are three lines of work related to this thesis: RDF benchmarks, stream processing benchmarks, and streaming RDF systems. In this chapter, we will introduce the first two parts. The streaming RDF systems will be discussed in Chapter 5.

2.1 RDF Benchmarks

With the increasing availability of systems supporting processing RDF triples, a number of benchmarks were proposed to evaluate these systems. Popular benchmarks such as LUBM [22], BSBM [14], DBSB [27] and SP2Bench [34] evaluate systems' performance of handling different use cases of SPARQL. For example, LUBM [22] focuses on testing reasoning capability and storage mechanism of RDF triple stores. Compared with LUBM, SP2Bench [34] contains more SPARQL operations like FILTER, DESCRIBE, UNION, etc. BSBM [14] tests how well RDF triple stores handle large amounts of data. DBSB [27] collects real world workloads from the DBpedia query log. Its real-world datasets are obtained from sampling the DBpedia datesets.

Nowadays RDF triple stores need to consume heterogeneous data and handle diversified workloads. However, these RDF benchmarks do not have sufficiently diversified workloads [5]. For example, queries in LUBM are small, only containing at most 6 triple patterns. SP2Bench contains two types of queries, linear query with long chain and large star query with single center. However, there are no other query types. BSBM makes up this, but does not cover the two extreme types of queries like SP2Bench. DBSB only contains queries with few object-to-object join and subject-to-object join. Based on these observations, the Waterloo SPARQL Diversity Test Suite (WatDiv) is proposed to tackle this diversity workload challenge.

The basis of this thesis is WatDiv, which is an RDF benchmark focusing on diversified stress testing of RDF processing systems. Compared with other RDF benchmarks, WatDiv is developed to better cope with the heterogeneous data and diversified workloads that the current RDF processing systems need to handle, and it addresses many of the shortcomings of the earlier benchmarks listed above. Its diversity lies in two sets of query features: structural and data-driven. Here, we revisit the definition of these query features in WatDiv which could help understand the rest of the thesis. WatDiv defines query features over a basic fragment of SPARQL, namely constrained basic graph pattern (CBGP). Formally, a CBGP is a pair of $\overline{B} = \langle B, F \rangle$, where B denotes a BGP and F denotes a finite set of SPARQL filter expressions. The evaluation of $\overline{B}$ over RDF graph G denoted by $[\![\overline{B}]\!]_G$ is a bag of solution mappings.

Structural features	Data-driven features
Triple pattern count	Result cardinality
Join vertex count	Filtered triple pattern selectivity (f-TP selectivity)
Join vertex degree	BGP-Restricted f-TP selectivity
Join vertex types	Join-Restricted f-TP selectivity

Table 2.1: Query features defined in WatDiv

The structural features differentiate the structure of the queries. *Triple pattern count* calculates the number of triple patterns in a query. *Join vertex count* calculates the number of join vertices in a query. *Join vertex degree* measures the number of triple patterns whose subject or object is x for each join vertex x of $\overline{B}$. *Join vertex types* differentiates three types of joins, subject-to-subject join, subject-to-object join and object-to-object join. Join vertices involving these joins are of types SS^+, SO^+ and OO^+, respectively.

Take the query shown in Figure 2.1 as an example, the triple pattern count is 7. As join vertices are marked in orange, the join vertex count is three. The corresponding join vertex degree of join vertices from left to right are 4, 2, and 3. And the corresponding join vertex type is SS^+, SO^+ and OO^+.

The data-driven features include the query features that are related to data. Figure 2.2 gives a simple query example and a small dataset, which together with Figure 2.3, 2.4, 2.5, 2.6 will be used later to better demonstrate data-driven features.

Result cardinality: assume Ω denotes the set underlying the bag of $[\![\overline{B}]\!]_G$, $card_{[\![\overline{B}]\!]_G}$ denotes the function to map each solution mapping $\mu \in \Omega$ to its cardinality in the bag, the

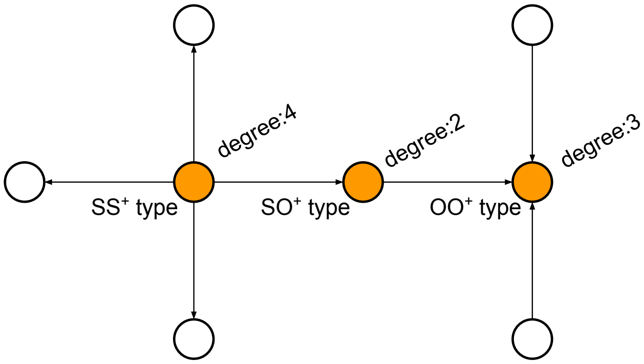


Figure 2.1: Query example to demonstrate structural features

result cardinality of $\overline{B}$ over G can be represented as:

$$CARD(\overline{B}, G) = \sum_{\mu \in \Omega} card_{[\![\overline{B}]\!]_G}(\mu). \tag{2.1}$$

Result cardinality measures the number of solutions of evaluating query over the data. Figure 2.3 shows the result of evaluating the query over the data in Figure 2.2. The result cardanility of this query example is 2.

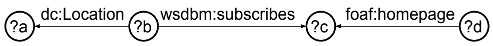
Filtered triple pattern selectivity (f-TP selectivity): given a CBGP $\overline{B} = \langle B, F \rangle$ and $tp \in B$, we use $\lambda^F(\{tp\})$ to present the CBGP $\overline{B'} = \langle B', F' \rangle$, with $B' = \{tp\}$ and $F' = \{f \in F | vars(f) \subseteq vars(tp)\}$, where $vars(\cdot)$ represents the variables in a filter expression or a BGP. Let Ω represents the set underlying the bag of $[\![\lambda^F(\{tp\})]\!]_G$, then the f-TP selectivity of a triple pattern $tp \in B$ in a CBGP $\overline{B} = \langle B, F \rangle$ over an RDF graph G, denoted by SEL_G^F is defined as:

$$SEL_G^F = \frac{|\Omega|}{|G|}. \tag{2.2}$$

In Figure 2.4, triples matching the triple pattern <?c, foaf:homepage, ?d> are marked in orange, the size of which is 4. The total triple number in the dataset is 9. So the F-TP selectivity of triple pattern <?c, foaf:homepage, ?d> is $\frac{4}{9}$.

BGP-Restricted f-TP selectivity: for any triple pattern $tp \in B$ in a CBPG $\overline{B} = \langle B, F \rangle$, let Ω and Ω' denote the set underlying the bag of $[\![\lambda^F(\{tp\})]\!]_G$ and $[\![\overline{B}]\!]_G$ respectively, then

```
#namespace      dc=http://purl.org/dc/terms/
#namespace      foaf=http://xmlns.com/foaf/
#namespace      wsdbm=http://db.uwaterloo.ca/~galuc/wsdbm/
```



subject	predicate	object
wsdbm:Product1	foaf:homepage	wsdbm:Website4
wsdbm:Product2	foaf:homepage	wsdbm:Website1
wsdbm:Product3	foaf:homepage	wsdbm:Website2
wsdbm:Product4	foaf:homepage	wsdbm:Website3
wsdbm:User1	wsdbm:subscribes	wsdbm:Website2
wsdbm:User2	wsdbm:subscribes	wsdbm:Website1
wsdbm:User3	wsdbm:subscribes	wsdbm:Website3
wsdbm:User1	dc:Location	wsdbm:City1
wsdbm:User2	dc:Location	wsdbm:City2

Figure 2.2: Query example and small dataset to demonstrate data-driven features

?a	?b	?c	?d
wsdbm:City1	wsdbm:User1	wsdbm:Website2	wsdbm:Product3
wsdbm:City2	wsdbm:User2	wsdbm:Website1	wsdbm:Product2

Figure 2.3: Result of evaluating query over dataset in Figure 2.2

the BGP-Restricted f-TP selectivity of tp over an RDF graph G is defined as:

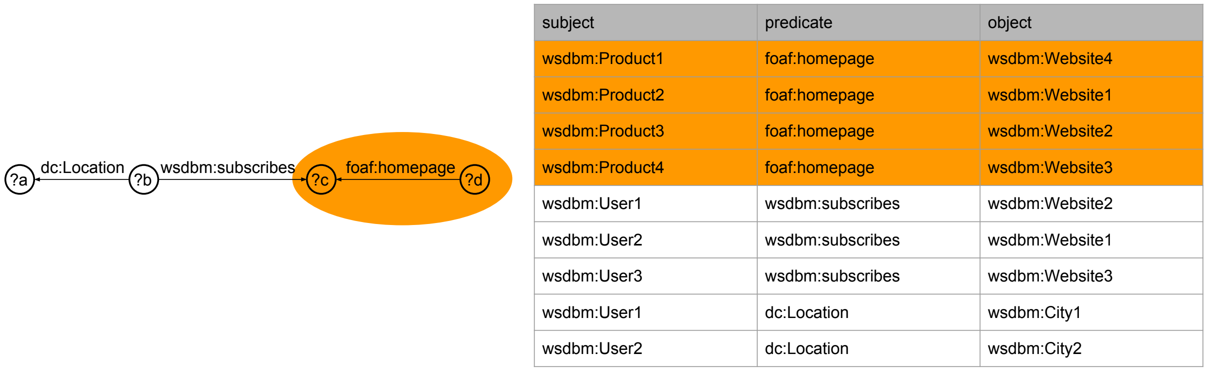
$$SEL_G^F(tp|\overline{B}) = \frac{|\mu \in \Omega \mid \exists \mu' \in \Omega' : \mu \ and \ \mu' \ are \ compatible|}{|\Omega|}. \tag{2.3}$$

BGP-restricted f-TP selectivity measures how much a triple pattern contributes to the overall query "selectivity". For example, in Figure 2.5, four triples match the triple pattern <?c, foaf:homepage, ?d>, while only two of them remain in the query result. So the BGP-restricicted f-TP selectivity of triple pattern <?c, foaf:homepage, ?d> is $\frac{1}{2}$.

Join-Restricted f-TP selectivity: given a CBPG $\overline{B} = \langle B, F \rangle$, a join vertex x of $\overline{B}$ and a triple pattern $tp \in B$ whose subject or object is x, let Ω and Ω' denote the set underlying the bag of $[\![\lambda^F(\{tp\})]\!]_G$ and $[\![\lambda^F(\{B^x\})]\!]_G$, respectively, where $B^x = \{tp \in B| \ subject \ or \ object \ of \ tp \ is \ x\}$, then the join-restricted f-TP selectivity of triple pattern tp related to join vertex x over an RDF graph G is defined as:

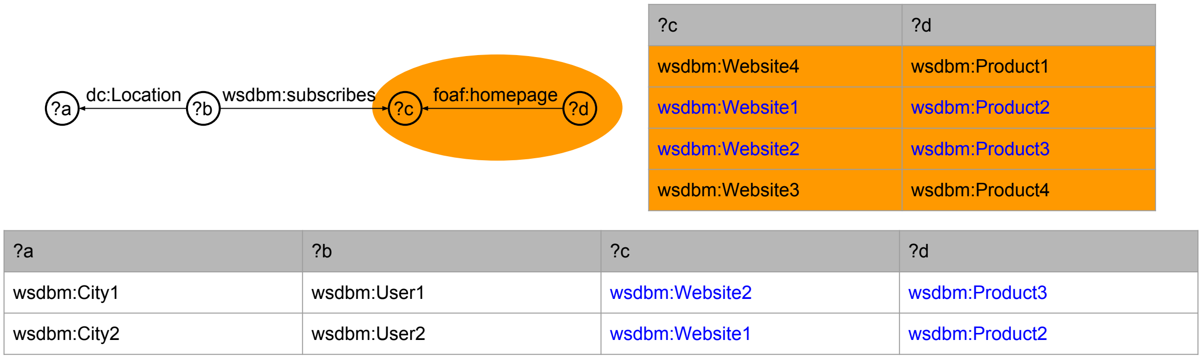
$$SEL_G^F(tp|x) = \frac{|\mu \in \Omega \mid \exists \mu' \in \Omega' : \mu \ and \ \mu' \ are \ compatible|}{|\Omega|}. \tag{2.4}$$

Join-restricted f-TP selectivity measures how much a triple pattern contributes to the overall "selectivity" of the join it participates in. For example, in Figure 2.6, the join vertex



subject	predicate	object
wsdbm:Product1	foaf:homepage	wsdbm:Website4
wsdbm:Product2	foaf:homepage	wsdbm:Website1
wsdbm:Product3	foaf:homepage	wsdbm:Website2
wsdbm:Product4	foaf:homepage	wsdbm:Website3
wsdbm:User1	wsdbm:subscribes	wsdbm:Website2
wsdbm:User2	wsdbm:subscribes	wsdbm:Website1
wsdbm:User3	wsdbm:subscribes	wsdbm:Website3
wsdbm:User1	dc:Location	wsdbm:City1
wsdbm:User2	dc:Location	wsdbm:City2

Figure 2.4: F-TP selectivity of the triple pattern marked in orange



?c	?d
wsdbm:Website4	wsdbm:Product1
wsdbm:Website1	wsdbm:Product2
wsdbm:Website2	wsdbm:Product3
wsdbm:Website3	wsdbm:Product4

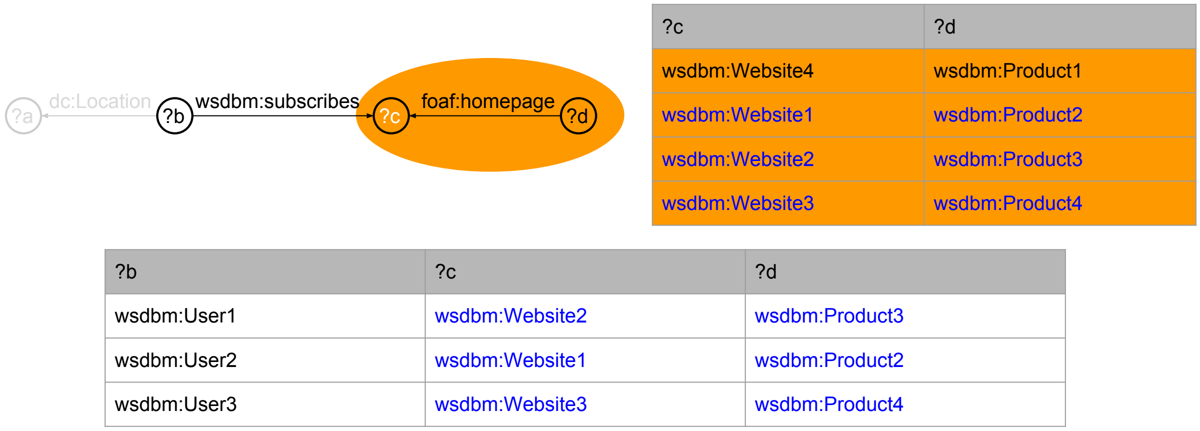
?a	?b	?c	?d
wsdbm:City1	wsdbm:User1	wsdbm:Website2	wsdbm:Product3
wsdbm:City2	wsdbm:User2	wsdbm:Website1	wsdbm:Product2

Figure 2.5: BGP-restricted F-TP selectivity of the triple pattern marked in orange

related to the triple pattern <?c, foaf:homepage, ?d> is ?c, the join related to the join vertex c? involves two triple patterns, <?c, foaf:homepage, ?d> and <?b, foaf:subscribes, ?c>. Among four triples which match the triple pattern <?c, foaf:homepage, ?d>, three of them are remained after the join operation. So the join-restricted f-TP selectivity of triple pattern <?c, foaf:homepage, ?d> related to ?c is $\frac{3}{4}$.

