Generating a benchmark for RDF stream processing

by

KASSARAS EMMANOUIL
(MTH285)

MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING SCHOOL OF ENGINEERING

HELLENIC MEDITERRANEAN UNIVERSITY

2024

Approved by:

Major Professor

Dr. Nikolaos Papadakis

Δημιουργία benchmark για την επεξεργασία ροής RDF

ΚΑΣΣΑΡΑΣ ΕΜΜΑΝΟΥΗΛ
(MTH285)

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΜΕΤΑΠΤΥΧΙΑΚΟ ΕΠΙΣΤΗΜΩΝ

ΤΜΉΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ

ΕΛΛΗΝΙΚΟ ΜΕΣΟΓΕΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ

2024

Acknowledgments

As this master thesis comes to the end, I have to express my deepest gratitude to my beloved people. First of all to my parents for their unconditional support and encouragement during the whole journey. Secondly, to my brother and sister for reminding me all the time to never give up even in the most difficult times. Last but not least, I want to thank my supervisor professor Dr. Nikolaos Papadakis and his assistant Dr, Charidimos Kondylakis for their scientific and technical advice which was very helpful.

Abstract

RDF Stream Processing (RSP) has become a vital area of research, driven by the growing need to process and analyze continuous streams of RDF data in real-time. However, the domain still faces several challenges, particularly in the areas of standardization, system performance, and benchmarking. This paper explores the current state of RSP systems, with a specific focus on the evaluation of existing benchmarks. We assess these benchmarks based on key performance choke points, such as parallel join execution, full-text search capabilities, and duplicate elimination, identifying significant gaps in their ability to comprehensively evaluate system performance.

To address these shortcomings, a benchmark was implemented in Java, built on top of RSP4J and uses the RSP-QL language to create queries. This benchmark tries to introduce additional evaluation criteria aimed at overcoming existing limitations and provides a more robust measure of system performance across various scenarios. Through this work, the goal is the advance of the field in RDF stream processing by contributing both to the ongoing standardization efforts and to the development of more effective and comprehensive benchmarking tools.

Σύνοψη

Η Επεξεργασία Ροών RDF (RDF Stream Processing - RSP) έχει γίνει ένας σημαντικός τομέας έρευνας, ο οποίος οδηγείται από την αυξανόμενη ανάγκη για επεξεργασία και ανάλυση συνεχών ροών δεδομένων RDF σε πραγματικό χρόνο. Ωστόσο, ο τομέας εξακολουθεί να αντιμετωπίζει αρκετές προκλήσεις, ιδιαίτερα στους τομείς της τυποποίησης, της απόδοσης των συστημάτων και της αξιολόγησης. Αυτή η εργασία εξετάζει την τρέχουσα κατάσταση των συστημάτων RSP, με συγκεκριμένη εστίαση στην αξιολόγηση των υπαρχόντων benchmarks. Αξιολογούμε αυτά τα benchmarks με βάση βασικά σημεία απόδοσης, όπως η παράλληλη εκτέλεση join, οι δυνατότητες αναζήτησης πλήρους κειμένου και η εξάλειψη των διπλότυπων, εντοπίζοντας σημαντικά κενά στην ικανότητά τους να αξιολογήσουν συνολικά την απόδοση των συστημάτων.

Για την αντιμετώπιση αυτών των αδυναμιών, υλοποιήθηκε ένα benchmark σε Java, το οποίο βασίζεται στο RSP4J και χρησιμοποιεί την γλώσσα RSP-QL για την δημιουργία επερωτήσεων. Αυτό το benchmark εισάγει πρόσθετα κριτήρια αξιολόγησης με στόχο την υπέρβαση των υπαρχόντων περιορισμών και προσπαθεί να παρέχει ένα πιο αξιόπιστο μέτρο απόδοσης των συστημάτων σε διάφορα σενάρια. Μέσω αυτής της εργασίας, ο στόχος είναι η προώθηση του τομέα της επεξεργασίας ροών RDF, συνεισφέροντας τόσο στις συνεχιζόμενες προσπάθειες τυποποίησης όσο και στην ανάπτυξη πιο αποτελεσματικών και ολοκληρωμένων εργαλείων αξιολόγησης.

Table of Content

Acknowledgments 3
Abstract 4
1. Introduction 7
    1.1 Evolution of the Web 8
    1.2 Semantic Web and its key concepts 9
2. RDF 11
    2.1 RDF streaming 12
    2.2 RDF Stream Processing Systems 13
3. State of the art 17
    3.1 RDF Stream Processing Benchmarks 19
    3.2 Choke points and their meaning 21
    3.3 Comparison of existing benchmark based on choke points 24
4. Development of an RDF Streaming benchmark 26
    4.1 Hardware and Software 26
    4.2 Programming languages and libraries 26
        4.2.1 Java 27
        4.2.2 RSP4J 28
        4.2.3 RSP-QL 29
    4.3 Approach & Implementation 30
5. Conclusion & Future Work 46
    5.1 Conclusion 46
    5.2 Future Work 47
References 49

1. Introduction

In recent years, Computer Science (CS) has affected in a positive way other sciences as well as various aspects of our daily lives. Many breakthroughs changed the world and the perspective that we understand it drastically. The World Wide Web may be the most significant result of the CS. Thanks to this, people and organizations are able to interact, communicate, and share information not only for entertainment reasons but also for education and business. The Web has evolved during the years with significant changes among the various phases which will be analyzed in the next chapter.

The way that data in the Web is manipulated and used by the users and the Computer Scientists also has been through various stages along with the evolution of the Web. Static and streaming data are the two main categories that have been concerning the scientific community for over thirty years. Static data are more common in simple web applications and they remain relatively stable and unchangeable over time **[1][2]**. They are often stored in databases and files that have been updated at a specific time. For example, static data includes customer information in a relational database, product catalogs, and historical sales records **[1]**. Static data is ideal for simple websites or e-shops but due to the limitations in memory that traditional databases suffer, rapid and continuous loading of individual data items is not supported as well the approach of the continuous queries. On the other hand, stream data or real-time data are capable when there is a continuous flow from data generated by sensors, social media, or other applications. This kind of data is often dynamic, unbounded, and often updated. The pace of is being generated is rapid and is exploited by businesses and organizations **[1][2]**.

In this work, streaming data will be examined in terms of RDF (Resource Description Framework) stream processing which is a relatively new field in Computer Science focusing on the way that this type of data behaves when they are expressed in RDF graphs. Moreover, a custom benchmark has been implemented to handle and perform queries on data streams and to deal with some of the vulnerabilities (also they will be referred to as **choke points**) that other benchmarks fail to do. In the end, possible improvements and future work will be mentioned not only as regards the benchmark tool but also the RDF streams in general,

1.1 Evolution of the Web

The World Wide Web (WWW or simply the Web) is an information system designed to make content sharing over the Internet accessible to a broad audience, beyond just IT professionals and enthusiasts. It enables users to access documents and other resources online by following rules established by the Hypertext Transfer Protocol (HTTP).

Created by British computer scientist Tim Berners-Lee at CERN in 1989, the Web was made publicly available in 1991 and was envisioned as a "universal linked information system." Content such as documents and media is hosted on web servers and can be accessed via web browsers. Servers and resources are identified by uniform resource locators (URLs).

The main technology that enables web pages to be displayed in browsers is HTML (Hypertext Markup Language), which supports text, images, videos, audio, and scripts for interactive functions. Other key technologies include CSS (Cascading Style Sheets), which provides advanced styling for HTML elements, and JavaScript, which enhances functionality, interaction, and dynamic content on web pages.

Web 1.0, often referred to as the informational web, represents the first generation of the web. In this era, users could only read and share information on web pages, without the ability to interact with the content, such as by commenting or contributing. Tim Berners-Lee, the creator of the Web, envisioned this as a read-only platform, where a small number of creators produced interlinked web pages, and a large audience accessed them via browsers on the Internet. The key technologies powering Web 1.0 included HTML, HTTP, and URI, with protocols like XML, XHTML, and CSS introduced later. Both server-side and client-side scripting were used, such as ASP, PHP, JSP, CGI, and PERL for server-side, and JavaScript, VBScript, and Flash for client-side scripting.**[3]**

However, Web 1.0 faced several issues, including its slow and cumbersome nature, requiring page refreshes every time new information was added. It didn't support two-way communication, relying entirely on a client-pull model (HTTP) initiated by the client. The search technologies of Web 1.0 were considered inadequate, focusing on the size of the index while ignoring relevance, making effective searching difficult.

A major flaw in Web 1.0 was its disregard for the power of network effects. With only a few content creators and a large number of readers, the network became slow, leaving users wanting more resources. The concept that a network becomes more valuable as more people use it was overlooked, as Web 1.0 was strictly read-only. Web 1.0 viewed the web as a

publishing platform rather than a participatory one, allowing users to only read information without interacting with web pages. This approach misunderstood the web's potential, treating software as an application rather than a service and relying on outdated business models.

Web 2.0, also known as the read-write web, represents a shift in how existing internet technologies are used. Unlike Web 1.0, where users could only read content, Web 2.0 allows users to write, modify, and update content online, promoting collaboration and collective intelligence. This new generation of networking services transformed the web into a platform, enabling content exchange and sharing through applications like wikis, blogs, widgets, and mashups.

The technology infrastructure of Web 2.0 includes:

1. **Content Syndication**: Technologies like RSS, Atom, and RDF are used to create Web 2.0 services for sharing content.
2. **Ajax-based Internet Technology**: Ajax (Asynchronous JavaScript and XML) makes web pages more interactive and responsive by reducing the data that needs to be reloaded for client-server interactions, improving speed and usability compared to traditional HTML sites.
3. **DOM**: The Document Object Model (DOM) represents HTML or XML documents as a tree structure, allowing better manipulation of content.
4. **REST**: Representational State Transfer (REST) is an approach for retrieving information from web pages in a more efficient way.
5. **XML and CSS**: XML is used for managing information and creating custom markup languages for various data types, while CSS (Cascading Style Sheets) provides styling to web pages. **[3]**

1.2 Semantic Web and its key concepts

The Semantic Web is an evolving extension of the World Wide Web where the meanings of information and services are defined, enabling the Web to understand and fulfill requests from both people and machines. This concept stems from W3C director Tim Berners-Lee's vision of the Web as a universal medium for the exchange of data, information, and knowledge. The current web functions as a hypermedia system, a digital library, and a collection of documents

known as web pages linked through hypermedia, a database, an application platform, and a multimedia platform. It also serves as a common portal for applications accessible via web pages and provides a unique naming scheme for documents.

