Politecnico Di Milano

Department of Electronic, Informatics and Bioengineering

School of Industrial and Information Engineering

An infrastructural view of Cascading Stream Reasoning using micro-services

Supervisor: Prof. Emanuele Della Valle

Co-Supervisor: Riccardo Tommasini

Author: Philippe Scorsolini

Personal Number: 878664

This dissertation is submitted for the degree of

Master of Science

April 2019

Abstract

In the Web of Data, in which Data are increasing in Volume, Variety, and Velocity, the Stream Reasoning and RDF Stream Processing research areas try to make sense of these continuous streams of schemaless data, by taking advantage of both the vast Stream Processing literature and Semantic Web technologies at the same time. State-of-the-art solutions proposed so far mostly proved that taming Variety and Velocity simultaneously is feasible.

In this Thesis, we propose μCR, an infrastructural redesign based on micro-services of two existing approaches, i.e., Cascading Reasoning (CR) and Network of Stream Reasoners (NoSR). Our proposal, taking inspiration from the Unix philosophy, merges the two existing approaches, producing a network of simple reasoners, each with its reasoning expressiveness, that can be used as rewriting target for continuous queries. To enable this approach, we present a Stack and propose our solutions for its foundational layers.

Finally, to produce a Proof of Concept of a μCR enabled stream reasoner, we focus on rewriting directly to Raw Stream Processing systems based on the Dataflow execution model. In fact, accordingly Stonebraker's principles, we recognize as a necessity the ability to perform little expressive reasoning on fast changing data in an horizontally scalable way.Therefore, we present Metamorphosis, i.e., a proof-of-concept implementation for RDF Stream Processing over Kafka Streams using the Dual Streaming model. We then prove the feasibility of our approach using SRBench benchmark for query expressiveness.

Estratto

Nel Web of Data di oggi, per il quale gli aspetti di "Volume", "Variety" e "Velocity" sono sempre più di crescente importanza, Stream Reasoning e RDF Stream Processing si propongono come campi di ricerca volti al dare un significato a questo continuo stream di dati, utilizzando al contempo gli strumenti dello Stream processing e le tecnologie del Semantic Web. Le soluzioni proposte finora si sono concentrate principalmente sulla fattibilità di gestire contemporeamente "Variety" e "Velocity" dei dati.

In questa tesi proponiamo μCR, una reinterpretazione infrastrutturale basata su micro servizi di due approcci già proposti in letteratura, "Cascading Reasoning" (CR) e "Network of Stream Reasoners" (NoSR). La nostra proposta, ispirata alla filosofia Unix, unisce i due approcci sopra citati, producendo un network di "reasoners" semplici, ognuno caratterizzato da una specifica espressività di ragionamento, che possono essere utilizzati come obbiettivo per la riscrittura di query continue. Al fine di attuare questo approccio, abbiamo proposto uno "Stack" e delle soluzioni per i suoi livelli fondamentali.

Infine, per produrre un esempio di Stream Reasoner abilitato ad interagire all'interno della rete μCR, ci siamo concentrati sulla riscrittura verso sistem di "Raw Stream Processing" basati sul modello di esecuzione Dataflow. Infatti, seguendo i principi proposti da Stonebraker, riconosciamo come una necessità l'essere capaci di eseguire ragionamenti poco espressivi su dati in rapido cambiamento scalando orizzontalmente, per poter gestire anche la componente "Volume" nei dati. Per questi motivi, presentiamo Metamorphosis, un esempio di implementazione di RDF Stream Processing utilizzando "Kafka Streams" e il "Dual Streaming model". Quindi, proviamo la fattibilità del nostro approccio testando l'espressività delle query implementabili grazie al benchmark SRBench.

Table of contents

List of figures

List of listings

Chapter 1

Introduction

1.1 Motivations

In the Web of Data, in which Data are increasing in Volume, Variety, and Velocity, the RDF Stream Processing research area tries to make sense of these continuous streams of schemaless data, by taking advantage of the vast Stream Processing literature and thanks to Semantic Web technologies and techniques. State-of-the-art solutions proposed so far mostly proved that taming Variety and Velocity simultaneously is feasible, but few have addressed the Volume aspect through horizontal scaleability.

Therefore, in this thesis, we propose an infrastructural redesign of two existing approaches for Stream Reasoning/RDF Stream Processing, i.e., Cascading Reasoning and Network of Stream Reasoners [31], in order to be able to address also Volume. Our proposed approach will be based on micro-services. In particular, we will focus onto targeting Big Data Stream Processing systems based on the Dataflow model.

1.2 Research questions

The general research question the RSP and SR fields try to answer focus on taming Velocity on the Web of Data without neglecting Variety, trying to explore the tradeoffs between these two aspects. In this general framework, we try to focus on trying to tame also the Volume aspect by applying the Cascading Reasoning approach to the Network of Stream Reasoners, trying to overcome the limitations of the former with the strengths of the latter. In the context of these questions, we will focus mostly on the fundamental choices necessary to enable such an approach. Therefore, exploring the space of possible solutions for what we will call Streaming Platform and Orchestrator, to build the underlying infrastructure for our vision.

1.3 Contributions

We contribute a vision, that of μCR, which consists of a novel framework combining Cascading Reasoning (CR) and Network of Stream Reasoners (NOSR) by Stuckenschmidt et al. [37], that could enable rewriting streaming queries against a microservice architecture of differently capable Stream Processors. Therefore we propose a Stack of choices that have to be made. We then frame some of the existing solutions through this stack and show what they are missing in order to be able to fit into our framework. We tackle at first the choice for a common Data Infrastructure and then focus on the choice of a Stream Processing system and model for a proof of concept of a μCR enabled reasoner, Metamorphosis, which constitutes our second contribution. We have chosen to target a system built upon the Dataflow model, recognizing it as a necessary component to achieve our vision and to be able to tackle also the Volume dimension of the Web of Data. To do so, we propose a model to represent and process RDF streams using a Dataflow system. Finally, we choose a State of the Art orchestrator and deploy the Streaming Platform on it and as future work we outline how it could be instrumented to handle the μCR Network of Stream Reasoners completely.

1.4 Outline of the thesis

This thesis develops in the following chapters:

1 **Background** We provide an overview of the main research areas related to this thesis. We introduce the Semantic Web technologies stack and research field of RDF Stream Processing, presenting some of the existing systems. We cover the CQL model over which many of the existing systems have been based. Then we explore the state of the art of streaming query rewriting. And go on outlining the Stream Processing research field, its core principles and requirements, some of the modern stream processing engines, giving a more in depth description of Apache Kafka and its Dual Streaming model. We end with a description of the Dataflow processing model.

2 **Problem Statement** We introduce the context for our research, Velocity on the web, reducing then our scope to that of Expressive Stream Reasoning and presenting our μCR vision as a Cascading Reasoning system over a Network of Stream Reasoners

and propose a stack that have to be covered to achieve our vision and focus our research on the lower layers of the stack.

3 **Design** We present the choices we had to make to enable our vision, explaining the motivations that brought us to choose a PubSub system as Data Infrastructure and a Dataflow model to implement our proof of concept of a μCR enabled reasoner targeting Raw Stream Processing through query rewriting directly to its DSL. We then propose a model to bridge the gap between the Dual Streaming model and RDF data.

4 **Implementations** We present the implementation of the choices done in the previous chapter, the choice of a specific PubSub system, the technologies used to allow efficient communication and our proposed proof of concept for a μCR enabled Dataflow reasoner. We propose an algorithm to translate RSP-QL queries into logical plans executable as a specific Dataflow system topologies. Then we show how we deployed our solution in a state of the art orchestration system. We then present an evaluation of the feasibility of our approach by checking the compatibility of our system with queries in the SRBench benchmark[39].

Chapter 2

Background

2.1 Semantic Web

The *Semantic Web* as first presented by Berners Lee et al. [8] is a research area enabling data interoperability in the World Wide Web. Semantic technologies allow to serve and consume data with clear semantics and, therefore, understandable also to automated agents. The *World Wide Web Consortium* (W3C) proposed the Semantic Web stack, i.e., a framework that organizes semantic technologies in layers (Figure 2.1).

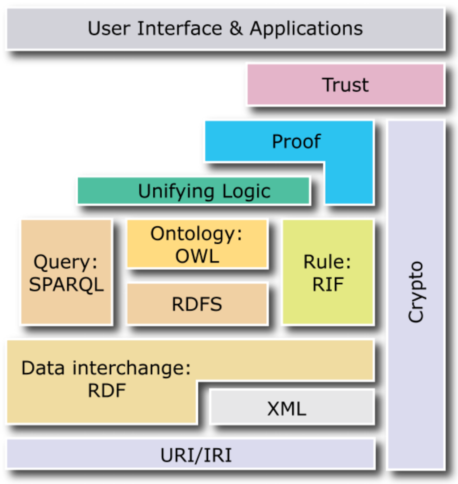


Fig. 2.1 The *semantic web* stack.

In the following, we give details about those technologies that were more relevant for this thesis, i.e., RDF, SPARQL, and OWL.

The *Resource Description Framework* (RDF) is a graph data model to publish semantically enriched information on the Web. RDF combined with the *Web Ontology Language* (OWL) can be used to allow *reasoning* on these data.

2.1.0.1 Resource Description Framework (RDF)

RDF is the W3C specification for data interchange and information representation for the *Semantic Web* [19]. Information is organised in sets of statements in the form of **(subject,predicate,object)** triples, formally:

Definition 1. An RDF statement d is a triple

$$(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$$

A set of RDF statements is an RDF graph.

Triples' elements can be IRIs, Blank nodes or Literals: (i) *IRI*s (Internationalized Resource Identifiers), string uniquely identifying a generic resource, allowing to enable equality checking among nodes of different RDF graphs. (ii) *Blank nodes*, representing anonymous resources, disjoint from IRIs and literals. (iii) *Literals*, representing values through strings associated with an IRI identifying their datatype.

RDF is a data model, therefore multiple possible representations and serialization formats have been presented. The main syntax for RDF models standardized from W3C is RDF/XML[1], which defines an XML syntax for encoding RDF. A recently proposed format is JSON-LD[2], a JSON based serialization format allowing to transparently exploit the RDF format in systems already using JSON.

RDF Schema (RDFS)[3] is a vocabulary to model RDF data, standardized by the W3C, providing a way to describe groups of related resources and their relationships.

2.1.0.2 Web Ontology Language

The Web Ontology Language (OWL) is the ontology language for the *Semantic Web*, standardized by the W3C in 2004, updated in 2012 to OWL2[4]. It allows to represent ontologies, which are explicit representations of concepts, modeling domain-specific knowledge.

[1]https://www.w3.org/TR/rdf-syntax-grammar/
[2]https://w3c.github.io/json-ld-syntax/
[3]RDFS https://www.w3.org/TR/rdf-schema/
[4] https://www.w3.org/TR/owl2-overview/

OWL builds on the RDF Schema Language, providing classes, properties, individuals, and data values. Ontologies itself can be serialized in RDF graphs using the RDF/XML syntax.

OWL enables *reasoning*, i.e., the possibility to infer implicit knowledge from asserted axioms relying on the theoretic semantics of *Description Logics*. Both OWL and OWL2 are in the worst case highly intractable, for this reason OWL2 defines three different tractable **profiles**[5], language subsets with useful computational properties. These are **OWL EL**, **OWL QL** and **OWL RL**.

OWL 2 EL, was designed for applications employing ontologies with a very large numbers of properties and/or classes. It is a subset of OWL 2 that allows to perform basic reasoning in polynomial time w.r.t. the size of the ontology. EL reflects the profile's basis in the *EL* family of Description Logics, providing only Existential quantification.

OWL 2 QL, instead, was aimed at applications that use very large volumes of instance data, and where the most important reasoning task is query answering. Conjunctive query (CQ) answering can be implemented using conventional RDBMS systems. Therefore, sound and complete CQ answering can be performed in LOGSPACE w.r.t. the size of the data. The QL acronym reflects the fact that query answering can be implemented through query rewriting into standard relational Query Languages.

Finally, OWL 2 RL, was designed for applications requiring scalable reasoning without sacrificing too much expressive power. Reasoning systems using this profile can be implemented using rule-based reasoning engines. Ontology consistency, class expression satisfiability, class expression subsumption, instance checking, and conjunctive query answering problems can be solved in polynomial time w.r.t. the ontology size.

2.1.0.3 SPARQL

SPARQL is the W3C standard[6] RDF query language. SPARQL is a graph-matching query language, therefore it allows to express queries through the specification of graph patterns an algebra operators to combine them.

SPARQL offers four different types of queries, each of which can have a WHERE block to specify graph-matching patterns: (i) **SELECT** queries return a set of variables and their possible values in the input graph. (ii) **CONSTRUCT** queries allow returning multiple RDF graphs created through templating directly from query results. (iii) **ASK** queries return a boolean value signifying whether or not the query pattern has a solution. (iv) **DESCRIBE** queries allow obtaining an RDF graph containing RDF data about the retrieved resources.

[5]OWL2 Profiles http://www.w3.org/TR/owl2-profiles/

[6]SPARQL https://www.w3.org/TR/rdf-sparql-query/

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
SELECT ?name ?email
FROM <http://www.w3.org/People/Berners-Lee/card>
WHERE {
    ?person foaf:name ?name .
    OPTIONAL { ?person foaf:mbox ?email }
} ORDER BY ?name LIMIT 10 OFFSET 10
```

Listing 2.1 Example SPARQL query.

SELECT queries are composed of the following clauses:

1. *PREFIX*: allows associating a prefix label to an IRI within the query, to ease user experience.

2. *SELECT*: specifies variables to be returned and their formats, like in the SQL counterpart.

3. *FROM*: allows specifying the source RDF dataset over.

4. *WHERE*: provides the graph pattern, which can be of various types [7] to be matched against the source data graphs specified in FROM clauses.

5. *Solution modifiers* like *ORDER BY*, *LIMIT*,... allows to modify the results of the query, much like their SQL counterparts.

SPARQL 1.1 extends the 1.0 version with additional features, like aggregations and subqueries. SPARQL queries can take as input explicitly given graphs or can work under some **entailment regime**[8], therefore taking into account also inferable RDF statements given an *entailment relation* (e.g., RDF entailment, RDFS entailment).

Given an RDF graph, SPARQL queries typically contain one or more triple patterns, called a Basic Graph Pattern (BGP). Triple patterns are similar to RDF triples, but they may contain variables in place of resources.

Definition 2. A triple pattern t_p is a triple (sp, pp, op) s.t.

$$(sp, pp, op) \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$$

where V is the infinite set of variables. Therefore, a BGP is a set of triple patterns.

[7] simple graph patterns, group patterns, optional patterns, ...
[8] Entailment Regimes https://www.w3.org/TR/sparql11-entailment/

BGPs in a SPARQL query can include other compound patterns defined by different algebraic operators, such as OPTIONAL, UNION and FILTER.

The semantic of evaluation of a SPARQL query is based on the notion of solution mappings.

Definition 3. A solution mapping μ is a partial function

$$\mu : V \rightarrow I \cup B \cup L$$

from a set of Variables V to a set of RDF terms.

Given an RDF graph, a SPARQL query solution can be represented as a set of solution mappings, each assigning terms of RDF triples in the graph to variables of the query. SPARQL operators therefore are defined over mapping.

Definition 4. Defined ω_1 and ω_2 as multisets of solution mappings. A JOIN is defined as follows:

$$Join(\omega_1, \omega_2) = \{merge(\mu_1, \mu_2) \mid \mu_1 \in \omega_1 \wedge \mu_2 \in \omega_2 \wedge \mu_1 \text{ and } \mu_2 \text{ are compatible}\}$$

Where mappings' compatibility is defined as follows:

Definition 5. μ_1 and μ_2 are compatible iff:

$$\forall x (\in dom(\mu_1) \cap dom(\mu_2) \Rightarrow \mu_1(x) = \mu_2(x))$$

Moreover, SPARQL queries operate over collections of one or more RDF graphs, named RDF datasets.

Definition 6. An RDF Dataset DS is a set:

$$DS = \{g_0, (u_1, g_1), (u_2, g_2), \ldots, (u_n, g_n)\}$$

where g_0 and g_i are RDF graphs, and each corresponding u_i is a distinct IRI. g_0 is called the default graph, while the others are called named graphs. During the evaluation of a query, the graph from the dataset used for matching the graph pattern is called *Active Graph*. Multiple graphs can become active during the evaluation.

Finally, SPARQL defines four Query Forms, i.e. ASK, SELECT, CONSTRUCT, and DESCRIBE, and other constructs such as solution modifiers, e.g. DISTINCT, ORDER BY, LIMIT, applied after pattern matching.

Definition 7. A SPARQL query is defined a tuple (E, DS, QS) where E is a SPARQL algebra expression, DS an RDF dataset and QF a query form.

The evaluation sematics of a SPARQL query algebra expression w.r.t. an RDF dataset is defined for every operator of the algebr as $eval(DS(g), E)$ where E denotes an algebra expression and $DS(g)$ a dataset DS with active graph g.

2.1.0.4 Reasoning

Reasoners, given an RDF graph and an ontology, are able to infer the implicit knowledge in the data.

The main problem of reasoning is its computational cost. Reasoning procedures, if too expressive, can become computationally too expensive or even undecidable and, therefore, have been applied initially only to static data. However, balancing expressiveness, e.g. choosing appropriate an OWL2 profile, and designing specific tools, performances can be optimized, allowing to apply *reasoning* also on dynamic data.

For a given ontology, a number of reasoning tasks can be performed:

1. consistency checking: Checking whether a specific ontology has any model, i.e. its axioms are not contradictory.
2. instance checking: checking whether an individual belongs to a specific class.
3. subsumption checking: checking wether a class is subclass of another one.
4. class satisfiability checking: checking whether a class can have instances.

2.2 Stream Processing

2.2.1 The 8 Requirements of Real-Time Stream Processing

In [36], Stonebraker et al., defined the following 8 requirements that a stream processing system should satisfy:

R1 Keep the Data Moving
process messages “in-stream”, without any requirement to store them to perform any operation or sequence of operations. Ideally the system should also use an active (i.e., non-polling) processing model.

R2 Query using SQL on Streams (StreamSQL)
support a high-level "StreamSQL" language with built-in extensible stream-oriented primitives and operators.

R3 Handle Stream Imperfections (Delayed, Missing and Out-of-Order Data)
have built-in mechanisms to provide resiliency against stream "imperfections", including missing and out-of-order data, which are commonly present in real-world data stream.

R4 Generate Predictable Outcomes
must guarantee predictable and repeatable outcomes.

R5 Integrate Stored and Streaming Data
have the capability to efficiently store, access, and modify state information, and combine it with live streaming data. For seamless integration, the system should use a uniform language when dealing with either type of data.

R6 Guarantee Data Safety and Availability
ensure that the applications are up and available, and the integrity of the data maintained at all times, despite failure.

R7 Partition and Scale Applications Automatically
have the capability to distribute processing across multiple processors and machines to achieve incremental scaleability. Ideally, the distribution should be automatic and transparent.

R8 Process and Respond Instantaneously
must have a highly-optimized, minimal-overhead execution engine to deliver real-time response for high-volume applications.

Moreover, they classified the software technologies for stream processing, distinguishing three main classes of that can potentially be applied to solve high-volume low-latency streaming problems: Database Management Systems (DBMSs): widely used for storing large data sets and processing human-initiated queries. Their in main-memory variant can provide higher performance than traditional DBMSs by avoiding the disk for most operations, given sufficient main memory. Rule Engines: systems like the early 1970's PLANNER and Conniver or later Prolog. Typically accepts condition/action pairs, watches an input stream for any conditions of interest, and then takes appropriate action when the condition is met. Stream processing engines (SPEs): specifically designed to deal with streaming data. They perform SQL-like processing on the incoming messages as they fly by, without necessarily storing them. Comparing them w.r.t. the previously listed requirements R0-8:

2.2.2 CQL

Arasu et al. in [3] presented CQL, a continuous query language, in the form of an expressive SQL-based language for registering continuous queries against streams and stored relations. They argued that stream processing queries execution semantics in previous works was usually left unclear and therefore they try to give it a precise abstract semantics for continuous queries, based on two data types, stream and relations, and three classes of operators: Relation-to-stream (R2S), stream-to-relation (S2R) and relation-to-relation (R2R) operators. They give the folowing definitions for the two data types:

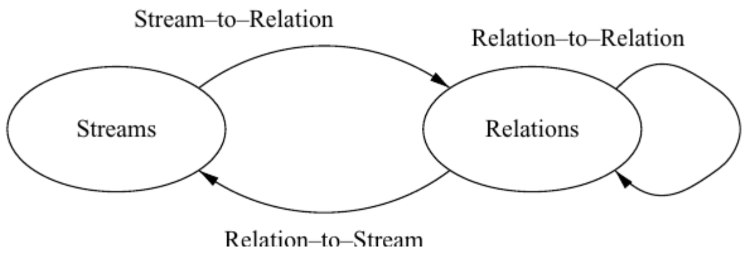


Fig. 2.2 CQL's interaction between streams and relations.

1. **Stream**: A stream S is a (possibly infinite) bag (multiset) of elements $\langle s, \tau \rangle$, where s is a tuple belonging to the schema of S and $\tau \in T$ is the timestamp of the element.
2. **Relation**: A relation R is a mapping from each time instant in T to a finite but unbounded bag of tuples belonging to the schema of R.

They defines as *instantaneous relation* the bag of tuples in a relation at a given point in time, therefore given relation R, R(τ) denotes an instantaneous relation. Given the semantics of the three class of operators they define the continuous semantics of a query Q, as a composition of any type-consistent composition of operators and a set of inputs streams and relations, whose result is then computed at time τ, assuming all inputs up to τ are available. They split two cases w.r.t. the outermost operator in Q:

1. It is a R2S operator, therefore the outcome will be a stream S.
2. It is a S2R or R2R operator, therefore the outcome will be a relation R.

In both cases the outcome will be the result of recursively applying the operators in Q to the input streams and relations up to τ.

They then present all the S2R, R2R and R2S operators available in CQL. R2R ones are the usual relational operators adapted to handle time-varying relations. S2R are all based on the concept of a sliding window over a stream, divided in three subclasses: *time-based*, *tuple-based* and *partitioned*. Time-based and tuple-based sliding windows are windows defined respectively on its time duration and number of tuples in the window, by default tumbling windows are assumed, but sliding windows can be specified through the addition of a sliding parameter in both cases. Partitioned windows instead behaves similarly to the SQL GROUP BY construct, it takes a stream S as input and a subset of S's attributes as parameters. It splits the input stream into different substreams based on the equality of requested attributes and then can be composed with tuple or time-based windows which will be applied to all the substreams generated.

R2S operators are the following:

1. **Istream** ("insert stream"): streaming out all new entries w.r.t. the previous instant.
2. **Dstream** ("delete stream"): streaming out all deleted entries w.r.t the previous instant.
3. **Rstream** ("relation stream"): streaming out all elements at a certain instant in the source relation.

Istream and Dstream could be derived from Rstream along with time-based sliding windows and some relational operator, but they preferred keeping them as syntactic sugar to facilitate writing queries.

In the STREAM prototype they proposed streams' imperfections were handled through an additional "metainput" heartbeat stream, consisting in a stream of timestamps signifying that the system won't be receiving no elements with timestamps previous to the heartbeat's one. Offering multiple possible implementations for heartbeats' streams.

2.2.3 Modern Stream Processing Engines

We here present a list of modern stream processing engines:

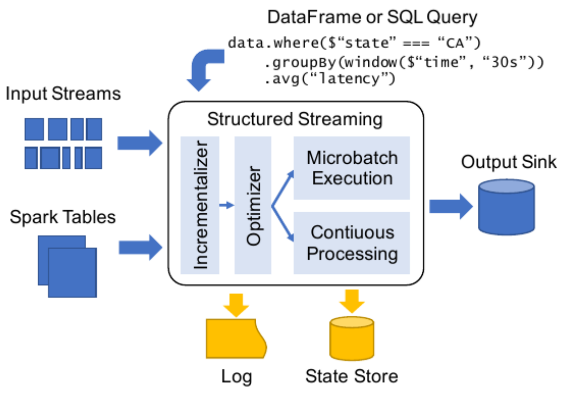
2.2.3.1 Apache Flink

Apache Flink [17] is an open source platform for distributed stream (DataSet API) and batch (DataStream API) data processing. Flink's core is a streaming dataflow engine that provides data distribution, communication, and fault tolerance for distributed computations over data streams. Both the API for stream and batch processing are available in Java and Scala. Apache Flink features two relational APIs - the Table API and SQL - for unified stream and batch processing. The Table API is a language-integrated query API for Scala and Java that allows the composition of queries from relational operators such as selection, filter, and join in a very intuitive way. Flink's Streaming SQL support is based on Apache Calcite which implements the SQL standard. Queries specified in either interface have the same semantics and specify the same result regardless whether the input is a batch input (DataSet) or a stream input (DataStream). Has to be said that Table and SQL API are still in beta at the best of our knowledge.When executed, Flink programs are mapped to streaming dataflows, consisting of streams and transformation operators. Each dataflow starts with one or more sources and ends in one or more sinks. The dataflows resemble arbitrary directed acyclic graphs (DAGs). This allow to exploit parallelization through stream partitioning when possible. Flink is growing in adoption and the Table and SQL API should become stable soon.

2.2.3.2 Apache Spark Streaming

Apache Spark [4] is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming. At a high level, every Spark application consists of a driver program that runs the user's main function and executes various parallel operations on a cluster. The main abstraction Spark provides is a resilient distributed dataset (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. Spark SQL is a Spark module for structured data processing. Unlike the basic Spark RDD API, the interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed and so the engine can perform extra optimizations. There are several ways to interact with Spark SQL including SQL and the Dataset API. When computing a result the same execution engine is used, independent of which API/language you are using to express the computation. This unification means that developers can easily switch back and forth between different APIs based on which provides the most natural way to express a given transformation. When running SQL from within another programming language the results will be returned as a Dataset/DataFrame. Recently Structured Streaming have been intruduced, it is a scalable and fault-tolerant stream processing engine built on the

Spark SQL engine. Streaming and batch computation can be expressed in the same way. Then the Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive. Dataset/DataFrame API are available in Scala, Java, Python and R. Finally, the system ensures end-to-end exactly-once fault-tolerance guarantees through checkpointing and Write Ahead Logs. Spark is one the most used batch processing systems nowadays and thanks to the increasing reliability and number of features added with new releases it's becoming also really used to perform streaming jobs. Spark Streaming leverages the spark infrastructure to perform streaming computation, it offers a purely declarative API based on automatically incrementalizing a static relational query (expressed using SQL or DataFrames). The engine runs in a microbatch execution mode by default but it can also use a low-latency continuous operators for some queries because the API is agnostic to execution strategy.