Queries containing triples with various selectivity can be used to test how the engine generates the query execution plan. To differentiate queries with different combination of selective triples, WatDiv calculates the mean and standard deviation of the triple selectivity in the query. WatDiv shows a good coverage of the value of all these features. The way to achieve its diversity lies in the design of dataset and the choice of workloads.

WatDiv generates a synthetic dataset based on the schema of an e-commerce database. The schema is shown in Figure 2.7. There are 16 types of entities in the scheme. Major



?c	?d
wsdbm:Website4	wsdbm:Product1
wsdbm:Website1	wsdbm:Product2
wsdbm:Website2	wsdbm:Product3
wsdbm:Website3	wsdbm:Product4

?b	?c	?d
wsdbm:User1	wsdbm:Website2	wsdbm:Product3
wsdbm:User2	wsdbm:Website1	wsdbm:Product2
wsdbm:User3	wsdbm:Website3	wsdbm:Product4

Figure 2.6: Join-restricted F-TP selectivity of the triple pattern marked in orange

part of the data describes users' activities on the e-commerce website. Users can purchase products, write reviews; retailers can provide offers of products. Details about these activities are also recorded. The data schema also contains the social network content. Users can follow other users, subscribe website and like products. Other background data such as users' age, gender, location, products' category are also included.

WatDiv provides a parameter called scale factor to generate datasets with different sizes. The instance number of some entities will increase proportional to scale factor, while some entities have constant instance number. Table 2.2 shows the default entity instance counts. To achieve diversity, the generated dataset has several properties:

- For each entity, its attributes are divided into several sets, each of which can have one or multiple attributes. Every set has a generating probability. When generating an instance of an entity, this value determines whether or not this set of attributes will appear in the database instance. For example if the probability for set A_i is 0.5, it means that there is 50% chance that the attributes in set A_i will exist for that entity instance. This probability make different instances of the same entity not guaranteed to have the same sets of attributes, so when a query visits certain attributes of an entity type, the triple patterns will have different selectivity.

- Entities have various attributes with literal values, which can be integer, string or date. Every literal attribute is instantiated within its specific value range and with its specific distribution type.

- Relationships between entities are varied. For each relationship, the model file defines

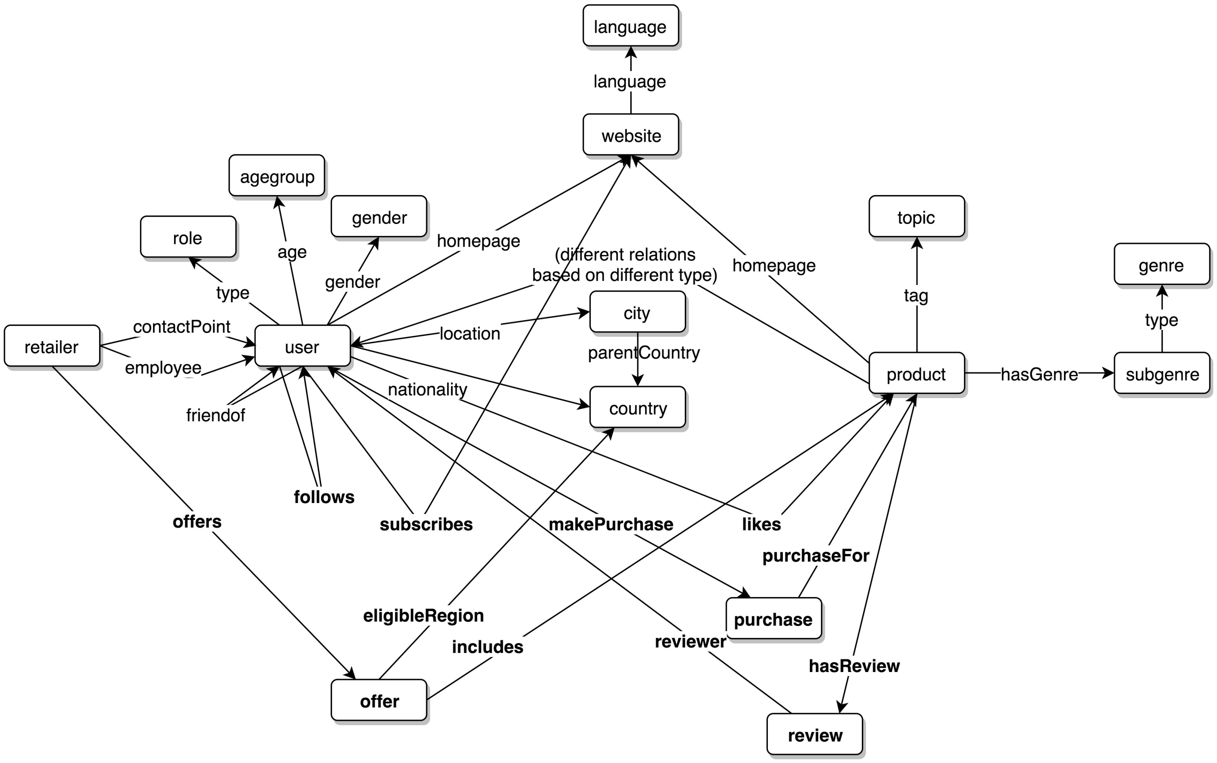


Figure 2.7: WatDiv data schema

the entity types, object cardinality range and subject cardinality range. Exact subject and object cardinality values are determined according to a probability distribution that is user-defined. In this way, selectivity of different triples are diversified. These triples with various selectivity value are joined together either through subject vertices or object vertices, creating a data graph with various structures such as linear, star, snowflake, and other more complex ones. Each join vertex also has various join vertex degree.

WatDiv's workloads only cover the basic fragment of SPARQL, namely basic graph patterns (BGPs) with filter expression. Generating the workloads includes two steps. The query template generator produces a set of query templates with placeholders. The query generator will instantiate these query templates by replacing the placeholders with real RDF terms. The query templates are created by performing a random walk in the data graph starting from a random node in the data graph. At each step, a random triple is selected from triples that are connected to the query graph and will be added to form a

entity type	instance number
Purchase	1500
User	1000
Offer	900
Product	250
Website	50
Retailer	22
Topic	250
City	240
SubGenre	145
Language	25
Country	25
Genre	21
ProductCategory	15
AgeGroup	9
Role	3
Gender	2

Table 2.2: Entity instance number (when scale factor is 1)

new query graph. The query template generator will uniformly select a random number as the query template size. Users can set the maximum triple pattern number to control the size of the query.

The advantage of WatDiv is the combination of the data model and the query generator. Since the generated data already have diversified structure and selectivity, randomly walking in the data graph will result in queries with different shapes and gathering triples with different selectivity. Our work extends WatDiv to enable the diversified testing of SRP engines by (a) modifying the data generator to generate streaming data in addition to the static data, and (b) modifying the query generator to allow specifying time windows to the queries, changing the target query language from SPARQL to continuous SPARQL. We remain all the query features defined in WatDiv, they proved to be useful as well in streaming setting. We focus on measurement metrics which are more interesting in streaming setting, such as output correctness, latency and memory consumption.

2.2 Streaming Benchmarks

Linear Road [7] Benchmark is the widely used benchmark to evaluate streaming engines. As more SRP engines have emerged, there have been efforts in designing SRP-specific benchmarks with their specific performance criteria, evaluation methodologies, and testbed infrastructure. In this section, we first present Linear Road Benchmark, then introduce recent streaming RDF benchmarks.

Linear Road Benchmark is a well-established benchmark to evaluate data stream management systems (DSMSs). It simulates an expressway toll system in a linear city. In the city, there are 10 expressways, and vehicles will be charged with dynamic tolls according to traffic congestion and accident occurrence. The benchmark generates both stream data (vehicle positions and accident information) and historical data (10 weeks data of vehicle and toll system). The workloads include continuous queries and also one-time historical queries. The benchmark measures response time and maximum query load of DSMSs. Although Linear Road Benchmark contains both steam data and historical data, it does not interlink stream data with historical data. Moreover, it does not capture the properties of RDF data model. Yahoo also proposed a benchmark [18] to compare Apache Storm with other distributed stream processing engines. However, most existing SRP engines are centralized, and their query operators and system architectures are quite different from distributed stream processing systems like Apache Storm. The only one of two distributed SRP engines which is built upon Apache Storm is not open source.

In SRP domain, there have already been several benchmarks. Here, we will list some representative ones and compare them with our benchmark.

- SRBench [39] is the first SRP benchmark. It uses real world weather data and contains seventeen well-selected weather-related queries. The main goal of the benchmark is functional evaluation, so these seventeen queries capture various features of SPARQL 1.1 query language. SRBench demonstrates that the existing SRP engines are quite primitive and do not support many of these features. This benchmark does not include any performance measurement.

- CSRBench [19] extends SRBench to include correctness verification. It is the first benchmark to propose oracle-based correctness verification method. The oracle offline evaluates the query over the data and generates the correct result for each query execution. Later, this correct result can be used to validate the engine's output. It takes different engines' query execution mechanisms into account. For example, periodically-executing engines such as C-SPARQL[1]. [12, 9, 8, 11, 10] execute the query when the window closes, while eagerly-executing engines such as CQELS [24] execute the query whenever the window content changes, so the oracle will mimic the same query execution behavior for different engines. CSRBench finds none of the streaming RDF processing engines successfully passes the correctness test and provides the explanation of engines' failure at some specific queries. One big shortcoming of CSRBench is its small workloads. It only measures the correctness and performance based on this small set of queries.

- LSBench [25] goes further than SRBench by considering correctness verification and performance evaluation. It simulates a social network environment, and contains twelve queries with different complexity. The output correctness is verified by calculating the mismatch between different engines' output considering the results returned by a majority of the systems as correct. It measures the maximum throughput of engines that are determined to produce correct results. In our benchmark, we adopt the oracle-based verification method.

- CityBench [4] considers several real world applications in Smart Cities. The experiments mainly focus on evaluating systems' scalability of running multiple concurrent queries, and the scalability of handling multiple data streams. The benchmark is application-driven. It does not contain a set of queries with complexity from low to high. What's more, no correctness verification is included in the experiments.

- [33] is a recent work measuring the performance of two SRP engines, CQELS and C-SPARQL. It generates water management data and has a small query set. It defines a complete set of test criteria to deeply understand the performance of these two engines. Compared with this work, our objective is to produce a benchmark that can be used to test a wider set of SRP engines over a more varied workload.

Previous works have shortcomings in either their workloads or their measurement metrics. Stream WatDiv addresses both of these shortcomings. The data generator and query generator can generate a scalable dataset and workloads.

In addition to these benchmarks, there are experiment testbeds that have been developed to evaluate streaming RDF systems. We discuss these below.

- YABench [23] implements an SRP engine testbed. It consists a set of tools to generate the data stream, verify the output correctness, monitor the resource usage, measure

[1] We discuss streaming RDF systems in Chapter 5

the performance and visualize the result. It also adopts oracle-based verification method. Compared with CSRBench, YABench provides richer measurement metrics such as per-window output accuracy, and measures the memory usage and CPU utilization of tested engines. YABench uses the same dataset and workloads from CSRBench, and provides a full evaluation platform. Rather than verifying every window result, we choose to measure the validity of overall result. The reason is that there is no efficient algorithm to compare mismatch of window results, even YABench's method has several restrictions. More importantly, we propose a diverse workload to detect issues that previous benchmarks are unaware of.

- Heaven [36] and RSPLab [37] are two recent works focusing on how to efficiently evaluate the SRP engines. Heaven proposes an experiment environment to enable comparative evaluation against SRP engines. Users can integrate their own datasets and workloads with the test-stand to conduct the experiment. RSPLab goes a bit further: it used RESTful API to communicate the SRP engine rather than the JAVA interface. Not all RSP engines are written in Java, they cannot be tested via Java interfaces. Instead, RSPLab uses a set of REST APIs to communicate with SRP engines. The REST APIs generalize the services such as stream registration, query registration and result consumption, so the usage of RSPLab is independent of specific engine implementation. In addition, it continuously monitors the performance and visualizes the data automatically.

We integrate our Stream WatDiv with YABench testbed environment. We discuss the integration in more detail in Chapter 4.2.

2.3 Analysis of Workloads of Previous Benchmark

During last ten years, although several SRP engines have been proposed, there is still no single standard for RDF stream query languages. Each SRP engine has its own query language. Therefore, when evaluating these engines, one of the major challenges is functionality test. Another common observation is that existing SRP engines are not mature enough, so verifying the correctness of query result is also necessary. Based on these two observations, some of the benchmarks design fine-selected workloads to fully cover query operations, while some of them have a small workload set to do correctness verification and performance evaluation. In this section, we will demonstrate that these existing benchmarks have not a diversified enough workload to illustrate a complete picture of strengths and weaknesses of SRP engines, and the engines' robustness in face of various queries.

We consider 5 benchmarks. Table 2.3 gives a summary of the workloads in these benchmarks. The analysis is based on two popular SRP engines, C-SPARQL [12, 9, 8, 11, 10] and CQELS [24], as 5 benchmarks all choose these two as target evaluation engines. We define a valid query as the query that only contains query operations supported by both C-SPARQL and CQELS. The successful query refers to the query that can successfully run and produce correct result on both C-SPARQL and CQELS. We count number of different queries.

- SRBench [39] has a fin-selected workload to conduct the functionality test. Among 17 queries, only 7 of them are supported by both C-SPARQL and CQELS. The correctness of the query result is not checked.
- CSRBench [19] builds an oracle to verify the query result of 7 queries. Among these 7 queries, three of them are actually the same query with different time windows. For these three queries, both C-SPARQL and CQELS produce the correct result.
- LSBench [25] verifies the correctness by calculating the mismatch between different engines' results. This method makes sense only if majority of target engines produce the correct results, which is not guaranteed. Among 12 queries in the workload, 7 of them are valid queries. None of them is guaranteed to be successful query.
- YABench [23] builds an oracle with richer measurement criterion. It uses the same workload from CSRBench.
- CityBench [4] is built upon smart city applications. All 13 queries in the workload are valid queries. No correctness verification is conducted for these queries.

query number	SRBench	CSRBench	LSBench	YABench	CityBench
total query	17	7	12	7	13
valid query	7	7	7	7	13
successful query	unknown	3	unknown	3	unknown

Table 2.3: Workloads in previous benchmarks

To sum up, both SRBench and LSBench have workloads covering almost all query operations, but only a few of the queries are supported by existing engines. CSRBench and

YABench check the query result correctness, but they do not provide a diversified workload. CityBench contains most number of valid queries. However, the workload is specific to smart city application scenarios.