The Semantic Web involves publishing data in the Resource Description Framework (RDF) language, specifically designed for data manipulation and integration similar to how data files are managed on a local computer. While HTML describes documents and their interconnections, RDF describes a wide range of entities such as people, meetings, and airplane parts. The building blocks of the Semantic Web include various formats and languages, such as Uniform Resource Identifiers (URI), Extensible Markup Language (XML) for documents, and Resource Description Framework (RDF) for statements. It also encompasses different data interchange formats (e.g., RDF/XML, N3) and notations like RDF Schemas (RDFS) and the Web Ontology Language (OWL), all aimed at providing formal descriptions of concepts, terms, and relationships.

The most essential component is the Resource Description Framework (RDF), which defines information on the web. RDF is a markup language for describing web information and resources. By embedding information in RDF files, computer programs, or "web spiders," can search, discover, collect, analyze, and process information from the web. The Semantic Web employs RDF to describe web resources, offering a data model and syntax that allow independent parties to exchange and utilize data. RDF is designed to be machine-readable rather than human-readable **[4][5]**.

2. RDF

The Resource Description Framework (RDF) is a W3C-recommended model for data interchange on the Web. It is designed to facilitate data integration from different schemas and supports schema evolution without requiring modifications from all data consumers. RDF leverages URIs to name the relationships between entities and the entities themselves, forming a "triple" structure. This model allows for the integration and sharing of structured and semi-structured data across various applications. RDF's linking structure forms a directed, labeled graph where edges represent named links between nodes (resources). This graph-based representation simplifies the conceptual understanding of RDF.

Although RDF was initially created to describe web resources like web pages, it is not limited to web-accessible resources. RDF can describe any entity, physical or abstract, by using named properties and their values. RDF Schema (RDFS) provides a vocabulary to describe these properties, classes of resources, and their interrelations.

RDF serves as the foundation for the Semantic Web, which, like the traditional Web, is a global infrastructure. However, the Semantic Web represents information in a machine-processable format, and its decentralized nature imposes constraints on how information is represented.

Unlike traditional resource-centric systems, RDF and RDFS adopt a decentralized, property-centric approach. In this framework, anyone can define new properties and use them to describe any class of resources, enabling a more flexible and open data description method.

RDF and RDFS are designed to enable decentralized development of information and vocabularies. This means anyone can create properties and describe any class without needing permission or coordination with class owners. This principle ensures that RDF and RDFS can accommodate diverse and evolving data descriptions.

RDF(S) is built upon URIs, the web's naming system, and XML, the standard syntax for web data representation. Extensions of RDF(S), such as the DAML+OIL and OWL ontology languages, add further capabilities. Ongoing research into querying the Semantic Web and expressing rules will likely influence future standards.

The subsequent content delves into RDF's abstract syntax, the role of blank nodes, data types, and its representation in XML. It also explores RDFS concepts including class hierarchies, properties, constraints, and predefined vocabularies. Additionally, there is a discussion on the main features and limitations of RDF(S), along with references to more advanced languages that address some of these limitations. It's important to note that this

information reflects the status of RDF specifications as of March 2003, and may not encompass the latest updates by the W3C RDFCore Working Group [6][7].

2.1 RDF streaming

RDF Stream Processing (RSP) systems enhance the SPARQL language to handle the dynamic and fast-changing nature of RDF data streams, allowing real-time queries. These systems differ in their syntax and capabilities, as they use various operators and extensions to SPARQL. Furthermore, they vary in how they execute and return query results, making it challenging to define, compare, and assess their performance.

To address this, efforts are underway to propose a unified model and query language for processing RDF streams, leading to the formation of the RSP Community Group within the W3C. The aim is to develop a standardized model that integrates key features from existing systems while precisely defining its semantics. This includes creating a formal foundation for the RSP query model, focusing on aspects such as correctness, continuous evaluation, execution timing, and operational semantics.

Semantic streams, which capture knowledge evolving over time, are important across multiple domains, including social networks, health monitoring, financial markets, and environmental monitoring. The Semantic Web community has worked on the challenges of handling and reasoning over these complex data streams, leading to the development of RDF Stream Processing and Stream Reasoning methods.

The Web is an ideal setting for semantic streams due to its vast amount of dynamic data, such as that generated by social networks and the Web of Things. RDF streams, which are potentially infinite sequences of time-stamped RDF items arranged chronologically, have been proposed as a means to implement semantic streams on the Web. While several definitions for RDF streams exist, the W3C RSP Community Group is working toward a formal standard based on time-annotated RDF graphs.

Beyond defining a common model, the community also needs to establish standard protocols for exchanging RDF streams. The lack of these standards currently limits the broader adoption and growth of RSP technologies on the Web.

RDF Stream Processing (RSP) refers to the methods and systems used to query and manage RDF data streams in real-time. RDF (Resource Description Framework) is a standard for data exchange on the Web. Unlike static RDF datasets, RDF streams are dynamic, continuous, and

potentially infinite sequences of RDF triples, each marked with a timestamp.

RSP is essential for applications requiring real-time data analysis and decision-making. For example, it supports use cases like monitoring social media for trends, detecting fraud in financial transactions, or tracking environmental data for disaster response. Ongoing efforts to standardize RSP models and protocols aim to enable the development of reliable, interoperable systems that can meet the needs of today's data-driven applications. [8][9]

2.2 RDF Stream Processing Systems

Semantic Web standards like RDF (Resource Description Framework) and SPARQL (SPARQL Protocol and RDF Query Language) offer structured approaches for producing and consuming Linked Data. However, despite their maturity and widespread use, these standards are not well-suited for managing data streams. RDF is static and lacks inherent time and order dimensions, which conflicts with the dynamic, potentially unbounded nature of time-ordered streams. Similarly, SPARQL operates on one-time query semantics, whereas stream processing requires continuous query capabilities.

Due to these limitations, the key challenge is how to effectively model and query streaming events within the Semantic Web. To address this, several RDF Stream Processing (RSP) engines have been developed to extend RDF and SPARQL for stream processing purposes, including:

C-SPARQL: An extension that allows continuous queries over RDF streams.

SPARQLstream: Another approach that modifies SPARQL to handle streaming data.

EP-SPARQL: Integrates event processing capabilities with SPARQL.

CQELS: Optimizes continuous queries over RDF streams for efficiency.

These extensions, while designed for different scenarios and possessing heterogeneous semantics, share common features such as similar RDF stream models, window operators, and continuous queries. However, a standardized set of these extensions has yet to be established, although efforts towards consensus are ongoing within the community.

The main distinction between traditional RDF/SPARQL processing and RDF Stream Processing lies in the time dimension. In RSP, time is crucial and must be incorporated into both the data and query models.

The RDF data model traditionally ignores the time dimension, as noted in the RDF 1.1 recommendation. To address this, the model needs extensions to include time, leading to two primary classes of data:

- RDF Stream: Represents dynamic, time-annotated RDF data that changes rapidly.
- Background Data: Includes static or quasi-static data that changes slowly compared to the stream rate. This data helps solve more complex queries, combining real-time data streams with more stable RDF graphs.
- Background data encompasses RDF stored in SPARQL endpoints, repositories, and other static sets. To incorporate time, concepts like time-varying and instantaneous graphs are used. A time-varying graph maps time instants to RDF graphs, capturing the dynamic evolution of data.

Incorporating the time dimension into the query model shifts from a one-time evaluation paradigm to a continuous one. SPARQL allows single-time queries, while RSP-QL (RDF Stream Processing Query Language) supports continuous queries that are registered once and evaluated repeatedly. The results of continuous queries are listed for each evaluation iteration. An RSP-QL query extends the traditional SPARQL query model to include time instants and streaming data:
SE (RSP-QL Algebraic Expression): Extends SPARQL algebra to support continuous evaluation.
SDS (RSP-QL Dataset): Includes both time-varying graphs and RDF streams.
ET (Evaluation Time Instants): The sequence of time instants for query evaluation.
QF (Query Form): Maintains compatibility with SPARQL's SELECT, CONSTRUCT, DESCRIBE, and ASK forms.

The notion of Evaluation Time instances (ET) is essential for modeling continuous queries. However, specifying an infinite sequence of time instants is impractical. Instead, policies (such as those in the SECRET framework) dictate when to evaluate the query, which could be periodically or based on the state of the data stream.

RSP-QL uses time-varying graphs and sliding windows to manage stream data. A time-based sliding window defines a subset of the RDF stream for evaluation, characterized by parameters such as width, slide, and start time. This method generates sequences of windows, allowing portions of streaming data to be queried as RDF graphs.

An RSP-QL dataset (SDS) includes an optional default graph, named graphs, and named sliding windows over streams, extending the traditional SPARQL dataset model to support continuous data. In the next paragraph, the engines and their focus points will be analyzed more.

RDF Stream Processing (RSP) engines extend traditional RDF and SPARQL to handle continuous data streams. They are designed to process dynamic and potentially infinite sequences of RDF data, enabling real-time querying and analysis. Key RSP engines include:

- **C-SPARQL (Continuous SPARQL):** Developed to support continuous queries over RDF streams. C-SPARQL introduces window operators to define finite subsets of the stream for evaluation and extends SPARQL syntax and semantics to handle streaming data. It is particularly useful for applications requiring real-time monitoring and event detection.
- **SPARQLstream:** Modifies SPARQL to process streaming data by adding time-based operators and constructs. It supports continuous queries and allows users to define windows over the data streams. SPARQLstream focuses on integrating streaming data with traditional RDF datasets, enabling complex queries that combine static and dynamic data.
- **EP-SPARQL (Event Processing SPARQL)**: Integrates event processing capabilities with SPARQL, allowing for the definition and detection of complex event patterns within RDF streams. EP-SPARQL extends SPARQL with constructs for event pattern matching and temporal reasoning, making it suitable for applications like event-driven monitoring and alerting systems.
- **CQELS (Continuous Query Evaluation over Linked Streams):** Optimizes continuous queries over RDF streams for efficiency. CQELS introduces a query engine specifically designed for low-latency and high-throughput processing of streaming data. It employs techniques like incremental query processing and adaptive indexing to enhance performance, making it ideal for scenarios with high-volume data streams.

All engines use window operators to define finite, manageable subsets of the potentially infinite RDF streams. Windows can be time-based (e.g., data from the last 10 minutes) or count-based (e.g., the last 100 data items). However, the main contrast is that RSP engines support continuous queries that run perpetually, producing incremental results as new data

arrives. This contrasts with traditional SPARQL queries that execute once and return a single result set.