```
data.where($“state” === “CA”)
    .groupBy(window($“time”, “30s”))
    .avg(“latency”)
```

Fig. 2.3 The components of Structured Streaming [4]

2.2.3.3 Apache Apex

Apache Apex [9] is an enterprise-grade streaming technology which processes data at a very high rate with low latency in a fault tolerant way while providing processing guarantees like exactly-once. Apex comes with Malhar, a library of operators for data sources and destinations of popular message buses, file systems, and databases. Recently thanks to Apache Calcite, which parses SQL and converts SQL Node Tree to Relational Algebra, optimized and used to create a processing pipeline in Apex. This pipeline of relational algebra is converted by Apache Apex engine to its set of operators to perform business logic on data-in-motion.

[9]https://apex.apache.org/

2.2.3.4 Apache Beam

Apache Beam [27] is an open source, unified model for defining both batch and streaming data-parallel processing pipelines. Using one of the available Beam SDKs a pipeline can be defined and then executed by one of Beam's supported distributed processing back-ends, which include Apache Apex, Apache Flink, Apache Spark, Apache Gearpump and Google Cloud Dataflow or locally for testing purposes.

The Beam SDKs provide a unified programming model, compliant with the Dataflow model proposed by Akidau et al. [1], that can represent and transform data sets of any size, whether the input is a finite data set from a batch data source, or an infinite data set from a streaming data source. The SDKs use the same classes to represent both bounded and unbounded data, and the same transforms to operate on that data. Beam currently supports language-specific SDKs for Java and Python, but a Scala interface is also available as Scio.

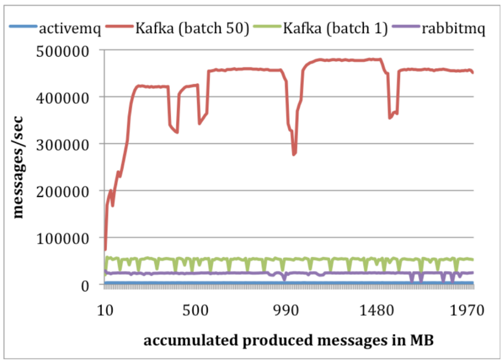
2.2.3.5 Apache Kafka

Apache Kafka [29] is a distributed streaming platform. Kafka is run as a cluster on one or more servers, called brokers, that can span multiple datacenters. The Kafka cluster stores streams of records in unbounded append only logs called topics. Each record consists of a key, a value, and a timestamp. Communication between the clients and the servers is performed through a high-performance, yet simple, language agnostic TCP protocol.

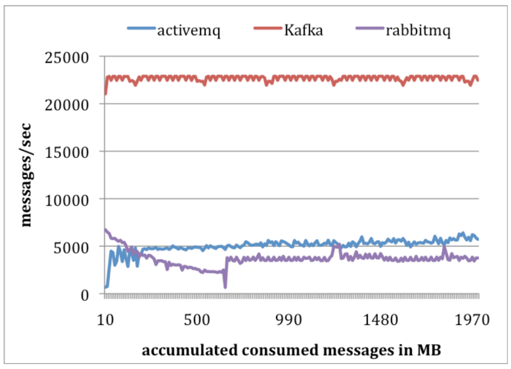
Compared to other Pub/Sub systems, such as RabbitMQ [10] and Apache ActiveMQ [11], both widely adopted message brokers, Kafka presents a noticeable performance difference [29] and it satisfying most of our requirements, we have in the end decided to opt for Apache Kafka. Moreover, we found Apache Kafka already adopted as an ingestion system in more than one recent RSP systems [32, 28], therefore it was quite a natural fit.

2.2.3.5.1 Topics and Logs: A topic is a category or feed name to which records are published. Multiple consumer can subscribe to the same topic. For each topic, the Kafka cluster maintains a partitioned log, the number of partition is a tunable parameter. Each partition is an ordered, immutable sequence of records at which new messages are continuously appended. Records in a partition are assigned a sequential id number called the offset, uniquely identifying each record within the partition. The Kafka cluster persists all published records, whether or not they have been consumed, for disk usage needs a retention period can be configured. For example, if the retention policy is set to a

[10]https://www.rabbitmq.com/
[11]http://activemq.apache.org/



(a) Producer performances.



(b) Consumer Performances.

Fig. 2.4 Kafka, RabbitMQ and ActiveMQ performance comparison from [29].

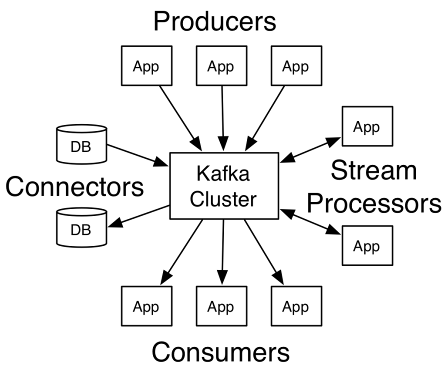


Fig. 2.5 Kafka components.

week, then it will be available for consumption for the following week, after which it will be deleted to free some disk. Kafka's performance is effectively constant with respect to data size so storing data for a long time is not a problem performancewise, it could be a problem for the size of the disk obviously. This is due to the fact that the only metadata retained for each consumer is the offset or position of that consumer in the log. This offset is controlled by the consumer, it advance its offset as it reads records, but since it controls it, it can consume records in any order it likes. Therefore it could reset to an older offset to reprocess data from the past or skip ahead to the most recent record and start consuming from "now". The number of consumers and producers can be dynamically modified at runtime. Partitioning allows the log to scale beyond a size that would fit on a single server. Each individual partition must fit on the servers that host it, but a topic may have many partitions so it can handle an arbitrary amount of data. Moreover, they act as the unit of parallelism. The partitions of the log are distributed over the servers in the Kafka cluster

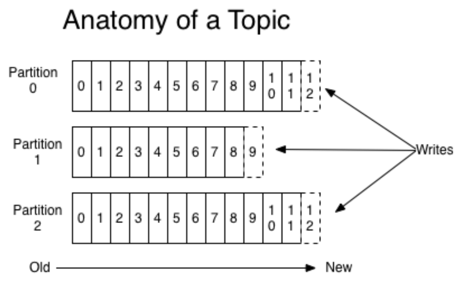


Fig. 2.6 Topics' partitioning.

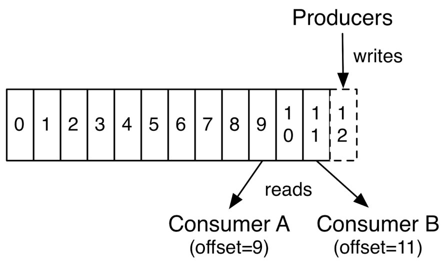


Fig. 2.7 Topics', producers' and consumers' interaction.

with each server handling data and requests for a share of the partitions. For fault tolerance each partition is replicated across a configurable number of servers. Each partition has one server acting as "leader" and zero or more servers as "followers". The leader handles all read and write requests for the partition while the followers passively replicate it. On a leader failure one of the followers will be elected as new leader and handle all read and write operations. Each server acts as a leader for some of its partitions and a follower for others so load is well balanced within the cluster. Total ordering is guaranteed only inside a single partition, not between different partitions in a topic. Key based partitioning combined with per-partition ordering is sufficient for most applications. However, if total ordering is required, it can be achieved with a topic that has only one partition, though this will imply that only one consumer thread per consumer group will be able to consume the topic.

Kafka gives these high level guarantees:

- Messages sent by a producer to a particular topic partition will be appended in the order they are sent.
- A consumer instance sees records in the order they are stored in the log.
- For a topic with replication factor N, up to N-1 server failures can be tolerated without losing any records committed to the log.

2.2.3.5.2 Architecture: In Kafka we have mainly four roles:

- **Brokers**: the servers handling the partitions.
- **Apache Zookeeper**: centralized service used to maintain naming and configuration data and to provide synchronization within distributed systems. Zookeeper keeps track of status of the Kafka cluster nodes and it also keeps track of Kafka topics,

partitions etc. The *Zookeeper atomic broadcast* (ZAB) protocol allows to act as an atomic broadcast system and issue ordered updates. A Kafka cluster uses Zookeeper to perform leader election, maintain topics' configuration and access control lists (ACLs) and keep track of cluster members' status.

- **Producers**: publishing data to required topics. They are responsible for choosing which record to assign to which partition within the topic. They send data directly to the broker leader for the partition they chose without any intervening routing tier, therefore the brokers need to offer information about the topics and their partitions' locations.

- **Consumers**: labeled together as a *Consumer Group*, each record published to a topic is delivered to one consumer instance within each subscribing consumer group. Consumer instances can be in separate processes or on separate machines. If all the consumer instances have the same consumer group, then the records will effectively be load balanced over the consumer instances. If all the consumer instances have different consumer groups, then each record will be broadcast to all the consumer processes. Consumption is implemented in Kafka by dividing up the partitions in the log over the consumer instances so that each instance is the exclusive consumer of a "fair share" of partitions at any point in time. This process of maintaining membership in the group is handled by the Kafka protocol dynamically. If new instances join the group they will take over some partitions from other members of the group; if an instance dies, its partitions will be distributed to the remaining instances.

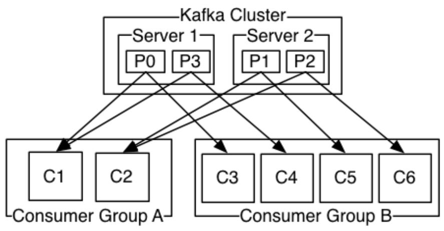


Fig. 2.8 Consumer groups and topics' partitioning.

2.2.3.5.3 APIs: Kafka has five core APIs:

- **Producer API** allows an application to publish a stream of records to one or more Kafka topics.

- **Consumer API** allows an application to subscribe to one or more topics and process the stream of records produced to them; seamlessly handles consumer groups.
- **Streams API** allows an application to act as a stream processor, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams. It is possible to do simple processing directly using the producer and consumer APIs. However, Streams API is suited for more complex transformations, adopting a Dataflow approach. This allows building applications that do non-trivial processing that compute aggregations off of streams or join streams together seamlessly using intermediate topics if necessary to perform rekeys.
- **Connector API** allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems, e.g. a connector to a relational database captures every change to a table.
- **KSQL** allows to use a SQL-like language (KSQL) to write queries against Topics present in Kafka, writing the results to console or to other topics, ready to be consumed by other components. Queries are converted to Kafka Streams topologies and executed.

2.2.3.5.4 Kafka Streams APIs: Kafka Streams is a client library for writing applications processing and analyzing data stored in Kafka, transforming input Kafka topics into output Kafka topics (if needed, calling external services, updating databases, etc.). It allows to process events using both event time and processing time, supports windowing and real-time querying of application state. It is designed to be embedded in existing Java application too and integrated with all packaging, deployment and operational tools, enriching applications with the features of Kafka. It has no external dependencies on systems other than Apache Kafka itself as internal messaging layer. It uses Kafka's partitioning model to horizontally scale processing, while still maintaining per partition ordering guarantees. It allows applications to take advantage of fault-tolerant local state, enabling stateful operations like windowed joins and aggregations. It supports exactly-once processing semantics, even in case of client or broker failure. It adopts a one-record-at-a-time processing, achieving millisecond processing latency. It supports event-time based windowing operations handling late arrival of records too. Moreover it offers both a high-level Streams DSL and a low-level Processor API.

A **Stream** is the key abstraction provided by these APIs, it represents an unbounded, continuously updating data set. It's an ordered, replayable and fault-tolerant sequence of immutable key-value pair records. Streams are consumed by stream processing applications, which defines the computational logic through one or more **processor topologies**. These are graphs of stream processing nodes connected through streams. Nodes represent a processing step to transform streams of data one input record at a time coming from its upstream node in the topology and producing one or more output records to downstream. **Sources** and **Sinks** are special kind of processor nodes, respectively without upstream and downstream processors and reading from or writing to, specific Kafka topics.

Kafka Streams is able to handle Event, Processing and Ingestion time, each data record is assigned a timestamp and the assignment can be customized through the implementation of an interface. These timestamps describe the progress of a stream with regards to time and are leveraged by time-dependent operations such as windowing operations.

Sax et al. [33] proposed the **Dual Streaming Model**, which is implemented in Kafka Streams. This model presents the result of an operator as a stream of successive updates, which induces a duality of results and streams, providing a natural way to cope with inconsistencies between the physical and logical order of streaming data in a continuous manner, without explicit buffering and reordering. They recognize the fact that in most existing streaming models, operators directly yield an output data stream and the state operator treatment is considered an implementation detail, e.g. CQL [3] ignores how physical and logical ordering inconsistencies of a stream may influence the computation of operator state and, therefore, the resulting output stream. Out-of-order records are neglected by models and are left to the implementation layer, therefore usually this translates into postponing any computation until the logical ordering of records has been established. As a consequence, the latency with which outputs are emitted grows linearly with the maximum allowed lateness of records. Therefore they propose a model in which operator result are updated continuously. Thus allows to drop any assumption on the consistency of the logical and physical order of records. Furthermore, operators' result can be seen either statically, as their materialization from processing a stream up to a certain record, or dynamically, as a stream of successive updates. Both views are just different ways of programming w.r.t. time, one may process a stream of result updates or continuously query the materialized result to achieve the same computational logic. The model comprises the notions of *table*, *table changelog stream* and a *record* stream. The static view of an operator's state is a *table*, updated for each input record. Tables, as Streams, have a primary key attribute. For instance, for an aggregator operator, grouping conditions define the primary key of a table. Out-of-order records increase the respective key as if they were in-order, therefore handling them does not increase the latency. Updating an older entry is not more expensive than updating a recent one, they are both a lookup in a hash table (Fig.2.9).

$$\begin{tikzcd}
\text{Table} \arrow[r, bend left, "\text{capture updates}"] & \text{Table Changelog Stream} \arrow[l, bend left, "\text{materialize updates}"]
\end{tikzcd}$$

Fig. 2.9 Duality of stream and tables.

The dynamic view on the result of an operator is a table changelog stream, consisting of records that are updates to a table. The semantics of an update is defined over both keys and timestamps. Replaying a table changelog stream allows to materialize the operator result as a table.

Applying a table changelog stream to an empty table results in a table with all the latest values per key. But, that's not sufficient for stream processing operators having temporal semantics (e.g. joins between tables), it is necessary to reason about the table content over time. Therefore, they consider tables as collections of versions, one for each point in time in which its content has been updated, using as version number the timestamp of the update and maintaining multiple versions at once. Out-of-order records update or add the corresponding version of the table and all the depending versions. To reason about the state of a table, they are assigned a *generation* number, which is incremented for each processed input record. Each table generation may consist of multiple table versions.

Finally, as said, *record streams* are part of the model too and represents facts, instead of updates. Record streams and changelog streams are special cases of streams. Record streams' model immutable facts, therefore each record must have a unique key other the whole strea, e.g. a unique id.

Stream processing operators are divided in stateless and stateful ones, may have one or multiple input streams and might be defined over special types of input streams only (Fig.2.10)

To abstract over ordered and unordered input streams, they define *stream equivalence* for record streams and table changelog streams. They see ordered streams as a canonical representation of an equivalence class of streams, both ordered and unordered. Therefore provide definitions for operators over ordered streams and then extend them to unordered

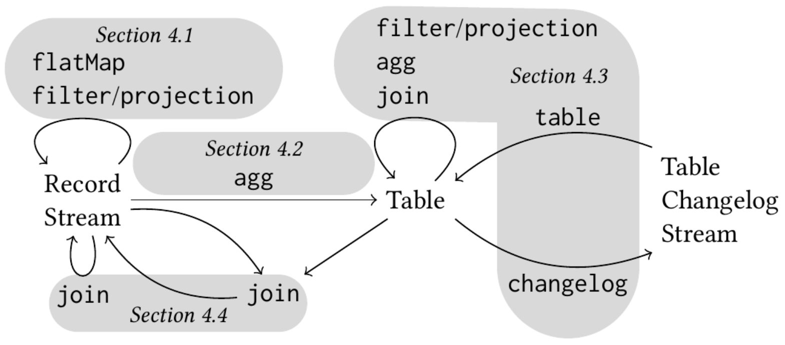


Fig. 2.10 Transformations between record streams, tables and changelog streams.

ones, using the equivalence. Two record streams are defined equivalent if and only if their corresponding multi-sets of timestamps-value pairs are equal.

Stateless operators process one record at a time and produce zero or more records, preserving ordering and the timestamp of the input records. A stateless operator is defined correct if and only if, given two equivalent streams, one ordered and one unordered, the resulting streams obtained by applying the operator to both streams are equivalent.

Record stream aggregation operators take as inputs a stream and produce an ever updating result table. Correctness for this kind of operator is defined through table equality, therefore given two equivalent streams the aggregation operator should yield two equivalent tables. Table equality implies that both tables contains the exact same versions, therefore timestamps have to be handled correctly and updates have to be emited for out-of-order input records. If no maximum lateness threshold is set, the whole history of the operator has to be kept in memory, if this is not possible a retention-time parameter can specified and defines the maximum delta w.r.t. current event time a record should have in order to be accepted and used to update the content of a table. The main difference between retention-time and watermark/punctuation mechanisms is that it does not introduce any latency, it represent just a mean to let the user choose between memory usage and correctness in case of out-of-order arrivals.

The model support windowing as a particular case of grouping by key, where the key has been modified to embed the windowing information.

Moreover, the model offers a set *table operators*, *table*, which materializes a table changelog stream, *filter*, limited *projection*, *map*, *aggregation* and primary key equi *join* for tables. All of them produce tables, which are updated at each source update. The semantics of these operators follow a temporal-relationa model, therefore updates on an input table version trigger update on output table with the same version. Table-table joins are defined over the corresponding table version.

A variety of equi-join operators for streams are included too, i.e. both stream-stream and stream-tables joins. Joins between streams are defined over a join attribute as well as a sliding window on event-time centered around the new element's timestamp. Two records join if their timestamps are "close to each other", the distance is less than or equal to the window size. Join results get the larger timestamp among the ones of the input records. Stream-table joins produce a stream by performing a key table lookup join over the key join attribute. It's a temporal join, the table lookup is done into the table version corresponding to the stream record's timestamp. Stream-table joins can also left-outer-join. Stream-stream instead can be inner, left-outer and full-outer joins.

Out-of-order records are handled differently in the various cases. Stream-table joins do not require special handling if the out-of-order comes from the stream, instead if it is an update to the table to arrive late, the stream input records needs to be buffered in the operator to be able to retrigger the join computation for late table updates. Updates in this case would overwrite previously emitted join records. For full and full-outer stream-stream joins the same techniques applies. Records that did not match are emitted eagerly and might be updated if a matching record appears in the other stream later on, therefore the resulting stream is no more a record stream. This allows to implement totally non-blocking joins. Inner stream-stream joins, being a monotonic operation, can never yield an early result that has to be updated later on, therefore producing a record stream in output.

Kafka Streams APIs implement the Dual Streaming Model through two first-class abstractions, KStream and KTable, respectively an abstraction of a record stream and both changelog stream and its materialized tables. Moreover KTables can be queried via DSL in real-time. All the operators of the model are implemented and can be seen in Fig.2.11

Kafka Streams Applications are able to handle partitioned topics, clustering, scaling number of instances and faults seamlessly by using the kafka cluster as intermediate state store whenever a rekey is necessary and to keep the index each consumer have reached of consumed topics. The messaging layer of Kafka partitions data for storing and transporting it. Kafka Streams partitions data for processing it. In both cases, this partitioning is what enables data locality, elasticity, scaleability, high performance, and fault tolerance. Kafka Streams uses the concepts of stream partitions and stream tasks as logical units of its parallelism model. An application's processor topology is scaled by breaking it into multiple stream tasks. More specifically, Kafka Streams creates a fixed number of stream tasks based on the input stream partitions for the application, with each task being assigned a list of partitions from the input streams (i.e., Kafka topics). The assignment of stream partitions to stream tasks never changes, hence the stream

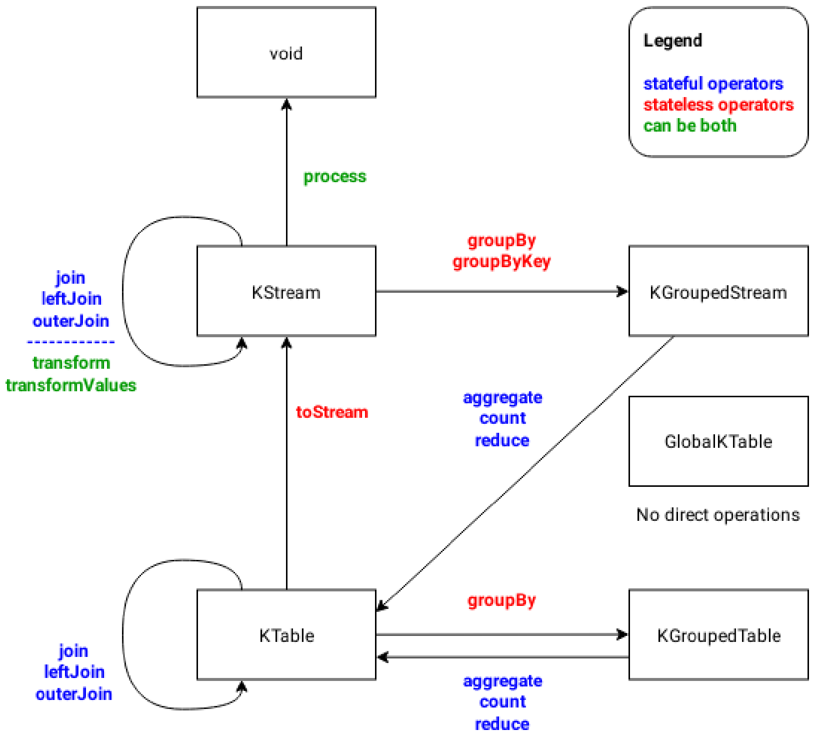


Fig. 2.11 Kafka Streams Api operators.

task is a fixed unit of parallelism of the application. Tasks can then instantiate their own processor topology based on the assigned partitions; they also maintain a buffer for each of its assigned partitions and process input data one-record-at-a-time from these record buffers. As a result stream tasks can be processed independently and in parallel without manual intervention. The maximum parallelism at which your application may run is bounded by the maximum number of stream tasks, which itself is determined by maximum number of partitions of the input topic(s) the application is reading from. Kafka Streams provides so-called state stores, which can be used by stream processing applications to store and query data. The Kafka Streams DSL, automatically creates and manages such state stores for stateful operators. Every stream task in a Kafka Streams application may embed one or more local state stores that can be accessed via APIs to store and query data required for processing. These state stores can either be a RocksDB database, an in-memory hash map, or another convenient data structure. Kafka Streams offers fault-tolerance and automatic recovery for local state stores.

Kafka Streams builds on fault-tolerance capabilities integrated natively within Kafka. Partitions are replicated; data persisted to Kafka is available even if the application fails and re-process it is needed. Tasks in Kafka Streams leverage the fault-tolerance capability offered by the Kafka consumer client. Tasks that run on a failed machine, are automatically restarted in one of the remaining running instances of the application. Kafka streams offers local state stores robust to failures. For each state store, it maintains a replicated changelog as a Kafka topic in which it tracks any state updates. These are partitioned as well so that each local state store instance has its own dedicated changelog topic partition. Log compaction is enabled on the changelog topics so that old data to reduce memory consumption. If tasks run on a machine that fails and are restarted on another machine, Kafka Streams guarantees to restore their associated state stores to the content before the failure by replaying the corresponding changelog topics prior to resuming the processing on the newly started tasks. As a result, failure handling is completely transparent to the end user.

2.2.4 Dataflow model

Akidau et al. in [1] propose a fundamental shift of approach to handle unbounded data, stopping waiting for data to be complete or assume they will ever be and just then process them, leaving the user the choice of the appropriate tradeoff between correctness, latency and cost. The Dataflow Model describes the processing model of Google Cloud Dataflow, which is based upon previous projects such as FlumeJava and MillWheel. Opposed to widely adopted batch systems such as MapReduce, FlumeJava and Spark, suffered of the latency problems inherent with collecting all data into a batch before processing it, the so called Lambda architecture. The streaming systems at the time fell short either on fault tolerance at scale, expressiveness or correctness.

The authors' main critics to the vast majority of streaming systems, as we said, is in the approach is their belief that at some point in time the streaming data will be complete and they define it as "fundamentally flawed when the realities of today's enormous, highly disordered datasets clash with the semantics and timeliness demanded by consumers". The Dataflow model instead is aimed at allowing computation of event-time ordered results, windowed by features of the data themselves over unbounded, unordered data source with tunable correctness, latency and cost. Therefore choosing between batch, micro-batch or streaming engine becomes a simple choice along these axes. Concretely this model translates into models for Windowing, Triggering and Incremental Processing and a set of core principles for the Dataflow model.

W.r.t. windowing they propose Fixed (a.k.a. Tumbling), Sliding and Sessions windows. Session windows are windows that capture some period of activity over a subset of the data, in this case per key, typically defined by a timeout gap. Windowing has two time domains over which can be applied, Event and Processing Time.

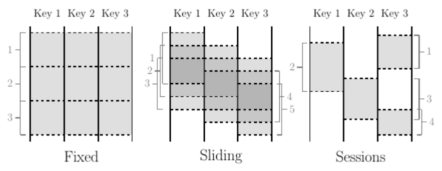


Fig. 2.12 Dataflow windowing model [1].

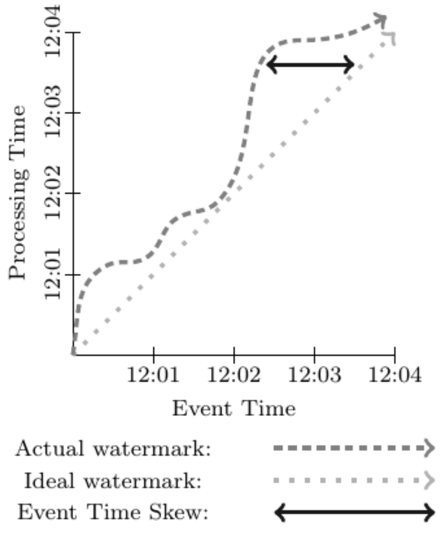


Fig. 2.13 Time Domain Skew [1].

The Dataflow model itself then is formally presented. It's two core primitives operating on the *(key, value)* pairs flowing through the system are **ParDo** for generic elementwise parallel processing producing zero or more output elements per input, and **GroupByKey** for key-grouping tuples, which collects data for a given key before sending them downstream for reduction, being data unbounded it needs windowing in order to be able to decide when it can output.

Windowing is usually treated as key modifier, this way GroupByKey will group also by window. Windows can be *aligned*, i.e. spanning the entirety of a data source, or *unaligned*, i.e. spanning only a subset of it. Key point of this model is that windowing can be split in two sub operations, **AssignWindows**, which assigns an element to zero or more windows, and **MergeWindows**, which merges windows at grouping time, allowing datadriven windows to be constructed as data arrive and are grouped together. The actual merge logic is defined by the windowing strategy, for example session windows get merged if they overlap. Window merging is then further split in five parts: DropTimestamps, GrouByKey, MergeWindows, GroupAlsoByWindow, ExpandToElements. All but the last step names are quite self explanatory and ExpandToElements takes care of expanding per-key, per-window groups of values into (key, value, event time, window) tuples, with new per-window timestamps.

After having described how windowing for unaligned windows is performed in the model, they address the problem of window completeness, being that watermarks can introduce incorrectness if set too fast or too high latency if too slow, they postulate that they are insufficient. To solve this issue they recognize a key aspect in the lamba architecture, that is that the fact that the streaming pipeline provides just the best low-latency estimate of a result, with the promise of eventual consistency and correctness once the batch pipeline runs, assuming that data will be complete at that time. Therefore being desirable to maintain a single pipeline, the streaming system should be able to provide both the lowest latency best answer possible, and the most complete one whenever possible. In order to be able to do so, the system needs a way to provide multiple answers for any given window and they call this feature **Triggers**.

They present them as a mechanism to stimulate the production of GroupByKeyAnd-Window results in response to external signals. Windowing and Triggering are complementary in that the first determines *when in event time* data are grouped together for processing whereas the second determines *when in processing time* the results of groupings are emitted. Multiple trigger implementations are provided, e.g. watermarks, percentile watermarks, at points in processing time, in response to data arriving, they can be composed with logical operators, loops, sequence and other constructs, or they can be user defined function too. The triggering system also provides ways to control how multiple panes for the same window have to be handled: **Discarding**, therefore upon triggering windows' contents are discarded and later results are independent from previous ones, allowing downhill consumers to assume inputs as independent; **Accumulating**, upon triggering windows' contents are left intact and later results become refinement of previous results; **Accumulating and Retracting**, upon triggering, in addition to the Accumulating semantics, a copy of the previous value is stored and used before the next trigger to retract the previous data.

The Principles they propose and that brought to the Dataflow model are the following:

1. Never rely on any notion of completeness.
2. Be flexible, to accommodate the diversity of known use cases, and those to come in the future.

3. Not only make sense, but also add value, in the context of each of the envisioned execution engines.

4. Encourage clarity of implementation.

5. Support robust analysis of data in the context in which they occurred.

2.3 Stream Reasoning

Stream reasoning is the research area investigating *"how to perform online logical reasoning over highly dynamic data"* [25]. RDF Stream Processing (RSP) is a sub research area of SR that investigates ways to query continuous RDF streams.

2.3.1 Inference Process