We want a diversified workload that can be successfully run on different SRP engines. Instead of building a workload covering as many query operations as possible, we restrict our workload to a small set of query operations. This small set of query operations is based on a basic fragment of SPARQL, namely, basic graph patterns(BGPs) with filter expressions. Since existing RDF stream query languages are extended from SPARQL, BGP is also the basic fragment of RDF stream query language. And other advanced query operations can be extended from BGP. In this way, the workload is supported by almost all SRP engines, independent of different RDF stream query languages.

Using this workload, we can study the performance influence of different query structures and various selectivity. Queries with different structures can be used to test engines' robustness in face of different query complexity. Analyzing a query's triple selectivity can help user understand how the engine generates execution plan. All of previous benchmarks are not able to study these topics, and our work will focus on these.

Chapter 3

Stream WatDiv Benchmark Suite

Previous benchmarks [39, 19, 25, 23, 4] provide a good foundation for the study of streaming RDF processing (SRP) engines. Different benchmarks emphasize different measurement targets. Some benchmarks cover as many query language features as possible, even though the features are not supported by existing engines. Some conduct correctness and performance experiments for a small set of queries. Overall, none of the existing benchmarks have diversified workloads that can be used to evaluate performance across existing engines. The RDF system benchmark WatDiv [5] has wide coverage of query workloads; therefore, we extend it with streaming features. In this chapter, we first discuss the design principles of Stream WatDiv and review the challenges that need to be addressed in extending WatDiv benchmark suite to incorporate streaming RDF. We then present stream WatDiv focusing on the details of the data generator and query generator.

3.1 Stream WatDiv Design Principles

The objective of Stream WatDiv is to facilitate performance evaluation of SRP engines. To achieve this, we identify the following design principles.

Compatibility with existing systems. There is no standard streaming RDF query language. Each engine (e.g., C-SPARQL and CQELS) defines its own query language, but they all extend SPARQL with time window operations. They also all support basic SPARQL operations such as basic graph pattern matching. However, they implement different sets of advanced operations. For example, CQELS does not support *NOT EXIST*, while C-SPARQL does; C-SPARQL does not support defining multiple windows on the same stream, while CQELS does. Furthermore, most of these engines do not support new operations introduced in SPARQL 1.1 such as *Negation* and *Subqueries*. We wish Stream WatDiv to take these differences into account. If the query is not executable on the target engines, it does not help to evaluate the performance across the engines. Previous benchmarks have focused mostly on functionality tests to show the features supported by each engine; we wish to have a workload that is compatible with most engines so as to conduct performance evaluation across them. At the same time, we want the query generator to be extensible to accommodate future developments in terms of functionality.

Diversity of the workload coverage. When processing a query, a SRP engine may generate the query execution plan that tries to minimize the intermediate result in each step. For example, suppose a simple query contains several triple patterns $\{tp_1, tp_2, tp_3, tp_4\}$, CQELS will join tp_1 with tp_2 first because the intermediate result by joining these two triple patterns is smallest, then in each of the following steps, the engine will always pick a remaining triple pattern which has the smallest set of matched triples. However, this greedy strategy may produce sub-optimal query execution plans. For example, in the previous simple query, if each triple pattern is almost equally selective, but only some of them contribute to pruning the intermediate result, the greedy strategy may not be aware of this and not execute these triple patterns first in the query execution plan. Furthermore, this sub-optimal query execution plan may generate a huge intermediate result that cannot fit into the memory. Existing engines are immature and may crash due to running out of memory. In our benchmark, we want the workload including these cases. In particular, the queries in the workloads should contain various combinations of triples with different selectivity.

Scalablitly. New SRP engines are continuously being developed and their features are being improved. In the future, these engines should be able to handle larger amounts of data and process more complex queries. A benchmark should be able to both test current immature engines, but also simulate future complex workloads. More specifically, the data generator should be able to generate various size of the static dataset and big enough stream dataset to produce RDF streams with high stream rates. The query generator should be able to generate queries with different complexity.

Customizability. The benchmark should provide necessary parameters so users can customize their own settings. For example, when generating the datasets, users should be able to decide the size of static dataset and length of RDF stream. When generating the workload, users should be able to control the query complexity.

Reproducibility. The benchmark results should be reproducible. This means that the performance evaluation platform should ensure that the results are deterministic for a given dataset, workload and experiment setting.

3.2 Challenges

In the previous chapter, we analyzed previous benchmarks whose workloads only contain very limited set of queries to evaluate existing SRP engines. Based on our design principles, we want to focus on testing engines' capability in handling a basic fragment of stream query language, namely basic graph pattern matching, but with a diversified workload. WatDiv conducts a diversified stress testing of RDF data management systems that manage static data sets. However, WatDiv cannot be directly used for evaluating SRP engines. To obtain our objective of developing a streaming RDF benchmark based on WatDiv consistent with the principles identified above, it is necessary to address the following challenges.

Streaming RDF triples. SRP engines take streaming RDF triples as input. These time-annotated RDF triples ordered by the timestamps are sent to the engines, forming the RDF stream. Each triple is valid for some time and expires when its timestamp exceeds the active time window range. Streaming RDF triples are required to be interlinked with static RDF dataset, so complex queries can match triples from both streaming data and static data.

In Stream WatDiv, we meet this challenge by modifying and dividing the data schema into a streaming data schema and a static data schema. The whole data schema describes an e-commerce website with social network functions. The streaming data simulates users' activities on the website. For example, users can purchase or write reviews for products. The static data is interlinked with the streaming data by providing metadata of entities that appeared in the streaming data.

Query features of workloads. WatDiv's diversity lies in two sets of query features: structural features and data-driven features (see Section 2.1). Both of these can be directly applied to streaming RDF queries if they only involve streaming data. However, for queries involving both static and streaming data, these query features represent a challenge due to the following factors.

The first factor is that SRP engines might cache the intermediate result. In RDF data management systems, the query will only run on-demand, while in streaming RDF processing systems, the queries are typically continuous and are re-executed periodically. This gives the engines the chance to cache intermediate results between consecutive query executions. More specifically, in SRP engines, the query may need both streaming data and static data. The streaming data is updated when the window slides, while static data is rarely updated. Therefore, the static part of the query result can be materialized and stored in a cache. In this case, when the triple selectivity in the query is computed, connected components in the static part of the query graph are considered as a whole. For example in Figure 3.1, the static part of the query contains four triple patterns $\{t_2, t_3, t_5, t_6\}$. When we calculate the triple selectivity of static part of the query, the selectivity of $\{t_5, t_6\}$ and $\{t_2, t_3\}$ are considered, instead of $\{t_2\}$, $\{t_3\}$, $\{t_5\}$, $\{t_6\}$.

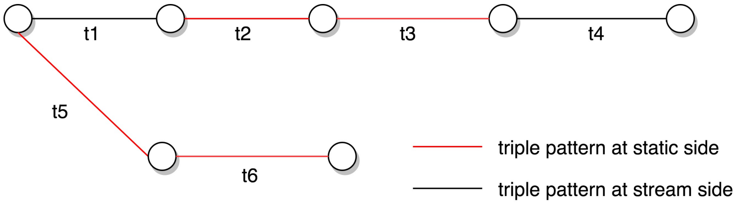


Figure 3.1: If the SRP engine caches intermediate results of static part of the query between consecutive query execution, when calculating the mean and standard deviation of triple selectivity, the selectivity of $\{\{t_1\}, \{t_2, t_3\}, \{t_4\}, \{t_5, t_6\}\}$ is considered rather than that of $\{\{t_1\}, \{t_2\}, \{t_3\}, \{t_4\}, \{t_5\}, \{t_6\}\}$.

The other factor is that SRP engines have different approaches to processing data compared with RDF data management systems. In RDF data management systems, a SPARQL query is run upon a single static data set. However, a continuously streaming RDF query can involve both static data and streaming data. In SRP engine, each query execution is normally run upon both the static data and active window content. So the definition of a f-TP selectivity need to be modified to fit in streaming setting. Formally, let W_i denotes the active window content, G denotes the static data, Ω represents the set underlying the bag of $[\![\lambda^F(\{tp\})]\!]_{W_i \cup G}$, then the f-TP selectivity of a triple pattern $tp \in B$ in a CBGP $\overline{B} = \langle B, F\rangle$, denoted by $SEL^F_{W_i \cup G}$ is defined as:

$$SEL^F_{W_i \cup G} = \frac{|\Omega|}{|W_i \cup G|}. \tag{3.1}$$

As we can see, the value of triple pattern selectivity in a streaming RDF query can be influenced by the static data and the active window content. This is very different from a SPARQL query considered in WatDiv where the selectivity is only determined by the static data. So in Stream WatDiv, only when the static data, streaming data, stream rate, window length and sliding step are fixed, the triple pattern selectivity of queries in the workload is fixed.

3.3 Stream WatDiv Design

Stream WatDiv benchmark suite contains multiple tools that enable performance evaluation across streaming RDF processing engines. In this section, we focus on the two components of Stream WatDiv: the data generator and the query generator. Both of these are extended from WatDiv.

WatDiv data generator generates scalable datasets based on a data model as discussed in the previous chapter.

WatDiv query generator consists of two parts:

- *Query template generator* performs the random walk in the data graph, and produces a set of diversified query templates. Users can specify the number of query templates needed, control the size of query templates and restrict the number of placeholders in a query.
- *Query generator* takes query templates as input, replace the placeholders in query templates with real RDF terms from the dataset, and generates the query instances of query templates.

To extend WatDiv to test streaming RDF processing engines, these components are modified to accommodate streaming data. The data generator generates both static data and streaming data. For the latter, each triple is assigned a timestamp. The timestamp interval determines the stream rate. Once the data is generated, the query template generator produces a set of query templates based on both static data and streaming data. We note that in streaming extension, we are only interested in pure stream queries (those that only access the streaming part of the RDF database) and hybrid queries that access both static and streaming data. Finally, the query generator will replace the placeholders in the query template with real RDF terms from datasets. Since there is no standard streaming RDF query language yet, the query generator re-writes the queries with different streaming RDF query languages by providing translators for the existing SRP engines. All these tools provide various parameters to customize the datasets and workloads. The following sections will describe each tool in detail.

3.3.1 Data Generator

Performance evaluation across SRP engines need both a static dataset and a streaming dataset. In order to generate two types of dataset, we modify WatDiv's data generator and the data schema. Figure 3.2 illustrates the logical schema of static data and streaming data. This schema extends the online shopping website schema implemented by WatDiv.

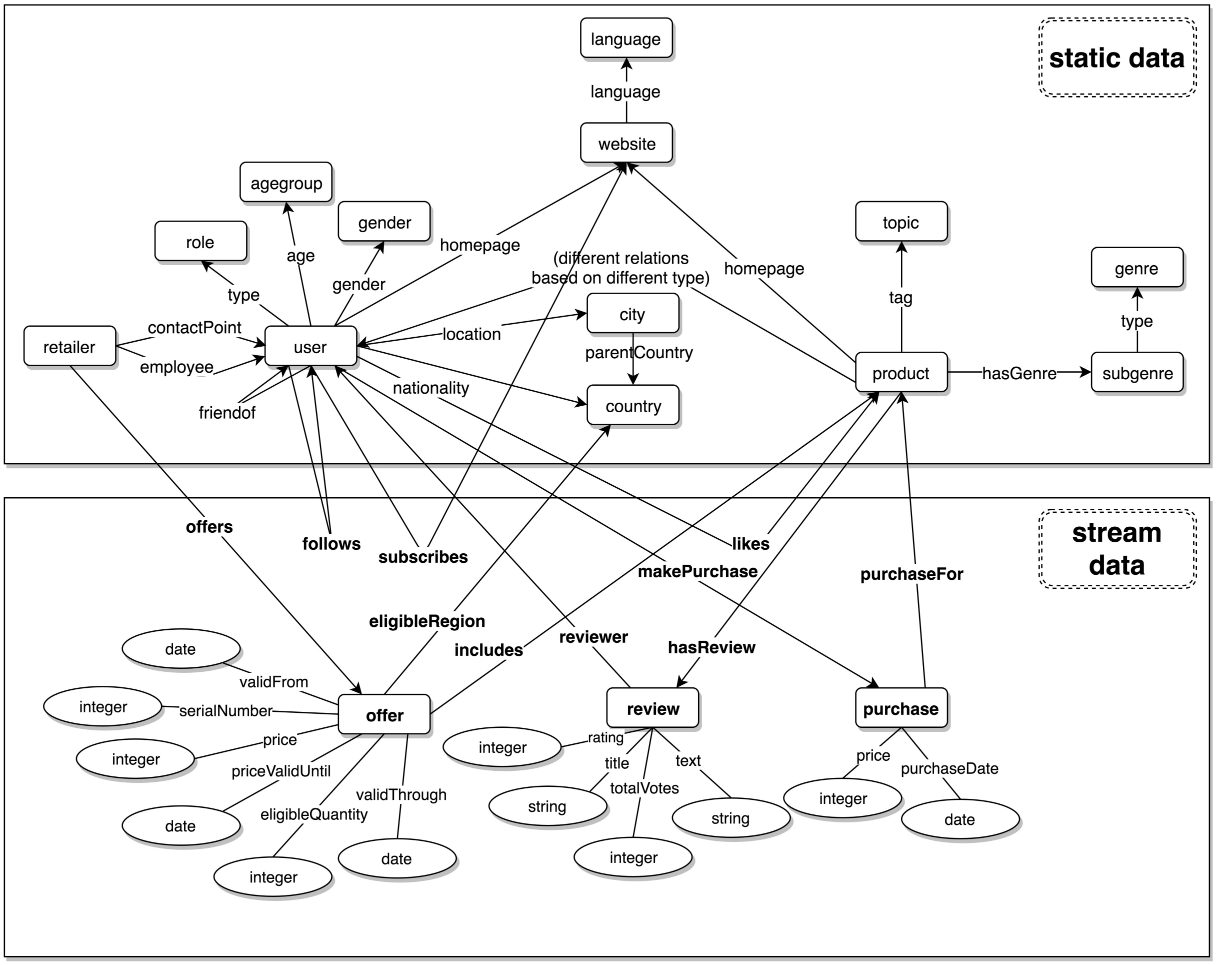


Figure 3.2: Data schema of static data and streaming data

Streaming data simulates users' activities on the shopping website. More specifically, streaming data contains six activities. Three of them are more complex, having their own entity types in the streaming data.

- Purchase: The most important activity on the website is the purchase. Users are able to purchase various products. Each purchase entity will record a user's purchase information for a single product, such as price and date.