RDF Stream Processing engines such as C-SPARQL, SPARQLstream, EP-SPARQL, and CQELS are essential tools for extending RDF and SPARQL to handle continuous data streams. They share common features like window operators and continuous queries but differ in their specific implementations and scenarios. Current prototypes focus on query evaluation and optimization, with ongoing efforts to standardize these extensions to support broader, asynchronous, and dynamic RDF stream processing.

3. State of the art

This chapter includes a detailed analysis of existing technologies in RDF stream processing that have been proposed by researchers what challenges and problems have been solved and what needs to be addressed in the future.

RDF (Resource Description Framework) and SPARQL (SPARQL Protocol and RDF Query Language) are well-established standards for data interchange and querying on the Web. These standards enable the creation and consumption of Linked Data, providing a robust framework for addressing data integration challenges. However, traditional RDF and SPARQL models are inherently static and lack the temporal dimension required to handle the dynamic and continuous nature of data streams.

The growing need to process continuous streams of data such as events from social networks, financial transactions, traffic data, and sensor alerts has led to the development of RDF Stream Processing (RSP). RSP extends RDF and SPARQL to manage data streams, which are potentially unbounded sequences of time-varying RDF statements or graphs. Several RSP systems have emerged, including C-SPARQL, SPARQLstream, EP-SPARQL, and CQELS, each proposing different syntaxes and semantics to handle streaming data.

One of the significant challenges in RSP is the lack of a standardized approach. Existing RSP systems differ in their stream models, window operators, and continuous query semantics. This heterogeneity complicates the process of interpreting, integrating, and comparing streaming data from different sources. Furthermore, the tight coupling between stream producers and RSP engines in current implementations poses scalability, elasticity, and resilience issues.

To address these challenges, researchers have proposed formal models to define the semantics of RSP systems. Notably, RSP-QL has been introduced as a comprehensive model to unify the semantics of existing RSP systems. RSP-QL extends the SPARQL model and integrates concepts from existing streaming data models such as CQL (Continuous Query Language) and SECRET. CQL provides a continuous extension of SQL with three types of operators (S2R, R2R, and R2S) for processing streams and relations. SECRET offers a framework for characterizing and analyzing the semantics of window operators in stream processing. RSP-QL formalizes the evaluation semantics of continuous queries in RSP systems, allowing for a unique solution for each query over input data and capturing the operational semantics of prominent SPARQL extensions for time-stamped statements. This formalization defines correctness criteria for RSP systems, enabling the assessment of

whether implementations produce accurate query results.

RDF stream processors handle the production, querying, and consumption of RDF streams. However, current RSP prototypes focus primarily on query evaluation and optimization, often overlooking the broader scenario where RDF stream systems need to reactively produce and consume RDF events asynchronously. This oversight leads to tightly coupled systems where the stream producer and RSP engine are hard-wired together, causing scalability and resilience issues.

Continuous queries are central to RSP, allowing queries to be registered and continuously evaluated over streaming data. Data Stream Management Systems (DSMS) and Complex Event Processors (CEP) have been developed to manage continuous queries, leveraging pull and push-based data access and emphasizing pattern matching and complex event definitions. Despite these advancements, commercial CEP solutions generally lack semantically rich annotation capabilities.

The development of formal models like RSP-QL represents a significant advancement in RDF stream processing, providing a unified framework to describe, compare, and evaluate different RSP systems. Despite the progress, challenges remain in standardizing RSP approaches and addressing the tight coupling in current implementations. Ongoing efforts in the community aim to establish common standards and improve the scalability, elasticity, and resilience of RSP systems, paving the way for more robust and interoperable solutions in the future [10][11].

3.1 RDF Stream Processing Benchmarks

In computing, benchmarking refers to running a computer program or multiple programs to assess a system's performance through standardized tests. This process allows for the comparison of different components or subsystems, typically using specialized benchmarking tools.

Benchmarks are commonly used to assess the performance features of computer hardware, like a CPU's floating-point calculation abilities. However, benchmarking can also be applied to software, including the evaluation of compilers or database management systems (DBMS). [12].

Benchmarks play a crucial role in various aspects of computing by providing a consistent method to assess and compare the performance of systems or components. In performance evaluation, benchmarks are essential for determining how well a system performs under

specific conditions, which is particularly important for tasks that require high computational power. They enable the comparison of different systems or components by offering a standard set of tests, thus identifying which options deliver the best performance.

Over the years, research has concentrated on developing methodologies for assessing the performance of RDF Stream Processing (RSP) systems, highlighting key challenges and dimensions that benchmarks need to address. These approaches have identified a range of performance metrics to be measured. The literature has proposed several dimensions for these benchmarks.

1. **Combination of Streaming and Static Data:** This dimension evaluates a system's capability to effectively integrate static and streaming data, which influences processing strategies, such as joint operations.
2. **Inference Support:** Providing logical inference in streaming environments is complex, as it requires the system to manage rapidly changing data while incorporating background knowledge, like rules and semantic annotations on data streams.
3. **Response to Burst:** This dimension examines the system's ability to sustain service quality during sudden spikes in input stream rates. Handling bursts is especially difficult in distributed systems, requiring strategic task management, load balancing, and sometimes back pressure mechanisms.

RDF stream processors can be categorized using models like the SECRET model, which defines the operational semantics of window operators used by the engines. By describing how a query is executed with an input stream in a specific system, the model enables validation of RDF stream engines by comparing expected outputs with actual results. This approach has been used to extend the SRBench benchmark to verify engine correctness.

LSBench and Citybench are two tools that feed RDF streams into an RSP engine for testing. Unlike RSPLab, they are tied to specific benchmarks and do not minimize errors by isolating components. Heaven, a proof-of-concept testbed, has a similar architecture to RSPLab but lacks a programmatic environment to simplify experiments. It is also limited to single-thread, window-based RSP engines and depends on a Java Facade for communication, which constrains the processing model. Heaven also lacks features like automated reporting and data visualization.

LOD Lab aims to reduce the human effort in evaluating approaches, offering features like

data cleaning and dataset selection through metadata. However, it does not focus on RDF streams or testing RSP engines. It only offers data provisioning and a command-line interface without a programmatic workflow environment.

OLTP-Bench, a benchmarking framework for relational databases, shares similarities with RSPLab in supporting distributed environments and data visualization. However, it lacks support for RDF streams and does not have a programmatic setup for experiment execution or report generation. It includes a workload manager and SQL translation module, which is adaptable to SQL-based systems but not RDF stream or RSP domains.

In conclusion, the benchmarking landscape for RDF Stream Processing is varied, with different tools addressing various aspects of performance evaluation and system comparison. These benchmarks and models are critical for ensuring that RSP systems handle dynamic data streams efficiently and correctly. [13][14][15].

3.2 Choke points and their meaning

In RDF stream processing benchmarks, "choke points" are specific aspects of query execution that are known to be challenging for current engines. These choke points are used to design continuous queries that stress test the performance and capabilities of RDF stream processors. By isolating these choke points, benchmarks can more effectively evaluate and compare the performance of different systems. Choke points that has been proposed are the following:

CP1: Join Ordering:

- **Goal**: Assess the engine's ability to balance the time spent finding the best execution plan against the quality of the output plan.
- **Considerations**: Evaluate the engine’s capability to handle cardinality constraints and various schema constraints like functional and inverse functional properties in OWL ontologies.

CP2: Aggregation:

- **Goal**: Test how well the optimizer handles sub-select operations within SPARQL queries.

- **Considerations**: Ensure the optimizer correctly evaluates operations included in sub-selects first.

CP3: Optional & Nested Optional Clauses:

- **Goal**: Measure the optimizer's efficiency in executing optional triple patterns last, as they do not reduce the size of intermediate results.
- **Considerations**: The ability to handle complex query structures involving optional clauses.

CP4: Reasoning:

- **Goal**: Evaluate the engine's proficiency in processing RDFS and OWL constructs efficiently.
- **Considerations**: Performance in handling reasoning tasks essential for RDF data.

CP5: Parallel Execution of Joins:

- **Goal**: Test the engine's capability to execute union operations in parallel.
- **Considerations**: Optimization of query plans for parallel execution.

CP6: Filters:

- **Goal**: Assess how early the engine can eliminate a large number of intermediate results.
- **Considerations**: Efficiency in applying filters to reduce the data set size early in the query execution process.

CP7: Ordering:

- **Goal**: Test the engine's ability to choose query plans that facilitate result ordering.
- **Considerations**: Efficiency in ordering results, which is critical for many queries.

CP8: Geo-spatial Predicates:

- **Goal**: Evaluate the system's capability to handle queries involving geospatial data.
- **Considerations**: Performance in processing and indexing spatial information.

CP9: Full-text Search:

- **Goal**: Test the engine's ability to evaluate regular expressions on data value properties.
- **Considerations**: Effectiveness in integrating full-text search capabilities.

CP10: Duplicate Elimination:

- **Goal**: Measure the engine's ability to identify and eliminate duplicate entries during intermediate result creation.
- **Considerations**: Efficiency in maintaining data integrity by removing duplicates.

CP11: Complex Filter Conditions:

- **Goal**: Assess the engine's handling of complex conditions like negation, conjunction, and disjunction.
- **Considerations**: Breaking down complex filters into conjunctions of filters and executing them in parallel.

Choke points play a critical role in benchmark design by stressing essential technical functionalities of RDF stream processing systems. A well-designed benchmark should reflect realistic scenarios that users can relate to, ensuring the benchmark's workload tests significant technical aspects of the systems. This approach not only validates current technologies but also encourages improvements in areas where performance bottlenecks still exist.

The Linked Data Benchmark Council (LDBC) has pioneered this concept, systematically identifying and documenting choke points relevant to specific scenarios. By keeping a close watch on which queries test which choke points, benchmarks can achieve comprehensive coverage with a limited number of queries, ensuring a thorough evaluation of system capabilities.