In the *stream reasoning* context, processing of streams requires a critical design choice about the moment in which the inference process must be taken into account. This choice can impact both performances and expressiveness and characterizes the different solutions proposed to approach the *stream reasoning* problem.

The discussed RSP Engines do not implement, as default, any entailment regime and do not consider ontologies and inferential processing. C-SPARQL and CQELS can operate under RDFS entailment regime, but this deteriorates performances. To face this issue, *Barbieri et al.* proposed an approach to incrementally maintain the entailed triples (materialization) avoiding recomputing it from zero at each window change [6].

2.3.1.1 Query rewriting

The problem of answering queries over an ontology has been approached in different ways, but query rewriting represents an important and promising technique under some expressiveness constraints. Based on the idea of transforming the original query into an expanded query capturing the information of the ontology TBox and then executing it over the ABox, providing answers that extract implicit knowledge from the data, this approach has been successfully adopted in scenarios such as OBDA (Ontology Based Data Access), in which data are stored in relational databases, but user queries are written against an higher level ontologies, rewritten and then translated to SQL queries. Various Description Logic (DL) languages have been explored and used to specify the TBox over which rewriting can be performed. The DL-*Lite* family of languages presented by Calvanese et al. [15] represented the first milestone in this area, derived into . The first including functionality restrictions on roles and the latter disjointness assertions between roles. These logics are first-order reducible or FOL-rewritable, meaning that rewritten queries are first-order queries, allowing them to be converted to languages like like SQL without the need for recursion, with tractable complexity. *OWL2 QL* was inspiderd by this family of DLs and designed to keep low the rewriting complexity as first-order rewritability. Main difference *OWL2 QL* introudced was the lach of Unique Name Assumption (UNA).

The $\mathcal{ELHIO}^{\neg}$ description logic used by Perez-Urbina et al. [31] is more expressive, it extends DL-$Lite_{\mathcal{R}}$, but does not preserve first-order rewritability property, meaning that depending on the query and the ontology's expressiveness the generated Datalog may contain recursive predicates. Therefore some queries cannot be rewritten to Union of Conjunctive Queries (UCQ) and have to be rewritten to recursive Datalog. Nonetheless, the computational complexity of the rewriting remains tractable (PTime-complete).

Query rewriting is based on the idea of expanding the original query taking into account the ontology TBox and then evaluate the expanded query over a given ABox, providing answers that extract implicit knowledge of the data.

Given an input query q and an ontology $\mathcal{O} = (\mathcal{T}, \mathcal{A})$, a *query rewriting* algorithm transforms q using $\mathcal{T}$ into a query q', such that for every ABox $\mathcal{A}$ the set of answers that q' obtains from $\mathcal{A}$ is equal to the set of answers that are entailed by q over $\mathcal{T}$ and $\mathcal{A}$. As long as some restrictions are imposed to the ontology language, q' can be unfolded and expressed as *union of conjunctive queries* (UCQ). Assuming the ontology $\mathcal{T}$ as statical, rewriting can be executed only once.

The ontology TBox can be described using different languages or logics. Calbimonte et al. [14] focused on $\mathcal{ELHIO}$ logic, as it is one of the most expressive logics currently used for query rewriting. Nonetheless, it is not FOL-reducible in general, in fact if cyclic axioms are present, it may produce recursive datalog programs, but still remain tractable (PTime-complete) for the rewriting process. For these reasons we decided to adopt acyclic $\mathcal{ELHIO}$ for describing TBoxes.

As we said, to be sure queries are rewriteable we have to constraint them to the form of *conjunctive queries* (CQ):

$$q_h(x) \leftarrow p_1(x_1) \wedge ... \wedge p_n(x_n)$$

where q_h is the head of the query, x a tuple of distinguished variables, $x_1 ... x_n$ tuples of variables or contraints, $p_i(x_i)$ are unary or binary atoms in the query's body. Certain answers for this kind of queries can be defined as the set:

$$cert(q, \mathcal{O}) = \{\alpha | q \cup \mathcal{T} \cup \mathcal{A} \models q_h(\alpha)\}$$

The rewriting algorithm produces a query q' in the form of UCQ such that:

$$cert(q,\mathcal{O}) = \{\alpha | q' \cup \mathcal{A} \models q_h(\alpha)\}$$

To apply query rewriting to continuous queries, a usual assumption is to consider the TBox static, while the ABox dynamic, therefore Calbimonte et al. defined an instantaneous ABox $\mathcal{A}(t)$ and represented the stream as a sequence of ABoxes over time.

Calbimonte et al. also implemented and open-sourced a prototype, *StreamQR*, consisting of two separate components: a query rewriter *kyrie*, using $\mathcal{ELHIO}$ ontologies, and an RSP query engine CQELS, which executes the rewritten queries, producing continuous answers.

2.3.1.2 RSP-QL

Dell'Aglio et al. in [21] proposed the RSP-QL model, a unifying semantic for RSP engines and continuous SPARQL extensions. This model proposes semantics similar to the one used in DSMSs [2].

The RSP-QL model can explain the operational semantics of existing RSP Engines and can be used to describe the semantics of the different continuous SPARQL extensions proposed by these engines.

RSP-QL defines **RDF streams** as an unbounded sequence of pairs (G_i, t_i), being t_i the associated timestamp and G_i either an RDF triple or an RDF named graph RDF graph. similarly to CQL, RSP-QL defines time-based window operators using three parameters, the starting time t_0, the window width α and the sliding parameter β. Given an RDF Stream S, a time-based sliding window operator $\mathbb{W}$ applied to S defines a *Time-Varying Graph*, a function returning for each time instant an *Instantaneous RDF Graph*.

RSP-QL extends the SPARQL definition of a query to a quadruple (SE, SDS, ET, QF), where SE is an RSP-QL algebraic expression, SDS is an RSP-QL dataset, ET is the sequence of time instants on which the evaluation occurs and QF is the Query Form. While the QF values are the same of SPARQL (i.e. SELECT, CONSTRUCT, DESCRIBE and ASK), SDS and SE are extended w.r.t. SPARQL's DS and E to take into account the time dimension.

An RSP-QL *streaming dataset* (SDS) is composed of:

1. an optional default graph G_0
2. n named graphs $(u_1, G_1), ..., (u_n, G_n)$
3. m named time-varying graphs obtained applying a sliding window operator over a stream $(w_1, W_1(S_1)), ...(w_m, W_m(S_o)$.

Evaluation time instants (ET) are defined by mean of reporting policies that can be specified over the inputs windows and combined using logical operators, "On Content Change" (CC) if the window content changes, "Non-empty Content" (NC) if the current window is not empty, "On Window Close" (WC) if the current window closes, and "Periodic" (P) reporting at regular intervals.

Outputs of an RSP-QL query may be either a sequence of solution mappings, i.e., a sequence of compliant SPARQL answers, or a sequence of timestamped solutions mappings if a streaming operator is applied. Streaming operators, *RStream, IStream* and *DStream*, are defined similarly to the R2S operators in [3] and append at each evaluation a new set of elements to the output stream with respect to the logic they implement.

RSP-QL evaluation semantics is defined for a query with n windows as:

$$eval(SDS(G_0,...,G_n),SE,t)$$

where $SDS(G_1,...,G_n)$ is a streaming dataset having n active Time-Varying Graphs. The evaluation is computed over the instantaneous graphs $Gi(t)$ as $eval(SDS(Gi,t),SE)$.

2.3.1.3 C-SPARQL engine

Continuous SPARQL (C-SPARQL) is a continuous extension of SPARQL allowing to register continuous queries over RDF Streams presented by Barbieri et al. in [7] that adopted an approach comparable to that of CQL [2]. RDF Streams' elements are represented as timestamped RDF triples. Queries written in C-SPARQL are executed in CSPARQL-engine, an RSP engine whose architecture consists into a DSMS piped into a SPARQL engine, therefore windowing is delegated to the DSMS and the remainder of the query to the SPARQL engine. It has a fixed reporting policy, on window close and non-empty content (WCNC), and provides RStream operator only as a streaming operator.

A C-SPARQL query is first parsed and then sent to an orchestrator, which splits it into a dynamic part then sent to the DSMS and a static part for the SPARQL engine. The static queries will be only issued at registration time, therefore no further updates to the non-stream data is taken into consideration. The static parts are then loaded into materialized relations as inputs for the DSMS. These relations together with the RDF streams are then used to compute the queries result via cascading views created as CQL queries; the first views in the query pipeline are window views over RDF streams, that are then joined with the static data and aggregated if neededd.

2.3.1.4 Morph-stream

Calbimonte et al. in [13] presented SPARQLstream, a SPARQL streaming extension to query virtual RDF streams composed of timestamped RDF triples. Morph-stream is the engine translating SPARQLstream queries into several DSMS queries over relational streams of data, by means of R2RML mappings. The mapping language is S_2O, an ad-hoc extension of R_2O, to include time constructs as windowing. Has a fixed reporting policy WC.NC. and all the R2S operators are available.

2.3.1.5 Strider

Ren et al. [32] presented Strider, a system aimed at efficiently integrating reasoning services within a real-time processing platform, through inference materialization and query rewriting. They pose particular attention on performances and therefore present optimizations to reduce stored knowledge memory usage and an adaptive query processing (AQP) for continuous SPARQL queries, mixing static heuristic-based and dynamic cost-based query optimization approaches.

They recognize the need for a distributed stream processing system as target system for executing queries in order to meet their performance requirements, therefore they adopt Apache Kafka as communication bus and Apache Spark as engine (Fig. 2.14).

As can be seen in Figure 2.14, Strider's architecture is split in multiple layers.

Due to the nature of the incoming data streams they have to translate data source messages into an RDF serialization, therefore the first layer of the architecture is dedicated to this.

Upfront to any data stream processing, an Encoding layer encodes concepts predicates and instances of registered KBs, this is to provide an efficient encoding scheme and data structures to support reasoning tasks. They target the ρdf subset of RDFS. Then encoded terms are used to encode incoming data streams.

The Optimization layer tries to optimize the execution of queries through a static and adaptive hybrid strategy in order to reduce network usage due to data shuffling required by the numerous joins that could have to be performed. The optimization layer creates a Undirected Connected Graph (UCG), where vertices correspond triple patterns and edges to joins between them. Vertices and edges are assigned weights corresponding to the selectivity of the triple pattern and joins respectively. This graph is then used during Static Optimization phase to generate, independently of the effective data, the logical plan for the query execution, which will be kept during its whole lifetime. The static plan could lead to suboptimal performances, therefore they have added an adaptive query

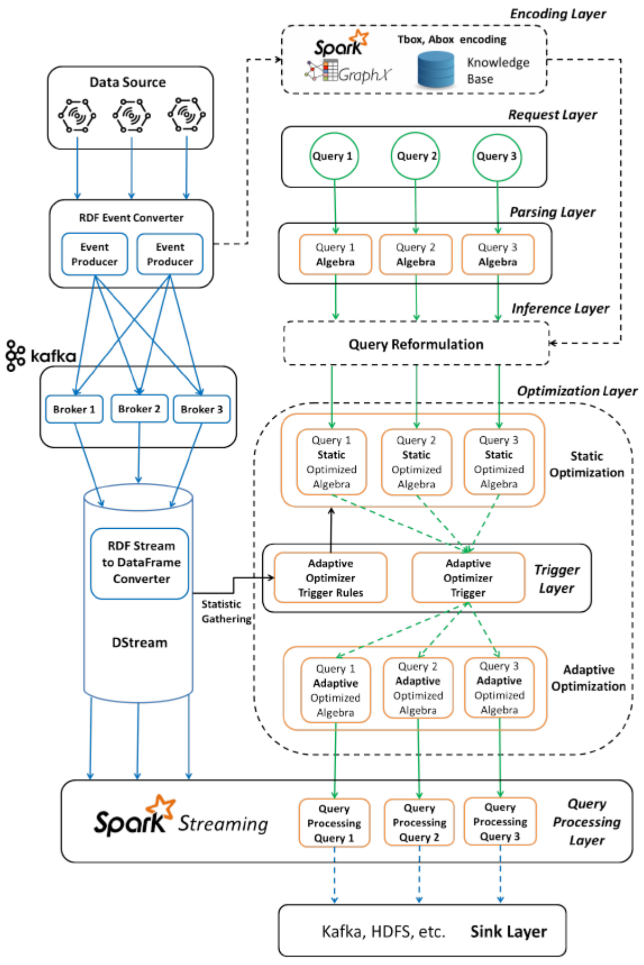


Fig. 2.14 Strider architecture [32].

optimization component. A component takes care of watching for specific conditions to be met, specified through simple rules defined over execution parameters and then notify the adaptive optimization component. Then UCG elements' weight will be recomputed on the basis of statistics of the data and the query plan recomputed and redeployed.

2.3.1.6 CQELS

Le-Phuoc et al. [30] criticize existing systems's "black box" approach, which delegate the processing to other Stream Processing Engines and SPARQL Query Processors by translating to their provided languages. To address this issue, the authors propose CQELS (Continuous Query Evaluation over Linked Streams), a native and adaptive query processor for unified query processing over Linked Stream Data and Linked Data. CQELS uses a "white box" approach and implements the required query operators natively to avoid the overhead and limitations of the aforementioned approach. Moreover, CQELS is able to dynamically adapt the query execution plan to changes in the input data. In fact, during query execution, it continuously tries to reorder operators according to defined heuristics, in order to improve the query execution performance.

The adaptive processing model of CQELS defines two types of data sources, RDF streams and RDF datasets and builds relational trees on top of windowed or static sources, that are then executed incrementally. The CQELS query engine allows to register Continuous queries using an homonymous language. Moreover, they propose to adopt the *dictionary encoding*, to reduce the number of required accesses to disk for data collections not fitting in main memory. This approach, commonly used by triple stores, allows to map RDF node to integers, reducing their size considerably. Moreover, since data comparison is now done on integers, rather than strings, pattern matching operations' performance result improved. Although, converting high rate data on the fly might not always be convenient, in fact, the cost of updating a dictionary and decoding the data might worsen performance instead. Therefore, they propose an optimized technique to encode RDF nodes in at most 64-bit integers, not encoding strings shorter than 63 bits.

2.3.1.7 Streaming MASSIF

Bonte et al. presented Streaming MASSIF [11] which applies a Cascading Reasoning [37] approach to perform efficient processing of IoT data streams. They combine RDF Stream Processing, expressive DL reasoning and Complex Event Processing to perform expressive and temporal reasoning over high volatile stream. They manage to combine CEP and DL, enabling the definition of patterns using high-level concepts, integrating complex domain models within CEP and a temporal notion in DL (Fig.2.15) . They present an associated query language bridging to allow defining the required temporal patterns.

They propose a generalized vision over Cascading Reasoning (Fig.2.16) , originally the role of RSP and raw stream processing was limited to streaming data integration, but they can support reasoning tasks too, through entailment regime or query rewriting.

Streaming MASSIF presents a layered architecture, composed of:

1. Input Module, the entrypoint of the platform.

2. Annotation Module, where raw data can be semantically annotated if necessary.

3. Selection Module which implements both the Stream Processing and the Continuous Information Integration Layer of the Cascading Reasoning approach and selects, through RSP, those parts of the RDF stream that are relevant. It uses YASPER, an

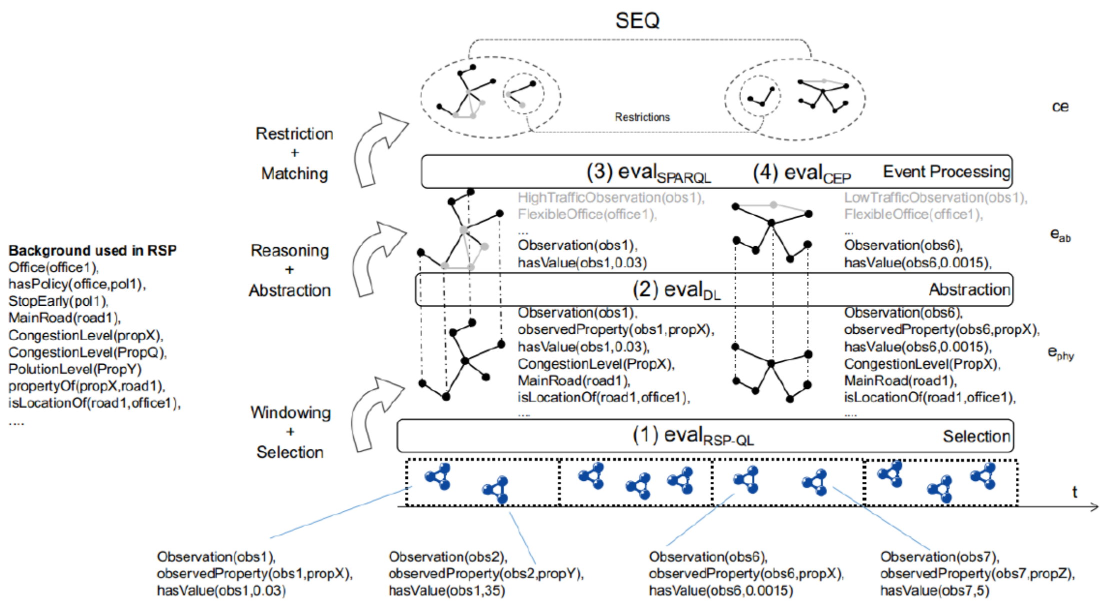

SEQ
Restrictions
ce
Restriction + Matching
(3) eval SPARQL
(4) eval CEP
Event Processing
HighTrafficObservation(obs1), FlexibleOffice(office1),
LowTrafficObservation(obs1), FlexibleOffice(office1),
Observation(obs1), hasValue(obs1,0.03)
Observation(obs6), hasValue(obs6,0.0015),
e ab
Background used in RSP
Office(office1), hasPolicy(office,pol1), StopEarly(pol1), MainRoad(road1), CongestionLevel(propX), CongestionLevel(PropQ), PolutionLevel(PropY) propertyOf(propX,road1), isLocationOf(road1,office1), ...
Reasoning + Abstraction
(2) eval DL
Abstraction
Observation(obs1), observedProperty(obs1,propX), hasValue(obs1,0.03), CongestionLevel(PropX), MainRoad(road1), isLocationOf(road1,office1),
Observation(obs6), observedProperty(obs6,propX), hasValue(obs6,0.0015), CongestionLevel(PropX), MainRoad(road1), isLocationOf(road1,office1),
e phy
Windowing + Selection
(1) eval RSP-QL
Selection
t
Observation(obs1), observedProperty(obs1,propX), hasValue(obs1,0.03)
Observation(obs2), observedProperty(obs2,propY), hasValue(obs1,35)
Observation(obs6), observedProperty(obs6,propX), hasValue(obs6,0.0015)
Observation(obs7), observedProperty(obs7,propZ), hasValue(obs7,5)

Fig. 2.15 MASSIF's Cascading Reasoning Approach from [11].

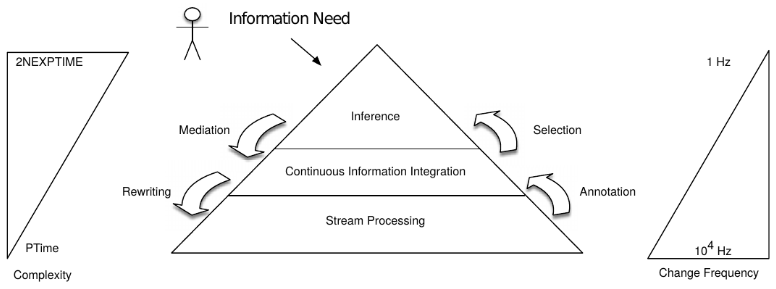


Fig. 2.16 MASSIF's generalized Cascading Reasoning piramid.

RSP engine fully implementing RSP-QL semantics and consuming time annotated graphs. Multiple RSP engines can be deployed in parallel to handle multiple streams.

4. Abstraction Module implementing the DL inference sub-layer. Receives selected events from the Selection Module and abstracts them to high-level concepts. It allows for semantic publish/subscribe mechanism, here services in the service module will be able to subscribe to events by a generic description. It operates on a OWL reasoner (HermiT).

5. Event Processing Module, implementing the temporal reasoning sub-layer. If needed this module is used to execute EPL queries over the abstracted events coming from the Abstraction module.

6. Service Module, receives the processed data and can perform additional analysis. Information needs can be formulated though this module using a custom DSL.

Chapter 3

Problem Statement

In this chapter, we define the Distributed RDF Stream Processing problem, we discuss its relevance, and we provide the research question, its hypotheses and requirements.

3.1 Velocity on the Web

Nowadays, Web applications need to continuously manage streams of data and detect events reactively, therefore taming data Velocity is a necessity. To this extent, Web technologies evolved to embrace these new challenges. For instance, one of the web pillars, HTTP was extended into HTTP Long-Polling[1], HTTP Streaming[2], WebSocket[3] to provide streams of continuous updates like Twitter's stream feeds, or WebHooks[4], see Github repositories' hooks, to provide reactive update.

The Web of Data (WoD) is an extension of the World Wide Web, aiming at data shareability and intoperability. The WoD builds on the Web infrastructure and approaches intoroperability by providing the following guidelines,i.e.,the Linked Data Principles [9]:

1. Use URIs as names for resources.
2. Use HTTP URIs so that people can look up those names.
3. Provide useful information using the standards (RDF, SPARQL) on URI resolution.
4. Include links to other URIs, so that related informations can be retrieved.

[1]https://realtimeapi.io/hub/http-long-polling/
[2]https://realtimeapi.io/hub/http-streaming/
[3]https://realtimeapi.io/hub/http-streaming/
[4]https://www.w3.org/TR/websub/

The WoD is an evolving environment and, although built on top of this up-to-date web infrastructure, its main abstractions, RDF Triples, Named Graphs and RDF data sets are not adequate to tackle Velocity, as they were only designed to tame Variety.

During the last 10 years, the Semantic Web (*SW*) community investigated under the name of Stream Reasoning [20], the following research question:

> **Q1** Can we tame Velocity on the Web of Data without neglecting Variety?

The initial effort of the community was put into proving the approach feasibility, providing data models, query languages and artifacts for RDF Stream Processing (RSP), empirically proving SR effectiveness. RSP represents the junction point between previous approaches such as Complex Event Processing (CEP), Data Stream Management Systems (DSMSs) and Semantic Web technologies. The goal of RSP is enriching the stream processing capability to tame Velocity with the ability to tackle Variety. Nevertheless, most of the existing solutions do not prioritize scaleability. Instead, they address Variety and Velocity simultaneously, tackling Volume only through vertical scaleability, due to the intrinsic difficulty of parallel graph computation, even just on static data. Requirements for RSP systems have been specified in [24].

3.2 Expressive Stream Reasoning

In order to tackle Variety, the Semantic Web world represents knowledge using formal languages and derive implicit information through the use of reasoning algorithms. Intuitively, opting for more expressive languages to model the domain requires reasoning algorithms with high computational complexity. To handle the increasing Volume and Velocity of the WoD, Semantic technologies have to perform reasoning scalably and fastly, but that's not always possible for highly complex and expressive methods.

Indeed, existing Stream Reasoning solutions only partially address this problem, presenting either highly expressive solutions with limited throughput or little expressive ones with better performances.

Stuckenschmidt et al. [37] envisioned two complementary approaches, both aimed at reaching the highest expressiveness possible in a context where data Velocity and Volume are of central importance. From an architectural point of view, they propose to connect multiple agents in a *Network of Stream Reasoners* (NoSR) (Fig.3.1) able to collectively process streams of data. These stream reasoners would be able to perform tasks otherwise impossible to the single, by taking advantage of each reasoner's strengths. The obvious drawback of a completely decentralized solution, as the NoSR, is the need for the agents

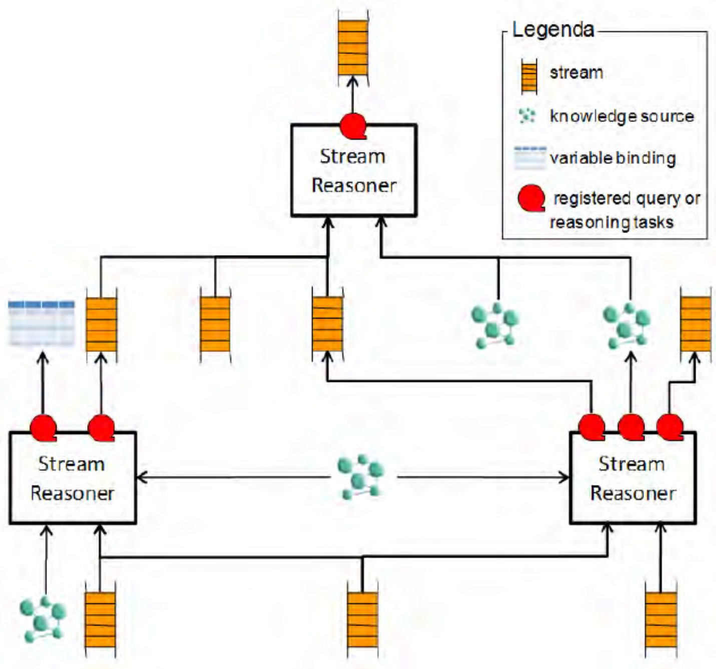


Fig. 3.1 A network of stream reasoners from [37].

to discover each other and qualify their capabilities in an open environment, like the Web, before being able to collectively perform any task.

The other approach focus on the applicative aspect. Stuckenschmidt et al. envisioned to split the reasoning complexity across multiple layers, each optimized for a particular set of tasks. This model goes by the name of *Cascading Reasoning* (CR) (Fig.3.2). Less expressive reasoning methods are used to extract only the needed parts of a stream and then more complex and expressive methods can be applied to a reduced amount of data, which they will be able to handle. And this pipeline can be of arbitrary length

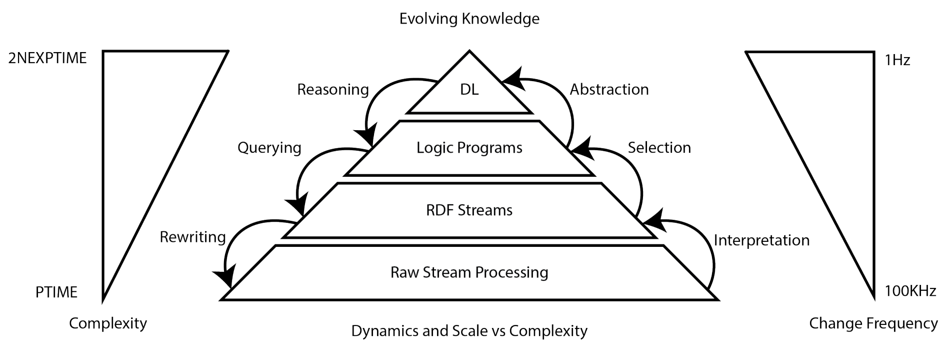


Fig. 3.2 Cascading Reasoning pyramid.

with increasingly complex reasoning methods, achieving the expressiveness of the most complex ones, but over large Volumes of streaming data, pushing subtasks down to little expressive, yet more efficient reasoners whenever possible. Tasks could be rewritten into smaller subtasks accordingly to each layer's capabilities, therefore a coordinator that has full knowledge of the actors involved.

Streaming MASSIF [11] is the first and only example of a system adopting the CR approach, performing expressive reasoning and complex event processing over high velocity streams.

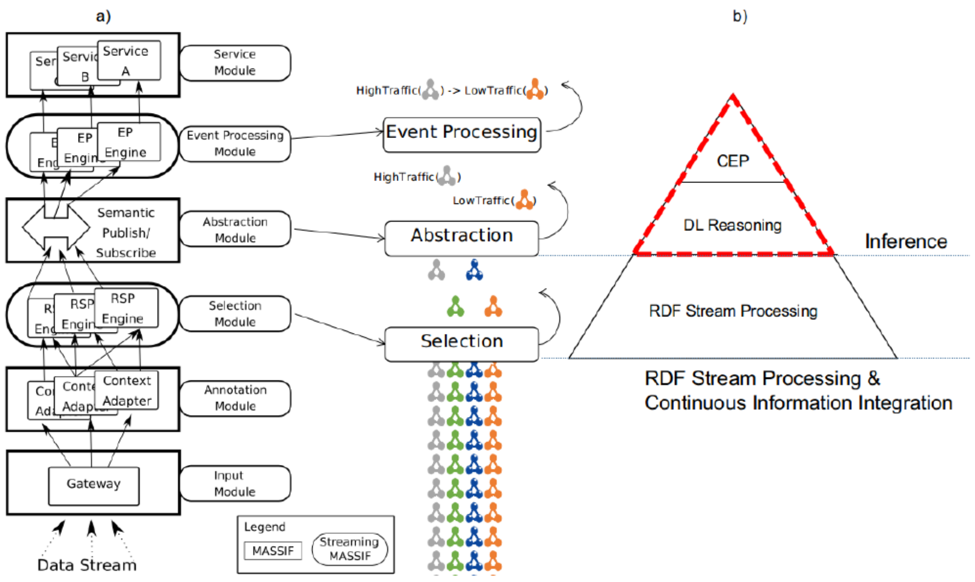


Fig. 3.3 a) MASSIF architecture. b) Covered parts w.r.t. the Cascading Reasoning pyramid. [11]

Being Streaming MASSIF mainly focused on the feasibility of a Cascading Reasoning approach, it presents the following limitations with respect to a solution embracing **A1** too: (i) Layers are embedded in the same artifact, therefore reasoners are not independently scalable and deployable. (ii) Communication between layers is specific for each step, so reasoners are not interchangeable in their order during planning. (iii) Does not targets a Raw Stream Processing directly directly, limiting therefore the change frequency it can handle.

In order to be able to handle Volume, Velocity and Variety at the same time, both architectural and applicative aspects have to be considered simultaneously. We envision a new approach that combines the best of both NoSR and CR, targeting the reasoners in the network for rewriting, each with different reasoning powers, collaborating as needed to produce results to streaming queries of arbitrary complexity and expressiveness. Tasks could be split both across *layers* based on the complexity of the sub-tasks and layers could be organized in subnetworks too. Agents belonging to the same layer of expressiveness would be able take advantage of data locality or just share the load if possible. Therefore we focus on the following research question which highlights the need of proving the feasibility of the approach:

> **Q2** *Can we combine multiple Stream Reasoners into a Network of Stream Reasoners, exploiting different expressive powers through Cascading Reasoning to tame Velocity and Volume on the WoD?*

Streaming MASSIF's approach to CR (Fig.3.2) targets only the top two level of the aforementioned pyramid, omitting to address Raw Stream Processing directly. Therefore, being able to handle complex reasoning for lower frequency changing RDF stream inputs, but in order to achieve the *Network of Stream Reasoner* vision over the Web of Big Data, also its basis have to be addressed.

We have identified two approaches as the endpoints of our solution space:

> **A1** *Network of Streaming Reasoners driven.*
>
> **A2** *Cascading Reasoning driven.*

Solutions totally embracing **A1** are distributed, highly reconfigurable systems of independent and interoperable agents, in which qualified resources can be used to execute tasks through planning. Therefore the available resources have to be discovered and catalogued. An example of this approach could be Semantic Web Services. Main drawbacks these systems have are the fact that discovery have to be handled dynamically, agents' qualification has to be flexible and therefore planning can be hard or even impossible.

Solutions oriented completely towards **A2** are monolithic systems of tightly coupled components, each with well defined and known characteristics, used in a prescribed order to optimally rewrite queries. Achieving theoretically high performances, but without the ability to scale horizontally and with limited expressiveness given the difficulties to extend them.

3.3 Microservices for CR

The NoSR strengths are that it is decentralized, reconfigurable, fault tolerant, but has drawbacks too, like the need to discover, catalogue and describe reasoners; CR can be more efficient, being layers tightly coupled, but is bounded in its expressiveness and reconfigurability. We envision hybrid solution, which we will call μCR (microservices Cascading Reasoning), in which the environment is closed as in CR, but still based on small independent reasoners performing single tasks each as in the NoSR, and thanks to their description can be reorganized in layers and used as rewriting target.

The Microservice architecture has become the de facto standard for modern Web application development. It structures an application as a collection of loosely coupled fine-grained services usually talking through light weight protocols, mostly HTTP. This approach allows each microservice to be developed, deployed and operated in parallel by different teams, enabling Continuous Integration and Continuous Deployment. These services need to cooperate to achieve their goals, therefore service discovery is a central issue in the microservice world. In fact, new services could be added by developers, needing to speak with old ones or vice versa.

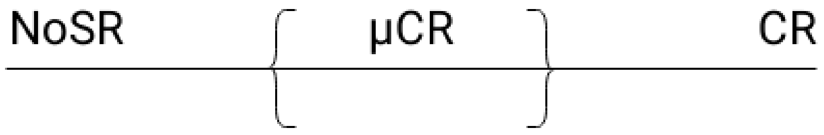


Fig. 3.4 The solution space and μCR positioning.

Therefore, the question we pose ourselves is the following:

Q3 *Can we overcome Cascading Reasoning limitations targeting the Network of Stream Reasoners for rewriting?*

Being the enabling of the NoSR mainly an architectural shift in the approach to RSP w.r.t. CR, we have identified an architectural stack (Fig.3.5) that should be covered in order to achieve our vision and answer **Q3**. The stack is composed of:

- **Infrastracture Orchestrator**: organizing and managing agents' bookkeeping and taking care of their orchestration.
- **Streaming Platform**: allowing streams' storage, access and processing. Split it in two components:
  1. **Stream Processing Engine**: which will be agent specific.
  2. **Data Infrastructure**: which instead will be common to all agents, no matter their implementation, allowing interoperability.

- **Domain Abstraction** through which to interpret the single streams' elements. Expressing and answering the information needs that our μCR infrastructure is designed to solve.

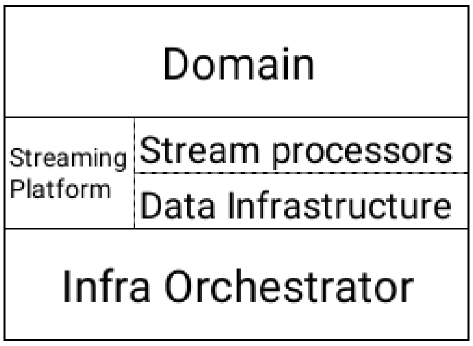


Fig. 3.5 Proposed stack to enable the NoSR.

The Data Infrastructure should allow Reasoners to be independently scalable and deployable, allow complete decoupling between consumers and producers and its necessary for part of the Network of Stream Reasoners to be able to target Raw Big Streams.

Analyzing Streaming MASSIF w.r.t. this stack we can say that it uses *Events* as Domain Abstraction. W.r.t. the two bottom layers Streaming MASSIF adopts an embedded solution, where all the components are part of a single artifact adopting OSGI[5], a *custom tailored component* as Orchestrator for the various layers and *multiple Stream Processors,* i.e., YASPER[38], HermiT and Esper. Not necessitating of any networking, they use the *main memory* as Data infrastructure to handle communication between layers.

We decided to focus mainly on the Streaming Platform layer and to produce an example of NoSR enabled Reasoner, adopting the industry standard regarding the Orchestrator layer and leaving as future works its instrumentation. We will describe a possible approach to it using the chosen orchestrator.

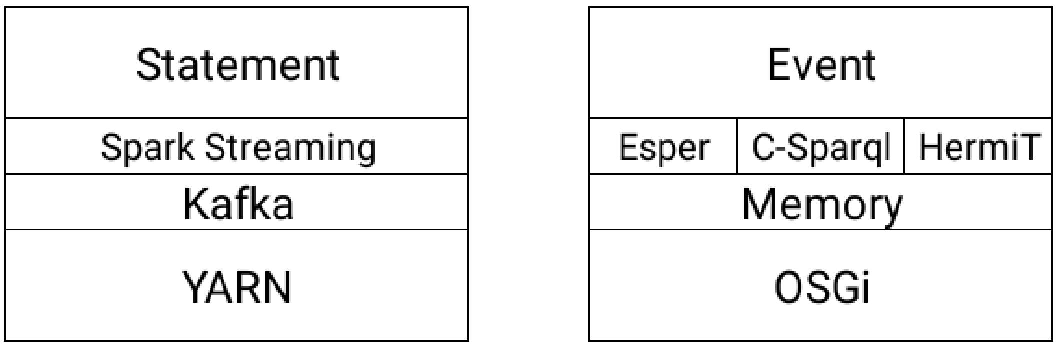
Once we will have chosen a common *Data Infrastructure,* our focus will be onto choosing which existing *Stream Processing Engine* could be used to create an agent covering also the lower part of the *Cascading Reasoning* pyramid, performing rewriting directly against Big Data Raw Stream Processing systems.

Few approaches have been already explored in the literature in this sense, targeting systems offering SQL-like interfaces. Morph-Stream [13], for example, rewrites queries to EPL and then execute them on Esper, which is a mainly vertically scalable system[6]. Another example of this approach has been proposed in Strider [32] where SPARQL operators

[5]https://www.osgi.org/

[6]At least in it's not enterprise version

are mapped to Spark SQL[7] ones and then the plan executed in microbatched mode, by doing so, Strider is able to handle Volume through horizontal scaling, but is limited in expressiveness and not easily extensible. Therefore w.r.t. our proposed stack, Strider has chosen Apache Yarn to orchestrate only the Stream Processor, Apache Spark Streaming, and not the Data Infrastructure, for which they use Apache Kafka as ingestion layer. Its formalization w.r.t. our stack can be seen in Figure 3.6a.