- Review: Users can also write reviews for products. The review entity contains information like rating, title, content and votes.
- Offer: A retailer might make some offers of products. The offer may have restrictions on eligible region and valid time period, so the offer entity should include these details.

The remaining three activities are more straightforward, only including a single triple.

- Likes: Users can show interest in products by "liking" them.
- Follows: Users can make friends by following each other.
- Subscribes: If a user wants to get updates on other users or some products, the most convenient way is to subscribe to their websites.

Static data contains all user and product metadata, and other necessary background information. We assume the static data rarely changes. User, Product and Retailer entities appearing in the streaming data are linked with the entity metadata in the static data.

Scale factor: The WatDiv data generator allows users to set the scale factor to control the size of the dataset. When the scale factor increases, the number of instances of most entity types will increase proportionally. After modifying the data generator to produce a streaming dataset and a static dataset, we want to have separate scale factors as well. The stream scale factor controls the size of the streaming dataset. If the stream scale factor increases, more user activities are generated. The static scale factor controls the size of the static dataset. If the static scale factor increases, more metadata of the website is generated. It is notable that increasing static scale factor will increase streaming dataset as well, while increasing stream scale factor has trivial influence on the static dataset. Table 3.1 shows how scale factors change the size of two datasets.

{Static Scale Factor, Stream Scale Factor}	Static Triples	Streaming Triples
{1, 1}	55,504	56,978
{1, 100}	59,251	5,576,489
{100, 1}	5,536,852	4,754,669
{100, 100}	5,572,466	475,211,828

Table 3.1: Influence of static scale factor and stream scale factor on datasets

Timestamp: As noted earlier, an RDF stream is an ordered sequence of pairs, where each pair consists of an RDF triple and an integer timestamp.

$$< subject,\ predicate,\ object,\ timestamp >$$

In our benchmark, once we generate a sequence of RDF triples, we need to attach a timestamp to each triple and transform the streaming data to an RDF stream. The timestamp interval between two consecutive triples indicates when the next triple should be sent to the engine. If the stream rate is high, the timestamp interval will be small. We allow the user to set the stream rate when using the data generator, and compute the interval accordingly. The time unit of the timestamp is milliseconds. If the stream rate is higher than 1000 triples/second, then certain number of triples will be sent to the engine in a batch with the same timestamp. We use the batch size to represent the number of triples which should be sent to the engine together at the same time. Formally, if the user-defined stream rate is $\mathcal{R}$, the batch size is $\mathcal{B}$, the timestamp interval is $\mathcal{I}$, and the RDF stream is $S = \{< s_0, p_0, o_0, \tau_0 >, < s_1, p_1, o_1, \tau_1 >, ..., < s_n, p_n, o_n, \tau_n >\}$, then the relation between $\mathcal{R}$, $\mathcal{B}$ and $\mathcal{I}$ is indicated below. Intuitively, based on stream rate $\mathcal{R}$, for every $\mathcal{B}$ triples, we increase the timestamp by $\mathcal{I}$.

$$\mathcal{B} = max\{\frac{\mathcal{R}}{1000}, 1\}. \tag{3.2}$$

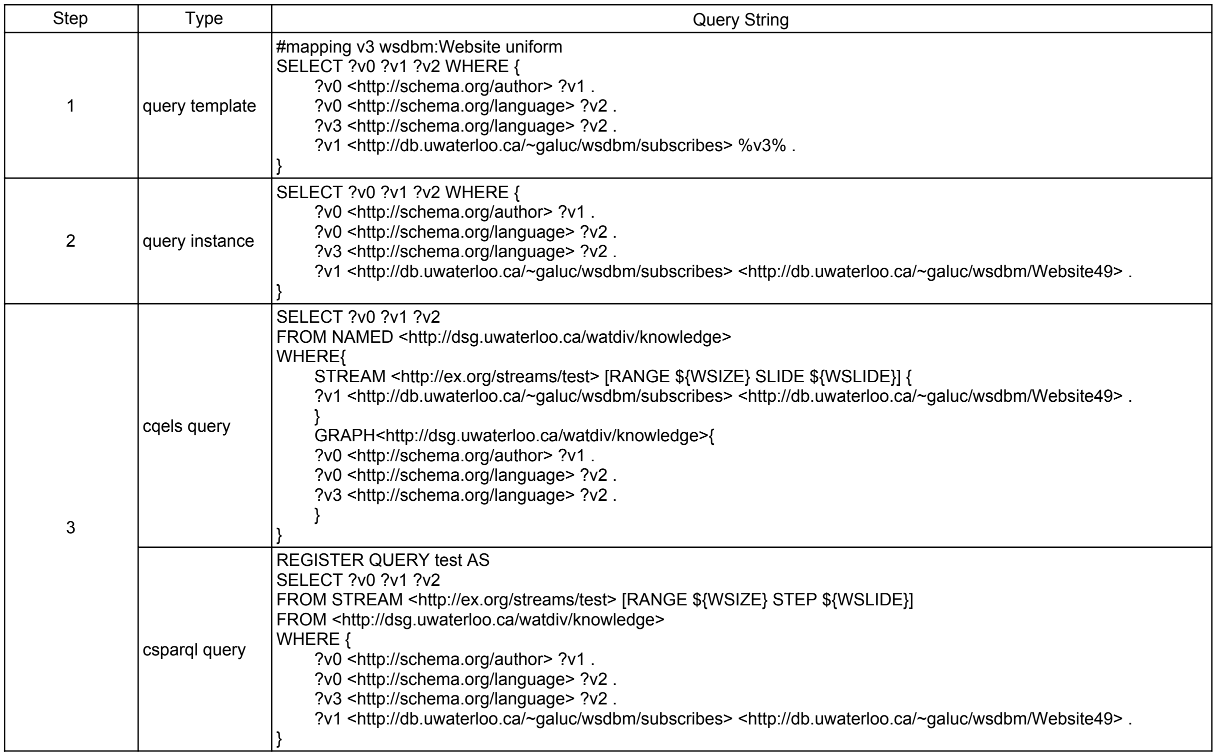
$$\mathcal{I} = max\{\frac{1000}{\mathcal{R}}, 1\}. \tag{3.3}$$

$$\tau_i = \tau_0 + \frac{i}{\mathcal{B}} * \mathcal{I}. \tag{3.4}$$

Reproducibility: As noted in design principles, one requirement of a benchmark is reproducibility. The key point is to make the test data generation reproducible. WatDiv's data generator relies on a random number generator to generate the data. For example, assume the type of user-follow-user relationship is many-to-many, and each involved user can follow at most 15 other users, then for each user, the data generator will use random number generator to uniformly pick a number between 2 to 15, then uniformly select this number of users from candidate user set to pair with that user. In WatDiv's data generator, the random seed of the random number generator is the system time. So each time the data generator is run, the random number generator will produce a different random number sequence and, thus, the data generator will generate a different dataset. This becomes an issue if we want to re-run the experiment on another machine and the original dataset has been unavailable. The sequence of the stream data may happen to be critical to verify the result. In stream WatDiv, the data generator takes a random seed as a parameter. Each time the data generator is run with the same random seed and other parameters, the same dataset will be produced.

3.3.2 Query Generator

Stream WatDiv's query generator follows WatDiv's two-step approach: query template generation followed by query instance generation according to these templates. However, Stream WatDiv requires one extra step to translate the queries into different streaming query languages. Steps in Figure 3.3 give an example of the entire query generating process. Users can determine the number of placeholders in the query templates, query instances for each query and the size of queries. We first describe the differences in the implementation of the first two steps in WatDiv and Stream WatDiv. Later, the query translation process is discussed.



Step	Type	Query String

1	query template	#mapping v3 wsdbm:Website uniform
		SELECT ?v0 ?v1 ?v2 WHERE {
		?v0 <http://schema.org/author> ?v1 .
		?v0 <http://schema.org/language> ?v2 .
		?v3 <http://schema.org/language> ?v2 .
		?v1 <http://db.uwaterloo.ca/~galuc/wsdbm/subscribes> %v3% .
		}

2	query instance	SELECT ?v0 ?v1 ?v2 WHERE {
		?v0 <http://schema.org/author> ?v1 .
		?v0 <http://schema.org/language> ?v2 .
		?v3 <http://schema.org/language> ?v2 .
		?v1 <http://db.uwaterloo.ca/~galuc/wsdbm/subscribes> <http://db.uwaterloo.ca/~galuc/wsdbm/Website49> .
		}

3	cqels query	SELECT ?v0 ?v1 ?v2
		FROM NAMED <http://dsg.uwaterloo.ca/watdiv/knowledge>
		WHERE{
		STREAM <http://ex.org/streams/test> [RANGE ${WSIZE} SLIDE ${WSLIDE}] {
		?v1 <http://db.uwaterloo.ca/~galuc/wsdbm/subscribes> <http://db.uwaterloo.ca/~galuc/wsdbm/Website49> .
		}
		GRAPH<http://dsg.uwaterloo.ca/watdiv/knowledge>{
		?v0 <http://schema.org/author> ?v1 .
		?v0 <http://schema.org/language> ?v2 .
		?v3 <http://schema.org/language> ?v2 .
		}
		}

	csparql query	REGISTER QUERY test AS
		SELECT ?v0 ?v1 ?v2
		FROM STREAM <http://ex.org/streams/test> [RANGE ${WSIZE} STEP ${WSLIDE}]
		FROM <http://dsg.uwaterloo.ca/watdiv/knowledge>
		WHERE {
		?v0 <http://schema.org/author> ?v1 .
		?v0 <http://schema.org/language> ?v2 .
		?v3 <http://schema.org/language> ?v2 .
		?v1 <http://db.uwaterloo.ca/~galuc/wsdbm/subscribes> <http://db.uwaterloo.ca/~galuc/wsdbm/Website49> .
		}

Figure 3.3: One example showing the process of generating a query

Query type: In WatDiv query template generator, query templates are selected through random walks starting from a random node in the dataset. As noted earlier, the streaming dataset and static dataset are interlinked (see Figure 3.2). Stream WatDiv's query template generator conducts a random walk across both of these datasets. The selected query templates could be either pure stream query templates, hybrid query templates or pure static query templates. Stream WatDiv extends original query template generator over static data by including only pure stream queries and hybrid queries. Stream WatDiv's query generator instantiate the query templates to query instances in the same way as WatDiv. Like data generator, we also include the random seed in the query generator to guarantee the workload generation reproducible.

Query translation: As noted earlier, there is no standard streaming RDF query language at the moment. To test a specific engine, the queries in the workload need to be translated into that engine's query language. In the current version of Stream WatDiv, we provide translators for two engines, C-SPARQL and CQELS, that we evaluate (see next chapter). In C-SPARQL, the stream source is defined in *FROM STREAM* clause. The time window is defined with *RANGE* and *STEP* keywords. In CQELS, the stream source is defined in *GRAPH* clause. The streaming triple patterns should be explicitly indicated by grouping together in the same scope. The time window is defined with *RANGE* and *SLIDE* keywords. Step 3 in Figure 3.3 shows final queries for CQELS and C-SPARQL.

Chapter 4

SRP Evaluation Using Stream WatDiv

As SRP engines have started to emerge over the past ten years, one important challenge is to evaluate performance across SRP engines. In this chapter we use Stream WatDiv to conduct the correctness test and performance evaluation of two systems: C-SPARQL and CQELS. Our objective is not to perform an exhaustive performance evaluation, but to primarily demonstrate that Stream WatDiv can be effectively used to evaluate these systems better than previous benchmarks. A secondary objective is to validate the performance results previously reported about these systems.

We first start with a more detailed discussion of C-SPARQL and CQELS. In particular, we highlight modifications that we made to C-SPARQL in order to make it more robust across workloads. Other SRP engines are introduced as well. We then discuss the testbed and the experimental setup. Our testbed is built upon YABench [23]. The experiments are divided into four parts. First, we check how many of the queries in the workload successfully pass the correctness test in both engines. Second, for those failed queries, we analyze the failure reasons. Third, for the successful queries, we measure the latency under different stream rates, and compare the performance across these two engines. Fourth, we group the queries based on the query features and measure how well C-SPARQL and CQELS handle specific types of queries.

4.1 Streaming RDF Processing Engines

As noted earlier the existing systems are quite primitive. Their query languages extend SPARQL with window operations. Among the existing systems, we have chosen C-SPARQL and CQELS for the performance study because they are more widely known and they have been previously evaluated. Furthermore, both of them are open source and well-maintained [1][2]. More importantly, they have lots of differences in design that we highlight below, and these allow us to test the impact of these different design decisions.

4.1.1 C-SPARQL

C-SPARQL [12, 9, 8, 11, 10] is an SRP engine built upon existing DSMSs/CEPs and RDF data management systems. The C-SPARQL version we test in this thesis adopts Esper [3] and Jena. Esper is a CEP engine which have features like highly scalable, in-memory computing and memory-efficient. It targets to detect patterns from inputs below 1 millisecond. Jena is an open-source framework for building linked data applications. It contains a high performance RDF store and SPARQL query engine.

C-SPARQL adopts periodic execution strategy. Whenever the active window closes, the engine will trigger the query execution, and report the output. C-SPARQL utilizes Esper and Jena to respectively handle the stream data and process the SPARQL query. Figure 4.1 shows the components of C-SPARQL engine. When a query is registered in C-SPARQL, it will be divided into two parts. The time window operation is sent to Esper, and the query body is translated into an equivalent SPARQL query. Esper takes the responsibility for window management. It will take the snapshot of window content whenever the active window closes, and send it to C-SPARQL's query executor. Once the query executor get the window content snapshot, it clears the Jena RDF database, load the static data and window content snapshot, and run the SPARQL query on it. Since it uses these external engines for query processing, C-SPARQL itself does not have the control of query processing. The optimization depends on the underlying CEP engine and RDF data management system.