In addition to the listed choke points, modern RDF stream processing systems must also consider several other aspects to ensure comprehensive evaluation and improvement. Efficiently managing sliding and tumbling windows is crucial for processing continuous data streams, which directly impacts the system's ability to handle real-time data effectively. Minimizing latency is another key consideration, as it reduces the time between data ingestion and query result generation, thus enhancing the responsiveness of the system. Scalability is vital to ensure that the system can accommodate increasing data volumes and query complexities without degrading performance. Furthermore, fault tolerance is essential for

maintaining system reliability and performance even in the face of failures or data inconsistencies. Addressing these aspects, along with the established choke points, ensures that benchmarks provide a holistic view of RDF stream processing systems' capabilities, guiding the development of more robust and efficient solutions. [16][17][18]

3.3 Comparison of existing benchmark based on choke points

In the realm of RDF Stream Processing (RSP), benchmarks are essential for evaluating and comparing the performance of various engines. These benchmarks simulate real-world data streams and provide a standardized set of queries and metrics for assessment. The primary benchmarks in this domain include LSBench, SRBench (along with its extensions CSRBench and YABench), and CityBench. Each benchmark has unique characteristics and focus areas, making them suitable for different aspects of RSP evaluation.

In the absence of a standard benchmark, Le-Phuoc introduced LSBench, a framework aimed at enabling the comparison of streaming-linked data processing engines. LSBench features a data generator (S2Gen) that simulates realistic social media data and provides a set of twelve queries. The framework produces data streams in three categories: a GPS stream tracking user locations over time, a social media stream with posts, likes, comments, and tags, and a multimedia stream containing photos and related attributes.

Additionally, LSBench encompasses static data such as user profiles and social network relationships. Each user has a forum for posting, and users can subscribe to forums to interact with posts and comments. LSBench is scalable, portable, and supports three testing categories: query language expressiveness, which evaluates the complexity of supported queries; execution throughput and scalability, which assesses performance with varying numbers of queries and static data sizes; and result discrepancies, which compare output from different engines to check correctness.

The benchmark includes twelve queries, primarily using SPARQL constructs like filters and joins. Some queries test advanced features such as subqueries, negation, and top K. However, only five queries involve both static and streaming data and none require inference capabilities. LSBench evaluates feature coverage, which tests if an engine supports specific queries; output comparability, which checks if engines produce identical results; and performance, which measures the maximum input rate under different scalability conditions.

SRBench and its extensions aim to provide benchmarks tailored for RDF Stream Processing engines. SRBench uses data from the LinkedSensorWeb, focusing on weather observations

during hurricanes, augmented with static information from DBpedia and GeoNames. SRBench defines seventeen queries to assess core RDF Stream Processing tasks, focusing on query language coverage to evaluate if an engine can execute specified queries, and stream and background data integration, with queries Q1 to Q7 about streaming data, Q8 to Q11 involving both streaming and background data, and Q12 to Q17 including external static data.

CSRBench extends SRBench by considering operational semantics variations such as reporting policies, windowing mechanisms, and timestamping. It introduces aggregated queries, queries joining triples with differing timestamps, and parametric sliding window queries, and provides an oracle for automated query result verification.

YABench builds on CSRBench, emphasizing both correctness and performance. It uses the same weather observation dataset and query set as CSRBench, adding comprehensive correctness metrics like precision and recall for each window.

CityBench provides real-world data streams from the CityPulse project, including vehicle traffic, weather, parking, user location, and air pollution data, and also incorporates static datasets related to cultural events. It features thirteen continuous queries applicable to smart city scenarios, expressed in C-SPARQL and CQELS QL dialects. These queries include advanced reporting, aggregates, and Top K features. CityBench allows parameterization for scalability experiments, testing the number of sensor streams, the number of concurrent queries, the background data size, playback time, and the input streaming rate.

The discussed benchmarks are designed for evaluating web stream processing engines, combining static and stream data, and considering inference to varying extents. Table 5 highlights the choke points each benchmark addresses, such as optional clauses, reasoning, join ordering, aggregation, filters, geospatial predicates, and complex filters. Most benchmarks neglect parallel execution of joins, full-text search, and duplicate elimination.

Besides these specific RSP benchmarks, other relevant benchmarks include the Linear Road Benchmark, used for evaluating stream processing systems like C-SPARQL engines; LUBM, a reasoning benchmark with a scalable data generator and expressive queries, related to incremental RDFS maintenance; the Berlin SPARQL Benchmark, which focuses on an e-commerce use case and evaluates systems like CQELS and Sparkwave; and SP2Bench, which contains eleven queries based on the DBLP RDF dataset and is used for evaluating CQELS. [17]

4. Development of an RDF Streaming benchmark

In this chapter, we will thoroughly analyze the entire implementation process of the RDF Streaming Benchmark I developed. This includes detailed discussions on the hardware and software utilized, as well as the programming languages and libraries involved. Furthermore, key parts of the benchmark code and the specific queries executed to extract performance metrics based on identified choke points will be illustrated with the help of code screenshots. Lastly, we will discuss some of the challenges encountered during the implementation and provide relevant disclaimers.

4.1 Hardware and Software

The proposed benchmark has been developed in a 15" inch Asus laptop which consists of a quad-core CPU Intel i5 processor, 8GB of RAM, 256 GB SSD, and running on Windows 10 Operating System. The IDE (Integrated Development Environment) I used is Eclipse developed by Eclipse Foundation in version 4.3

4.2 Programming languages and libraries

Prior to implementation, research focused on finding innovative technologies capable of seamless interoperability and supporting complex systems like the one in question. These technologies needed to be maintainable and allow for future expansions. A key criterion for selection was their popularity and the presence of a large community offering open-source and pre-built solutions for various implementation challenges. Additionally, these technologies had to be easily adoptable by other developers in the future, as the code would be open source. The chosen technologies for implementing the system are:

- Java: The main programming language for creating the proper classes (e.g. Stream Generator) and the manipulation of the data.
- Rsp4j: This library provides tools for managing data streams, processing them reactively, and integrating them with various data sources.
- RSP-QL: RSP-QL (Reactive Stream Processing Query Language) is a query language specifically designed for reactive stream processing in dynamic and real-time environments. It extends traditional SPARQL to handle continuous queries over RDF

data streams, allowing for the querying and reasoning over data that is constantly changing, enabling real-time insights and actions.

4.2.1 Java

Java is a high-level, object-oriented programming language designed to reduce implementation dependencies. It allows developers to "write once, run anywhere" (WORA), meaning Java code can be compiled once and executed on any system that supports Java without requiring recompilation. Java programs are compiled into bytecode, which can run on any Java Virtual Machine (JVM), regardless of the system's hardware. Its syntax is similar to C and C++, though it offers fewer low-level features. Java also supports dynamic capabilities like reflection and runtime code modification, which are typically absent in traditional compiled languages.

Since its release, Java quickly gained popularity and remains widely used. As of 2022, it ranked as the third most popular language on GitHub, though its usage has been declining with the rise of other JVM languages.

Java was created by James Gosling at Sun Microsystems and launched in May 1995 as part of Sun's Java platform. Initially, Sun distributed Java compilers, virtual machines, and class libraries under proprietary licenses, but in 2007, they re-licensed most Java technologies under the GPL-2.0-only license to meet the Java Community Process standards. Today, Oracle provides the HotSpot JVM, but the primary reference implementation is the OpenJDK JVM, which is open-source and the default JVM for most Linux systems.

The JVM provides notable performance benefits through just-in-time (JIT) compilation, which translates bytecode to native machine code during execution, optimizing performance across different hardware platforms. JVMs also manage memory efficiently via garbage collection, reducing memory-related issues often encountered in lower-level programming languages.

Java’s syntax, while simpler than C and C++, remains powerful and versatile, supporting multiple programming paradigms. Its balance of performance, simplicity, and cross-platform functionality has played a major role in its enduring popularity and extensive use. [19]

4.2.2 RSP4J

RSP4J is an API designed to facilitate the rapid development of RDF Stream Processing (RSP) engines and applications that follow RSP-QL semantics. Much like the Apache Beam project, which integrates batch and stream processing, RSP4J provides a unified interface for existing RSP engines. It comprises several core modules—Core, Streams, IO, Querying, and Engine—that offer essential abstractions for defining RSP engines according to RSP-QL semantics.

While space constraints prevent a detailed discussion of the Querying module, which enables the creation of RSP programs using RSP-QL syntax, and the Engine module, which includes tools for controlling and monitoring the lifecycle of RSP engines, RSP4J's other modules are described thoroughly. The IO, Stream, and Core modules simplify the setup and operation of RSP engines. Additionally, RSP4J includes YASPER, a basic RSP engine implementation intended for use as a case study and educational tool.

RSP4J also supports the description and publication of web streams, which can be consumed by RSP4J engines or other engines. For example, using RSP4J's Publishers pattern, one can easily create a web stream description that aligns with the Linked Data life-cycle, utilizing VoCalS for stream descriptions. Moreover, RSP4J is an open-source project, offering canonical citations, permanent web identifiers, and comprehensive documentation for developers. This flexibility and comprehensive support make it a valuable tool for systematic and comparative research in RDF stream processing. [20][21]

4.2.3 RSP-QL

RDF (Resource Description Framework) and SPARQL (SPARQL Protocol and RDF Query Language) are well-established standards for data interchange and querying on the Web. Despite their widespread applicability, these technologies fall short when dealing with streams of data, which are inherently continuous. To address this limitation, the field of RDF Stream Processing (RSP) has emerged, proposing extensions to RDF and SPARQL to handle continuous data processing. Various models and implementations have been developed under RSP, each offering alternative syntaxes but aiming for similar semantics. However, due to differences in their operational semantics, these implementations can produce varied results for the same queries on the same data streams at different times, complicating the understanding and comparison of continuous query results.

To resolve these issues, **RSP-QL (RDF Stream Processing Query Language)** has been proposed as a comprehensive model that formally defines the semantics of an RSP system. RSP-QL clarifies the implicit assumptions of current RSP systems and establishes a formal notion of correctness for RSP query results, explaining the discrepancies observed in existing implementations.

RSP-QL is an extension of SPARQL designed to query streams of RDF graphs. The key design requirements of RSP-QL are:

1. Queries evaluated over input data should yield a unique solution.
2. RSP-QL should encapsulate the operational semantics of major SPARQL extensions for timestamped statements, including C-SPARQL, CQELS, and SPARQLstream.

A significant distinction between SPARQL and RSP-QL lies in query evaluation. SPARQL supports one-time queries, which are evaluated once by the SPARQL engine. In contrast, RSP-QL supports continuous queries, which are registered once and continuously evaluated, producing results from each evaluation iteration. This aligns with the data stream management system (DSMS) nomenclature.

RSP-QL extends the capabilities of existing RSP languages, enhancing their application domains. It allows queries to be applied directly to multiple RDF triples grouped within graphs, enabling real-time reasoning over sensor data, urban computing, and social semantic data. Unique features of RSP-QL, not present in other RSP languages, include the FROM NAMED WINDOW clause in dataset declarations and the WINDOW keyword in the WHERE clause, further broadening its utility and effectiveness in processing continuous data streams. [11]

4.3 Approach & Implementation

In this section will be an overview of the system that has been implemented showing the key parts of the source code, the queries that have been used as well a comparison of the choke points that the proposed benchmark satisfies compared to the already existing benchmarks.