(a) Strider (b) MASSIF

Fig. 3.6 Proposed Architectural Stack already implemented.

The other possible approach we see is to target directly a Dataflow Stream Processing system, but no previous attempt has been proposed in literature for RDF streaming graph computation.

To choose the Data Infrastructure for the Streaming Platform we took inspiration from the Web, following Ian Robinson's principle for microservices architecture design: "Be of the Web, not behind the Web". Indeed, the Web has its roots in two important aspects: interoperability and composability.

Similarly, the idea of NoSR prescribes to build a network of "simple" reasoners, each performing a single specific task that can be combined to achieve greater goals and communicating through simple yet versatile interfaces. This approach strongly reminds the so called Unix Philosophy, which is at the core of all modern *nix systems, that recommends developing minimal tools communicating through the simplest format possible, i.e., text.

Moreover, Stonebreaker et al. [36] described a set of requirements for Stream Processing systems (see Section 2.2.1) and we followed them to guide our choice for a Stream Processor for the implementation of a reasoner enabled to be part of a Network of Reasoners, i.e., a Dataflow System [1] directly rather than a microbatched one. Part of these very same requirements then guided us in the choice of the target Streaming Platform.

[7]https://spark.apache.org/sql/

To be more specific, **R3** ("Handle Stream Imperfections") and **R8** ("Process and Respond Instantaneously") backed our choice for Stream Processor built on top of the Dataflow execution model, given that it allows natively to handle stream imperfections without sacrificing latency, allowing to have always updated results even in presence of late arrivals brought us to choose such approach over a microbatched one. This choice respects also **R1** ("Keep the Data Moving"), **R4** ("Generate Predictable Outcomes"), **R5** ("Integrate Stored and Streaming Data") and **R7** ("Partition and Scale Applications Automatically").

Indeed, producing timely results (**R8**) implies that results can be updated in the future, therefore downstream operators have to be able to manage late updates to already received messages, possibly indefinitely if not instructed differently. Therefore, a really important shift has to be made in order to embrace the Dataflow model: results are always correct w.r.t. the data received so far.

R2 ("Query using SQL on Streams") guided us in the choice of a standardized streaming language to encode queries the stream processor should execute.

The choice for the Streaming Platform has been guided by **R1** and **R5** again and **R6**("Guarantee Data Safety and Availability"), given that reasoners should be able to read from streams both as soon as a new element appears and as late as needed to fit their consumption pace.

In the following Chapter, we approach the design of a system satisfying the requirements we have intrisically exposed here, mainly focusing on the Data Infrastructure and Stream Processors layer. These requirements can be summerized as follows:

1. Data Infrastructure MUST:

   DI.R1 "Be of the Web, not behind the Web"

   DI.R2 Enable Interoperability providing a common communication layer

   DI.R3 Generate Predictable Outcomes [8]

   DI.R4 Guarantee Data Safery and Availability [9]

2. Stream Processors MUST:

   SP.R1 Keep the Data moving [10]

   SP.R2 Querying according to domain abstractions

[8] **R4** from Stonebreaker et al. [36]
[9] **R6** from Stonebreaker et al. [36]
[10] **R1** from Stonebreaker et al. [36]

SP.R3 Integrate Stored and Streaming Data [11]

SP.R4 Speak the same language, i.e., enabling interoperability

[11] **R5** from Stonebreaker et al. [36]

Chapter 4

Design

In this chapter, we explain the design choices we had to make and the reasons behind them. We have split the choice for a *Streaming Platform* in two parts: a common *Data Infrastructure* and a *Stream Processor* for each NoSR enabled stream reasoner.

4.1 Choosing the Data Infrastructure

In order to enable composability and interoperability of reasoners in a network a common data model and an agreement over a communication medium is needed. The Web's ability to scale has its roots in the *Client/Server* architectural pattern it has adopted since its first days, allowing servers to serve thousands of clients at a time. The evolution of this architectural pattern oriented toward Velocity is the Publish-Subscribe (*PubSub*) architecture, allowing publishers to not know in advance to which receivers their messages have to be sent and instead leaving it as subscribers' duty to declare their interests for specific kinds of events. This communication paradigm allows to achieve full decoupling of the communicating entities w.r.t. time, space, and synchronization [26]. It provides subscribers with the ability to express their interest in an event or a pattern of events, so to be notified of any event that matches their registered interests.

The PubSub architecture can sustain the communication in the Network of Stream Reasoners, providing in particular,it meets our need to decouple stream producers and consumers, enabling composability, and implies the necessity for a centralized system able to manage streams, store them and serve them asynchronously to multiple consumers.

Modern implementations of this architecture all have a unique component at their heart, a broker, handling all the communication. At best they can be deployed as a cluster, to be able to handle faults. This reflects our choice of a closed world of known reasoners over which to perform planning, what we have called *μCR*.

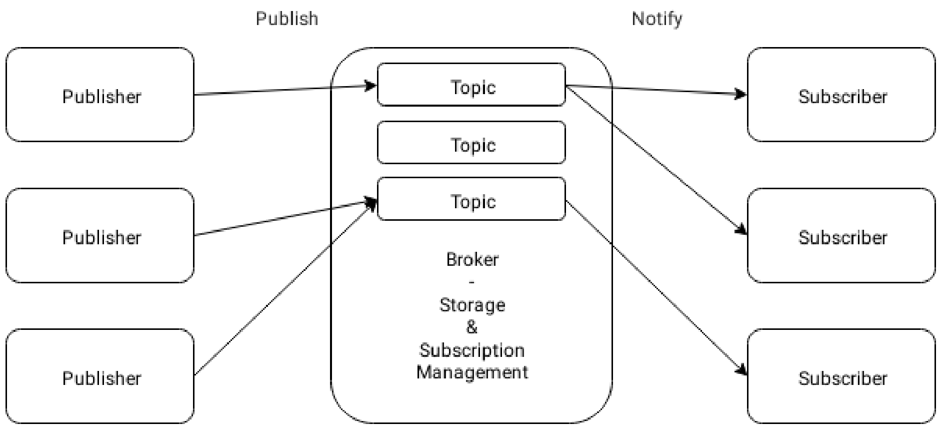


Fig. 4.1 A PubSub system example.

4.2 Choosing the Stream Processor

The second requirements for our streaming platform is enabling interoperability between reasoners.

Semantic Web technologies cover this providing a common data model, RDF, and logical languages to design and share vocabularies describing the domain. The RSP community extended such model to support streaming data by introducing RDF Streams. Therefore, we decided to adopt RDF Stream as common language, to be able to leverage the whole semantic technologies stack. Moreover, RSP-QL is a SPARQL streaming extension [22], that we can use for our stream processing layer.

Dell'aglio et al. [22] formalised this approach into RSP-QL, a model to describe the behaviour of the different existing RSP dialects. In [23] they presented an homonymous language which covers the needed constructs to tame velocity and variety simultaneously.

RSP-QL model too has its roots in the CQL [3] approach to stream processing.

CQL allows to switch between the streaming and relational world only through Stream-to-Relation and Relation-to-Stream operators as in Fig.4.2, where windowing is an example of the first kind. Due to it being focused mostly on demonstrating the feasibility of the RSP approach. They decided to put windowing operators, as separator between the streaming and static worlds. Such an approach allowed to reuse the whole SPARQL algebra without modifications, taking as inputs the windowed data in batches and executing queries as if they were static data.

This approach does not take advantage of any optimization possible in the streaming realm, executing queries only at fixed instant in time, defined through reporting policies.

Moreover, RSP-QL has already shown its limitations as an actual set of implementation guidelines as described by Tommasini et al. [38] obliging to choose between the ability to

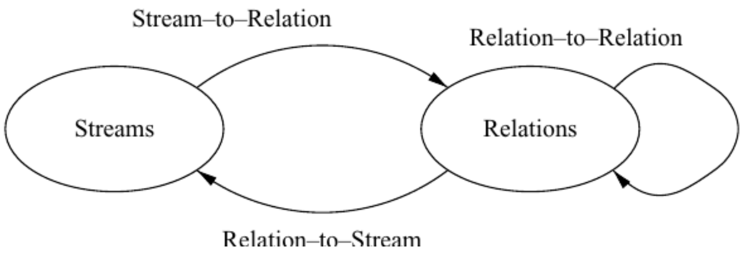


Fig. 4.2 CQL's interaction between stream and relation.

handle late arrivals and reporting latency. This lack is due to its interpretation of the CQL model influenced by Botan et al. SECRET model's [12] rigid execution semantic, separating between Streaming and Relational world and performing batched computation only at fixed time instants, named ticks, when data can be considered to be complete.

Most existing solutions only manage to achieve low latency, e.g. CQELS, or high throughput, e.g. C-SPARQL or MorphStream. CQELS represents a first step toward timely computation of results, implementing itself the execution logic instead of offloading it to an external "black box" DSMS systems such as Esper, it is able to perform adaptive query execution, but still without any horizontal scaleability.

We decided to address the Raw Stream Processing layer of the Cascading Reasoning pyramid proposed by Stuckenschmidt et al. [37], targeting directly a Big Data Stream Processing Engine for our PoC because the only example in literature to date of a CR system, Streaming MASSIF [11], did not target this layer. Being able to address also Big Data Streaming Systems, in our opinion, is a necessary step toward enabling the NoSR and being able to handle the increasing Volume and Velocity of the modern Web of Data.

A Dataflow approach would allow to achieve both low latency and high throughput, always having the most recent results, being the result updated at each new message. Obvious drawback is that of result correctness, that has to be relaxed in order to allow for later updates if needed. The Dataflow execution model could be applied with the CQL model to guarantee timely results in a horizontally scalable manner, still maintaining the well defined relational semantic of the R2R operators.

In the following, we describe how we mapped RSP-QL to the Datalofow execution model.

4.3 From RSP-QL to Dataflow

To be able to perform reasoning tasks on Big Data streams, as per the CR model [37], we have to find the maximum reasoning expressiveness that can still be pushed down to a

Big Data Stream Processing system. In literature, one of the proposed approaches consists into mapping SPARQL queries directly to other languages. This method has already been successfully adopted in scenarios such as OBDA (Ontology Based Data Access) where data, stored in one or more relational databases, can be queried in terms of high-level ontological model, hiding to the final user the internal schema of the relational data sources. This approach is named *query rewriting* and has already been shown to be applicable over RDF streams too [14]. This rewriting is performed through the usage of mappings from the relational schema to the RDF graphs and queries are translated to a SQL-like target languages, e.g. Ontop with R2RML mappings by Calvanese et al. [16].

Few existing works have already explored the translation and execution of SPARQL queries over RDBMS by constraining the data's schema to RDF tables, therefore producing algorithms to convert the SPARQL algebra into Relational Algebra over these data, e.g. Chebotko et al. [18]. To be able to adopt this approach too, we need to find a model bridging the Dataflow execution model and relational world, and a way to represent RDF streams that would allow it to be processed through Relational Algebra operators.

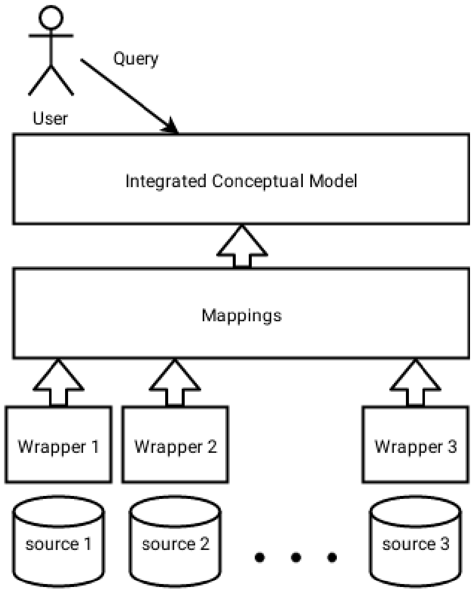


Fig. 4.3 The Architecture of Information Integration Systems.

Therefore, to execute UCQ RSPQL queries we had two options. The one adopted in the OBDA world in which the data is given in some specific form, usually relational, and therefore mappings are needed to be able to virtually query these data as RDF graphs. Mappings implies that knowledge of the original data and the windowing semantic in this context is still not clear in the literature. The other approach is to rethink the SPARQL query algebra and map it to another language, constraining the data underneath it, without needing any mapping. Given the lack of knowledge about the data in our context and the need for minimal human intervention, we opted for the second approach.

Chosen as our target the Dataflow execution model, to be able to execute SPARQL UCQs we need to map the SPARQL algebra extended with windowing to the Dataflow operators described by Akidau et al. [1]. Although, the Dataflow model itself has only few generic operators (mostly ParDo and GroupByKey) and is really just an execution model, not a language. For these reasons we decided to tackle first the choice of a specific Dataflow system and then work directly with its Domain Specific Language (DSL) operators.

As a bridging model we have found the Dual Streaming Model by Sax et al. [33]. This model presents Streams and Tables as two alternative views that can be always reconducted the one to the other. Tables can be seen as a materialization of a Stream and Streams as a stream of updates to a Table. This allows to think as for the CQL model in terms of R2R operators, but thanks to the Dataflow execution model, still be able to produce timely results without increasing latency and being constrained to complicated reporting policies with dubious semantics as in SECRET.

$$\begin{tikzcd}
\text{Table} \arrow[r, bend left, "\text{capture updates}"] & \text{Table Changelog Stream} \arrow[l, bend left, "\text{materialize updates}"]
\end{tikzcd}$$

Fig. 4.4 Duality of stream and tables.

Models such as CQL and SECRET completely ignored inconsistencies between physical and logical ordering of a stream and operators directly yielded an output data stream, while the treatment of operator state is considered to be an implementation detail, resulting into buffering data until their logical ordering has been established.

Sax et al. propose the Dual Streaming model in which operators results are updated continuously, allowing to drop any assumption over the consistency between logical and physical ordering. The model presents the notions of *tables*, *table changelog stream* and *record stream*. The static view on the current result of an operator is a *table*, updated for each new input record. Out of order records are processed updating the respective key similarly to in-order records, therefore not increasing the operator latency. The dynamic view on the result of an operator is a *table changelog stream* which consists of records representing updates to a table. The semantic of updates is defined over keys and timestamps, therefore record keys in this case have primary key semantics. Replaying a changelog stream allows to materialize the operator state as a table and that's the stream and table duality represented in Figure 4.4. Operators with temporal semantics need to maintain multiple versions at once, therefore a table is a collection of table versions and new versions can be added while older ones can be updated, triggering updates in downstream operators' corresponding versions. Record Streams, finally, are streams of facts, each having a unique id with an infinite key-space. Practically temporal operators achieve to handle windowing by enriching original keys with data about the assigned windows.

On top of this stream and table duality model all the usual relational operators have been defined, but due to the key-value nature of the underlying Dataflow execution model they mainly take in consideration the keys semantic, leaving the value's content handling up to the implementer. In order to be able to use the Dual Streaming model to process relational data in the value part of the messages, the model has to be further extended.

We took the approach by Chebotko et al. as inspiration to translate RSP-QL queries to Relational algebra, adapting it to the Dual Streaming Model and then mapping the produced relational logical plan into Kafka Streams' operators. This mapping will be straight forward given that the Dual Streaming Model allows to build the same logical plans on tables, with the advantage of timely update computation due to the Dataflow model it is built upon. But as said, the Dual Streaming model describes how to handle the relational operators w.r.t. keys, so we will need to define appropriate data structure and semantic also for values.

To sum up, thanks to the Streams and Tables duality we could be able to execute relational plans, produced from RSP-QL queries, but with a Dataflow execution model.

4.3.1 Representing RDF streams

In the Dataflow model stream items are seen as a timestamped key-value pair that flows through a Directed Acyclic Graph (DAG) of stateful and stateless operators. Therefore in order to be able to process RDF streams with a Dataflow system we have to decide how should its elements be represented in the key-value format.

Values can be of any form, but we have found two approaches in literature to represent RDF streams, both representing elements as tuples $< o, \tau >$ where τ is a timestamp and o

can be either an RDF triple [5]

$$[..., (\langle subj_i, pred_i, obj_i \rangle, \tau_i), (\langle subj_{i+1}, pred_{i+1}, obj_{i+1} \rangle, \tau_{i+1}), ...]$$

$$\forall i \geq 0 (\tau_i \geq \tau_{i+1})$$

or an RDF graph [24, 38]. Therefore, the choice of the Value format implies the choice of a corresponding Key.

Choosing to use graphs as Values implies to choose a vertex or edge in the graph or any combination of multiple of them as Key. Being the key used to compute the physical positioning of the data, in the case of stateful operation that need to operate on multiple streams simultaneously, it would be necessary to decide ahead of time which key would allow to match all the elements that have to match of the two streams. This to be sure they were consumed by the same processor, but in the context of the NoSR, where no prior knowledge of the consumers is given to the consumers, this approach can not be applied.

Keys in Dataflow systems, in a distributed scenario, are used to ensure that elements with a specific characteristic, encoded through the key, end up to the same processor and therefore can correctly concur to the computation of outputs. This is particularly useful for stateful operations, like simple counts or more complex joins. For example, joins can be executed only on co-partitioned streams, i.e., having the same key, otherwise we have no guarantees over the location of the data. If this is not the case a rekey operation is needed which plays in the Dataflow context a similar role to the reshuffling phase in MapReduce systems and its derivatives.

On the other hand, the alternative is to use timestamped triples, where the key can be one among Subject, Predicate or Object or a combination of them. It has already been shown in literature that triples can be indexed in six different ways, nonetheless we'll assume that all the streams will always have the Subject as Key, because in absence of prior knowledge about their successive usages, at least, we guarantee that simple star pattern can be recognized without any rekeying operation. If a diffent key will be needed we will always be able to perform a rekey obtaining the desired new key or if instead we want to leave the keying strategy up to the data producers we should associate the keying strategy to the stream and inform reasoners about it before consuming the stream and they would in case perform a rekey before consuming.

In the following we propose a model to represent RDF streaming data as relational data, compatibly with the Dataflow execution model, in order to then be able to execute relational logical plans over it.

The basic intuition behind our model is to see input and output streams as infinite tables with four columns $(subject, predicate, object, timestamp)$ and a key associated to each row, where new triples can only be appended and every update is streamed out to consumers, rows with the same key are guaranteed to end up in the same downstream processing node, Fig. 4.5.

Definition 8. A **Stream** s is a:

$$s = (K, [..., t_i, t_{i+1}, ...])$$

$$t_k = (\langle s_k, p_k, o_k \rangle, ts_k)$$

having $\langle s_k, p_k, o_k \rangle \in ((I \cup B) \times I \times (I \cup B \cup L))$ where I, B and L are sets of IRIs, blank nodes and literals respectively and $K(t_k) \in (s_k, p_k, o_k)$. K is a function s.t. $K(t_k)$ = key of the triple t_k.

	subj	pred	obj	ts
:s1	:s1	:p1	:o1	t0
:s2	:s2	:p2	:o1	t1
:s3	:s3	:p1	:o2	t2
:s1	:s1	:p3	:o1	t0

Fig. 4.5 A Stream in our representation.

In the following we describe the operators we have defined to cover the part of RSP-QL we were interested in. The main stream-to-stream (S2S) operator we have taken into consideration is the **Filter**, that takes a stream as input and produce a stream as output, maintaining schema and key of the original stream. W.r.t. the Dataflow model Filter is clearly just a specific instantiation of the ParDo operator, a stateless operator over a single element at a time that can run in parallel on different keys guaranteeing per-key ordering. Fig. 4.6

Definition 9. A **filter** can be defined over a stream $s = (K, [..., t_i, ...])$, in which case:

$$filter(f, s) = s' \text{ s.t. } s' = (K, [..., t_i, ...]) \text{ iff } f(t_i) = true$$

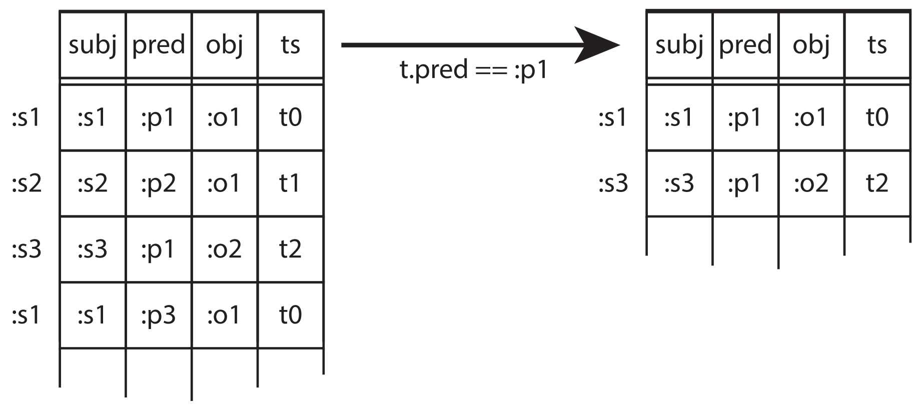


Fig. 4.6 A Filter over a Stream.

where f is a function checking a condition over a triple t_i and returning a boolean, e.g. $f(t_i) = (t_i.subj == ":s1")$

In order to be able to apply more complex operators, e.g. joins, we have to pass to the Tables. This is done via grouping by key or windowing, the latter assigning stream elements to zero or more windows and is always followed by a groupByKey operation where the key have been enriched with metadata regarding the window. As in the CQL model we can see this as a S2R operation. Windowing will take a Stream as input and produce what we call a Cube for each window, Fig. 4.7.

To allign RSP-QL to the stream and table model it is convenient to have a more intuitive view of data in table form w.r.t. their actual distribution due to the underlying Dataflow execution, therefore we introduce the next definition.

Definition 10. A **Cube** c is defined as:

$$c = (S, K, T)$$

Where S is a schema, K is a key belonging to the schema header S, T is a set of pairs $\langle key, t$ s.t. t is a table, intended as representation of Relations in the Relational Model, having each row r_i has $r_i.K = key$

Definition 11. Windowing is defined over a stream $s = (K, [..., t_i, t_{i+1}, ...])$, given a windowing function W, and a triple pattern $\tau = (s, p, o)$, where s,p and o can be variables or associated to a term, as follows:

$$Window(W, s) = [..., (W_i, c_i), ...]$$

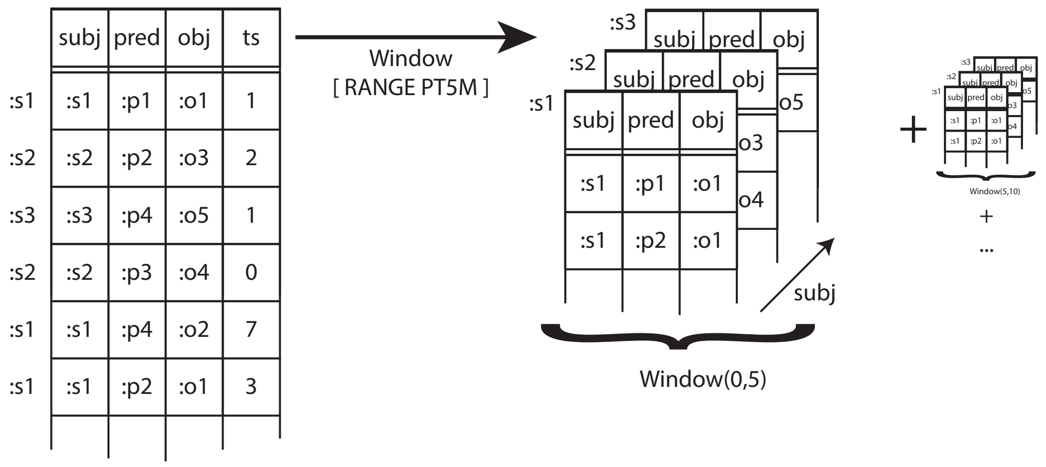


Fig. 4.7 Windowing applied to a filter to obtain a Cube.

having $W_i = (\alpha_i, \beta_i)$, α_i opening and β_i closing time associated with the i-th window. And c_i is the cube:

$$c_i = (S, K', T)$$

where the columns of $S = \tau$, $K' = K(\tau)$ and

$$T = [\langle k, \text{q(k)} \rangle \text{ for each } k \text{ in the set of the values } v \text{ s.t. } \exists r \text{ in } s\ (\ v = K(s)\ )\ ]$$

where $q(k)$ = "SELECT subj AS $\tau.s$, pred AS $\tau.p$, obj AS $\tau.o$ FROM s WHERE $K = k$ AND $\alpha_i \leq ts \leq \beta_i$".

A single Cube has a fixed schema s and is composed by multiple tables with the same schema s. A table for each value the key has assumed so far in the stream, e.g. if the input stream had key k with values $k1, k2, k3$, applying the window operator will produce a cube with three tables, one for each value of the key S. Each table of a Cube is associated to a single key and all the rows it contains have that exact key. Timestamps are dropped after windowing and out of order arrivals will be handled directly by the Kafka Streams APIs through Generations and Versions as explained by Sax et al. [33].

The Cube representation allows us to have a direct intuition of the physical distribution, in fact each table in a cube could end up in a different processor, on a different machine, based on the assigned key. As we'll see this will allow us to simplify the join logic still correctly computing results achieving the Dataflow model's timely updates, because no check will have to be done on the join condition as long as the keys are matching and therefore as long as the join sides are correctly keyed. In fact, this will allow us to perform joins without the need for a scan, similarly to the usage of indexes in RDBMS.

Specifically, the Join operator, in all its variants (Inner, Left, Outer), will take as input two Cubes and produce a single Cube, having as schema the merge of the 2 schemas of the input cubes, with columns corresponding to keys merged. The number of tables of the output Cube will depend on the variant of join, e.g. for Inner-joins it will be the size of the intersection between the values the key assume in both input cubes' schemas:

$$nr\_tables(output) = size(values\_key(input1) \cap values\_key(input2))$$

Rows of the output tables for key k will be the cartesian product of the rows of the input tables associated with key k in both input cubes, Fig. 4.8.

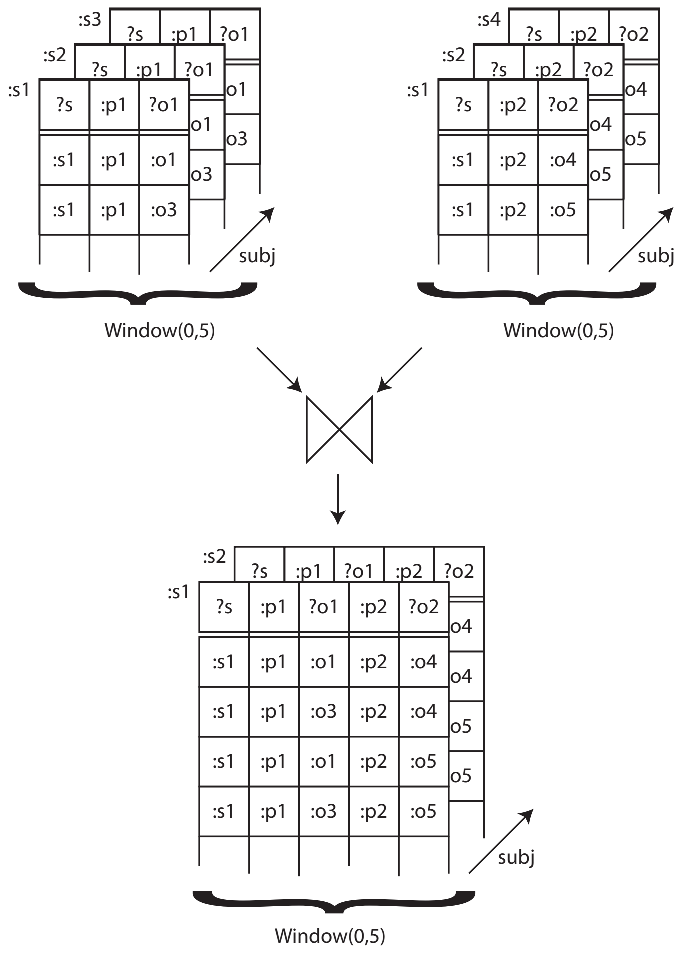


Fig. 4.8 A join between two cubes.

In order to be joined cubes have to share the same key, as said, but if this is not the case and we need to join them on a different column w.r.t. the key, a rekey operation is needed.

Definition 12. Rekey takes a Cube $C = (S, K, T)$ keyed by a column c, a new column c' and produce a Cube having the same schema and a number of tables equal to the number of values assumed by c', having c' as key, Fig. 4.9.

$$rekey(C, c') = C' = (S, K', T')$$

where $K' = c'$ and T' is formed by the same rows in all the tables in T, but grouped by the new key c'.

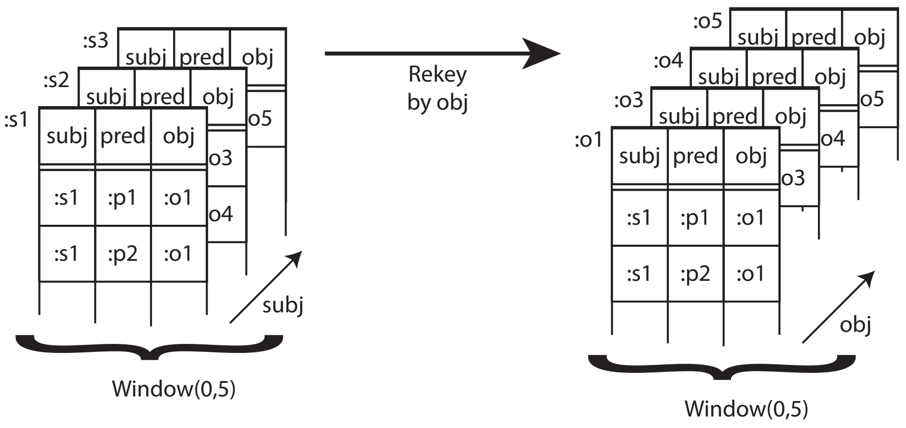


Fig. 4.9 A rekey operation over a cube.

Definition 13. Joins, of any type, can be performed only if the keys K_i and K_j of the two input cubes $c_i = (S_i, K_i, T_i)$ and $c_j = (S_j, K_j, T_j)$ are equal and they are defined as follow:

$$\text{join}(c_i, c_j) = c_l \text{ s.t. } c_l = (S', K, T')$$

where S' is the merge of the schemas S_i and S_j, $K = K_j = K_i$ and T' is the result of the join of t_i and t_j for each $\langle k_i, t_i \rangle$ in T_i, $\langle k_j, t_j \rangle$ in T_j in which $k_i = k_j$

A rekey operation over a cube $c = (S, K, T)$ given a new key K' produces a cube $c' = (S, K', T')$ where T' is equal to T but keyed by K' instead of K.

Definition 14. Filters can also be defined over a cube $c = (S, K, T)$:

$$filter(f, c) = c' \text{ s.t. } c' = (S, K, T')$$

where T' is defined as follow:

$$T' = (\langle k, \text{"SELECT * FROM t WHERE f holds"} \rangle \text{ for each } \langle k, t \rangle \text{ in T})$$

More in general, any classical Relational Algebra operator can be applied to a Cube by applying it to one table at a time; joins can be applied to two cubes if and only if they have the same key and aggregations can be performed only if the key of the cube is part of the grouping key.

In the next chapter we'll see how we have implemented these operators and what data structure we have chosen in order to reduce to the minimum the cost of incremental update, postponing as much as possible any computation.

Chapter 5

Implementation

In this Chapter, we will evaluate the feasibility of our design. Moreover, we will test the potential of our solution using SRBench, i.e. a benchmark for RSP query expressiveness.

In the proposed stack, the Data Platform has to be split in two parts, one common to all μCR Stream Reasoners, the Data Infrastructure, and one specific for each of them, the actual Stream Processor they are built upon.

In Chapter 3, to enable our μCR vision of a Network of cooperative Stream Reasoners we have proposed a stack (Figure 3.5). In Chapter 4, we have presented and motivated the design choices of a distributed PubSub for the Data Infrastructurea and a Dataflow system for the RDF Stream Processor.

In the following, starting from the bottom of our stack, we present tech technological choices, i.e., Kubernetes as, Apache Kafka as Data Infrastructure and Kafka Streams as a target Dataflow System.

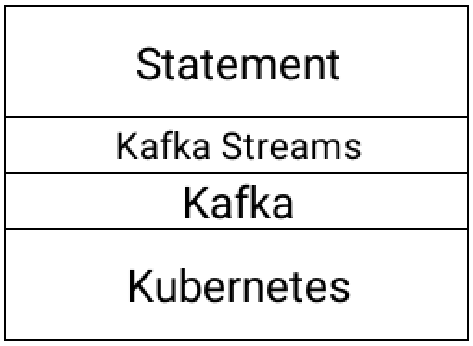


Fig. 5.1 Implemented stack choices enabling the $muCR$.

5.1 Technical Stack

In this Section, we will describe more in details the technologies we have used, Kubernetes, Apache Kafka, Apache Avro and the Kafka Streams APIs.

5.1.1 Kubernetes as Orchestrator

Given its rising adoption, we decided to adopt **Kubernetes** as *Infrastructure Orchestrator*. Kubernetes[1] is an open-source system for automating deployment, scaling, and management of containerized applications, born from a project by Google and then opensourced under the Cloud Native Computing Foundation (CNCF). It offers a declarative API through which define different kinds of workloads, with different characteristics, through manifests written in YAML. In order to be able to use Kubernetes as Orchestrator we have produced manifests to deploy a cluster of a configurable number of instance of Kafka brokers and Zookeper's instances, which can be then be used by other applications deployed in the Kubernetes using the resolvable service name.

5.1.2 Apache Kafka as Data Infrastructure

We have decided to adopt Apache Kafka, as we needed a PubSub distributed system, able to handle, store and serve streams of data for multiple Consumer and Producers.

Apache kafka reads and writes serializable messages. XML and JSON formats are widely adopted due to their flexibility, but parsing them, especially JSON, can impact performances, being instead AVRO a binary format. For this reason Apache Kafka offers a native integration with a data serialization system, **Apache Avro** [2], through an additional component included in the Confluent Platform [3], the **Schema Registry** [4].

Apache Avro is a data serialization system providing compact, fast, binary data serialization and code generation tools for multiple languages (e.g. Java) for improved performance. Through the definition of *schemas*, which have to be known by both the receiver and the sender for each message, data are fully self-describing. Being the schema shared among all the communicating parts, no per-value overhead is needed. When Avro data are stored in a file, their schema is stored along contestually, allowing any program to be able to read it even in absence of the schema. Avro can also be used for RPC, in which case schema are exchanged during the connection handshake, therefore correspondence

[1]https://kubernetes.io/
[2]https://avro.apache.org/
[3]https://www.confluent.io/product/confluent-platform/
[4]https://docs.confluent.io/current/schema-registry/docs

between same name, missing or extra fields can be easily resolved. Schemas are defined through JSON. Schemas allows to formally specify the data in the messages, with types, meaning and also human readable documentation.