Previous works have identified two issues of C-SPARQL: its incapability to handle large static data and imprecise time window management. Even though these two issues have been reported, none of previous works have successfully located the factors causing these

[1] https://code.google.com/archive/p/cqels
[2] https://github.com/streamreasoning/C-SPARQL-engine
[3] http://www.espertech.com/esper

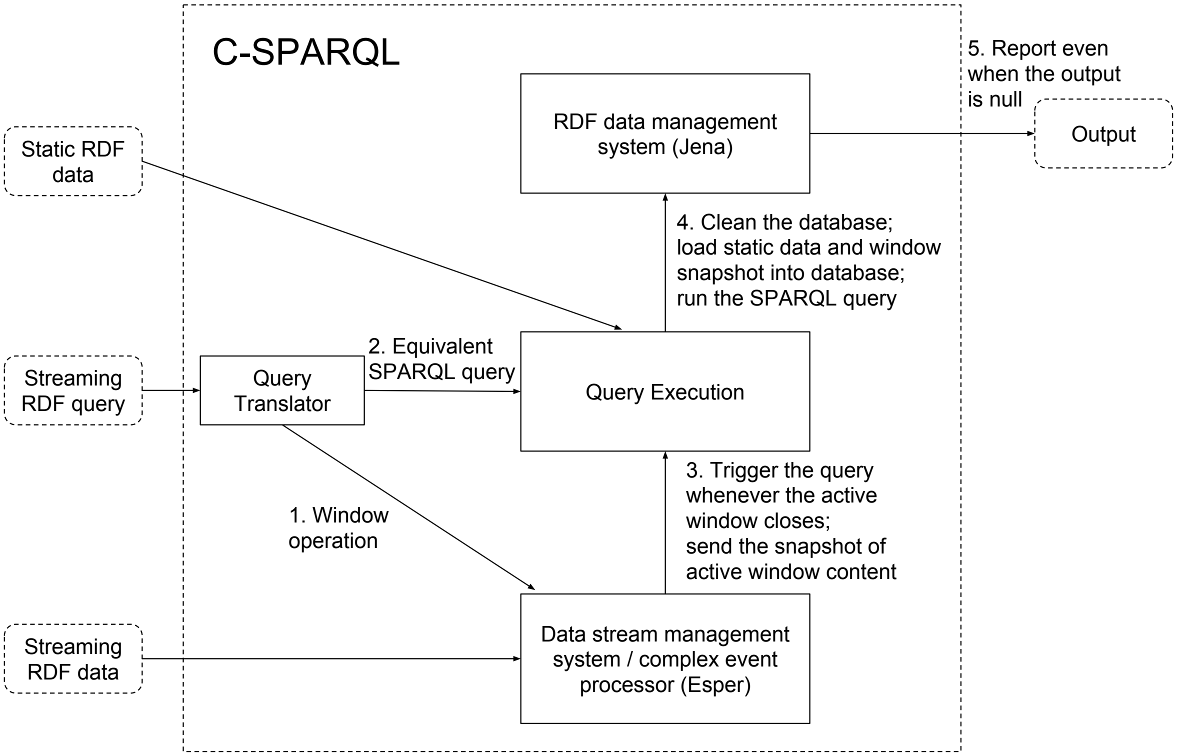

C-SPARQL
Static RDF data
Streaming RDF query
Streaming RDF data
Query Translator
RDF data management system (Jena)
Query Execution
Data stream management system / complex event processor (Esper)
Output
1. Window operation
2. Equivalent SPARQL query
3. Trigger the query whenever the active window closes; send the snapshot of active window content
4. Clean the database; load static data and window snapshot into database; run the SPARQL query
5. Report even when the output is null

Figure 4.1: Steps to process a streaming RDF query in C-SPARQL

issues, or have solved them. In our work, we have analyzed them in-depth and provide potential solutions to address them.

In our experiments, we observe that when running a query involving large static dataset, C-SPARQL will crash, reporting *ConcurrentModificationException*. The issue is caused by how C-SPARQL executes the query. C-SPARQL relies on Jena to generate the final result. Whenever re-executing the query, C-SPARQL will clear the Jena database, load all static data and active window content into Jena database, then run the equivalent SPARQL query to generate the final result, and finally wait for the next query execution. Consecutive query executions all use the same Jena database instance. If the static dataset is too big, loading data into Jena or cleaning data from Jena may take longer than the execution time, and this causes *ConcurrentModificationException*. To fix this issue, the static data are loaded into Jena database once when the first query execution is executed, consequent query executions only replace the active window content in Jena database. We show later that, even with this modification, when the stream rate is high and window content is large, this issue will re-appear, but at least less frequently than before.

Another big issue is that C-SPARQL cannot precisely manage the data around the window borders. We observe from the experiments that data around the window borders will leak from current window to the next window, and the window tends to close earlier than expected. We observed over several rounds of the experiments that the window closes earlier at different degrees. The results from YABench are consistent with our observations. YABench reports the occurrence of negative latency of C-SPARQL query execution, which means C-SPARQL reports the query result even before the window closes. Unfortunately, YABench does not give an explanation of this weird behaviour. Inspection of the source code of C-SPARQL revealed that the reason lies in how C-SPARQL sets the initial time, which is the time the first window starts. C-SPARQL sets the initial time with the timestamp of first arrived stream data. The time unit of C-SPARQL is seconds, while the timestamp is represented in milliseconds, so C-SPARQL rounds the timestamp downward. For example, if the timestamp of the first stream data is 1507791779400 (millisecond), C-SPARQL will set the initial time to $\frac{1507791779400}{1000} = 1507791779000$. Through this process, the first actual window starts 400 milliseconds earlier than expected. Since the window length is fixed, this causes all the subsequent windows to close earlier. Since the timestamp of the first stream data is determined by the time when the data is sent to C-SPARQL, different rounds of the experiment show varying degrees of this weird behavior. We modified C-SPARQL to set the initial time in milliseconds. Although minimum time unit supported in C-SPARQL query language is still in seconds, management of window content is now conducted in milliseconds.

4.1.2 CQELS

CQELS [24] is an SRP engine built from scratch. It differs from C-SPARQL in that instead of using external systems to manage windows and to process the query, CQELS implements these functions naively – it parses the query, generates the query execution plan and conducts the continuous query execution. Unlike C-SPARQL, even though CQELS extends SPARQL with the *SLIDE* keyword to define the slide step, the slide step has no effect on when the query execution should be triggered.

CQELS adopts the eager execution strategy in contrast to the periodic execution strategy of C-SPARQL. Whenever the new stream data arrives and the active window content changes, the engine will trigger the query execution. CQELS will not report the result if the result is null, while C-SPARQL will report the result for query execution even if the query result is null.

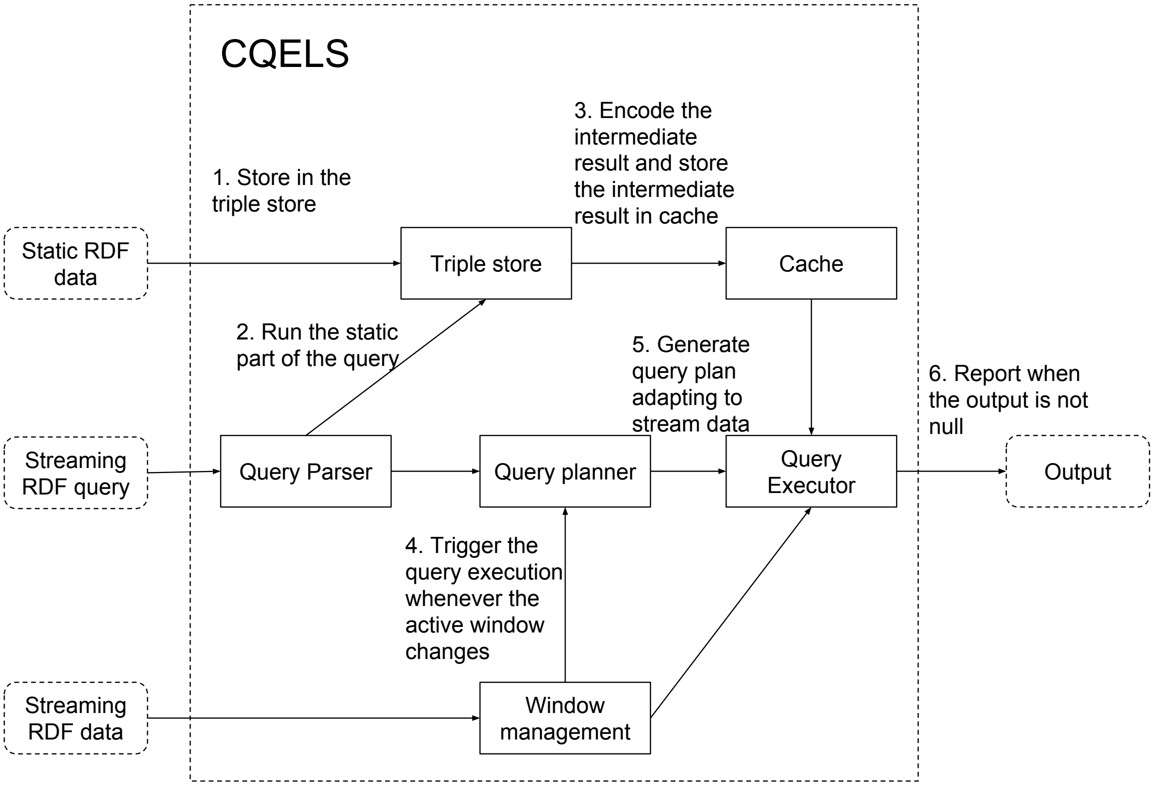

CQELS
1. Store in the triple store
3. Encode the intermediate result and store the intermediate result in cache
Static RDF data
Triple store
Cache
2. Run the static part of the query
5. Generate query plan adapting to stream data
6. Report when the output is not null
Streaming RDF query
Query Parser
Query planner
Query Executor
Output
4. Trigger the query execution whenever the active window changes
Streaming RDF data
Window management

Figure 4.2: Steps to process a streaming RDF query in CQELS

Instead of relying on the RDF triple store to process both the stream data and the static data as C-SPARQL does, CQELS implements its own query processing framework for both the stream data and the static data. Figure 4.2 illustrates how CQELS processes a query. When a query is registered in CQELS and the query involves static data, the static data will be loaded into the triple store and the static part of the query will be executed on it. This starts even before processing the streaming data. The intermediate result of the static part of the query will be encoded and cached for following query executions. Then based on the window content, the query planner will generate a dynamic query execution plan. The execution order of the operations in the query plan aims to minimize the intermediate result. The triple pattern matching of the static part of the query and the streaming part of the query are processed separately, and the results are joined together based on the query plan. The native processing engine implementation allows CQELS to perform a number of optimizations. For example, encoding the RDF nodes such as subjects, predicates and objects, as integers could reduce the data size, so as to load as many data as possible into memory and reduce the external disk access. This will certainly accelerate the query execution. In addition, operations on integers will be more efficient than on strings. For static RDF data, all RDF nodes in the intermediate result are encoded as integers before storing into the cache. For streaming data, since they are updated frequently and needs to be processed in real-time, encoding every RDF nodes may hinder the performance. So only the ones which cannot be presented in 63-bits are mapped to integers. CQELS also caches and indexes the intermediate result, especially for static part of the query. For streaming part of the query, indexing the intermediate result could accelerate the join operation. However, the streaming query result is updated frequently, so only if the index is updated faster than the stream rate, index will be built on streaming query result.

4.1.3 Other SRP Engines

In addition to the two engines that we discussed in previous sections, there are a few others that have been developed. We highlight them in this section for completeness.

- Morph-stream [17] is an ontology-based data access system implemented based on $SPARQL_{stream}$ [16]. It represents stream data through ontological models, while internally it manages the data in data stream management systems or complex event processing engines. It translates $SPARQL_{stream}$ queries into ones suitable to DSMS/CEP engines like Esper, GSN [3] and SNEE [21].

- EP-SPARQL [6] extends SPARQL with event processing and stream reasoning capabilities. Both the stream data and triple patterns in query graph are maintained in a temporal order. Then the graph pattern matching is conducted based on this order information.

- CQELS-Cloud [26] is the first distributed SRP engine. It uses Apache Storm [4] to process stream data. It implements parallel incremental evaluation algorithms for both stateless operations such as select and filter, and stateful operations such as aggregation and join. The experiments illustrate the scalability when increasing the number of deployed machine nodes, and the scalability of handling multiple parallel queries. However, the system as a whole is not open source.

[4] http://storm.apache.org

- Strider [32] is a most recent distributed SRP engine. It is implemented upon Apache Kafka [5] and Spark Streaming [6]. Kafka is used to manage the stream data, and Spark Streaming is used to process query operations. Queries are re-executed every micro-batch interval specified by the *BATCH* clause. It proposes two adaptive algorithms to generate the run-time query plan. The experiments claim its advantage over other engines in coping with high stream rate. Currently, Strider does not support querying static background data.

4.2 Experiment Setup

The testbed we use to conduct the experiments is based on YABench. YABench implements the oracle approach to verify the engines' output and evaluate the engines' performance in terms of reporting latency for each window. The oracle will calculate the standard output for each query execution. We treat the oracle output as the correct result, and use it to validate the engine output. We modify and reuse the oracle from YABench, rewrite the data player which issues the stream data to the test engines. Figure 4.3 presents the overall architecture of our testbed.

The stream dataset contains a sequence of RDF triples annotated with monotonically non-decreasing timestamps. The data player will read these stream data, send them to target SRP engines based on the timestamp. Meanwhile, the query will be registered on the target SRP engine. To verify the engine's output, an oracle is built to offline calculate the correct output, and compare the engine's output with the oracle output. The oracle will load each window content and static data into Jena, then an equivalent SPARQL query will be run on it. By comparing the oracle output with the engine output, we perform correctness check and performance measurement. The two SRP engines under evaluation, C-SPARQL and CQELS, are integrated with the testbed. In this section, we will discuss some modifications we made to the YABench testbed.

YABench checks the correctness and measures the latency for each query execution taking into account different engines' query execution mechanisms. As noted earlier, C-SPARQL executes the query and outputs the result when the active window closes, while CQELS executes the query and outputs the result when the window content changes and the result is not empty. The oracle simulates the same execution mechanisms. So for each engines' output, there will always be a corresponding oracle output to compare with.

[5] https://kafka.apache.org

[6] https://spark.apache.org/streaming

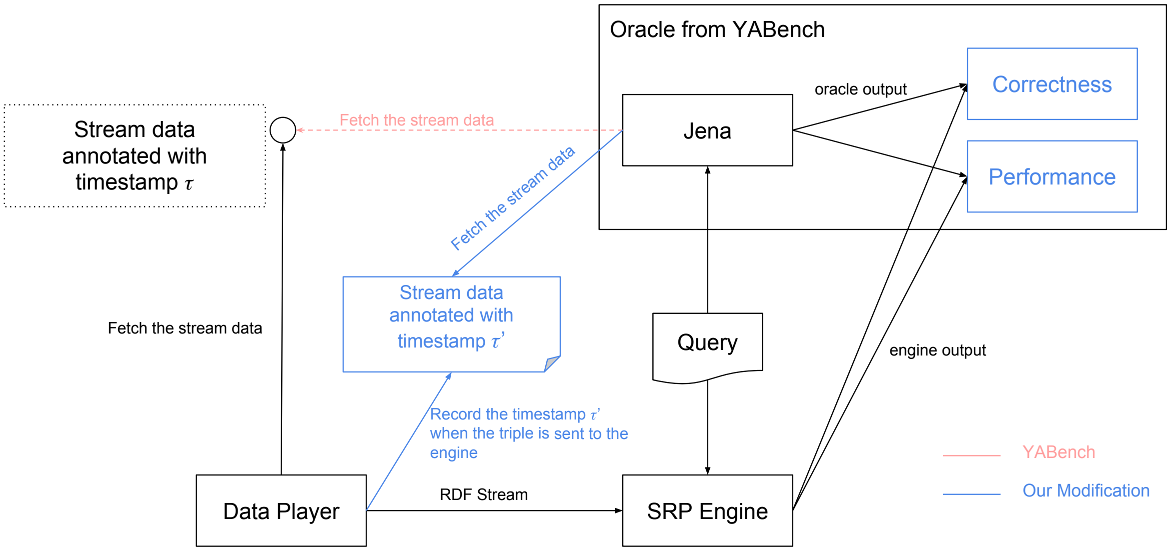


Figure 4.3: The architecture of testbed

It's easy to check C-SPARQL's query result, since one output represents the query result of one specific window. However, CQELS triggers the query execution for triples with same timestamp. If CQELS drops some data or fails to delete some expired data, then the engine's output mismatches the oracle output, and the oracle cannot determine which incoming triples contribute to the current engine output. What will happen next is that all the subsequent outputs will mismatch the oracle output. This problem cannot be efficiently solved without additional information from CQELS. In our benchmark, rather than checking the correctness for each query evaluation, we check the overall correctness. In other words, we let the engine run for a while, and compare the engine output with oracle output. If there is a mismatch, we manually check which window or which triple causes the mismatch.