Firstly, the dataset utilized for this system is based on the CityBench benchmark, which encompasses various types of urban data to support smart city initiatives that started in 2014 for Denmark. This benchmark includes information on the real-time and total capacity of parking lots, pollution levels, and details about various city events. Typically, such data is

collected through sensors and IoT devices deployed across the city. By leveraging this dataset, the system aims to enhance urban management and improve the quality of life in smart cities. For the needs of the project, however, I chose to use only data for four parking lot names and their real-time capacity which is randomly generated (1-100) in each observation stream. Despite the fact that there is no real-time data or the data are finite, there is no concern about the final purpose of this work which is to examine how well handle queries in the RDF final graph and if the correct results are returned.

The project consists of 2 main Java files (along with the other included libraries packages and dependencies such as rsp4j). The first file named "StreamGenerator2.java" includes an array of some parking names and an integer variable that is randomly generated in the range (1-100) representing the current seats of each parking. Also, it has a variable that represents the final RDF graph with the triplets in each observation as well as methods for creating these triples, starting and stopping the stream after a specific time period (20 seconds) has been passed.

RDF (Resource Description Framework) works with triplets, commonly referred to as triples. An RDF triple consists of three components:

1. **Subject**: The entity or resource being described.
2. **Predicate**: The property or relationship of the subject.
3. **Object**: The value or another resource related to the subject.

So, the triples are in the form: of "ParkingName" (object) "hasSeats " (predicate) "NumberOfSeats" (subject). Below there is an indicative figure of the triples that the StreamGenerator2 file handles.

```
<http://rsp4j.io/parkingStream/names/NORREPORT> <http://rsp4j.io/parkingStream/names/hasSeats> "76" .
<http://rsp4j.io/parkingStream/names/BRUNNS> <http://rsp4j.io/parkingStream/names/hasSeats> "15" .
<http://rsp4j.io/parkingStream/names/SKOLEBAKKEN> <http://rsp4j.io/parkingStream/names/hasSeats> "36" .
<http://rsp4j.io/parkingStream/names/NORREPORT> <http://rsp4j.io/parkingStream/names/hasSeats> "48" .
<http://rsp4j.io/parkingStream/names/MAGASIN> <http://rsp4j.io/parkingStream/names/hasSeats> "71" .
<http://rsp4j.io/parkingStream/names/BRUNNS> <http://rsp4j.io/parkingStream/names/hasSeats> "86" .
<http://rsp4j.io/parkingStream/names/NORREPORT> <http://rsp4j.io/parkingStream/names/hasSeats> "25" .
<http://rsp4j.io/parkingStream/names/SKOLEBAKKEN> <http://rsp4j.io/parkingStream/names/hasSeats> "5" .
<http://rsp4j.io/parkingStream/names/BRUNNS> <http://rsp4j.io/parkingStream/names/hasSeats> "47" .
<http://rsp4j.io/parkingStream/names/BRUNNS> <http://rsp4j.io/parkingStream/names/hasSeats> "17" .
<http://rsp4j.io/parkingStream/names/SKOLEBAKKEN> <http://rsp4j.io/parkingStream/names/hasSeats> "77" .
```

Figure 1 - RDF triples for the parking data

It is common that the Object and the Predicate are represented by URI (Uniform Resource Identifier) because usually, this data exists in resources such as webpages. However, the

object can be either a literal value (like a string or a number) or another resource identified by a URI. In this example (and of course in all triples) the number of the parkingSeats is a literal value of an integer after the proper modification that needs to be in the rsp4j environment.

The below figure shows the data and the variables declared in the StreamGenerator2 file that will help to build the data stream. There is the prefix constant in order to construct the URIs of the triples, a timeout constant for the timeout of the thread that creates the stream, the nameStream variable which represents the final RDF graph, an array of strings that represents parking names, a boolean (true or false) variable that enables the stream to start or stop and random type final constant which the role will be shown below. Also, there are some functions (getters) for the stream and for getting a random parking name of the array.

```
1  package org.streamreasoning.rsp4j.bigdata2021.utils;
2
3  import org.apache.commons.rdf.api.Graph;
25
26 public class StreamGenerator2 {
27     private static final String PREFIX = "http://rsp4j.io/parkingStream/";
28     private static final Long TIMEOUT = 1000L;
29
30     private final AtomicBoolean isStreaming;
31     private DataStream<Graph> nameStream;
32     private String [] parkingNames = {"NORREPORT","SKOLEBAKKEN","MAGASIN","BRUNNS"};
33     private final Random randomGenerator;
34
35
36
37     public StreamGenerator2() {
38         this.isStreaming = new AtomicBoolean(false);
39         this.nameStream = new DataStreamImpl<>(PREFIX+"names");
40         randomGenerator = new Random(1337);
41
42     }
43
44     public DataStream<Graph> getNameStream() {
45
46         return nameStream;
47     }
48     public String getParkingName(String[] parkingNames,int index) {
49
50         return parkingNames[index];
51     }
52
```

Figure 2 - Declaration of the variables in the StreamGenerator2.java file

In the next figure will be explained one of the most important routines of the file, the one that triggers the process of creating the stream and also the call that creates the observation event. The startStreaming method is designed to initiate a background task that simulates a streaming process. It begins by checking if streaming is currently inactive using this.isStreaming.get(). If streaming is not active, it sets isStreaming to true to indicate that streaming will start. Next, a Runnable task is defined using a lambda expression, which will

run in a separate thread. Inside the task, a timestamp variable ts is initialized to 0. The task then enters a while loop that continues as long as this.isStreaming.get() returns true, meaning streaming is active. Within the loop, the current timestamp value is captured in finalTs, and a random number is generated. If this random number is greater than or equal to 0.5, the method nameStream.put(createRandomVehicleCount(parkingNames), ts) is called to simulate streaming data. After this, the timestamp is incremented by 5 minutes (in milliseconds). The thread then sleeps for a duration specified by TIMEOUT. If interrupted, it catches the InterruptedException and prints the stack trace. Finally, a new Thread is created to run the defined task, and the thread is started with thread.start(), initiating the background streaming process. This method sets up a simple simulation of streaming data in a multithreaded environment, with periodic checks and intervals to control the data generation process.

```
public void startStreaming() {
    if (!this.isStreaming.get()) {
        this.isStreaming.set(true);
  Runnable task =
      () -> {
        long ts = 0;
        while (this.isStreaming.get()) {
          long finalTs = ts;
            if (randomGenerator.nextDouble()>=0.5){
                nameStream.put(createRandomVehicleCount(parkingNames),ts);
            }
          ts += 5*60*1000;
          try {
            Thread.sleep(TIMEOUT);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }
      };

        Thread thread = new Thread(task);
        thread.start();
    }
}
```

Figure 3 - startStreaming method

In the next figure, The createRandomVehicleCount method generates a random vehicle count for a given set of parking names and creates an RDF graph representing this information. It begins by instantiating two Random objects: one for generating a random index and the other for generating a random vehicle count. Using the first Random object, it selects a random index within the bounds of the parkingNames array and retrieves the corresponding parking name by calling getParkingName(parkingNames, parkingNameIndex).

An RDF instance is then obtained using RDFUtils.getInstance(), which provides methods for creating RDF components. A new RDF graph is created with instance.createGraph(). The method defines an IRI (Internationalized Resource Identifier) for the RDF type using instance.createIRI("http://www.w3.org/1999/02/22-rdf-syntax-ns#type").

Next, a random number between 1 and 100 is generated to represent the number of seats in the parking area. An IRI for the parking name is created with instance.createIRI(nameStream + "/" + parkingName), and another IRI for the "hasSeats" property is created with instance.createIRI(nameStream + "/" + "hasSeats"). A literal value for the number of seats is generated using instance.createLiteral(String.valueOf(seats)).

The method then adds a triple to the RDF graph representing the relationship between the parking name and the number of seats with graph.add(instance.createTriple(parkingNameIRI, hasIRI, seatsLiteral)). This triple consists of the subject (parking name IRI), predicate (hasSeats IRI), and object (seats literal). The resulting graph is printed to the console and returned by the method. This method effectively creates and returns an RDF graph that associates a randomly selected parking name with a randomly generated vehicle count.

```
public Graph createRandomVehicleCount(String [] parkingNames){

    Random random = new Random();
    Random randomIndexName = new Random();
    int parkingNameIndex = randomIndexName.nextInt(parkingNames.length);
    String parkingName = getParkingName(parkingNames,parkingNameIndex);

    RDF instance = RDFUtils.getInstance();
    Graph graph = instance.createGraph();
    IRI a = instance.createIRI("http://www.w3.org/1999/02/22-rdf-syntax-ns#type");
    int seats = random.nextInt(101);
    IRI parkingNameIRI = instance.createIRI(nameStream + "/" +parkingName);
    IRI hasIRI = instance.createIRI(nameStream + "/" +"hasSeats");
    Literal seatsLiteral = instance.createLiteral(String.valueOf(seats));

    IRI parkingnameIRI = instance.createIRI("http://example.org/ontology#parkingname");
    graph.add(instance.createTriple(parkingNameIRI, hasIRI, seatsLiteral));
    System.out.println(graph);
    return graph;
}
```

Figure 4 - createRandomVehicleCount method

The stopStreaming method is straightforward. It sets the isStreaming flag to false, which will stop the streaming process when the startStreaming method's while loop checks this flag.

The main method serves as the entry point of the program. Inside the main method, an instance of the StreamGenerator2 class is created by calling its constructor (StreamGenerator2 generator = new StreamGenerator2()). Once the instance is created, the startStreaming method

is invoked on this instance (generator.startStreaming()), which starts the streaming process in a new thread.