```
{"namespace": "example.avro",
    "type": "record",
    "name": "user",
    "fields": [
        {"name": "name", "type": "string"},
        {"name": "favorite_number",  "type": "int"}
    ]
}
```

Listing 5.1 An Apache Avro schema example

The Schema Registry provides a serving layer for Avro schemas associated with messages in Kafka topics. It provides RESTful APIs for storing and retrieving them, allowing to keep a versioned history of each. Schema evolution is also provided, accordingly to configured compatibility settings. It also provides pluggable Serializers for Kafka clients, that at boot will read the correct schema from the schema registry and use it to deserialize data.

5.1.2.1 Data Infrastructure deployment on Kubernetes

The Kafka deployment consists of two StatefulSets, one handling the Kafka brokers' cluster and one handling the ZooKeeper instances' one.

Statefulsets are the Kubernetes API object used to handle stateful workloads, they allow to handle multiple replicas of the same application, keeping for each of them a unique identity w.r.t. network and volumes associated. Each instance of an application runs as a Pod, the minimum schedulable unit in Kubernetes, which in this case are created and restarted in case of failure by the associated StatefulSet.

The ZooKeeper cluster's is reachable from the Kafka Pods through a Service, which can be seen as a loadbalancer associated with an internally resolvable domain. A special kind of Service, named Headless Service, is needed for the instances to form a cluster, both for Kafka and ZooKeeper.

In order to able to use AVRO as a serialization format for messages in Kafka topics, as previously said, a Confluent Schema Registry is needed to share schemas and handle schema versioning. Given that the Schema registry uses Kafka topics as storage we do not need a StatefulSet to deploy it. Instead, we used a Deployment, another Kubernetes API object more suitable for stateless applications, able to handle rollouts and rollbacks between new and old versions of a same application through the usage of multiple ReplicaSets, which are responsible of keeping the number of replicas coherent with the desired state of the cluster. A single Service is needed to talk with the Schema registry from inside the Kubernetes cluster.

Other API objects were needed in order to be able to deploy correctly the aforementioned components, e.g. ServiceAccounts, Roles and Rolebindings for Zookeeper and Kafka pods to be able to talk with the Kubernetes API server to discover other members of their cluster thanks to the Endpoints associated to the Headless Service.

A graphic representation of the main components can be found in Fig.5.2.

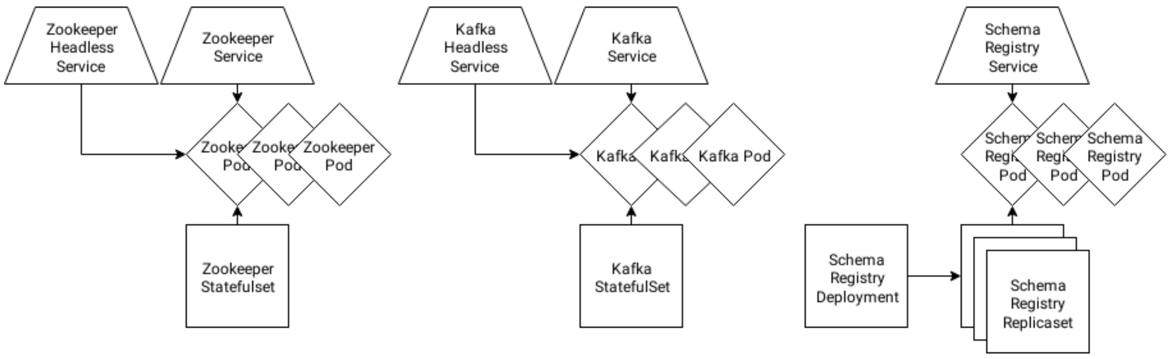


Fig. 5.2 The needed Kubernetes API objects to deploy Kafka, Zookeeper and the Schema Registry.

5.1.3 Kafka Streams APIs

Kafka Streams APIs offer a DSL to handle streams as both Streams and Tables, implementing the Dual Streaming model. Therefore, it offers most of the relational operators directly or that can be implemented depending on the value data structure, on top of the Dataflow execution model. It uses keys to partition topics, so stateful operations can only be performed on co-partitioned streams, otherwise a rekey operation is needed before being able to apply them.

Kafka Streams consists of a library that can be embedded in any Java application, used to process data from and to Kafka topics and allows to take advantage of many features automatically e.g. failure handling, consumer groups work partitioning and organization. Application having the same consumer group id will be automatically handled as a single consumer groups, therfore instances of such applications can be dynamically scaled up and down and the library will take care of repartitioning topics' partitions to consumers and splitting topologies' tasks among them whenever possible.

Topologies in Kafka Streams are split in tasks, each containing a part of the topology, called subtopology, joined by specific topics and that can be executed even on different hosts. Tasks distribution is transparently handled by the Kafka Streams library. Topologies are split in subtopologies whenever a rekey operation is needed. Rekeying through new kafka topics allows to reshuffle correctly the data by writing each message to the appropriate partition of the destination topic, then read by the right consumer which will receive all the messages for a certain subset of the key space. Subtopologies can be executed in parallel, on different instances of the same Kafka Streams application. Notably, the max number of parallel processors equal to the minimum number of partition of the input topic or the number of partitions for all the co-partitioned input topics.

5.2 Metamorphosis

We here present Metamorphosis, our proof of concept for a μCR enabled Dataflow RDF Stream Processor based on the Kafka Streams APIs.

Given a conjunctive RSP-QL query we will have to perform the following 4 steps:

S1 Parse the RSP-QL query and produce a parse tree.

S2 Convert the parse tree to a logical query plan in form of a DAG.

S3 Apply optional optimizations to the logical query plan.

S4 Convert to a physical query plan targeting Kafka Streams APIs.

5.2.1 Representing RDF in AVRO

First of all, we have to decide a schema for both the Key and the Values of the messages. *Keys* will be simply a wrapper around a String (Listing 5.4), while for the value we have more complex structures, the single triples will be represented as *SJSONTriple* (Listing 5.2) and will be used for input and output between the reasoners, while *SJSONTripleMap* (Listing 5.3), a wrapper around multiple lists of triples, will be used as internal schema in Methamorphosis.

With reference to the Cube and Streams model presented in Section 4.3.1, SJSON-TripleMap encodes a single table of a cube, the arrays it contains are the triples coming from the source represented by the key of the map. The data structure has been chosen with the purpose to delay as much as possible the joining and windowing computational cost and memory and network usage. In fact, these operations are not performed until