When the data player sends the stream data to the target engine based on the timestamp τ, there is a delay between the timestamp τ and the timestamp τ'. τ is the time when a data is expected to be sent to the engine, this timestamp is generated by the data generator. τ' is the time when the data is actually sent to the engine. The delay between τ and τ' is caused by the I/O cost. It becomes very obvious when the stream rate is high. If the oracle uses the timestamp τ to calculate the oracle result, then for C-SPARQL, the mismatch between engine output and oracle output will become worse for the following windows, while for CQELS, the latency will continue increasing. YABench reports the error results without realizing this problem. To overcome this, we re-write the data player in the testbed.

The data player fetches the stream data and sends the streaming data to target engine, at the same time, the data player records the time τ' and write them to a file. Later, when the oracle start evaluating the engine's outputs, it reads this file, evaluate the query over the data annotated with τ' instead of τ. This guarantees both the oracle and the engine are evaluating the query over the same stream, which largely reduces the experimental errors caused by the testbed.

In the experiments, we use Stream WatDiv data generator to generate one static dataset and one streaming dataset. The static scale factor is set to 1 and the stream scale factor is set to 100. 100 random queries are generated to form our workload. The random seed used to generate the workloads and datasets is initialized to 1024, the maximum triple number of a query is restricted to 5, while all the other default parameters are used. The experiments are executed on the server node with 2x Intel E5-2620v2 @2.1GHz (12 physical cores total) with 32 GB RAM running Ubuntu 16.04.3. The whole experiment environment is implemented and run in JAVA. We set the JAVA max heap size to 6.98GB.

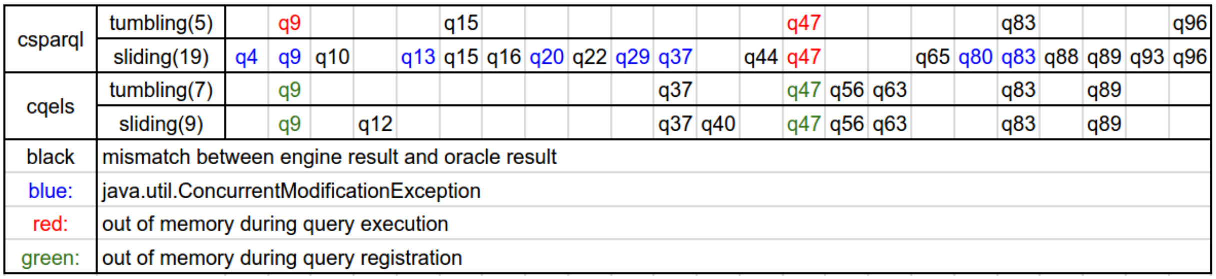
4.3 Correctness Test

The objective of this experiment is to see whether a SRP engine is able to process various queries under certain streaming rate. The workload includes both pure-streaming and hybrid queries. The hybrid queries involve a static dataset with $59,251$ triples. All of these queries cover different values of query features as defined in WatDiv. For C-SPARQL and CQELS, we test both tumbling window and sliding window queries. The window size is set to 5 seconds (for each query in the workload), and for sliding window queries the slide step is set to 1 second. Higher window size can be applied to test more robust SRP engines in the future.

We set the streaming rate to $10,000$ triples per second. We perform a "cold run" for each query. We first start the engine, register the query in the engine, start sending the stream data and let it run for a while, then shut down the engine. If the engine stops without crashing, the oracle computes the oracle output and measure the correctness.

This experiment result divides the queries into two sets, successful query set and failed query set. The successful queries are those that successfully run and get correct output on both C-SPARQL and CQELS. These queries will be used to measure the latency across C-SPARQL and CQELS, so they should at least run correctly on both engines. When running a failed query, the engine either crashes, or produces wrong result (produces extra results or misses results relative to the oracle results). The failed query set is shown in the

Figure 4.4 separated by their type. The queries marked in blue and red make C-SPARQL crash, and the queries marked in green make CQELS crash. All of these queries are not reported before. There exists queries that make either C-SPARQL or CQELS produce the wrong output. We mark these queries in black. This type of query is reported in previous benchmarks [19, 23, 25].



csparql	tumbling(5)		q9				q15								q47					q83				q96
	sliding(19)	q4	q9	q10		q13	q15	q16	q20	q22	q29	q37		q44	q47			q65	q80	q83	q88	q89	q93	q96
cqels	tumbling(7)		q9									q37			q47	q56	q63			q83		q89		
	sliding(9)		q9		q12							q37	q40		q47	q56	q63			q83		q89		
black	mismatch between engine result and oracle result																							
blue:	java.util.ConcurrentModificationException																							
red:	out of memory during query execution																							
green:	out of memory during query registration																							

Figure 4.4: Failed queries under 10,000 triples/second

4.4 Failed Queries Analysis

In this section, we explore the failure reason of the failed queries. For those queries that make the engine crash, none of the previous works have captured these query cases. Stream WatDiv helps us detect these failure queries and locate the failure reasons through analyzing the workloads.

Green queries: CQELS runs out of memory when registering Q9 and Q47 (see Listing 4.1 and 4.2). We observe that these two queries have a special pattern that causes this failure. The failure happens at step 3 in Figure 4.2. When a query is registered in CQELS, even before the arrival of stream data, CQELS will pre-calculate the intermediate result of the static part of the query. Sometimes triple patterns in the static part of the query do not have a shared subject or object, CQELS will still join them together, so the joining is actually a cross-product of two result sets. If both of the triple patterns have high result cardinality, then the cross product produces a new huge result set. Q9 and Q47 happen to be this kind of query.

```
SELECT ?v0 ?v1 ?v2 ?v3 ?v4 ?v5
FROM NAMED <http://dsg.uwaterloo.ca/watdiv/knowledge>
WHERE{
        STREAM <http://ex.org/streams/test> [RANGE ${WSIZE} SLIDE ${WSLIDE}] {
        ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/follows> ?v2 .
        ?v2 <http://db.uwaterloo.ca/~galuc/wsdbm/follows> ?v3 .
        ?v4 <http://db.uwaterloo.ca/~galuc/wsdbm/makesPurchase> ?v5 .
        }
        GRAPH <http://dsg.uwaterloo.ca/watdiv/knowledge>{
        ?v0 <http://schema.org/email> ?v1 .
        ?v3 <http://db.uwaterloo.ca/~galuc/wsdbm/friendOf> ?v4 .
        }
}
```

Listing 4.1: Q9 query string

```
SELECT ?v0 ?v1 ?v2 ?v3 ?v4 ?v5
FROM NAMED <http://dsg.uwaterloo.ca/watdiv/knowledge>
WHERE{
        STREAM <http://ex.org/streams/test> [RANGE ${WSIZE} SLIDE ${WSLIDE}] {
        ?v2 <http://purl.org/stuff/rev#rating> ?v3 .
        ?v2 <http://purl.org/stuff/rev#reviewer> ?v0 .
        ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/follows> ?v4 .
        }
        GRAPH <http://dsg.uwaterloo.ca/watdiv/knowledge>{
        ?v0 <http://xmlns.com/foaf/givenName> ?v1 .
        ?v4 <http://db.uwaterloo.ca/~galuc/wsdbm/friendOf> ?v5 .
        }
}
```

Listing 4.2: Q47 query string

As noted earlier, CQELS encodes each element of (subject, predicate, object) triple as an integer. This saves significant memory allowing intermediate result set to fit into RAM during query processing. Furthermore, computations on integers are faster than computations on variable-length strings. Both of these can help improve the system performance. CQELS utilizes Jena's own encoding function for this purpose. When a hybrid query is registered, CQELS loads all the data in Jena to build the encoding table that is stored on disk. During execution, whenever CQELS looks up a triple element, Jena will use the $B+$ tree index to fetch the block of the table, copy it to the heap, then lookup the corresponding element ID. This encoding process contributes to system crash when the intermediate results are too large as discussed above: CQELS encodes every element in intermediate result as integers; this encoding process will call the look-up function; each look-up operation in Jena will copy a file block to heap, and quickly, the heap memory is used up. More generally, if the triple patterns in the static part of the query are not connected, and have high result cardinality, there is a high chance to reproduce this issue in CQELS.

Blue queries: Lots of sliding window queries get *ConcurrentModificationException* when running in C-SPARQL. The common feature of these queries is that their query execution time is longer than the slide step. This failure happens in step 4 in Figure 4.1. As mentioned earlier, consecutive query executions will share the same Jena instance in C-SPARQL, if the query execution time is longer than the slide step, two consecutive query executions will concurrently modify the same Jena instance. More generally, if the query execution time is longer than the slide step in C-SPARQL, the engine will crash due to this exception.

Red queries: When running Q9 and Q47 in C-SPARQL, the queries exhaust the memory, which causes C-SPARQL to crash as well. The failure happens in Jena database shown in Figure 4.1. Consider Q47 as an example, Figure 4.5b shows the memory usage and CPU usage when it is run with sliding window in C-SPARQL. We observe that at the end of around the first minute, C-SPARQL has already used up 8GB memory, which exceeds the Java heap size (6.98GB). After that, the CPU is highly occupied to run the garbage collector to release the used memory.

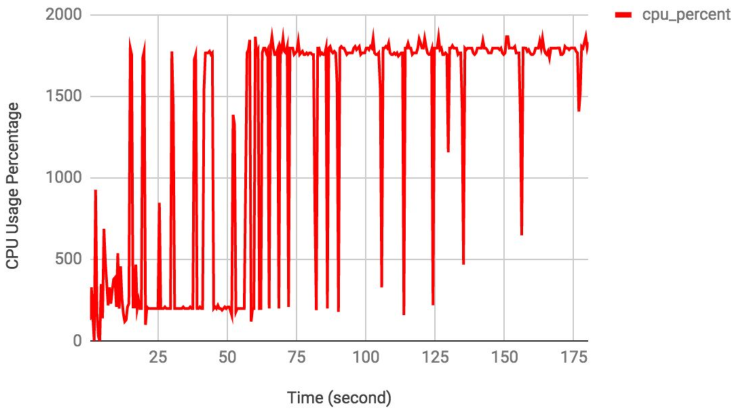
Black queries: Black queries cause the SRP engines produce wrong results.

- The general problem of C-SPARQL is that it misses results when the query execution is too long that consecutive query execution are running concurrently. This issue happens a lot when C-SPARQL runs the sliding window queries. In our experiment, the slide step of sliding window is 1 second, if the query execution time is longer than that, then new query execution instance tries to add new stream data into database, while the previous query execution instance is still clearing the previous window content from the same database. In some cases, this causes loss of data.

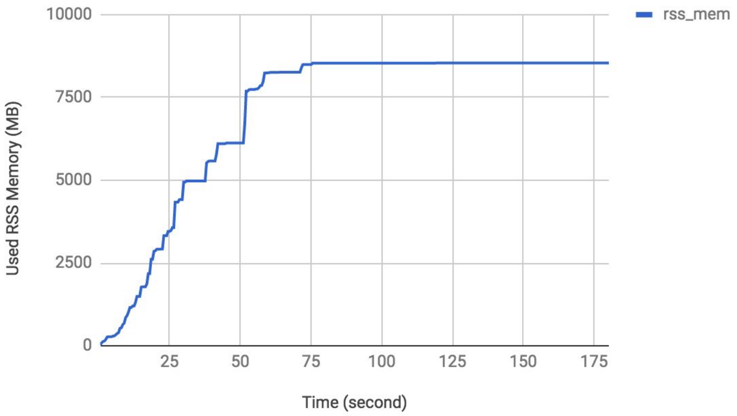
- When the stream rate is high, CQELS starts dropping some data. It places incoming triples on a queue to be processed. This happens in the window management component in Figure 4.2. If the stream rate is high, the queue fills up and the system cannot catch-up with the arrival rate, resulting in the drop of new incoming triples. That means that the results generated will not cover the dropped triples.

4.5 Latency Measurement

Latency measures the time consumed by a SRP engine to generate the output for each window. For the two systems under study, latency is computed differently. In C-SPARQL,



(a) CPU percentage



(b) Memory usage

Figure 4.5: Memory and CPU Consumption During Running Q47 on C-SPARQL

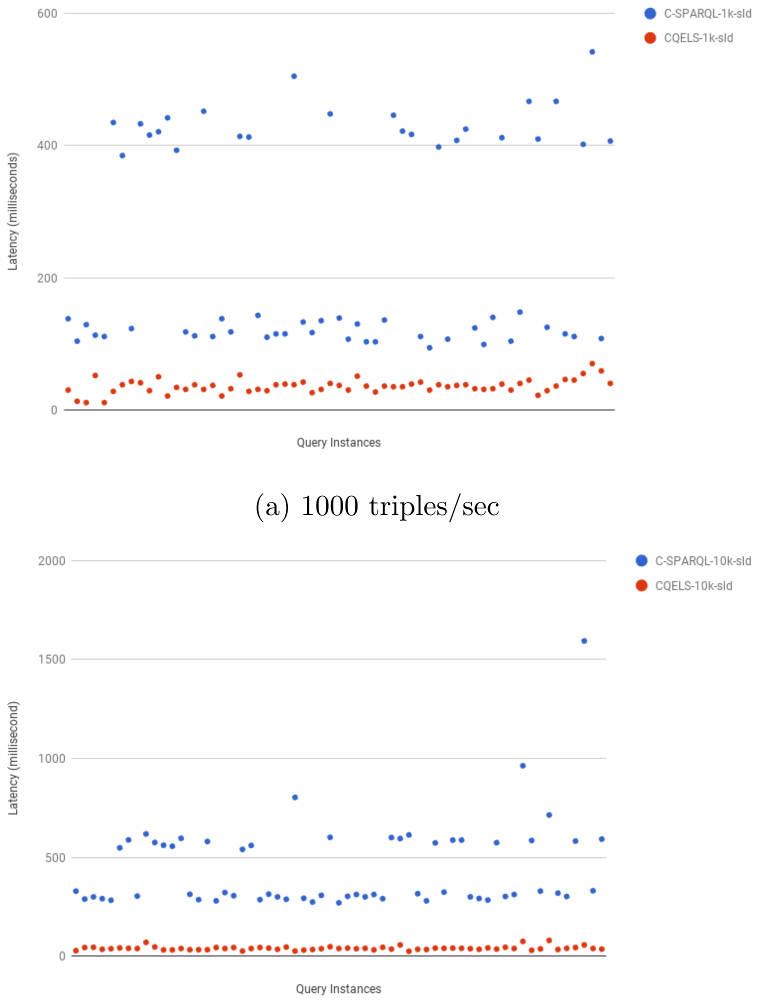
latency is the time interval between the window close time and the engine's output time, because the results are generated every time the window closes. CQELS, however, eagerly executes the query for each incoming triple and generates the output only if the output is not empty. Therefore, latency is the time interval between the timestamp of the last triple in the window that triggers the engine to compute a result and the time when the engine output arrives. We assume that if the streaming rate is high, the last triple that makes CQELS generate a result is near the window border, so we can compare the latency of C-SPARQL and CQELS. We evaluate the latency of C-SPARQL and CQELS with increasing stream rate for both sliding window queries and tumbling window queries.