In summary, the stopStreaming method allows the streaming process to be stopped by setting a flag, and the main method initiates the streaming process by creating an instance of StreamGenerator2 and calling the startStreaming method on it. This setup ensures that streaming starts when the program runs. The main method is necessary to run the file separately and check that the RDF graph with the triples is properly created.

```
public void stopStreaming() {
    this.isStreaming.set(false);
}

public static void main(String[] args) {
    StreamGenerator2 generator = new StreamGenerator2();
    generator.startStreaming();


}
```

Figure 5 - stopStreaming and main method of StreamGenerator2 file

The above figures show the flow of the StreamGenerator2 file the creation of the data streams and the addition to the final Graph. In the next section, there is a similar presentation about the other main java file named QueryProcessingSolution2. In the first figure, there is the declaration of the basic two variables. Variable with the name of "generator" has the type of StreamGenerator2 which allows the connection with StreamGenerator2 file (where the final RDF graph is stored after the stream starts). Variable with name "nameStream" has the value of the stream URL "<http://rsp4j.io/parkingStream/names/> ". These objects are created in the constructor of the file.

```
public class QueryProcessingSolution2 {
    private StreamGenerator2 generator;
    private DataStream<Graph> nameStream;

    public QueryProcessingSolution2() throws InterruptedException {
        generator = new StreamGenerator2();
        nameStream = generator.getNameStream();
    }
```

Figure 6 - QueryProcessingSolution2 variables and constructor

Afterwards, there is the main function which is the starting point of every Java program. In the main function, there is the call of all methods which are the queries towards the RDF graph.

An object named “querySol2” type of QueryProcessingSolution2 is necessary to call these methods.

```
public static void main(String[] args) throws InterruptedException {
    QueryProcessingSolution2 querySol2 = new QueryProcessingSolution2();
    querySol2.showAllParkings();
    //querySol2.showParkingSpecificName();
    //querySol2.showParkingSeats20();
    //querySol2.showParkingsSeatsOver60();
    //querySol2.showParkingsSeatsUnder40();
    //querySol2.showSeatsOver20AndUnder70();
    //querySol2.showSeatsBetweenRange();
    //querySol2.queryCountParkingName();
    //querySol2.queryMinSeats();
    //querySol2.subQuery();
    //querySol2.complexFilter();
    //querySol2.reasoningQuery();
}
```

Figure 7 - main method in QueryProcessingSolution2 file

In this section, each method will be analyzed separately and also the results of each query will be shown. The majority of all methods have the same logic, so it will be analyzed once and only the query code will be described separately. The TPQueryFactory.parse method is used to define a continuous query in RSP-QL. The query registers an output stream (RSTREAM <http://out/stream>) and selects triples with subjects (?s), predicates (?hasSeats), and objects (?o) from a named window. The window is defined over the stream: names with a range of 10 minutes (PT10M) and a step size of 1 minute (PT1M).

A TaskOperatorAPIImpl instance is created using a query task builder, which takes the defined continuous query as input. A ContinuousProgram is built to link the input RDF stream (nameStream), the task operator (t), and the output stream of the query. A hash join algorithm is specified for the join operations within the continuous program.

A consumer is added to the query's output stream to process and print the results along with their timestamps. The generator.startStreaming() method starts the data streaming process. The Thread.sleep(20_000) method allows the streaming to run for 20 seconds. Finally, generator.stopStreaming() stops the data streaming process.

```
// Create the RSP4J Task and Continuous Program
TaskOperatorAPIImpl<Graph, Graph, Binding, Binding> t =
        new QueryTaskOperatorAPIImpl.QueryTaskBuilder().fromQuery(query).build();

ContinuousProgram<Graph, Graph, Binding, Binding> cp =
        new ContinuousProgram.ContinuousProgramBuilder()
                .in(nameStream)
                .addTask(t)
                .out(query.getOutputStream())
                .addJoinAlgorithm(new HashJoinAlgorithm())
                .build();
// Add the Consumer to the stream
query.getOutputStream().addConsumer((el, ts) -> System.out.println(el + " @ " + ts));

// Start streaming
generator.startStreaming();

// Stop streaming after 20s
Thread.sleep(20_000);
generator.stopStreaming();
```

Figure 8 - Common code snippet for all the methods

The query in the first method returns all the results without any restriction on the RDF graph after the stream has been started.

```
"PREFIX : <http://rsp4j.io/parkingStream/> "
        + "PREFIX rsp4j: <http://rsp4j.io/> "
        + "PREFIX names: <http://rsp4j.io/parkingStream/names/> "
        + "REGISTER RSTREAM <http://out/stream> AS "
        + "SELECT ?s ?hasSeats ?o "
        + "FROM NAMED WINDOW rsp4j:window ON :names [RANGE PT10M STEP PT1M] "
        + "WHERE {"
        + "  WINDOW rsp4j:window {?s ?hasSeats ?o ."
        + "}"
        );
```

Figure 9 - First query

The results are in the below figure expressed as RDF triples (object (name of the parking) , predicate (hasSeats), and subject (the number of the parking seats).

```
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "20"} @
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "20"} @
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "93"} @
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "93"} @
{s -> <http://rsp4j.io/parkingStream/names/MAGASIN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "92"} @ 9
{s -> <http://rsp4j.io/parkingStream/names/MAGASIN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "45"} @ 1
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "70"}
{s -> <http://rsp4j.io/parkingStream/names/BRUNNS>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "46"} @ 48
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "88"}
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "79"} @
```

Figure 10 - Returned Results of the first query

The second query returns all the parking data with a specific name. This is achieved with the use of the keyword FILTER in rsp-ql which allows to set restrictions either for comparison between strings or numeric operations (greater than or less than etc). This query will give all the parking with the name NORREPORT.

```
"PREFIX : <http://rsp4j.io/parkingStream/> "
        + "PREFIX rsp4j: <http://rsp4j.io/> "
        + "PREFIX names: <http://rsp4j.io/parkingStream/names/> "
        + "REGISTER RSTREAM <http://out/stream> AS "
        + "SELECT ?s ?hasSeats ?o "
        + "FROM NAMED WINDOW rsp4j:window ON :names [RANGE PT10M STEP PT1M] "
        + "WHERE {"
        + "  WINDOW rsp4j:window {?s ?hasSeats ?o.}"
        + "  FILTER(?s = names:NORREPORT)"
        + "}"
        );
```

Figure 11 - showParkingSpecificName query

```
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "8"}
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "8"}
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "40"}
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "19"}
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "75"}
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "75"}
```

Figure 12 - Results of showParkingSpecificName query

The third query which is in method showParkingSeats20() shows all the parkings with 20 total seats.

```
"PREFIX : <http://rsp4j.io/parkingStream/> "
        + "PREFIX rsp4j: <http://rsp4j.io/> "
        + "PREFIX names: <http://rsp4j.io/parkingStream/names/> "
        + "REGISTER RSTREAM <http://out/stream> AS "
        + "SELECT ?s ?hasSeats ?o "
        + "FROM NAMED WINDOW rsp4j:window ON :names [RANGE PT10M STEP PT1M] "
        + "WHERE {"
        + "  WINDOW rsp4j:window {?s ?hasSeats ?o.}"
        + "  FILTER(?o = 20)"
        + "}"
        );
```

Figure 13 - showParkingSeats20() query

```
{s -> <http://rsp4j.io/parkingStream/names/MAGASIN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "20"} @
{s -> <http://rsp4j.io/parkingStream/names/MAGASIN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "20"} @
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "20"
```

Figure 14 - Results of showParkingSeats20() query

Query in the showParkingSeatsOver60() returns all the records where the parking seats are over 60. The code is shown in the below figure.

```
"PREFIX : <http://rsp4j.io/parkingStream/> "
        + "PREFIX rsp4j: <http://rsp4j.io/> "
        + "PREFIX names: <http://rsp4j.io/parkingStream/names/> "
        + "REGISTER RSTREAM <http://out/stream> AS "
        + "SELECT ?s ?hasSeats ?o "
        + "FROM NAMED WINDOW rsp4j:window ON :names [RANGE PT10M STEP PT1M] "
        + "WHERE {"
        + "  WINDOW rsp4j:window {?s ?hasSeats ?o ."
        + "  FILTER(?o > 60)"
        + "}"
        );
```

Figure 15 - showParkingsSeatsOver60() query

The returned results are presented in this image.

```
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "98"}
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "98"}
{s -> <http://rsp4j.io/parkingStream/names/BRUNNS>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "66"} @ 60
{s -> <http://rsp4j.io/parkingStream/names/BRUNNS>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "77"} @ 90
{s -> <http://rsp4j.io/parkingStream/names/BRUNNS>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "66"} @ 90
{s -> <http://rsp4j.io/parkingStream/names/MAGASIN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "70"} @ 2
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "86"} @
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "92"}
{s -> <http://rsp4j.io/parkingStream/names/BRUNNS>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "71"} @ 87
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "71"} @
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "83"} @
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "71"} @
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "83"} @
```

Figure 16 - Results of showParkingsSeatsOver60() query

Also, it is possible to do a comparison with less than operator which will be shown in the next method named showParkingsSeatsUnder40 checking for all parking with capacity under 40.

```
"PREFIX : <http://rsp4j.io/parkingStream/> "
        + "PREFIX rsp4j: <http://rsp4j.io/> "
        + "PREFIX names: <http://rsp4j.io/parkingStream/names/> "
        + "REGISTER RSTREAM <http://out/stream> AS "
        + "SELECT ?s ?hasSeats ?o "
        + "FROM NAMED WINDOW rsp4j:window ON :names [RANGE PT10M STEP PT1M] "
        + "WHERE {"
        + "  WINDOW rsp4j:window {?s ?hasSeats ?o ."
        + "  FILTER(?o < 40)"
        + "}"
        );
```

Figure 17 - Query showParkingsSeatsUnder40

```
{s -> <http://rsp4j.io/parkingStream/names/MAGASIN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "39"} @ 2
{s -> <http://rsp4j.io/parkingStream/names/BRUNNS>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "4"} @ 480
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "26"} @
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "25"}
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "25"}
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "3"}
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "3"}
```

Figure 18 - Query results of showParkingsSeatsUnder40

In rsp-ql there is the ability to do also more complicated comparisons using operators like && (logical AND), || (logical OR) and != (NOT operator). This kind of is constructed in the method showSeatsOver20AndUnder70() where it will show the parking where the seats are over 20 and under 70.

```
"PREFIX : <http://rsp4j.io/parkingStream/> "
        + "PREFIX rsp4j: <http://rsp4j.io/> "
        + "PREFIX names: <http://rsp4j.io/parkingStream/names/> "
        + "REGISTER RSTREAM <http://out/stream> AS "
        + "SELECT ?s ?hasSeats ?o "
        + "FROM NAMED WINDOW rsp4j:window ON :names [RANGE PT10M STEP PT1M] "
        + "WHERE {"
        + "  WINDOW rsp4j:window {?s ?hasSeats ?o ."
        + "  FILTER(?o >= 20 && ?o <= 70)"
        + "}"
        );
```