```
{ "type": "record", "name": "SJSONTriple",
            "fields": [
              { "name": "s", "type": "string" },
              { "name": "p", "type": "string" },
              { "name": "o", "type":
                    {   "type": "record", "name": "TripleObject",
                        "fields": [
                            { "name": "value",
                              "type": ["null", "string", "int", "long", "float" ]
                                  },
                            { "name": "datatype", "type": [ "null", "string" ] },
                            { "name": "type",
                              "type": {
                                "name": "Kind", "type": "enum",
                                "symbols": [ "literal", "uri", "bnode"]}}]
                                }},
              { "name": "ts", "type": "long" }]}
```

Listing 5.2 The AVRO schema for the SJSONTriple.

```
{
  "type": "record",
  "name": "SJSONTripleMap",
  "namespace": "phisco.streams.polimi.it.avro",
  "fields": [
    { "name": "data",
      "type": {
        "type": "map",
        "values": {
          "type": "array",
          "items": { "type": "SJSONTriple" }
            }}]}
```

Listing 5.3 The Value AVRO schema.

```
{
  "type": "record",
    "namespace" : "phisco.streams.polimi.it.avro",
    "name": "SJSONtKey",
    "fields": [
      {"name": "key", "type": "string"}
    ]
}
```

Listing 5.4 The Key AVRO schema.

strictly necessary, e.g., when projecting the data for outputting. Consequently, the cost of updating reduces, because updating becomes a simple append to the appropriate list.

5.2.1.1 Parsing RSP-QL with Antlr

We have decided to use RSP-QL [22] as input language in order to describe the SPARQL queries over RDF streams that the system shall be able to execute. Given the choice of constraining the possible queries to conjunctive queries we have targeted a subset of the RSP-QL grammar, that we have defined in EBNF (simplified version can be found in Listing A1 in Appendix).

In order to parse an RSP-QL query we have decided to adopt ANTLR[5] (ANother Tool for Language Recognition). ANTLR is a parser generator for reading, processing, executing, or translating structured text or binary files. Given an input context-free grammar in Extended Backus–Naur Form (EBNF) it generates the corresponding lexer and a parser for that grammar that can automatically build parse trees. ANTLR can also generate tree walkers, in the form of Visitors or Listener, that can be used to visit the nodes of those trees and execute application-specific code.

We have written the necessary parts of the RSP-QL grammar in EBNF, then we have let ANTLR generate the needed parser, lexer and implemented the logic for translating the syntax tree into a logical query plan **S2** as an ANTLR Visitor. We have chosen to implement a Visitor, because it allows to modify the visit order through custom code, while Listeners only allow to be notified whenever specific conditions hold during a depth first descent of the tree, e.g. entering or exiting a specific node.

5.2.1.2 Converting the parse tree into a logical query plan DAG

In the following, we describe the a simplified version of the algorithm we have applied to convert the parsed syntax tree generated by ANTLR in the previous step **S1** from an RSP-QL query into a logical plan in form of DAG. We'll assume a simplified RSP-QL grammar that can be found in Appendix A1 [6].

We will use the two queries in Listings 5.5 and 5.6 as running examples.

```
SELECT ?s ?o
FROM NAMED WINDOW :win ON STREAM :source [ RANGE PT1H]
WHERE{
    WINDOW :win { ?s :p1 :?o . }
}
```

Listing 5.5 Example query 1.

[5]https://www.antlr.org/

[6]Integral version: https://github.com/phisco/thesis/blob/master/Metamorphosys/src/main/antlr4/phisco/streams/polim

```
SELECT ?s1 ?s2 ?o
FROM NAMED WINDOW :win ON STREAM :source [ RANGE PT1H]
WHERE{
    WINDOW :win {
        ?s1 :p1 :?o.
        ?s2 :p2 ?o.
    }
}
```

Listing 5.6 Example query 2.

Starting from a selectQuery at each datasetClause corresponds a source node, reading from a graph or a windowed stream, in the latter case both a source node and a window node will have to be added, Listing 5.7.

```
forest = map : nodename -> node
filters = list of filters
roots = list of top level nodes
i = 0

for datasetClause in datasetClauseList:
    forest[iri] = SourceNode(datasetClause)
        # e.g. AnonymousGraph, NamedGraph or NamedWindow
```

Listing 5.7 Data set clauses.

At each triple pattern in the whereClause will correspond a filter node downhill to the corresponding source, Listing 5.8. The association follows rewriting techniques already adopted for example query federation. In our running examples, both the SourceNode ":source" and WindowNode ":win" are added, Figures 5.6 and 5.7.

```
for graphPatternNotTriples in groupGraphPattern:
    source = source.varOrIri
    for triplesSameSubject in graphPatternNotTriples:
        s = triplesSameSubject.varOrTerm
        for property in propertyListNotEmpty:
            p = property.verb
            for object in objectList:
                add filter(s, p, object, source) to forest, filters and roots
```

Listing 5.8 Filter creation.

Once we have added all sources and filters, we have to join the filters accordingly to the graph pattern they have to match. To recognize necessary joins, we build a Undirected Connected Graph (UCG) [35] in which vertexes are triple patterns (what we call filters) and edges are labelled with variables present in both the connected triples, meaning these two triple patterns will have to be joined over at least that variable (Listing 5.9). Notably, a pair of nodes can be connected through multiple edges, e.g. (?s ?p :o1) and (?s ?p :o2), in which case the joining condition will be specified accordingly. The generated UCGs for our running examples can be seen in Figures 5.3 and 5.4, the former consisting of a single node, while the latter of two nodes connected through an edge labeled "?o" which is the common variable among the two triple patterns.

```
joinGraph = UndirectedGraph { vertexes = filters , edges = variables }

for f in filters:
    joinGraph.addVertex(f)

# all the possible couples of vertices
for f1, f2 in (filters × filters − (f1,f2) s.t. f1 == f2) :
    for var in (vars(f1) ∩ vars(f2)):
        joinGraph.addEdge(f1, f2, var)
```

Listing 5.9 Building the join UCG.

Once the Graph has been built, we need to choose the order in which the joins have to be performed. The ordering will take into account a customizable cost function. In our current implementation, the cost function takes into account the number of rekeys operations needed by one or both the sides of the join in order to be joinable. Keying metadata are propagated and updated consequently while building the DAG.

Each time two nodes, $n1$ and $n2$, are joined in the UCG their two nodes have be merged into a single node $n3$ substituting $n1$ and $n2$ in all the remaining edges, until no more edges remain (Listing 5.10).

Given the two UCGs for our running examples, we can see that only the second query (Listing 5.6) needs a join over the variable ?o, therefore the node "J1" is added. Assuming the input streams to be partitioned by subject both sides of the join will need to be rekeyed, but this operation will be performed before the windowing node to reduce its cost.

After the last iteration of the previous step, the number of remaining nodes will be in equal to the number of connected components initially present in the UCG. If more than one node is present, they will all have to be joined through a particular kind of join, that we call an unconditional join. An unconditional join does not have a joining condition, but both sides of the join have to have been satisfied at least before forwarding the combined result.

Then, a projection node specified by the query in the selectClause is added and if the query is in construct form also a construct node. Both our running examples presented a

```

numberOfVertices = sizeOf(ConnectedComponentsList(joinGraph))
# numberOfVertices will be the final number of vertices obtained

while( joinGraph.vertices > numberOfVertices):

    lessExpensiveEdge = getLessExpensiveEdge(sortedEdges, joinCostFunction)

    v1, v2 = lessExpensiveEdge.vertices
    v1v2edges = edgesBetween(v1,v2)

    variables = set of all variables on edges between v1 and v2
    remove all edges in v1v2edges from joinGraph

    newJoinNode = join(v1, v2, variables, ConditionalJoin)
    forest[newJoinNode.name] = newJoinNode
    # update roots consequently, removing v1 and v2 and adding the new node

    joinGraph.add(newJoinNode)
    add edges ( newJoinNode, k, l) from newJoinNode to k with label l iff an edge
        between f1 or f2 and k with label l exists

    joinGraph.remove(v1)
    joinGrap.remove(v2)
```

Listing 5.10 Reduce vertices adding joins to logical plan DAG.

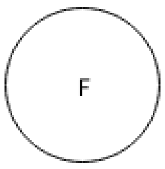


Fig. 5.3 Produced UCG for the example query 1.

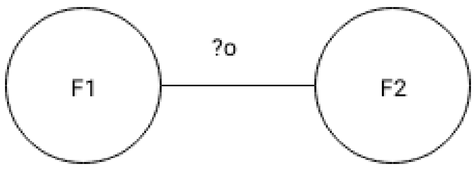


Fig. 5.4 Produced UCG for the example query 2.

select clause and therefore a projection node "P" is added to both (Figures 5.6 and 5.7). In the first query it will project the incoming triples to "?s ?o", while in the second to "?s1 ?s2 ?o".

```

DAG.head = project(vars or * from selectClause, DAG.head)

if Construct query:
    DAGHead = construct(construct graph pattern to be templetized, DAG.head)
```

Listing 5.11 Unconditional join, projection and construct node.

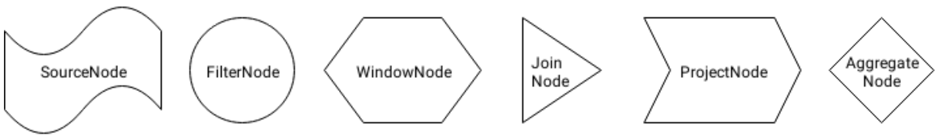


Fig. 5.5 Legend DAGs graphical representation.

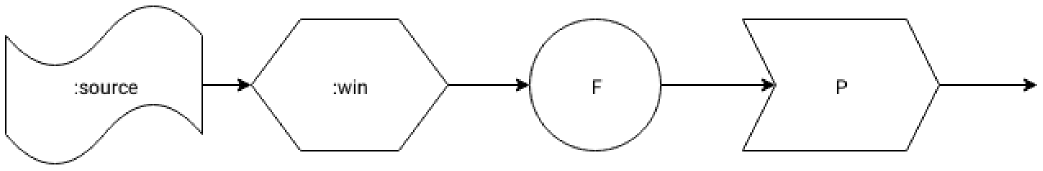


Fig. 5.6 DAG for example query 1.

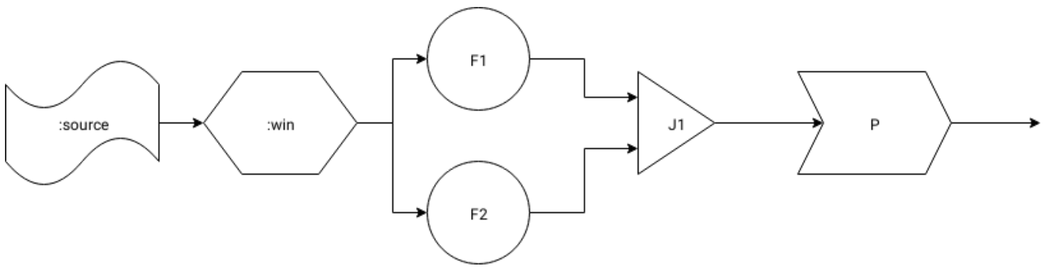


Fig. 5.7 DAG for example query 2.

5.2.1.3 Logical Plan Optimization

The DAG is then optimized through a simple rule engine. Rules are applied if specified preconditions are met until no more rules are applicable.

The main optimization performed at the moment is *PushFiltersBeforeWindows* which works as in Figure 5.8. W.r.t. our running examples the result of applying the PushFilters-BeforeWindows rule can be seen respectively in Figure 5.9 and 5.10.

5.2.1.4 From Logical to Physical plan

The generated DAG is then converted into a Kafka Streams Topology through a depth first visit, converting each node to its Streams APIs counterpart. Totally custom logic had to be implemented to handle correctly values using the classes generated by the aforementioned AVRO schemas.

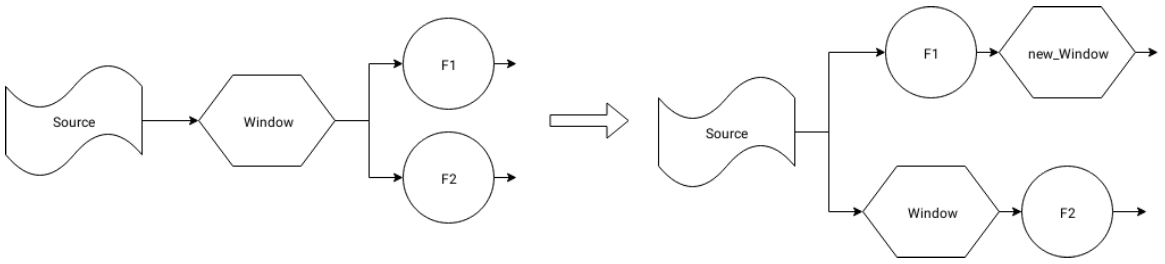


Fig. 5.8 Application of PushFilterBeforeWindows rule on node F1.

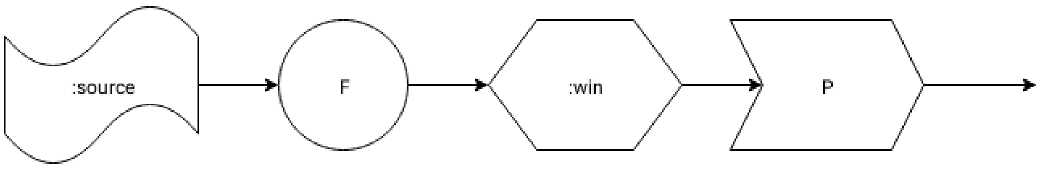


Fig. 5.9 Optimized DAG for example query 1.

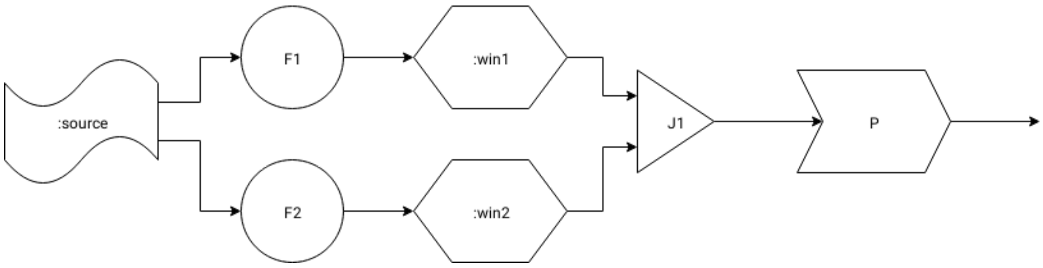


Fig. 5.10 Optimized DAG for example query 2.

The nodes of the Logical Plan are directly mapped to Kafka Streams APIs operators plus custom value handling logic. For example, Filter Nodes are implemented in Java, the only language officially fully supported by the APIs, in Listing 5.12.

5.2.1.5 Metamorphosis Software Organization

The software can be found on *GitHub* [34] and is split in five Packages:

1. **Antlr4**: which contains the classes and interfaces for lexer, parser, listener and visitor generated through a Maven plugin, from the provided RSP-QL grammar. Given a input stream the lexer produces the specified tokens which are then interpreted by the parser to build the parse tree, that can then be visited and used to perform any needed action in step **S1**.

2. **Algebra**: containing all the extension to the abstract class "RelNode", which will be used to build the DAG out of the parse tree in step **S2**. The available nodes are shown in Fig.5.12.

```java
package phisco.streams.polimi.it.executor;

@Accessors(fluent = true)
public class KafkaFilterNode extends KafkaNode {
    public KafkaFilterNode(KafkaExecutor executor, FilterNode node) {
        KafkaNode child = executor.nodes().get(node.children().get(0).name());
        if (child.stream() != null)
            this.stream(child.stream()
            .filter(getStreamingPredicate(node.filters())));
        else
            this.table( ... );
    }

    private Predicate<SJSONtKey, SJSONTriple> getStreamingPredicate(Filters
        filters){
        Predicate<SJSONtKey,SJSONTriple> p = (k,v) -> true;
        for (Map.Entry<Key, java.util.function.Predicate> e :
            filters.entrySet()) {
            final Predicate pf = p;
            switch (e.getKey()) {
                case S:
                    p = (k, v) -> pf.test(k, v) && e.getValue().test(v.getS());
                    break;
                case P:
                    p = (k, v) -> pf.test(k, v) && e.getValue().test(v.getP());
                    break;
                case O:
                    p = (k, v) -> pf.test(k, v) &&
                        e.getValue().test(v.getO().getValue().toString());
            }
        }
        return p;
    }
}
```

Listing 5.12 KafkaFilterNode implementation in Metamorphosis.

3. **Parser**: mostly composed of the implementation of the Visitor interface offered by Antlr4, "Gregor", which include all the logic to translate the parse tree produced by the parser into a DAG suitable to be executed **S2**. It also contains the abstract class "OptimizationRule" which can be extended to provide custom optimization rules to be applied on the produced DAG. To find the connected component in the join graph (UCG) needed by the algorithm we adopted an existing java graph library, jGraph [7].

[7]https://www.jgraph.com/

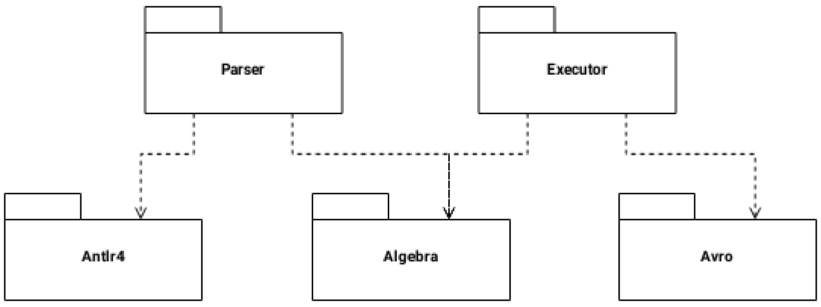


Fig. 5.11 Dependencies between the Metamorphosis packages.

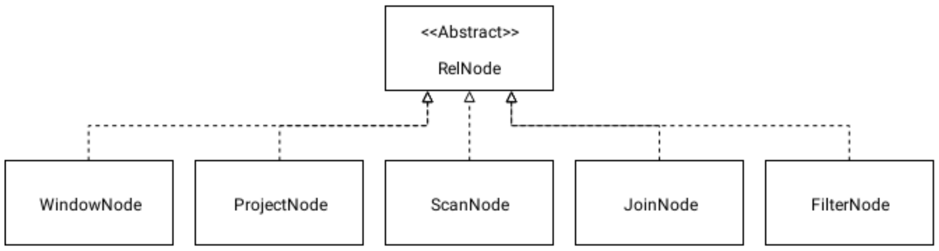


Fig. 5.12 Algebra Package nodes structure.

4. **Avro**: contains the classes generated by a Maven Plugin from the Avro schemas provided, which allows to serialize and deserialize the topics content.

5. **Executor**: an abstract class "Executor" and its extension "KafkaExecutor", plus all the Kafka Streams specific classes to translate the generic DAG into a Kafka topology and accomplish **S4**.

```
CharStream input = new ANTLRInputStream("... query ....")
RSP-QLLexer lexer = new RSP-QLLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
RSP-QLParser parser = new RSP-QLParser(tokens);
Gregor gregor = new Gregor();
gregor.visit(parser.queryUnit());
gregor.optimize(Arrays.asList(new PushFiltersBeforeWindows()));
Executor executor = new KafkaExecutor("executor-" + new
    Timestamp(System.currentTimeMillis().getTime());
executor.execute();
```

Listing 5.13 Example usage of Metamorphosijs to execute an RSP-QL query.

5.2.2 Metamorphosis Deployment on Kubernetes

To deploy the various Stream Reasoners, like Metamorphosis, various Kubernetes API objects can be used depending on the type of task to be performed. For long running tasks a Deployment can be more suitable, while Jobs can be used for tasks that should run to completion or CronJobs for Jobs that should run periodically completing a specific task each time. Seen that most of the Stream Reasoners should be long running applications interacting with the Kafka Cluster to read, process and write back data to specific topics, Deployments will be the most suitable choice. In fact, Metamorphosis has been set up as a Deployment, running a single query at a time, replicas will be handled out of the box by the Kafka Streams APIs, but the number of replicas will be limited by the number of partitions of the input topics.

A complete representation of the whol μCR stack can be seen in Figure 5.13.

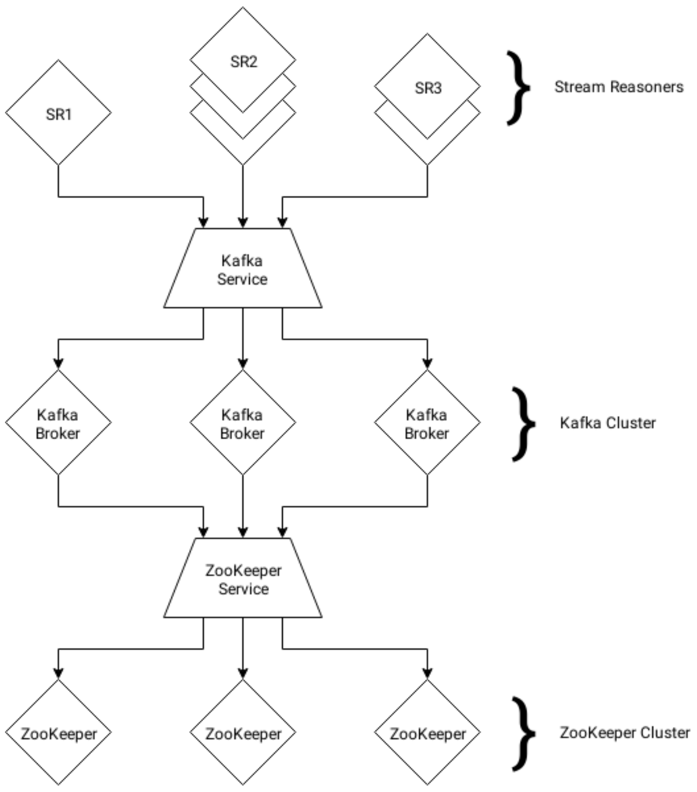


Fig. 5.13 Communication diagram with multiple Stream Reasoners.

5.3 Evaluation

Given our work mostly centering on the feasibility of applying the Dataflow approach through the usage of the Kafka Streams APIs to execute RDF Streaming queries, we decided to evaluate our system and model over the capabilities it has, underlining which limitations are due to the current implementation and which are due to intrinsic lacks of our model and approach.

5.3.1 SRBench

Zhang et al. proposed SRBench [39], a general-purpose benchmark for streaming RDF/SPARQL engines based on data from the Linked Open Data, the LinkedSensorData data set. This data set contains 1.7 bilion triples regarding the US weather data published by Kno.e.sis. Moreover it includes also data from DBpedia and GeoNames.

The SRBench benchmark consists of 17 queries, conceived to assess a system's capabilities at streaming RDF data, i.e.:

Q1 Get the rainfall observed once in an hour.

Q2 Get all precipitation observed once in an hour.

Q3 Detect if a hurricane has been observed.

Q4 Get the average wind speed at the stations where the air temperature is >32 degrees in the last hour, every 10 minutes.

Q5 Detect if a station is observing a blizzard.

Q6 Get the stations that have observed extremely low visibility in the last hour.

Q7 Detect stations that are recently broken.

Q8 Get the daily minimal and maximal air temperature observed by the sensor at a given location.

Q9 Get the daily average wind force and direction observed by the sensor at a given location.

Q10 Get the locations where a heavy snowfall has been observed in the last day.

Q11 Detecting if a station is producing significantly different observation values than its neighbouring stations.

Q12 Get the hourly average air temperature and humidity of large cities.

Q13 Get the shores in Florida, US where a strong wind, i.e., the wind force is between 6 and 9, has been observed in the last hour.

Q14 Get the airport(s) located in the same city as the sensor that has observed extremely low visibility in the last hour.

Q15 Get the locations where the wind speed in the last hour is higher than a known hurricane.

Q16 Get the heritage sites that are threatened by a hurricane.

Q17 Estimate the damage where a hurricane has been observed.

5.3.2 Implementable Queries

In the following we show which queries of the SRBench benchmark can be be implement and the motivations for the ones that are not:

Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**	**Q12**	**Q13**	**Q14**	**Q15**	**Q16**	**Q17**
Y	**N**	*Y*	*Y*	*Y*	*Y*	*Y*	*Y*	*Y*	*Y*	**N**	**N**	*Y*	*Y*	**N**	**N**	**N**

Q2, **Q15**, **Q16** and **Q17** are not executable due to the presence of an arbitrary length path query, which our model is not able to execute due to the missing recursion in the target language, Kafka Streams DSL.

Q11 and **Q12**, instead, are not executable because of the necessity of select subqueries, in fact only construct subqueries allow to pipe their output as source for the wrapping query.

5.3.3 Query Translation

We will now show the generated DAG for the queries **Q1** and **Q4** from SRBench [39].

For query **Q1** in Listing 5.14, the UCG in Fig. 5.14 is produced and then used to build the DAG in Fig. 5.15, where "srbench:observations" is the source stream, ":win" the windowing operator, "F*" are the filters, "J*" are all natural joins, given that filters rename input cubes schemas with triplepattern's terms; "P1" is a projection node, performing the projection to "?sensor ?value ?uom" as specified in the select clause. To be executed using the Kafka Streams APIs' Dataflow execution model the output of "J2" has to be rekeyed before being used as input for "J4", rekeying is not represented as a node in the DAG because it is an implementation detail due to how joins must be performed in the Dataflow execution model.

Q4 can be seen in Listing 5.15, the generated UCG in Figure 5.17 and the generated DAG in 5.18. Here three rekeys have to be performed: the output of "J2" before "J6" to match the key of the output of "J3"; the output of "J5" to match the key of "F10"; the output of both "J6" and "J7" will have to be rekeyed to be joined in "J8".