The workload we choose here is from the successful query set. Among 100 random queries, previous correctness test has shown that 23 of them will either cause engines crash or generate wrong answers. In addition, we also eliminate the queries that do not produce any result during the experiment time. After eliminating these queries, the workload consists of 61 queries. Each query will be initialized to a sliding window query and a tumbling window query. Figures 4.6–4.9 illustrate the results. Each plot in the figure represents the average latency aggregated from multiple consecutive windows.

First, we make a direct comparison between C-SPARQL and CQELS. We evaluate the latency of these two engines for both sliding window and tumbling window queries under low ($1,000$ triples/sec) and high ($10,000$ triples/sec) stream rate. From Figures 4.6 and 4.7, we observe the following:

- Overall, CQELS has lower latency than C-SPARQL. CQELS implements several optimization like adaptively generating query execution plan, caching and encoding the intermediate result, which makes its query execution more than 10 times faster than that of C-SPARQL. This finding is consistent with the result in [24].
- The latency of C-SPARQL queries clusters around two values. The group of queries with higher latency are hybrid queries, the ones with lower latency are pure stream queries. Due to C-SPARQL's query mechanism, it stores both streaming data and static data in an RDF triple store, and runs the SPARQL query on it. The bigger the static data size, the longer the query execution time will be. However, CQELS shows consistent performance of processing hybrid queries and pure stream queries. This finding is consistent with the result in [33].
- As expected, CQELS is not influenced by the slide step because of its eager-execution strategy. However, for C-SPARQL, the average latency of sliding window query is lower than that of tumbling window. Since C-SPARQL reuses the same Jena instance to process the query, smaller slide step results in more common data between consecutive executions, so there is more chance to cache the intermediate result.

Next, we analyze how the two engines handle the increasing stream rate. Figures 4.8 and 4.9 show the latency of C-SPARQL and CQELS with different stream rates (1,000 triples/sec, 10,000 triples/sec). To better illustrate the result, we sort the query instances based on the latency at high (10,000 triples/sec) stream rate.

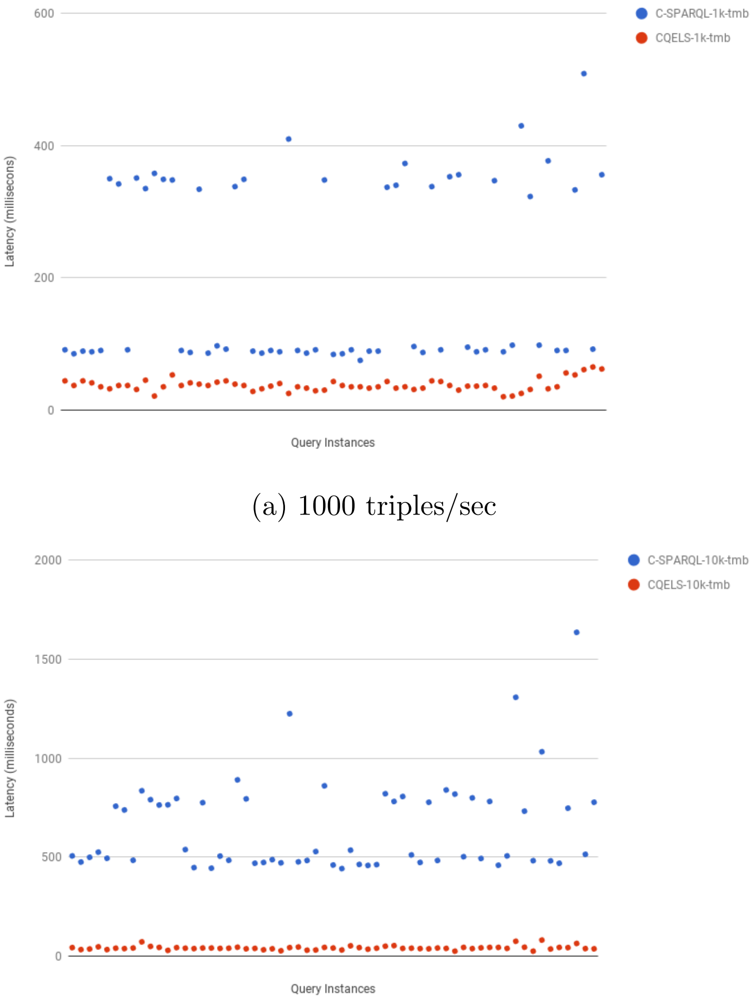


(a) 1000 triples/sec

(b) 10,000 triples/sec

Figure 4.6: Latency of sliding window queries

Figure 4.8 compares C-SPARQL's query latency under 1,000 triples/second and 10,000 triples/second. The gaps in the figure comes from the latency difference between hybrid queries and pure stream queries. We see that C-SPARQL scales well for most queries with increasing stream rate. Only the queries at the rightmost area of the figure have much longer latency at high stream rate. These queries have either high result cardinality or complex query graph structure.

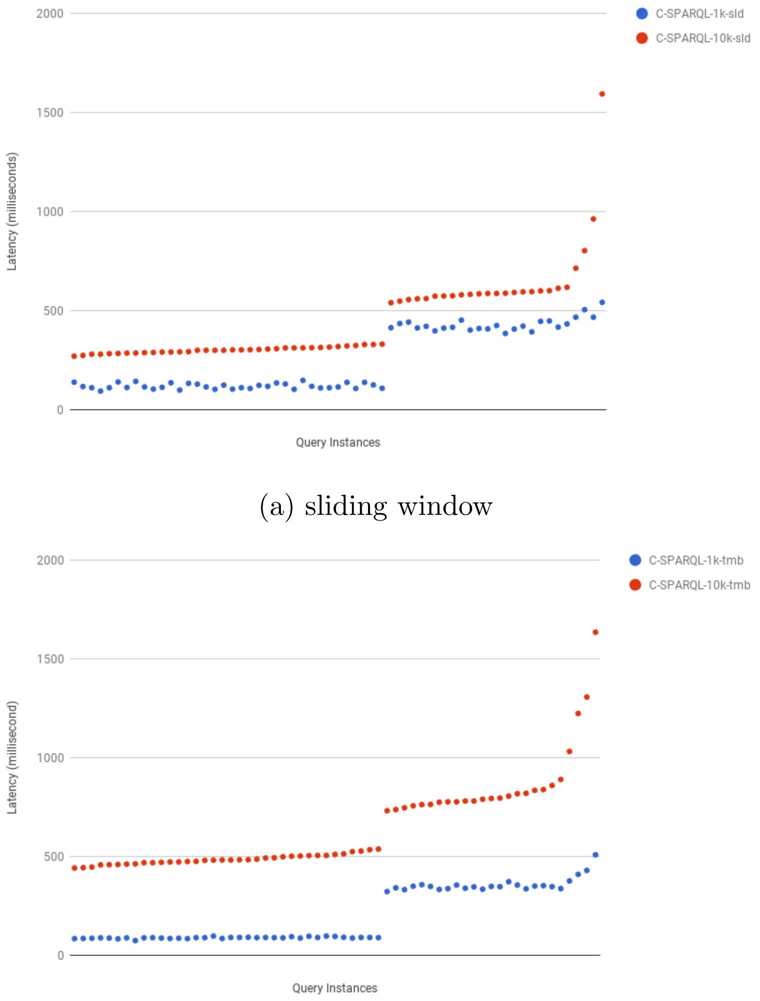


(a) 1000 triples/sec

(b) 10,000 triples/sec

Figure 4.7: Latency of tumbling window queries

From the results shown in Figure 4.9, we can observe that stream rate affects CQELS's latency marginally. Actually, some of the queries have lower latency when the stream rate is higher. A possible reason is that when the stream rate changes, the triple pattern matching results in the streaming part of the query changes, causing CQELS to choose a different query execution plan, which results in the latency difference. The average latency for all the queries under both stream rates is around 40 milliseconds.



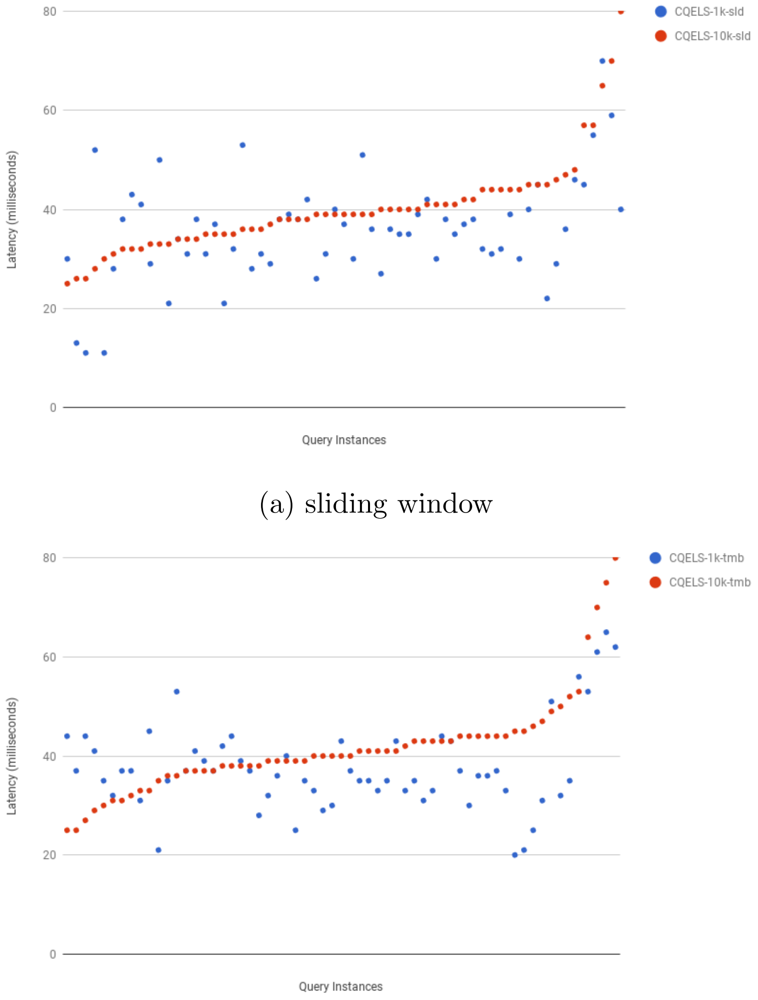
(a) sliding window

(b) tumbling window

Figure 4.8: Increasing stream rate for C-SPARQL

4.6 Detailed Analysis

In this section, we further use our workloads to analyze the performance. First, memory consumption of C-SPARQL and CQELS are conducted. The result is aligned with previous works. Then, we group the queries based on the defined query features and test how two engines handle certain types of the query. No other benchmarks conduct this analysis



(a) sliding window

(b) tumbling window

Figure 4.9: Increasing stream rate for CQELS

before, and the findings are new.

In correctness test, C-SPARQL and CQELS show a lot of *running out of memory* failure, this arouses our interest in exploring memory consumption of these two engines. As most successful queries show same memory consumption patterns, we take Q14 with the tumbling window under 10,000 triples/second as an example. As demonstrated in Figure 4.10, CQELS occupies almost twice memory as C-SPARQL does. The occupied memory in CQELS is used to achieve optimizations such as encoding the RDF nodes and caching intermediate result. The result is aligned with the observation in previous work [23].

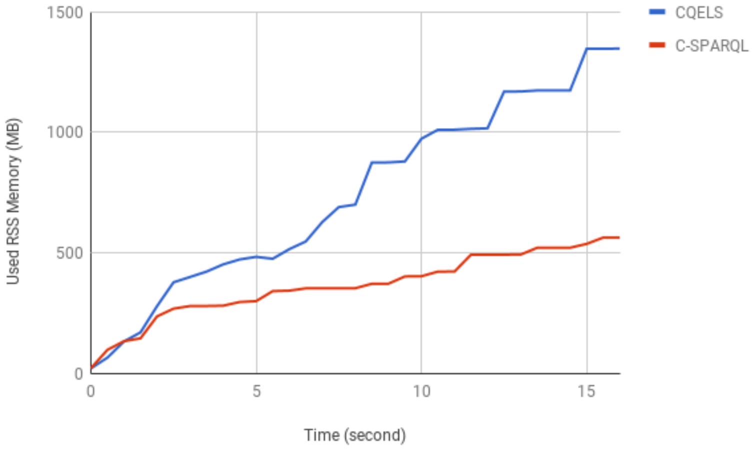


Figure 4.10: Q14 memory consumption

The 61 queries generated by Stream WatDiv have a diversified coverage of the values of query features, which can be further used to evaluate engines' features. In particular, we measure the engines' performance of integration with static data, handling queries with different graph structure and queries with different result cardinality and triple pattern selectivity. None of the previous benchmarks have diversified workloads to support these analysis. In all of the following analysis, we focus on high stream rate (10,000 triples/second). Since sliding window queries show similar result pattern with tumbling window queries, in this section, we only consider the tumbling window queries.