Figure 19 - showSeatsOver20AndUnder70 query

```
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "65"}
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "65"}
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "62"} @
{s -> <http://rsp4j.io/parkingStream/names/MAGASIN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "53"} @ 6
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "31"} @
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "31"} @
```

Figure 20 - Results of showSeatsOver20AndUnder70 query

The query in showSeatsBetweenRange method performs a more complicated comparison with the FILTER keyword where it returns all the parking with seats in a range between 20 and 30 or between 70 and 80. In the below image, there is the code of this query.

```
"PREFIX : <http://rsp4j.io/parkingStream/> "
        + "PREFIX rsp4j: <http://rsp4j.io/> "
        + "PREFIX names: <http://rsp4j.io/parkingStream/names/> "
        + "REGISTER RSTREAM <http://out/stream> AS "
        + "SELECT ?s ?hasSeats ?o "
        + "FROM NAMED WINDOW rsp4j:window ON :names [RANGE PT10M STEP PT1M] "
        + "WHERE {"
        + "  WINDOW rsp4j:window {?s ?hasSeats ?o ."
        + "  FILTER(?o >= 20 && ?o <= 30  ||  ?o >= 70 && ?o <= 80 )"
        + "}"
        );
```

Figure 21 - Query in showSeatsBetweenRange method

```
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "74"
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "74"
{s -> <http://rsp4j.io/parkingStream/names/BRUNNS>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "20"} @ 9
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "79"}
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "80"
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "71"}
```

Figure 22 - Results of the query in showSeatsBetweenRange method

Aggregate functions in RSP-QL (RDF Stream Processing Query Language) are used to perform calculations on sets of data within a stream. These functions include:

1. **COUNT**: Calculates the number of items in a dataset or stream.
2. **MAX**: Finds the maximum value in a dataset.
3. **MIN**: Determines the minimum value in a dataset.
4. **AVG**: Computes the average value of a numerical dataset.

These functions are essential for summarizing and analyzing continuous data streams in real time. For this reason, there is the query in method queryCountParkingName() where it is calculated how many parking with specific names there are in each window.

```
"PREFIX : <http://rsp4j.io/parkingStream/> "
+ "PREFIX rsp4j: <http://rsp4j.io/> "
+ "PREFIX names: <http://rsp4j.io/parkingStream/names/> "
+ "REGISTER RSTREAM <http://out/stream> AS "
+ "SELECT (COUNT(DISTINCT ?s) AS ?count) "
+ "FROM NAMED WINDOW rsp4j:window ON :names [RANGE PT10M STEP PT1M] "
+ "WHERE {"
+ " WINDOW rsp4j:window {?s ?hasSeats ?o .}"
+ " FILTER(?s = names:NORREPORT)"
+ "}");
```

Figure 23 - Query for calculating the instance of parking with the name NORREPORT

In the below image there are the results of this query, for better understanding the data from the RDF graph are also displayed. It is important to note that rsp-ql does not keep track of history from the passed streams so it will not show the final occurrences of the records for this restriction. In other words, if every window is parsed if the comparison is true then the count will be one (meaning that find one parking spot with name NORREPORT) otherwise the count will be zero.

```
<http://rsp4j.io/parkingStream/names/NORREPORT> <http://rsp4j.io/parkingStream/names/hasSeats> "32" .
{count -> <1>} @ 1500000
<http://rsp4j.io/parkingStream/names/SKOLEBAKKEN> <http://rsp4j.io/parkingStream/names/hasSeats> "15" .
{count -> <0>} @ 2400000
<http://rsp4j.io/parkingStream/names/SKOLEBAKKEN> <http://rsp4j.io/parkingStream/names/hasSeats> "27" .
{count -> <0>} @ 2700000
<http://rsp4j.io/parkingStream/names/NORREPORT> <http://rsp4j.io/parkingStream/names/hasSeats> "7" .
{count -> <0>} @ 4200000
<http://rsp4j.io/parkingStream/names/NORREPORT> <http://rsp4j.io/parkingStream/names/hasSeats> "95" .
{count -> <1>} @ 4800000
<http://rsp4j.io/parkingStream/names/MAGASIN> <http://rsp4j.io/parkingStream/names/hasSeats> "86" .
{count -> <1>} @ 5400000
<http://rsp4j.io/parkingStream/names/SKOLEBAKKEN> <http://rsp4j.io/parkingStream/names/hasSeats> "6" .
{count -> <0>} @ 5700000
```

Figure 24 - Results of Query for calculating the instance of parking with name NORREPORT

The next query also uses an aggregate function and more specifically the MIN function which calculates the minimum number of attributes. Particularly, the query finds which parking lot with the name NORREPORT has the least (minimum) seats.

```
"PREFIX : <http://rsp4j.io/parkingStream/> "
+ "PREFIX rsp4j: <http://rsp4j.io/> "
+ "PREFIX names: <http://rsp4j.io/parkingStream/names/> "
+ "REGISTER RSTREAM <http://out/stream> AS "
+ "SELECT ?s (MIN(DISTINCT ?o) as ?minSeats) "
+ "FROM NAMED WINDOW rsp4j:window ON :names [RANGE PT10M STEP PT1M] "
+ "WHERE {"
+ " WINDOW rsp4j:window {?s ?hasSeats ?o .}"
+ " FILTER(?s = names:NORREPORT)"
+ "}"
+ "ORDER BY ASC(?minSeats) "
+ "LIMIT 1"
);
```

Figure 25 - Query of finding the minimum seats of the parking lot named NORREPORT

Unfortunately, as the following image indicates there is a problem with function MIN (the same problem there is in the function MAX) that these functions are not found in the registry of the rsp4j library. However, the syntax and the logical approach of the query are correct.

```
<http://rsp4j.io/parkingStream/names/NORREPORT> <http://rsp4j.io/parkingStream/names/hasSeats> "57" .
[Thread-0] ERROR org.streamreasoning.rsp4j.operatorapi.ContinuousProgram  - Function MIN not found in Registry!
<http://rsp4j.io/parkingStream/names/NORREPORT> <http://rsp4j.io/parkingStream/names/hasSeats> "79" .
[Thread-0] ERROR org.streamreasoning.rsp4j.operatorapi.ContinuousProgram  - Function MIN not found in Registry!
<http://rsp4j.io/parkingStream/names/BRUNNS> <http://rsp4j.io/parkingStream/names/hasSeats> "0" .
[Thread-0] ERROR org.streamreasoning.rsp4j.operatorapi.ContinuousProgram  - Function MIN not found in Registry!
<http://rsp4j.io/parkingStream/names/BRUNNS> <http://rsp4j.io/parkingStream/names/hasSeats> "21" .
[Thread-0] ERROR org.streamreasoning.rsp4j.operatorapi.ContinuousProgram  - Function MIN not found in Registry!
```

Figure 26 - Results of queryMinSeats

Since rsp-ql is a query language like SPARQL it is possible to perform subqueries. A subquery, also known as an inner query or a nested query, is a query embedded within another query. The main purpose of a subquery is to provide a result set that can be used as a condition or a value in the outer query. This query, shown in the following image, retrieves triples where the predicate is ?hasSeats. Additionally, it performs a subquery to count the number of seats (?numSeats) for each subject (?s) and groups the results by ?s. The final output includes the subjects, predicates, and objects within the specified window.

```
"PREFIX : <http://rsp4j.io/parkingStream/>"
+ " PREFIX rsp4j: <http://rsp4j.io/> "
+ " PREFIX names: <http://rsp4j.io/parkingStream/names/> "
+ " REGISTER RSTREAM <http://out/stream> AS "
+ " SELECT ?s ?hasSeats ?o "
+ " FROM NAMED WINDOW rsp4j:window ON :names [RANGE PT10M STEP PT1M] "
+"  WHERE { "
+ " WINDOW rsp4j:window { "
+ " ?s ?hasSeats ?o ."
 +" { "
  + " SELECT ?s (COUNT(?hasSeats) AS ?numSeats) "
   +" WHERE { "
    +" ?s ?hasSeats ?o"
   + " } "
     +" GROUP BY ?s "
      + " } "
    +" } "
   +" }"
);
```

Figure 27 - Subquery in rsp-ql

```
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "21"}
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "59"}
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "21"}
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "73"} @
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "96"}
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "99"} @
{s -> <http://rsp4j.io/parkingStream/names/MAGASIN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "0"} @ 48
{s -> <http://rsp4j.io/parkingStream/names/MAGASIN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "93"} @ 5
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "61"}
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "37"} @
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "84"} @
{s -> <http://rsp4j.io/parkingStream/names/MAGASIN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "31"} @ 7
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "67"} @
```

Figure 28 - Results of Subquery in rsp-ql

It is common for the need to perform more complex comparisons and filtering in query languages like rsp-ql. For this purpose, there is the query in the complexFilter method where an arithmetic and comparison between strings is conducted. This query returns all the parking lots where the names are not NORREPORT and SKOLEBAKKEN or the total capacity of the seats is over 40 and less than 90.

```
"PREFIX : <http://rsp4j.io/parkingStream/>"
+ " PREFIX rsp4j: <http://rsp4j.io/> "
+ " PREFIX names: <http://rsp4j.io/parkingStream/names/> "
+ " REGISTER RSTREAM <http://out/stream> AS "
+ "SELECT ?s ?hasSeats ?o "
+ "FROM NAMED WINDOW rsp4j:window ON :names [RANGE PT10M STEP PT1M] "
+ "WHERE {"
+ "  WINDOW rsp4j:window {?s ?hasSeats ?o ."
+ "  FILTER(?s != names:SKOLEBAKKEN && ?s != names:NORREPORT ||  ?o >= 40 && ?o <= 90 ) "
+ "}"
);
```

Figure 29 - Complex filter in rsp-ql

```
{s -> <http://rsp4j.io/parkingStream/names/MAGASIN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "18"} @
{s -> <http://rsp4j.io/parkingStream/names/MAGASIN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "18"} @
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "50
{s -> <http://rsp4j.io/parkingStream/names/SKOLEBAKKEN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "50
{s -> <http://rsp4j.io/parkingStream/names/MAGASIN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "89"} @
{s -> <http://rsp4j.io/parkingStream/names/MAGASIN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "32"} @
{s -> <http://rsp4j.io/parkingStream/names/MAGASIN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "80"} @
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "45"}
{s -> <http://rsp4j.io/parkingStream/names/BRUNNS>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "52"} @
{s -> <http://rsp4j.io/parkingStream/names/MAGASIN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "65"} @
{s -> <http://rsp4j.io/parkingStream/names/MAGASIN>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "36"} @
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "43"}
{s -> <http://rsp4j.io/parkingStream/names/NORREPORT>, hasSeats -> <http://rsp4j.io/parkingStream/names/hasSeats>, o -> "43"}
```