```
PREFIX om-owl:
    <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX srbench: <http://www.cwi.nl/SRBench/>
SELECT ?sensor ?val ?uom
FROM NAMED WINDOW ON STREAM srbench:observations [RANGE PT1H] AS :win
WHERE {
    WINDOW :win {
        ?observation om-owl:procedure ?sensor ;          // -> F1 (corresponding
            filters)
            a weather:RainfallObservation ;              // -> F2
            om-owl:result ?result .                      // -> F3
        ?result om-owl:floatValue ?val ;                 // -> F4
            om-owl:uom ?uom .                            // -> F5
    }
}
```

Listing 5.14 SRBench first query.

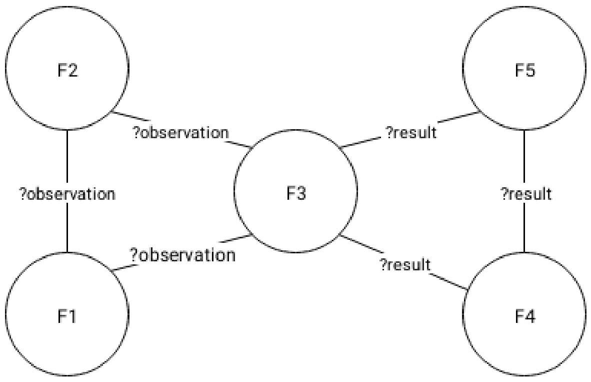


Fig. 5.14 Produced UCG for the SRBench query **Q1** in Listing 5.14.

5.3.4 Needed Tooling

Given that the LinkedSensorData [8] used by the SRBench benchmark was not in the format expected by our system, we had to write a producer able to stream the N-Triples serialized blob of RDF data and write it to a topic in the specified AVRO format in chronological order. We have chosen to split our solution in two components, both implemented in python.

The original data are split by huricanes, e.g. 2009 Bill or 2008 Ike; each of them is then split in multiple RDF files, with no clear ordering, therefore we load them in memory, query them to find the timestamps, sort them and then get the observations associated to that timestamp. To be as customizable as much as possible we have decided to split the

[8]http://wiki.knoesis.org/index.php/LinkedSensorData

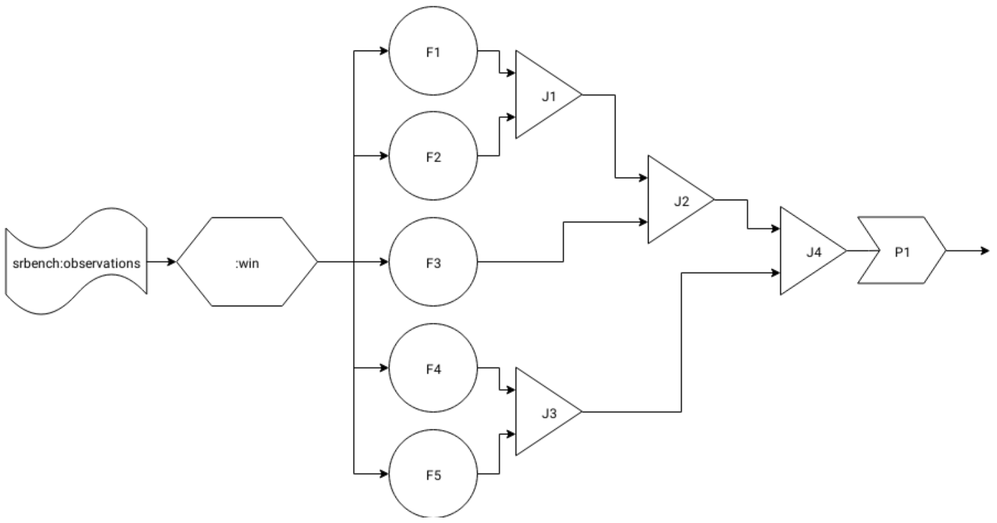


Fig. 5.15 DAG generated from the SRBench query **Q1** in Listing 5.14.

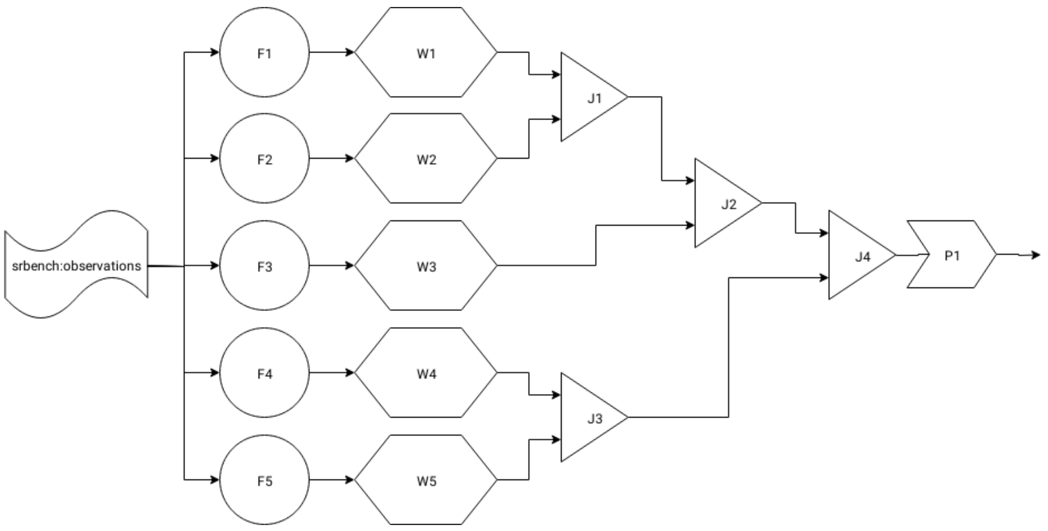


Fig. 5.16 DAG generated from the one in Fig.5.15 by applying PushFiltersBeforeWindows rule.

two queries and allow them to be passed as argument to the script. The first query can be seen in Listing 5.16 and the second in Listing 5.17 will be executed binding *?o* to each of the found timestamps.

So, the first component, given the two queries, a set of files and their serialization format, splits the files in batches and executes the two queries on each batch separately, assuming reasonably that data regarding a single observation are in same file or at least in the same batch. Each of the producers will write to a single file in json format, compatible with the StreamingTriples AVRO schema that will be used to serialize it to AVRO afterwards,

```
SELECT ?sensor (AVG(?windSpeed) AS ?averageWindSpeed)
               (AVG(?temp) AS ?averageTemperature)
FROM NAMED WINDOW :win ON STREAM srbench:observations [RANGE PT3H SLIDE PT10M]
WHERE {
  WINDOW :win {
    ?tempObs om-owl:procedure ?sensor ;                          // -> F1
                             a weather:TemperatureObservation ;  // -> F2
                             om-owl:result ?tempRes .            // -> F3
    ?tempRes om-owl:floatValue ?temp ;                           // -> F4
                         om-owl:uom ?uom .                       // -> F5
    FILTER(?temp > "32"^^xsd:float )                             // -> F6
    ?windSpeObs om-owl:procedure ?sensor ;                       // -> F7
                            a weather:WindSpeedObservation ;     // -> F8
                            om-owl:result ?result .              // -> F9
    ?result om-owl:floatValue ?windSpeed .                       // -> F10
}}
GROUP BY ?sensor                                                 // -> Agg
```

Listing 5.15 SRBench query **Q4** in RSP-QL.

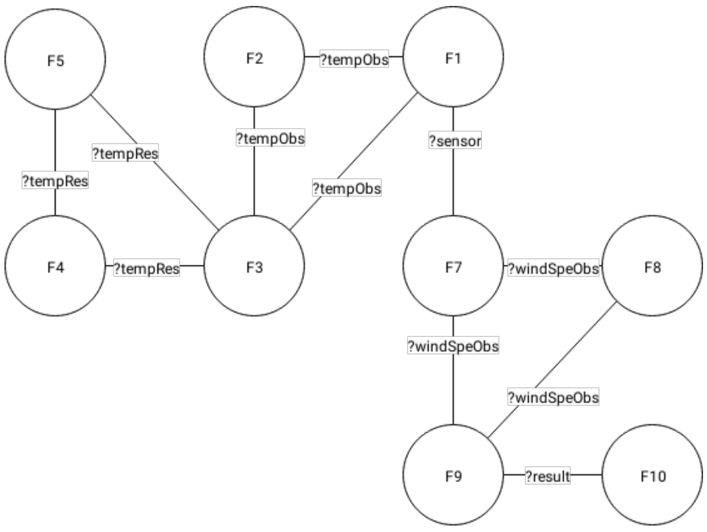


Fig. 5.17 UCG generated from SRBench's **Q4**

i.e. $\{"s": ..., "p": ..., "o": \{"value": ..., "type": ..., "literal": ...\}, "ts": ...\}$, and write all of them to *standard out*, if needed the output can be redirected into a file for later usage.

The second component will take directly from *standard in* a triple at a time, extrapolate the key and the timestamp and write each message to a configurable Kafka Topic using the specified AVRO schema.

The intended usage for these two components is to use the first to convert the whole dataset into a file containing a single triple per line and sort it by timestamp, if multiple process have been used, given that in such case ordering is not guaranteed. The first step

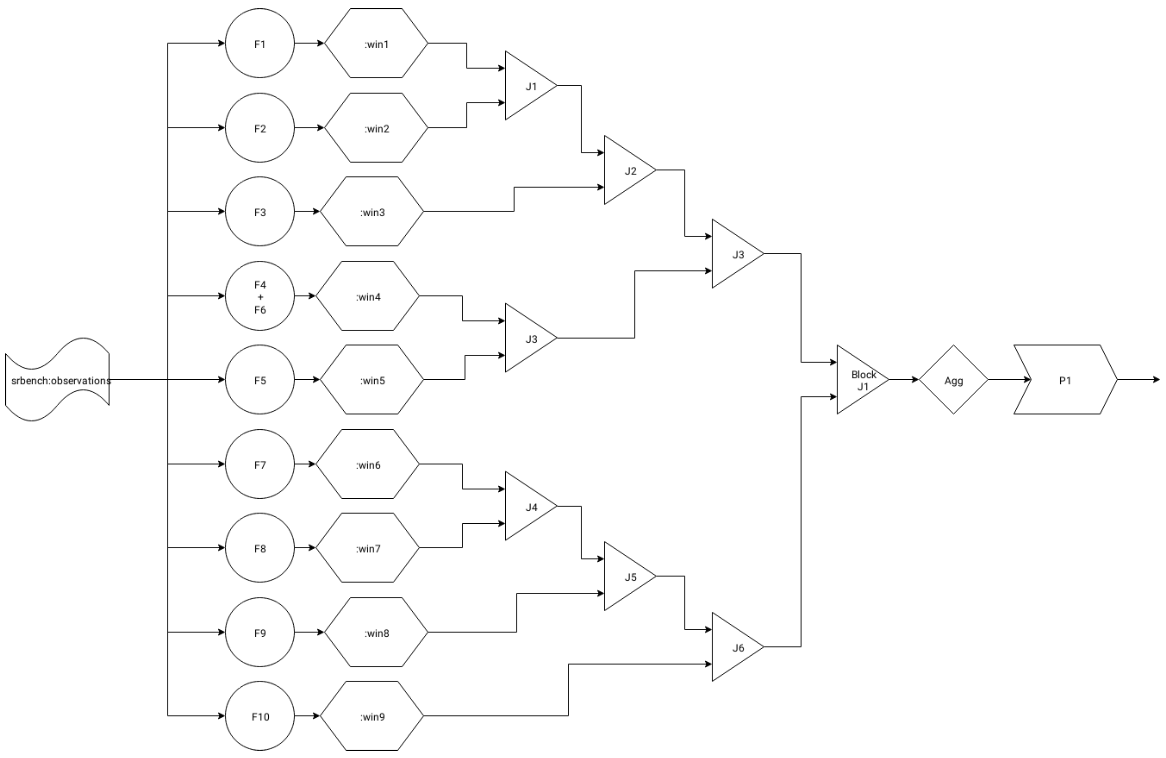


Fig. 5.18 Optimized DAG generated from SRBench's **Q4**

```
SELECT DISTINCT ?o
WHERE {
    ?s owl-time:inXSDDateTime ?o
}
```

Listing 5.16 Timestamp extraction query.

can be performed once and the second component can be deployed, as many times as needed, with the whole file to be streamed into a topic one triple at a time.

Both components have been containerized producing two distinct Docker container and the second component already comes with a Job manifest for deploying it on a Kubernetes cluster and a Makefile to run it locally.

The first component code can be seen in Listing A20 and the second in A21.

```
CONSTRUCT {
    ?s owl-time:inXSDDateTime ?o ;
        a ?o2 .
    ?s2 om-owl:samplingTime ?s ;
        a ?o3 ;
        om-owl:observedProperty ?o4 ;
        om-owl:procedure ?o5 ;
        om-owl:result ?s3 ;
        om-owl:samplingTime ?o6.
    ?s3 a ?o7 ;
        om-owl:floatValue ?o8 ;
        om-owl:uom ?o9 .
} WHERE {
    ?s owl-time:inXSDDateTime ?o ;
        a ?o2 .
    ?s2 om-owl:samplingTime ?s ;
        a ?o3 ;
        om-owl:observedProperty ?o4 ;
        om-owl:procedure ?o5 ;
        om-owl:result ?s3 ;
        om-owl:samplingTime ?o6.
    ?s3 a ?o7 ;
        om-owl:floatValue ?o8 ;
        om-owl:uom ?o9 .
}
```

Listing 5.17 Construct query.

Chapter 6

Conclusions and Future Work

In our Thesis Work, we proposed μCR, a novel approach to RDF Stream Processing and Stream Reasoning mixing Cascading Reasoning (CR) and Network of stream reasoners [31] to the the Microservice architecture. In order to enable our approach we have outlined a Stack (Figure 3.5), composed of an Orchestrator, a Streaming Platform, divided further in one single Data Infrastructure and multiple Stream Processors, and a Domain for Stream elements. We have proposed and motivated our choices w.r.t. this stack, Kubernetes as an Orchestrator, Apache Kafka as Data Infrastructure and Kafka Streams APIs as a Stream Processor. We have chosen to focus on producing a Proof of Concept (PoC) for a μCR enabled Stream Processor targeting the Raw Stream Processing layer of the CR pyramid (Figure 3.2) and therefore to target a Dataflow system [1]. To allow rewriting RSP-QL queries to a Dataflow system, mixing the approach on SPARQL query rewriting to relational algebra by Chebotko et al. [18] with the Dual Streaming Model by Sax et al. [33], we have proposed a model to execute queries on RDF streams compatible with the Dataflow distributed execution model.

6.1 Limitations and future work

In this section, we present the limitations we recognize w.r.t. our Work and enumerate the future work necessary to overcome these limitations or to further implement our vision.

L1 To evaluate our proposed PoC we have decided to use the SRBench benchmark [39]. Such benchmark focuses on the abilities of RSP dialects only, not checking correctness or performance in any way. To allow these really important aspects to be tested too an extension to SRBench has been proposed, CSRBench [1]. This second

[1]https://www.w3.org/wiki/CSRBench

benchmark offers a subset of the original queries and their results against which to test an RSP engine. Given that we have chosen to focus on describing our vision, explore its enabling pieces and then proposing a novel approach to target Dataflow systems for rewriting, we have decided to adopt the original benchmark based on query feasibility to be able to at first evaluate the capabilities of our proposed model, outlining what are the boundaries and limitations of our approach.

L2 We have only marginally explored the query rewriting aspect, deciding for our PoC to accept queries in the form of Union of Conjunctive Queries (UCQ), being it the rewriting output of recent systems such as Kyrie by Calbimonte et al. [14].

L3 In this Thesis Work, we have limited ourselves to use Kubernetes as an Orchestrator for workloads, therefore describing how to deploy the Data Infrastructure and Metamorphosis, our PoC. Although, in our vision the Orchestrator should also be able to handle more natively the Reasoners and the streaming query.

L4 The true power of the μCR is in the cooperation enabled by the centralized Data Infrastructure and common language we have chosen. Nonetheless, we have only shown how the Reasoners could interoperate and proposed PoC of a Stream Processor enabled to interoperate.

To overcome each of these Limitations respectively we formulated the following Future Work propositions:

FW1 We leave as future work the systematic and comparative performance and correctness benchmarking using CSRBench and COST [10].This will help us understand the bottlenecks and tune the PoC performance. Nonetheless, our hypothesis is that performance will be negatively influenced by the number of rekeying operations required, making such kind of systems suitable for simple queries on large volumes of data, condition that reflects exactly the CR model. Rekeying, as the reshuffling phase in other Big Data systems, will be one of the most costly w.r.t. query execution performance.

FW2 We have left as future work automating the integration between our PoC with Kyrie, but also to explore further ways to overcome the limitations of the only existing CR system to date, Streaming MASSIF [11], w.r.t. automatic query rewriting to reasoners with differents expressive powers.

FW3 In Appendix A4 we will propose how this could be achieved by extending Kubernetes APIs to handle custom resources, such as Reasoners, Data Infrastructure, Queries and Streams both incoming and outgoing. Which implementation is a necessary future work.

FW4 We leave as future work identifying existing systems that can be enabled to be used on the μCR network and if needed producing connectors for them to be able to consume and produce data from it. Therefore testing the real interoperability of our solution.

References

[1] Tyler Akidau, Eric Schmidt, Sam Whittle, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, and Frances Perry. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, August 2015. ISSN 21508097. doi: 10.14778/2824032.2824076. URL http://dl.acm.org/citation.cfm?doid=2824032.2824076.

[2] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[3] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006. ISSN 1066-8888, 0949-877X. doi: 10.1007/s00778-004-0147-z. URL https://link.springer.com/article/10.1007/s00778-004-0147-z.

[4] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 601–613, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4703-7. doi: 10.1145/3183713.3190664. URL http://doi.acm.org/10.1145/3183713.3190664.

[5] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-sparql: a continuous query language for rdf data streams. *Intl. J. Semantic Computing*, 4(01):3–25, 2010.

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

[7] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying RDF streams with C-SPARQL. *ACM SIGMOD Record*, 39(1):20, September 2010. ISSN 01635808. doi: 10.1145/1860702.1860705. URL http://portal.acm.org/citation.cfm?doid=1860702.1860705.

[8] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, page 4, May 2001. URL https://www-sop.inria.fr/acacia/cours/essi2006/Scientific%20American_%20Feature%20Article_%20The%20Semantic%20Web_%20May%202001.pdf.

[9] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data: The story so far. In *Semantic services, interoperability and web applications: emerging concepts*, pages 205–227. IGI Global, 2011.

[10] Christoph Boden, Tilmann Rabl, and Volker Markl. Distributed machine learning-but at what cost. In *Machine Learning Systems Workshop at the 2017 Conference on Neural Information Processing Systems*, 2017.

[11] Pieter Bonte, Riccardo Tommasini, Emanuele Della Valle, Filip De Turck, and Femke Ongenae. Streaming MASSIF: Cascading Reasoning for Efficient Processing of IoT Data Streams. *Sensors*, 18(11):3832, November 2018. ISSN 1424-8220. doi: 10.3390/s18113832. URL http://www.mdpi.com/1424-8220/18/11/3832.

[12] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J. Miller, and Nesime Tatbul. SECRET: a model for analysis of the execution semantics of stream processing systems. *Proceedings of the VLDB Endowment*, 3(1-2):232–243, 2010.

[13] Jean-Paul Calbimonte, Hoyoung Jeung, Oscar Corcho, and Karl Aberer. Enabling Query Technologies for the Semantic Sensor Web:. *International Journal on Semantic Web and Information Systems*, 8(1):43–63, January 2012. ISSN 1552-6283, 1552-6291. doi: 10.4018/jswis.2012010103. URL http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/jswis.2012010103.

[14] Jean-Paul Calbimonte, Jose Mora, and Oscar Corcho. Query Rewriting in RDF Stream Processing. In Harald Sack, Eva Blomqvist, Mathieu d'Aquin, Chiara Ghidini, Simone Paolo Ponzetto, and Christoph Lange, editors, *The Semantic Web. Latest Advances and New Domains*, volume 9678, pages 486–502. Springer International Publishing, Cham, 2016. ISBN 978-3-319-34128-6 978-3-319-34129-3. doi: 10.1007/978-3-319-34129-3_30. URL http://link.springer.com/10.1007/978-3-319-34129-3_30.

[15] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family. *Journal of Automated Reasoning*, 39(3):385–429, October 2007. ISSN 0168-7433, 1573-0670. doi: 10.1007/s10817-007-9078-x. URL http://link.springer.com/10.1007/s10817-007-9078-x.

[16] Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. Ontop: Answering SPARQL queries over relational databases. *Semantic Web*, 8(3):471–487, December 2016. ISSN 22104968, 15700844. doi: 10.3233/SW-160217. URL http://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/SW-160217.

[17] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache FlinkTM: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36 (4), 2015.

[18] Artem Chebotko, Shiyong Lu, and Farshad Fotouhi. Semantics preserving SPARQL-to-SQL translation. *Data & Knowledge Engineering*, 68(10):973–1000, October 2009.

ISSN 0169023X. doi: 10.1016/j.datak.2009.04.001. URL https://linkinghub.elsevier.com/retrieve/pii/S0169023X09000469.

[19] World Wide Web Consortium et al. RDF 1.1 Primer. 2014.

[20] Emanuele Della Valle, Stefano Ceri, Frank Van Harmelen, and Dieter Fensel. It's a streaming world! Reasoning upon rapidly changing information. *IEEE Intelligent Systems*, 24(6), 2009.

[21] Daniele Dell'Aglio, Emanuele Della Valle, Jean-Paul Calbimonte, and Oscar Corcho. Rsp-ql semantics: a unifying query model to explain heterogeneity of rdf stream processing systems. *Intl. J. on Semantic Web and Information Systems (IJSWIS)*, 10(4): 17–44, 2014.

[22] Daniele Dell'Aglio, Emanuele Della Valle, Jean-Paul Calbimonte, and Oscar Corcho. RSP-QL Semantics: A Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems. *International Journal on Semantic Web and Information Systems*, 10(4):17–44, October 2014. ISSN 1552-6283, 1552-6291. doi: 10.4018/ijswis.2014100102. URL http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/ijswis.2014100102.

[23] Daniele Dell'Aglio, Jean-Paul Calbimonte, Emanuele Della Valle, and Oscar Corcho. Towards a Unified Language for RDF Stream Query Processing. In Fabien Gandon, Christophe Guéret, Serena Villata, John Breslin, Catherine Faron-Zucker, and Antoine Zimmermann, editors, *The Semantic Web: ESWC 2015 Satellite Events*, volume 9341, pages 353–363. Springer International Publishing, Cham, 2015. ISBN 978-3-319-25638-2 978-3-319-25639-9. doi: 10.1007/978-3-319-25639-9_48. URL http://link.springer.com/10.1007/978-3-319-25639-9_48.

[24] Daniele Dell'Aglio, Emanuele Della Valle, Frank van Harmelen, and Abraham Bernstein. Stream reasoning: A survey and outlook: A summary of ten years of research and a vision for the next decade. *Data Science*, pages 1–25, October 2017. ISSN 24518492, 24518484. doi: 10.3233/DS-170006. URL http://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/DS-170006.

[25] Daniele Dell'Aglio, Emanuele Della Valle, Frank van Harmelen, and Abraham Bernstein. Stream reasoning: A survey and outlook. *Data Science*, 1(1-2):59–83, 2017.

[26] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.

[27] H. Karau. Unifying the open big data world: The possibilities of apache BEAM. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 3981–3981, December 2017. doi: 10.1109/BigData.2017.8258410.

[28] Houda Khrouf, Badre Belabbess, Laurent Bihanic, Gabriel Képéklian, and Olivier Curé. Waves: big data platform for real-time rdf stream processing. In *SR workshop at ISWC*, 2016.

[29] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[30] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, Abraham Bernstein, Lalana Kagal, Natasha Noy, and Eva Blomqvist, editors, *The Semantic Web – ISWC 2011*, volume 7031, pages 370–388. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-25072-9 978-3-642-25073-6. doi: 10.1007/978-3-642-25073-6_24. URL http://link.springer.com/10.1007/978-3-642-25073-6_24.

[31] Héctor Pérez-Urbina, Ian Horrocks, and Boris Motik. Efficient Query Answering for OWL 2. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Abraham Bernstein, David R. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, editors, *The Semantic Web - ISWC 2009*, volume 5823, pages 489–504. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-04929-3 978-3-642-04930-9. doi: 10.1007/978-3-642-04930-9_31. URL http://link.springer.com/10.1007/978-3-642-04930-9_31.

[32] Xiangnan Ren, Olivier Curé, Li Ke, Jeremy Lhez, Badre Belabbess, Tendry Randriamalala, Yufan Zheng, and Gabriel Kepeklian. Strider: an adaptive, inference-enabled distributed RDF stream processing engine. *Proceedings of the VLDB Endowment*, 10 (12):1905–1908, August 2017. ISSN 21508097. doi: 10.14778/3137765.3137805. URL http://dl.acm.org/citation.cfm?doid=3137765.3137805.

[33] Matthias J. Sax, Guozhang Wang, Matthias Weidlich, and Johann-Christoph Freytag. Streams and Tables: Two Sides of the Same Coin. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics - BIRTE '18*, pages 1–10, Rio de Janeiro, Brazil, 2018. ACM Press. ISBN 978-1-4503-6607-6. doi: 10.1145/3242153.3242155. URL http://dl.acm.org/citation.cfm?doid=3242153.3242155.

[34] Philippe Scorsolini. phisco/thesis v1.0.0, 2019. URL https://zenodo.org/record/2617307.

[35] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceeding of the 17th international conference on World Wide Web - WWW '08*, page 595, Beijing, China, 2008. ACM Press. ISBN 978-1-60558-085-2. doi: 10.1145/1367497.1367578. URL http://portal.acm.org/citation.cfm?doid=1367497.1367578.

[36] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, December 2005. ISSN 0163-5808. doi: 10.1145/1107499.1107504. URL http://doi.acm.org/10.1145/1107499.1107504.

[37] Heiner Stuckenschmidt, Stefano Ceri, Emanuele Della Valle, and Frank Van Harmelen. Towards expressive stream reasoning. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[38] Riccardo Tommasini and Emanuele Della Valle. Challenges & opportunities of rsp-ql implementations. In *WSP/WOMoCoE@ ISWC*, pages 48–57, 2017.

[39] Ying Zhang, Pham Minh Duc, Oscar Corcho, and Jean-Paul Calbimonte. SRBench: A Streaming RDF/SPARQL Benchmark. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Philippe Cudré-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jérôme Euzenat, Manfred Hauswirth, Josiane Xavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist, editors, *The Semantic Web – ISWC 2012*, volume 7649, pages 641–657. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-35175-4 978-3-642-35176-1. doi: 10.1007/978-3-642-35176-1_40. URL http://link.springer.com/10.1007/978-3-642-35176-1_40.

Appendix

A1 RSP-QL Grammar

```
selectQuery :  selectClause  datasetClause*  whereClause  solutionModifier? ;
selectClause : 'SELECT' ( ( resultVar )+ | STAR ) ;
resultVar :  var | ( '('  expression 'AS'  var')' ) ;
datasetClause : 'FROM' (  anonymousGraph=iri |  namedGraphClause |
    namedWindowClause ) ;
namedGraphClause : 'NAMED'  graphIRI=iri ;
namedWindowClause : 'NAMED' 'WINDOW' windowIRI=iri 'ON' streamIRI=iri '['
    windowDefinition ']' ;

// window
windowDefinition : physicalWindow | logicalWindow ;
physicalWindow :  physicalRange physicalStep? ;
physicalRange : 'ELEMENTS' INTEGER ;
physicalStep : 'STEP' INTEGER ;
logicalWindow : logicalRange logicalStep? ;
logicalRange : 'RANGE' DURATION ;
logicalStep : 'STEP' DURATION ;
DURATION : 'P' ( INTEGER 'Y' )? ( INTEGER 'M' )? ( INTEGER 'D' )? 'T'
            ( INTEGER 'H' )? ( mins=INTEGER 'M' )? ( INTEGER ( '.' INTEGER )? 'S'
                )? ;

whereClause : 'WHERE'  groupGraphPattern ;
groupGraphPattern : '{'  ( graphPatternSub )+ '}' ;
graphPatternSub : ( triplesSameSubject '.'? | graphPatternNotTriples );
graphPatternNotTriples :  graphGraphPattern | windowGraphPattern
graphGraphPattern : 'GRAPH'  varOrIri  '{' (triplesSameSubject '.'?)+ '}' ;
windowGraphPattern : 'WINDOW'  varOrIri  '{' (triplesSameSubject '.'?)+ '}' ;
varOrIri :  var | iri ;

triplesSameSubject :  triplesSameSubjectNoBlankNode ;

```

```
triplesSameSubjectNoBlankNode: varOrTerm propertyListNotEmpty;
propertyListNotEmpty : property ( ';' ( property )? )* ;
property : verb objectList ;
verb : varOrIri | TYPE ;
objectList : object ( ',' object )* ;
object : varOrTerm ;
varOrTerm : var | graphTerm ;
```

Listing A1 A simplified version of the RSP-QL grammar.

A2 SRBench Queries

```
PREFIX category: <http://dbpedia.org/resource/Category:>
PREFIX dbpprop: <http://dbpedia.org/property/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX gn: <http://www.geonames.org/ontology#>
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX srbench: <http://www.cwi.nl/SRBench/>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX wgs84_pos: <http://www.w3.org/2003/01/geo/wgs84_pos>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX yago: <http://dbpedia.org/class/yago/>
```

Listing A2 Common prefixes to all SRBench queries.