In the previous experiments, we see the latency difference between pure stream queries and hybrid queries. Among 61 successful queries, 36 of the queries are pure stream queries, and 25 of them are hybrid queries. We aggregate the average query execution time for pure stream queries and hybrid queries. The result is presented in Table 4.1. Overall, pure stream queries result in lower latency on both C-SPARQL and CQELS. C-SPARQL takes more than twice longer to process hybrid queries compared to pure stream queries. This is because C-SPARQL stores the window snapshot and static data in the same triple store and run the query on it. With a high stream rate and a five-second time window, each window content contains around 50,000,000 triples, and the static data set contains also around 59,000,000 triples. Introducing the static data when processing the query only poses little overhead on CQELS. CQELS will pre-calculate the intermediate result for the static part of the query, cache it and build the index. In this way, CQELS does not need to re-run the query among the static data for every query execution.

	tumbling window		sliding window	
	hybrid query	pure query	hybrid query	pure query
C-SPARQL	873.52	493.89	651.4	301.25
CQELS	46.28	38.44	42.88	38.44

Table 4.1: Average latency for pure stream queries and hybrid queries (millisecond)

Triple pattern count of a query is a straightforward feature to measure the query complexity. In our workloads, we restrict the maximum triple pattern count of a query to 5. While generating queries with higher triple pattern count is certainly feasible, existing engines already show various problems when handling queries with few triple patterns. We group the workload based on the triple pattern count. Queries with 4 and 5 triple patterns are grouped together, so each group contain around 15 queries. The result presented in Figure 4.11 shows the average latency for different groups of queries. C-SPARQL takes longer latency to process queries with larger amount of triple patterns. CQELS shows constant average latency to process different groups of queries.

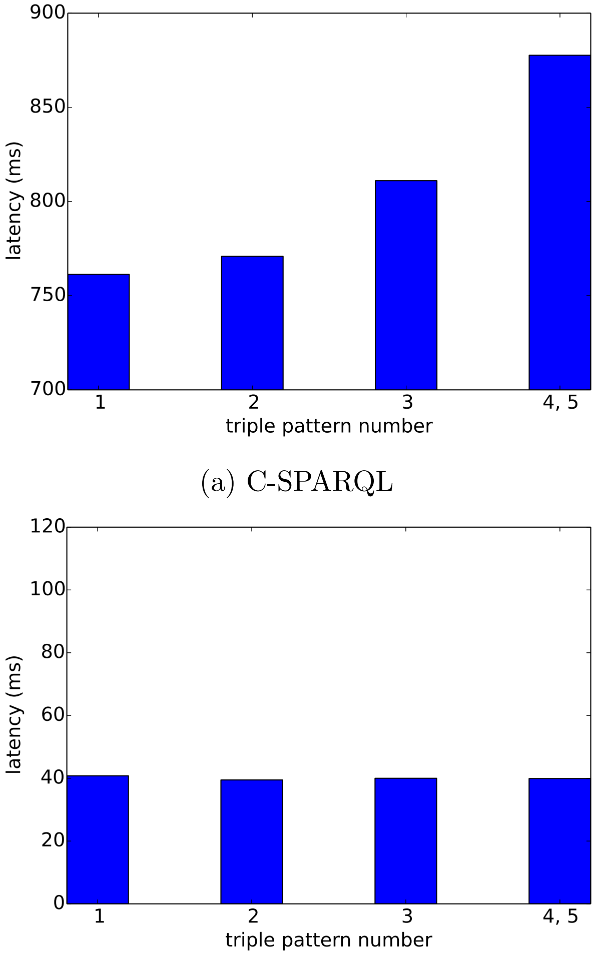
Join vertex number and join vertex degree can together differentiate linear-structure queries (join vertex number≥0 and join vertex degree≤2) and star/snowflake-structure (join vertex number≤2 and join vertex degree≥3) queries. According to the result in table 4.2, both C-SPARQL and CQELS are not biased toward processing queries with a particular graph structure.

	linear	star/snowflake
C-SPARQL	795.03	814.96
CQELS	40.65	38.88

Table 4.2: The latency of queries with different graph structure (millisecond)

Next, we want to see the affect of result cardinality on engine's latency. We calculate the average per-window result cardinality for each query. The values of result cardinality are divided into three intervals (low (0, 500], medium (500, 900] and high (900, $+\infty$)) so that each interval contains almost equivalent number of queries. Both C-SPARQL and CQELS do not show much difference of latency between queries with low result cardinality and medium result cardinality, but queries with high result cardinality have higher latency.

Last, we demonstrate how engines handle two different types of workloads, the one capturing queries with equally selective triple patterns and the one capturing queries with



(a) C-SPARQL

(b) CQELS

Figure 4.11: Triple pattern count

	low	medium	high
C-SPARQL	795	791.39	851.5
CQELS	39.13	38.83	44.2

Table 4.3: The latency of queries with low (0, 500], medium (500, 900] and high (900, +∞) result cardinality

extremely variously selective triple patterns. We pick 6 queries for each type of workloads from 61 successful queries and compute the average latency. As we shown before, the result cardinality of a query influences the latency, therefore, only queries whose per window result cardinality is below 900 are selected.

	equally selective	variously selective
C-SPARQL	801	818
CQELS	38.2	33.4

Table 4.4: The latency of queries with equally selective triple patterns and with extremely variously selective triple patterns

As expected, CQELS performs well for queries with variously selective triple patterns. The greedy strategy adopted when generating the query execution plan always picks a triple pattern with smallest result cardinality. However, for queries with equally selective triple patterns, this strategy may not find the optimal query execution plan that tunes the intermediate result largely.

Chapter 5

Conclusions and Future Work

This thesis proposes a new streaming RDF benchmark, Stream WatDiv. The main goal of this work is to design a benchmark whose workloads diversified to evaluate various SRP engines. Existing SRP engines are still quite primitive and only support limited number of query operations, while previous benchmarks either contain workloads that most engines do not support or contain a small workload set that is not sufficiently diverse. Instead, our benchmark only involves a basic fragment of the query language, namely basic graph pattern (BGP) matching that all systems support, while covering various query features like query graph structure and triple selectivity which affect the engine's performance. We modify the data generator and query generator of WatDiv to construct Stream WatDiv. More specifically, stream WatDiv can generate both RDF streams and static background data. The query generator can conduct random walk across both streaming data and static data to generate pure streaming query and hybrid query.

Two representative engines, C-SPARQL and CQELS are picked as the target test engines to demonstrate the use of Stream WatDiv. The experiments contain three parts: correctness test, failed query analysis and latency measurement. Our evaluation achieves two goals:

First, the evaluation results validate the results related to these two engines in previous works. (1) CQELS is more robust and efficient than C-SPARQL at processing streaming RDF queries in most cases. (2) increasing streaming rate and integrating static data will significantly influence C-SPARQL's performance, while CQELS is marginally affected. (3) C-SPARQL is more memory-efficient than CQELS when processing the query.

Second, the diversity of Stream WatDiv workloads makes Stream WatDiv capable of finding new observations. Failed queries capture engine issues that are not detected before. These queries either make engine crash or generate wrong output. Analyzing the failed queries helps to locate failure reason. Successful queries can be grouped into different types based on the query features. These types of queries can be used to evaluate a specific engine features. (1) Triple pattern count of a query influences C-SPARQL's performance. (2) Both C-SPARQL and CQELS shows a significant latency increase when the query has larger result cardinality. (3) Neither of these two engines are biased toward processing linear, star or snowflake queries. (4) CQELS is more efficient at handling queries with variously selective triple patterns, while C-SPARQL performs better for queries with equally selective triple patterns than queries with variously selective triple patterns.

We note that extending WatDiv to stream WatDiv is only a first step to enable performance evaluation across SRP engines. Improvements can be made to both the benchmark design and the experiment setup.

In our benchmark, we only cover the basic fragment of the query language, namely BGP matching. Even though existing engines are not mature enough to handle more advance query operations; they will be in the future. So it is necessary to explore the possibility to extend the workloads with more advance query operations like aggregations, reasoning and other new features introduced in SPARQL 1.1.

Distributed SRP engines are still in their early development stage. Our benchmark is capable to generate large datasets and complex queries to meet the needs of testing distributed SRP engines. As we noted, existing engines like Strider can only execute pure stream queries, and CQELS-Cloud is not open source. As more mature distributed SRP engines are developed, it would be necessary to re-visit Stream WatDiv to enhance its features..

References

[1] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proc. 33rd International Conference on Very large data bases*, pages 411–422, 2007.

[2] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. SW-Store: a vertically partitioned DBMS for semantic web data management. *The VLDB Journal*, 18(2):385–406, 2009.

[3] Karl Aberer, Manfred Hauswirth, and Ali Salehi. Global sensor networks. Technical report, LSIR-REPORT-2006-001, 2006.

[4] Muhammad Intizar Ali, Feng Gao, and Alessandra Mileo. CityBench: a configurable benchmark to evaluate RSP engines using smart city datasets. In *Proc. 14th International Semantic Web Conference*, pages 374–389, 2015.

[5] Güneş Aluç, Olaf Hartig, M Tamer Özsu, and Khuzaima Daudjee. Diversified stress testing of RDF data management systems. In *Proc. 13th International Semantic Web Conference*, pages 197–212, 2014.

[6] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *Proc. 20th International Conference on World Wide Web*, pages 635–644, 2011.

[7] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: a stream data management benchmark. In *Proc. 13th International Conference on Very Large Data Bases*, pages 480–491, 2004.

[8] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-SPARQL: SPARQL for continuous querying. In *Proc. 18th International Conference on World Wide Web*, pages 1061–1062, 2009.

[9] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Continuous queries and real-time analysis of social semantic data with C-SPARQL. In *Proc. 2nd Workshop on Social Data on the Web*, volume 10, 2009.

[10] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Incremental reasoning on streams and rich background knowledge. In *Proc. 7th Extended Semantic Web Conference*, pages 1–15, 2010.

[11] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, and Michael Grossniklaus. An execution environment for C-SPARQL queries. In *Proc. 13th International Conference on Extending Database Technology*, pages 441–452, 2010.

[12] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying RDF streams with C-SPARQL. *ACM SIGMOD Record*, 39(1):20–26, 2010.

[13] BBC: dynamic semantic publishing FIFA world cup web site. https://ontotext.com/company/customers/bbc-dynamic-semantic-publishing/. Accessed: 2017-12-15.

[14] Christian Bizer and Andreas Schultz. The Berlin SPARQL benchmark. *International Journal on Semantic Web and Information Systems*, 5(2):1–24, 2009.

[15] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: a generic architecture for storing and querying RDF and RDF schema. In *Proc. 1st International Semantic Web Conference*, pages 54–68, 2002.

[16] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair JG Gray. Enabling ontology-based access to streaming data sources. In *Proc. 9th International Semantic Web Conference*, pages 96–111, 2010.

[17] Jean-Paul Calbimonte, Ho Young Jeung, Oscar Corcho, and Karl Aberer. Enabling query technologies for the semantic sensor web. *International Journal on Semantic Web and Information Systems*, 8:43–63, 2012.

[18] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. Benchmarking streaming computation engines: Storm, Flink and Spark Streaming. In *Proc. 30th IEEE International Parallel and Distributed Processing Symposium*, pages 1789–1792, 2016.

[19] Daniele Dell'Aglio, Jean-Paul Calbimonte, Marco Balduini, Oscar Corcho, and Emanuele Della Valle. On correctness in RDF stream processor benchmarking. In *Proc. 12th International Semantic Web Conference*, pages 326–342, 2013.

[20] Aluç G. *Workload Matters: a robust approach to physical RDF database design.* PhD thesis, University of Waterloo, 2015.

[21] Ixent Galpin, Christian YA Brenninkmeijer, Alasdair JG Gray, Farhana Jabeen, Alvaro AA Fernandes, and Norman W Paton. SNEE: a query processor for wireless sensor networks. *Distributed and Parallel Databases*, 29(1):31–85, 2011.

[22] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: a benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.

[23] Maxim Kolchin, Peter Wetz, Elmar Kiesling, and A Min Tjoa. YABench: a comprehensive framework for RDF stream processor correctness and performance assessment. In *Proc. 16th International Conference on Web Engineering*, pages 280–298, 2016.

[24] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *Proc. 10th International Semantic Web Conference*, pages 370–388, 2011.

[25] Danh Le-Phuoc, Minh Dao-Tran, Minh-Duc Pham, Peter Boncz, Thomas Eiter, and Michael Fink. Linked stream data processing engines: facts and figures. *Proc. 11th International Semantic Web Conference*, pages 300–312, 2012.

[26] Danh Le-Phuoc, Hoan Nguyen Mau Quoc, Chan Le Van, and Manfred Hauswirth. Elastic and scalable processing of linked stream data in the cloud. In *Proc. 12th International Semantic Web Conference*, pages 280–297, 2013.

[27] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. DBpedia SPARQL benchmark–performance assessment with real queries on real data. In *Proc. 10th International Semantic Web Conference*, pages 454–469, 2011.

[28] Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.

[29] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010.

[30] M Tamer Özsu. A survey of RDF data management systems. *Frontiers of Computer Science*, 10(3):418–432, 2016.

[31] Peng Peng, Lei Zou, M Tamer Özsu, Lei Chen, and Dongyan Zhao. Processing SPARQL queries over distributed RDF graphs. *The VLDB Journal*, 25(2):243–268, 2016.

[32] Xiangnan Ren and Olivier Curé. Strider: a hybrid adaptive distributed RDF stream processing engine. *arXiv preprint arXiv:1705.05688*, 2017.

[33] Xiangnan Ren, Houda Khrouf, Zakia Kazi-Aoul, Yousra Chabchoub, and Olivier Curé. On measuring performances of C-SPARQL and CQELS. *arXiv preprint arXiv:1611.08269*, 2016.

[34] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SPˆ 2Bench: a SPARQL performance benchmark. In *Proc. 25th IEEE International Conference on Data Engineering*, pages 222–233, 2009.

[35] Nan Tang. Big RDF data cleaning. In *Proc. 31st IEEE International Conference on Data Engineering Workshops*, pages 77–79, 2015.

[36] Riccardo Tommasini, Emanuele Della Valle, Marco Balduini, and Daniele Dell'Aglio. Heaven: a framework for systematic comparative research approach for RSP engines. In *Proc. 13th Extended Semantic Web Conference*, pages 250–265, 2016.

[37] Riccardo Tommasini, Emanuele Della Valle, Andrea Mauri, and Marco Brambilla. RSPLab: RDF stream processing benchmarking made easy. In *Proc. 16th International Semantic Web Conference*, pages 202–209, 2017.

[38] Amrapali Zaveri, Anisa Rula, Andrea Maurino, Ricardo Pietrobon, Jens Lehmann, and Sören Auer. Quality assessment for linked data: a survey. *Semantic Web*, 7(1):63–93, 2016.

[39] Ying Zhang, Pham Minh Duc, Oscar Corcho, and Jean-Paul Calbimonte. SRBench: a streaming RDF/SPARQL benchmark. In *Proc. 11th International Semantic Web Conference*, pages 641–657, 2012.

[40] Lei Zou, Jinghui Mo, Lei Chen, M Tamer Özsu, and Dongyan Zhao. gStore: answering SPARQL queries via subgraph matching. *Proceedings of the VLDB Endowment*, 4(8):482–493, 2011.

[41] Lei Zou and M Tamer Özsu. Graph-based RDF data management. *Data Science and Engineering*, 2(1):56–70, 2017.

[42] Lei Zou, M Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang, and Dongyan Zhao. gStore: a graph-based SPARQL query engine. *The VLDB journal*, 23(4):565–590, 2014.