Figure 30 - Results of Complex filter in rsp-ql

The last query tests if an RDF literal is expressed in the schema efficiently. Many times there is the need to express an element (for example the predicate) as an RDF property (rdf:type), which is used to state that a resource is an instance of a class. The query in the figure does this as the variable named “a” is an RDF property that has been defined in the StreamGenerator file. The purpose of this query is just to ensure that if there is such kind of data no errors will be generated and for this reason, there is no point for the results to be shown.

```
"PREFIX : <http://rsp4j.io/parkingStream/>"
+ " PREFIX rsp4j: <http://rsp4j.io/> "
+ " PREFIX names: <http://rsp4j.io/parkingStream/names/> "
+ " REGISTER RSTREAM <http://out/stream> AS "
+ "SELECT ?s  ?hasSeats ?o "
+ "FROM NAMED WINDOW rsp4j:window ON :names [RANGE PT10M STEP PT1M] "
+ " WHERE { "
+ "  WINDOW rsp4j:window {?s ?hasSeats ?o ."
 + " ?s a <http://example.org/ontology#parkingname> ."
+ "}"
);
```

Figure 31 - Reasoning Query

5. Conclusion & Future Work

5.1 Conclusion

RDF (Resource Description Framework) stream processing is crucial because it allows for real-time analysis and reasoning over continuous data streams, which is essential in today's data-driven world. By enabling the immediate processing and integration of dynamic information, RDF stream processing supports timely insights and decision-making for applications such as monitoring systems, financial trading, and social media analytics. It also enhances interoperability and integration across diverse data sources by adhering to semantic web standards, facilitating the fusion of heterogeneous data in a coherent manner.

RDF benchmarks are essential tools for evaluating the performance and scalability of RDF data management systems. They provide standardized datasets and queries to assess various aspects such as data ingestion, query processing, and reasoning capabilities. Notable benchmarks include CityBench, which focuses on smart city applications and evaluates the processing of urban data streams; SRBench (Sensor RDF Benchmark), designed for benchmarking RDF stream processing engines using sensor data; and LSBench (Linked Stream Benchmark), which tests systems with linked data streams in social media contexts.

Based on all the above, the idea of this master thesis is to evaluate this benchmark in terms of the choke points against the other benchmarks that have already been developed by other scientists. For this need the below table presents a comparison of the choke points that are identified in the various benchmarks - the numbers represent the time a specific choke point is evaluated in the benchmark.

As shown, optional clauses (CP3) and reasoning (CP4) are neglected by most of the benchmarks and only SRBench considers them. Most of them consider join ordering (CP1), aggregation (CP2), filters (CP6), geospatial predicates (CP8), and complex filters (CP11), where only two of them consider ordering of the results (CP7). Finally, all benchmarks ignore parallel execution of joins (CP5), full-text search (CP9), and duplicate elimination (CP10), paving the way for future benchmarks in the domain.

	CP1	CP2	CP3	CP4	CP5	CP6	CP7	CP8	CP9	CP10	CP11
LSBench	7	2	0	0	0	2	1	0	0	0	1
SRBench	3	4	2	3	0	7	0	6	0	0	0
CSRBench	0	1	0	0	0	6	0	2	0	0	1
CityBench	5	3	0	0	0	2	2	6	0	0	1
YaBench	0	1	0	0	0	6	0	2	0	0	1

Figure 32 - Choke points and benchmarks

The next figure includes the benchmarks of this work.

	CP1	CP2	CP3	CP4	CP5	CP6	CP7	CP8	CP9	CP10	CP11
Paper's Benchmark	0	3	1	1	0	9	1	0	4	2	1

Figure 33 - Choke points and paper's benchmark

Choke points of the various benchmarks (CP1. Join Ordering, CP2. Aggregation, CP3. Optional & nested optional clauses, CP4. Reasoning, CP5: Parallel execution of joins, CP6: Filters, CP7. Ordering, CP8. Geo-spatial predicates, CP9. Full-text, CP10: Duplicate elimination, CP11: Complex filter conditions).

Choke point 8 is not satisfied because there is no need to use such kind of data (latitude and longitude) for the needs of the paper. Choke points 1 and 5 (Join ordering and Parallel execution of joins) can not be satisfied because there is no way to identify how the engine and the query optimizer implement the joins in the queries so it can not be evaluated.

5.2 Future Work

In the realm of RDF stream processing (RSP) systems, several challenges and opportunities for future work stand out. A notable gap exists in the absence of a strong community-based standardization body, leading to fragmentation in query language adoption. Many systems still rely on proprietary languages, and while standards like C-SPARQL and CQELS have gained some traction, their adoption remains limited. However, the introduction of RSP-QL has sparked a shift, with more systems starting to adopt it, signaling a move toward greater standardization in the field.

In terms of system capabilities, one area that requires significant attention is incremental updates. Only a few RSP systems currently handle incremental updates effectively, highlighting an opportunity for further research and development. Furthermore, many existing systems separate query evaluation from reasoning, relying on distinct components for each task. Unifying these processes would lead to more adaptive and efficient systems. Reasoning support is another underdeveloped aspect; the few systems that offer it often provide only

basic capabilities. Expanding reasoning expressiveness and implementing more advanced entailment mechanisms beyond simple graph-based approaches represent promising research directions. Additionally, the reliability of ingested data, along with conflict detection and resolution policies, remains largely unaddressed in the domain.

The exploration of distributed systems within RSP is still in its early stages, with only a handful of solutions proposed. This area demands more research, particularly in balancing scalability and reasoning expressiveness. For instance, while CQELS Cloud addresses cloud computing, the field remains underexplored. Notably, major RDF database systems like Stardog, GraphDB, and Virtuoso have yet to support continuous query languages, creating an opening for future commercial development.

Benchmarking is another critical area that requires improvement. Current benchmarks lack robustness, particularly in evaluating parallel join execution, full-text search, and duplicate elimination. Developing better and more comprehensive benchmarks is essential to advancing the RSP domain and ensuring that new systems can be effectively evaluated and compared.

References

[1] Some terminology about static and data stream is based to some degree from Hazelcast company https://hazelcast.com/glossary/streaming-data/

[2] Chen Weidong, Pan Yunhe **"**Survey of the study on frequent pattern mining in data streams"
https://www.researchgate.net/publication/224756690_Survey_of_the_study_on_frequent_pattern_mining_in_data_streams

[3] Keshab Nath, Sourish Dhar, Subhash Basishtha "Web 1.0 to Web 3.0 - Evolution of the Web and its Various Challenges"
https://www.researchgate.net/publication/269310255_Web_10_to_Web_30_-_Evolution_of_the_Web_and_its_various_challenges

[4] Miss. Komal U. Dhulekar , Miss. Madhuri P. Devrankar "A REVIEW ON SEMANTIC WEB"
https://www.granthaalayahpublication.org/ijetmr-ojms/index.php/ijetmr

[5] Grigoris Antoniou, Frank van Harmelen "A Semantic Web Primer"
https://www.csd.uoc.gr/~hy566/SWbook.pdf

[6] RDF - Semantic Web by W3C Organization
https://www.w3.org/RDF/

[7] Brian McBride Hewlett Packard Laboratories, Bristol, UK. Chapter 3 from paper The Resource Description Framework (RDF) and its Vocabulary Description Language RDFS
https://link.springer.com/chapter/10.1007/978-3-540-24750-0_3

[8] TripleWave: Spreading RDF Streams on the Web
https://www.zora.uzh.ch/id/eprint/132903/1/main.pdf

[9] Towards a Unified Language for RDF Stream Query Processing by Daniele Dell'Aglio, Jean-Paul Calbimonte, Emanuele Della Valle & Oscar Corcho

https://link.springer.com/chapter/10.1007/978-3-319-25639-9_48

[10] Reactive Processing of RDF Streams of Events "Jean-Paul Calbimonte" "Karl Aberer" https://www.researchgate.net/publication/314620579_Reactive_Processing_of_RDF_Streams_of_Events

[11] RSP-QL Semantics: A Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems by Daniele Dell'Aglio, Jean-Paul Calbimonte, Emanuele Della Valle & Oscar Corcho
https://www.researchgate.net/publication/277942246_RSP-QL_Semantics_a_Unifying_Query_Model_to_Explain_Heterogeneity_of_RDF_Stream_Processing_Systems

[12] Terminology and general content about Benchmarks
https://en.wikipedia.org/wiki/Benchmark_(computing)

[13] On Correctness in RDF Stream Processor Benchmarking Daniele Dell'Aglio,Jean-Paul Calbimonte, Marco Balduini, Oscar Corcho
https://www.researchgate.net/publication/257030218_On_Correctness_in_RDF_Stream_Processor_Benchmarking

[14] Web Stream Processing Systems and Benchmarks Chapter 5 Riccardo Tommasini, Pieter Bonte, Fabiano Spiga & Emanuele Della Valle
https://link.springer.com/chapter/10.1007/978-3-031-15371-6_5

[15] RSPLab: RDF Stream Processing Benchmarking Made Easy, Chapter 4 Related Work, Riccardo Tommasini, Emanuele Della Valle, Andrea Mauri & Marco Brambilla
https://link.springer.com/chapter/10.1007/978-3-319-68204-4_21#Sec4

[16] Towards a Benchmark for Expressive Stream Reasoning Riccardo Tommasini, Marco Balduini, Emanuele Della Valle

https://ceur-ws.org/Vol-1870/paper-03.pdf

[17] A Survey on Systems, Languages, and Benchmarks for Streaming Linked Data

Pieter Bonte · Christophe Call´e · Olivier Cur´e · Haridimos Kondylakis ·Riccardo Tommasini

[18] LDBC: benchmarks for graph and RDF data management Peter Boncz https://ir.cwi.nl/pub/22644/22644B.pdf

[19] General Information about Java https://en.wikipedia.org/wiki/Java_(programming_language)

[20] Web stream processing with RSP4J Riccardo Tommasini, Pieter Bonte https://www.researchgate.net/publication/352796292_Web_stream_processing_with_RSP4J

[21] RSP4J: An API for RDF Stream Processing Riccardo Tommasini, Pieter Bonte, Femke Ongenae & Emanuele Della Valle https://link.springer.com/chapter/10.1007/978-3-030-77385-4_34