```
SELECT DISTINCT ?sensor ?value ?uom
FROM NAMED WINDOW :win ON STREAM srbench:observations [RANGE PT1H]
WHERE {
WINDOW :win {
    ?observation om-owl:procedure ?sensor ;
                 a weather:RainfallObservation ;
                 om-owl:result ?result .
    ?result om-owl:floatValue ?value ;
            om-owl:uom ?uom .
}}
```

Listing A3 SRBench query **Q1** in RSP-QL.

```
SELECT DISTINCT ?sensor ?value ?uom
FROM NAMED WINDOW :win ON STREAM <http://www.cwi.nl/SRBench/observations> [RANGE PT1H]
WHERE {
    WINDOW :win {
        ?observation om-owl:procedure ?sensor ;
                     rdf:type/rdfs:subClassOf* weather:PrecipitationObservation ;
                     om-owl:result ?result .
        ?result ?p1 ?value .
        FILTER( REGEX(STR(?p1), "value", "i") )
        OPTIONAL {
            ?result ?p2 ?uom .
            FILTER( REGEX(STR(?p2), "uom", "i") )
        }
    }
}
```

Listing A4 SRBench query **Q2** in RSP-QL.

```
ASK
FROM NAMED WINDOW :win ON STREAM <http://www.cwi.nl/SRBench/observations> [RANGE PT1H STEP PT10M]
WHERE {
    WINDOW :win {
        ?observation om-owl:procedure ?sensor ;
                     om-owl:observedProperty weather:WindSpeed ;
                     om-owl:result [ om-owl:floatValue ?value ] .
    }
GROUP BY ?sensor
HAVING ( AVG(?value) ≥ "74"^^xsd:float )
```

Listing A5 SRBench query **Q3** in RSP-QL.

```
SELECT ?sensor (AVG(?windSpeed) AS ?averageWindSpeed)
               (AVG(?temperature) AS ?averageTemperature)
FROM NAMED WINDOW :win ON STREAM srbench:observations [RANGE PT3H SLIDE PT10M]
WHERE {
  WINDOW :win {
  ?temperatureObservation om-owl:procedure ?sensor ;
                          a weather:TemperatureObservation ;
                          om-owl:result ?temperatureResult .
  ?temperatureResult om-owl:floatValue ?temperature ;
                     om-owl:uom ?uom .
  FILTER(?temperature > "32"^^xsd:float )
  ?windSpeedObservation om-owl:procedure ?sensor ;
                        a weather:WindSpeedObservation ;
                        om-owl:result [ om-owl:floatValue ?windSpeed ]  .
}}
GROUP BY ?sensor
```

Listing A6 SRBench query **Q4** in RSP-QL.

```
CONSTRUCT { ?sensor om-owl:generatedObservation [a weather:Blizzard] }
FROM NAMED WINDOW :win ON STREAM srbench:observations [RANGE PT3H SLIDE PT10M]
WHERE {
    WINDOW :win {
        ?sensor om-owl:generatedObservation [a weather:SnowfallObservation] ;
                om-owl:generatedObservation ?o1 ;
                om-owl:generatedObservation ?o2 .
        ?o1 a weather:TemperatureObservation ;
            om-owl:observedProperty weather:_AirTemperature ;
            om-owl:result [om-owl:floatValue ?temperature] .
        ?o2 a weather:WindObservation ;
            om-owl:observedProperty weather:_WindSpeed ;
            om-owl:result [om-owl:floatValue ?windSpeed] .
    }
}
GROUP BY ?sensor
HAVING ( AVG(?temperature) < "32"^^xsd:float  ∧  # fahrenheit
         MIN(?windSpeed) > "40.0"^^xsd:float ) #milesPerHour
```

Listing A7 SRBench query **Q5** in RSP-QL.

```
SELECT ?sensor
FROM NAMED WINDOW :win ON STREAM <http://www.cwi.nl/SRBench/observations> [RANGE PT1H]
WHERE {
    WINDOW :win {
        { ?observation om-owl:procedure ?sensor ;
                        a weather:VisibilityObservation ;
                        om-owl:result [om-owl:floatValue ?value ] .
        FILTER ( ?value < "10"^^xsd:float)  # centimeters
        }
        UNION
        { ?observation om-owl:procedure ?sensor ;
                        a weather:RainfallObservation ;
                        om-owl:result [om-owl:floatValue ?value ] .
        FILTER ( ?value > "30"^^xsd:float)  # centimeters
        }
        UNION
        { ?observation om-owl:procedure ?sensor ;
                        a weather:SnowfallObservation .
        }
    }
}
```

Listing A8 SRBench query **Q6** in RSP-QL.

```
SELECT DSTREAM DISTINCT ?sensor
FROM NAMED WINDOW :win ON STREAM <http://www.cwi.nl/SRBench/observations> [RANGE PT1H]
WHERE {
    WINDOW :win {
        ?sensor om-owl:generatedObservation ?observation .
    }
}
```

Listing A9 SRBench query **Q7** in RSP-QL.

```
SELECT ( MIN(?temperature) AS ?minTemperature ) ( MAX(?temperature) AS ?maxTemperature )
FROM NAMED WINDOW :win ON STREAM <http://www.cwi.nl/SRBench/observations> [RANGE PT24H]
FROM NAMED GRAPH :sensors <http://www.cwi.nl/SRBench/sensors>
WHERE {
    WINDOW :win {
        ?sensor om-owl:generatedObservation ?observation .
        ?observation om-owl:observedProperty weather:_AirTemperature ;
                        om-owl:result [ om-owl:floatValue ?temperature ] .
    }
    GRAPH :sensors {
        ?sensor om-owl:processLocation ?sensorLocation .
        ?sensorLocation wgs84_pos:alt "%Altitude%"^^xsd:float ;
                           wgs84_pos:lat "%Latitude%"^^xsd:float ;
                           wgs84_pos:long "%Longitude%"^^xsd:float .
    }
}
GROUP BY ?sensor
```

Listing A10 SRBench query **Q8** in RSP-QL.

```
SELECT ( IF(AVG(?windSpeed) < 1,  0,
          IF(AVG(?windSpeed) < 4,  1,
           IF(AVG(?windSpeed) < 8,  2,
            IF(AVG(?windSpeed) < 13, 3,
             IF(AVG(?windSpeed) < 18, 4,
              IF(AVG(?windSpeed) < 25, 5,
               IF(AVG(?windSpeed) < 31, 6,
                IF(AVG(?windSpeed) < 39, 7,
                 IF(AVG(?windSpeed) < 47, 8,
                  IF(AVG(?windSpeed) < 55, 9,
                   IF(AVG(?windSpeed) < 64, 10,
                    IF(AVG(?windSpeed) < 73, 11, 12) ))))))))))))
         AS ?windForce )
       ( AVG(?windDirection) AS ?avgWindDirection )
FROM NAMED WINDOW :win ON STREAM <http://www.cwi.nl/SRBench/observations> [RANGE PT24H]
FROM NAMED GRAPH :sensors <http://www.cwi.nl/SRBench/sensors>
WHERE {
    WINDOW :win {
        ?sensor om-owl:generatedObservation ?o1 ;
                om-owl:generatedObservation ?o2 .
        ?o1 om-owl:observedProperty weather:_WindSpeed ;
                om-owl:result [ om-owl:floatValue ?windSpeed ] .
        ?o2 om-owl:observedProperty weather:_WindDirection ;
            om-owl:result [ om-owl:floatValue ?windDirection ] .
    }
    GRAPH :sensors {
        ?sensor om-owl:processLocation ?sensorLocation .
        ?sensorLocation wgs84_pos:alt "%Altitude%"^^xsd:float ;
                        wgs84_pos:lat "%Latitude%"^^xsd:float ;
                        wgs84_pos:long "%Longitude%"^^xsd:float .
    }
}
GROUP BY ?sensor
```

Listing A11 SRBench query **Q9** in RSP-QL.

```
SELECT ?lat ?long ?alt
FROM NAMED WINDOW :win ON STREAM srbench:observations [RANGE PT1H]
FROM NAMED GRAPH :sensors <http://www.cwi.nl/SRBench/sensors>
WHERE {
  WINDOW :w {
     ?sensor om-owl:generatedObservation [a weather:SnowfallObservation] .
  }
  GRAPH :sensors {
       ?sensor om-owl:processLocation ?sensorLocation .
       ?sensorLocation wgs84_pos:alt ?alt ;
                       wgs84_pos:lat ?lat ;
                       wgs84_pos:long ?long .
                  }
}
```

Listing A12 SRBench query **Q10** in RSP-QL.

```
SELECT DISTINCT ?sensor
FROM NAMED WINDOW :win ON STREAM <http://www.cwi.nl/SRBench/observations> [RANGE PT1H]
FROM <http://www.cwi.nl/SRBench/sensors>
WHERE {
  ?sensor om-owl:generatedObservation ?observation ;
          om-owl:hasLocatedNearRel [om-owl:hasLocation ?nearbyLocation] .
  ?observation a ?observationType ;
               om-owl:observedProperty ?observationProperty ;
               om-owl:result [ om-owl:floatValue ?value ] .
  { SELECT (AVG(?value2) AS ?avgValue)
    WHERE {
      ?sensor2 om-owl:generatedObservation ?observation2 ;
               om-owl:hasLocatedNearRel [om-owl:hasLocation ?nearbyLcation2] .
      FILTER ( sameTerm(?nearbyLocation, ?nearbyLocation2) )
      ?observation2 a ?observationType ;
                    om-owl:observedProperty ?observationProperty ;
                    om-owl:result [ om-owl:floatValue ?value2 ] .
    }
  }
  FILTER ( ABS(?value - ?avgValue) / ?avgValue > "0.10"^^xsd:float)
}
```

Listing A13 SRBench query **Q11** in RSP-QL.

```
SELECT ?name ( AVG(?temperature) AS ?avgTemperature ) ( AVG(?humidity) AS ?avgHumidity )
FROM NAMED WINDOW :win ON STREAM <http://www.cwi.nl/SRBench/observations> [RANGE PT24H STEP PT1H]
FROM <http://www.cwi.nl/SRBench/sensors>
FROM <http://www.cwi.nl/SRBench/geonames>
WHERE {
  ?sensor om-owl:generatedObservation ?temperatureObservation;
          om-owl:generatedObservation ?humidityObservation;
          om-owl:hasLocatedNearRel [ om-owl:hasLocation ?nearbyLocation ] .
  ?temperatureObservation om-owl:observedProperty weather:_AirTemperature ;
                          om-owl:result [ om-owl:floatValue ?temperature ] .
  ?humidityObservation om-owl:observedProperty weather:_RelativeHumidity ;
                       om-owl:result [ om-owl:floatValue ?humidity ] .
  { SELECT ?name
    WHERE {
      ?nearbyLocation gn:featureClass ?featureClass ;
                      gn:name • gn:officialName ?name ;
                      gn:population ?population .
      FILTER ( ?population > 15000 ∧ REGEX(?featureClass, "P" , "i") )
    }
  }
  UNION
  { SELECT ?name
    WHERE {
      ?nearbyLocation gn:parentFeature+ ?parentFeature .
      ?parentFeature gn:featureClass ?parentClass ;
                     gn:name • gn:officialName ?name ;
                     gn:population ?parentPopulation .
      FILTER ( ?parentPopulation > 15000 ∧ REGEX(?parentClass, "P" , "i") )
    }
  }
}
GROUP BY ?name
```

Listing A14 SRBench query **Q12** in RSP-QL.

```
SELECT ?shoreName ?lat ?long
       ( IF(AVG(?windSpeed) < 31, 6,
           IF(AVG(?windSpeed) < 39, 7, IF(AVG(?windSpeed) < 47, 8, 9)))
         AS ?windForce )
FROM NAMED WINDOW :win ON STREAM <http://www.cwi.nl/SRBench/observations> [RANGE PT1H]
FROM NAMED GRAPH :sensors <http://www.cwi.nl/SRBench/sensors>
FROM NAMED GRAPH :geonames <http://www.cwi.nl/SRBench/geonames>
WHERE {
    GRAPH :geonames {
        ?shore gn:featureClass ?shoreClass ;
               gn:name•gn:officialName ?shoreName ;
               gn:parentFeature+ ?florida ;
               wgs84_pos:lat ?lat ;
               wgs84_pos:long ?long .
        ?florida gn:name•gn:officialName ?floridaName .
        FILTER ( ( REGEX(?shoreClass, "L.CST" , "i") ∨ # coast
                   REGEX(?shoreClass, "T.BCH" , "i") ∨ # beach
                   REGEX(?shoreClass, "T.SHOR" , "i") ) ∧ # shore
                   REGEX(?floridaName, "Florida", "i") )

    }
    GRAPH :sensors {
            ?sensor om-owl:hasLocatedNearRel [ om-owl:hasLocation ?shore ] .
    }
    WINDOW :win {
        ?sensor om-owl:generatedObservation ?observation.
        ?observation om-owl:observedProperty weather:_WindSpeed ;
                     om-owl:result [ om-owl:floatValue ?windSpeed ] .
        FILTER ( 25 ≤ ?windSpeed ∨ ?windSpeed ≤ 54 ) # milesPerHour
    }
}
GROUP BY ?shoreName ?lat ?long
```

Listing A15 SRBench query **Q13** in RSP-QL.

```
SELECT DISTINCT ?airportName ?lat ?long
FROM NAMED WINDOW :win ON STREAM <http://www.cwi.nl/SRBench/observations> [RANGE PT1H]
FROM NAMED GRAPH :sensors <http://www.cwi.nl/SRBench/sensors>
FROM NAMED GRAPH :geonames <http://www.cwi.nl/SRBench/geonames>
WHERE {
  GRAPH :geonames {
        ?airport gn:featureClass ?airportClass ;
                 wgs84_pos:lat ?lat ;
                 wgs84_pos:long ?long ;
                 gn:name•gn:officialName ?airportName ;
                 gn:parentFeature+ ?city .
      ?city gn:featureClass ?cityClass .
      FILTER ( REGEX(?airportClass, "S.AIRP" , "i") ∧
                 REGEX(?cityClass, "P" , "i") )
  }
  GRAPH :sensors { ?sensor om-owl:hasLocatedNearRel [ om-owl:hasLocation ?city ] . }
  WINDOW :win {
        ?sensor om-owl:generatedObservation ?observation.
      { ?observation om-owl:procedure ?sensor ;
                        a weather:VisibilityObservation ;
                        om-owl:result [om-owl:floatValue ?value ] .
        FILTER ( ?value < "10"^^xsd:float)  # centimeters
      }
      UNION
      { ?observation om-owl:procedure ?sensor ;
                        a weather:RainfallObservation ;
                        om-owl:result [om-owl:floatValue ?value ] .
        FILTER ( ?value > "30"^^xsd:float)  # centimeters
      }
      UNION
      { ?observation om-owl:procedure ?sensor ;
                        a weather:SnowfallObservation .
      }
  }
}
```

Listing A16 SRBench query **Q14** in RSP-QL.

```
SELECT ?lat ?long ?alt ( AVG(?windSpeed) AS ?avgWindSpeed )
FROM NAMED WINDOW :win ON STREAM <http://www.cwi.nl/SRBench/observations> [RANGE PT1H]
FROM NAMED GRAPH :sensors <http://www.cwi.nl/SRBench/sensors>
FROM NAMED GRAPH :dbpedia <http://www.cwi.nl/SRBench/dbpedia>
WHERE {
    WINDOW :win {
        ?observation a weather:WindspeedObservation ;
                        om-owl:procedure ?sensor ;
                        om-owl:result [ om-owl:floatValue ?windSpeed ] .
    }
    GRAPH :sensors {
        ?sensor om-owl:processLocation ?sensorLocation .
        ?sensorLocation wgs84_pos:alt ?alt ;
                            wgs84_pos:lat ?lat ;
                            wgs84_pos:long ?long .
    }
    GRAPH :dbpedia {
        ?hurricane rdf:type/rdfs:subClassOf* yago:Hurricane111467018 ;
                    dbpprop:1MinWinds ?hurricaneWindSpeed.
        FILTER(?windSpeed > ?hurricaneWindSpeed)
    }
}
GROUP BY ?lat ?long ?alt
```

Listing A17 SRBench query **Q15** in RSP-QL.

```
SELECT ?heritage
FROM NAMED WINDOW :win ON STREAM <http://www.cwi.nl/SRBench/observations> [RANGE PT1H]
FROM NAMED GRAPH :sensors <http://www.cwi.nl/SRBench/sensors>
FROM NAMED GRAPH :dbpedia <http://www.cwi.nl/SRBench/dbpedia>
WHERE {
    WINDOW :win
        ?observation a weather:WindspeedObservation ;
                     om-owl:procedure ?sensor ;
                     om-owl:result [ om-owl:floatValue ?windSpeed ] .
        FILTER ( ?windSpeed ≥ "74"^^xsd:float ) #milesPerHour
    }
    GRAPH :sensors {
        ?sensor om-owl:hasLocatedNearRel [om-owl:hasLocation ?nearbyLocation] .
    }
    GRAPH :dbpedia {
        ?heritage owl:sameAs ?nearbyLocation ;
                  dcterms:subject ?category .
        ?category skos:broader* category:World_Heritage_Sites .
    }
}
```

Listing A18 SRBench query **Q16** in RSP-QL.

```
SELECT ?damage
FROM NAMED WINDOW :win ON STREAM <http://www.cwi.nl/SRBench/observations> [RANGE PT1H]
FROM NAMED GRAPH :geonames <http://www.cwi.nl/SRBench/geonames>
FROM NAMED GRAPH :sensors <http://www.cwi.nl/SRBench/sensors>
FROM NAMED GRAPH :dbpedia <http://www.cwi.nl/SRBench/dbpedia>
WHERE {
    WINDOW :win {
        ?observation a weather:WindspeedObservation ;
                   om-owl:procedure ?sensor ;
                   om-owl:result [ om-owl:floatValue ?windSpeed ] .
        FILTER ( ?windSpeed ≥ "74"^^xsd:float ) #milesPerHour
    }
    GRAPH :sensors {
        ?sensor om-owl:hasLocatedNearRel [om-owl:hasLocation ?nearbyLocation] .
    }
    GRAPH :dbpedia {
      ?hurricane dbpprop:areas [ foaf:name ?areaName ] ;
                 rdf:type/rdfs:subClassOf* yago:Hurricane111467018 ;
                 dbpprop:damages ?damage .
    }
    GRAPH :geonames {
        ?nearbyLocation gn:parentFeature* ?area .
        ?area gn:name•gn:officialName ?areaName .
    }
}
```

Listing A19 SRBench query **Q17** in RSP-QL.

A3 Python scripts for SRBench streamer

```python
#!/usr/bin/env python3
from rdflib import Graph
from rdflib.term import URIRef, Literal, BNode
import datetime, json, math, multiprocessing as mp, os, sys

def getSJSON(s,p,o,t):
    type(t)
    r = { "s":str(s), "p":str(p), "ts":t}
    if isinstance(o, Literal):
        r["o"]={ "value": o.value, "type": "literal", "datatype": str(o.datatype)
            }
    elif isinstance(o,URIRef):
        r["o"]={ "value": str(o), "type":"uri" }
    elif isinstance(o, BNode):
        r["o"]={ "value": str(o.n3()), "type": "bnode" }
    return r

def datetime_converter(o):
    if isinstance(o, datetime.datetime):
        return o.__str__()

def divide_chunks(l, number_of_chunks):
    n = math.ceil(len(l)/number_of_chunks)
    for i in range(0, len(l), n):
        yield l[i:i + n]

def batch(data):
    number_of_chunks  = 5
    for chunk in divide_chunks(data["files"], number_of_chunks):
        g = Graph()
        for f in chunk:
            g.parse(os.path.join(directory,f), format=syntax)
        timestamps = g.query(data["timestamp_query"])
        for ts in timestamps:
            t = int(ts[0].toPython().timestamp())
            rs = g.query(data["construct_query"], initBindings={"o": ts[0]})
            for s, p, o in rs:
                print(json.dumps(getSJSON(s,p,o,t), default=datetime_converter))


if __name__ == '__main__':
    if len(sys.argv) <2:
        print("Usage : script.py directory format construct_query timestamp_query
            [start end]", file=sys.stderr)
        print("e.g. : script.py rdf n3 construct_query.sparql
            timestamp_query.sparql 0 100",file=sys.stderr)
    else:
        directory, syntax, construct_query, timestamp_query = sys.argv[1:5]
        directories = sorted([el for el in os.listdir(directory) if
            el.endswith(syntax)])
        start, end = map(int,sys.argv[5:7]) if len(sys.argv)>=7 else (0,
            len(directories))
        directories = directories[start:end]
        construct_query = " ".join(open(construct_query).readlines())
        timestamp_query = " ".join(open(timestamp_query).readlines())
        cpus = mp.cpu_count()
        with mp.Pool(cpus) as p:
            p.map(batch, [ { "process": i, "files": files,
                    "timestamp_query":timestamp_query,
                        "construct_query":construct_query }
                for (i, files) in enumerate(divide_chunks(directories, cpus))] )
```

Listing A20 Python script of the first component.

```
from confluent_kafka import avro
from confluent_kafka.avro import AvroProducer
import json, sys, os

topic = os.getenv("TOPIC", "sorted_triples")
ts = os.getenv("TIMESTAMP_KEY", "ts")
key = os.getenv("KEY", "s")
key_schema_file = os.getenv("KEY_SCHEMA_FILE",
    "Streaming-triples-subject-key.avsc")
value_schema_file = os.getenv("VALUE_SCHEMA_FILE", "Streaming-triples.avsc")
bootstrap_server = os.getenv("BOOTSTAP_SERVER",'localhost:9092')
schema_registry = os.getenv("SCHEMA_REGISTRY", 'http://localhost:8081')

if __name__ == '__main__':
    with open(key_schema_file) as kf, open(value_schema_file) as vf:
        value_schema = avro.loads(" ".join([el.strip() for el in vf.readlines()]))
        key_schema = avro.loads(" ".join([el.strip() for el in kf.readlines()]))
        avroProducer = AvroProducer({'bootstrap.servers': bootstrap_server,
            'schema.registry.url': schema_registry },
            default_value_schema=value_schema, default_key_schema=key_schema )
        for line in sys.stdin:
            triple = json.loads(line)
            try :
                avroProducer.produce(topic=topic, value=triple, key={"key" :
                    triple["s"]}, timestamp=triple["ts"])
                avroProducer.flush()
            except BufferError as e:
                print(e, triple)
                avroProducer.flush()
```

Listing A21 Python script of the second component.

A4 Kubernetes Operator and CRDs as μCR Orchestrator

In this Section we will how Kubernetes could be instrumented through a custom Operator and Custom Resource Definitions (CRDs) to handle Reasoners, Data Infrastructure, Queries and Streams.

Since Kubernetes 1.7, Custom Controller have been introduced to allow developers to extend and add new functionalities, replace existing ones and automate administration tasks as if they were native Kubernetes components. An Operator is a set of application-specific custom controllers having direct access to Kubernetes APIs, meaning that they can monitor the cluster and react to any change or perform any needed action to achieve the desired state. They allow to write applications to fully manage the lifecycle of other applications.

As said, an operator is able to interact with existing Kubernetes APIs objects, but developers are also able to define custom resources through the usage of Custome Resource Definition. Resources defined become Kubernetes first citizens and can have a totally custom structure to carry all the needed informations.

A real example of Operator from which we took inspiration is the *Prometheus Operator*[2]. Prometheus is a monitoring system and time series database, with Kubernetes it is one of the few graduated project in the CNCF Landscape, it allows to store and query metrics scraped from application exposing them in a specific format. Rules can be specified to record aggregate data before storing them and to produce alerts if specific conditions are met and then send those alert to another provided component, the AlertManager, which will dispatch them to the desired receivers, e.g. emails, Slack channels. Being Prometheus a database its state has to be persisted and it has to be configured correctly with the endpoint it has to scrape and the queries and alerts it has to run. The Prometheus Operator comes with four different Custom Resource Definitions: Prometheus to define

[2]https://github.com/coreos/prometheus-operator

the Prometheus deployment's characteristics, like version, number of replicas and disks sizing; ServiceMonitors which declaratively specify how groups of services should be monitored and are used by the operator to generate Prometheus scrape configurations; PrometheusRules, defining the desired Prometheus rules then used by the operator to configure alerting and recording rules correctly; Alertmanager, specifying the Alertmanager deployment and the desired receiver for different kinds of alerts.

Operators are becoming increasingly popular because they allow to operate complex systems such as Databases or stateful applications in general, abstracting away most of the complexity behind clear declarative APIs.

In the very same way we could produce a μCR Operator and Custom Resources Definitions for the Data Infrastructure, Reasoners, Streaming Queries and Streams. The Data Infrastructure would allow to define a Kafka cluster with its associated Schema registry, while Reasoners would encode also the capabilities of each reasoner and thanks to containerization every reasoner could be deployed at need by the operator. Streaming Queries could be split between the available Stream Reasoners capabilities by the μCR operator and run by the single reasoners interacting through the Data Infrastructure. Streaming Sources and Sinks could also be defined to allow sending and consuming results to and from the external world.

Main obstacles to achieve this are the need for the operator to be able to translate a given input query to multiple queries or plan executable using the provided Reasoners, therefore the need to produce a standardized communication language to specify what each agent should do, otherwise it would be impossible to instrument the Operator to be able to translate each query to a custom language for each one of them. The chosen language should be flexible enough to allow for different expressiveness powers but also easy to comprehend for users writing them. In this sense RSP-QL could be a good starting point, given the flexibility of the graph model and its ability to take advantage of encoded knowledge in the form of ontologies.