Optimizing RDF Stream Processing for Uncertainty Management

Robin Keskisärkkä, Eva Blomqvist and Olaf Hartig
Linköping University, Linköping, Sweden

Abstract. RDF Stream Processing (RSP) has been proposed as a way of bridging the gap between the Complex Event Processing (CEP) paradigm and the Semantic Web standards. Uncertainty has been recognized as a critical aspect in CEP, but it has received little attention within the context of RSP. In this paper, we investigate the impact of different RSP optimization strategies for uncertainty management. The paper describes (1) an extension of the RSP-QL* data model to capture bind expressions, filter expressions, and uncertainty functions; (2) optimization techniques related to lazy variables and caching of uncertainty functions, and a heuristic for reordering uncertainty filters in query plans; and (3) an evaluation of these strategies in a prototype implementation. The results show that using a lazy variable mechanism for uncertainty functions can improve query execution performance by orders of magnitude while introducing negligible overhead. The results also show that caching uncertainty function results can improve performance under most conditions, but that maintaining this cache can potentially add overhead to the overall query execution process. Finally, the effect of the proposed heuristic on query execution performance was shown to depend on multiple factors, including the selectivity of uncertainty filters, the size of intermediate results, and the cost associated with the evaluation of the uncertainty functions.

Keywords. RSP, CEP, Uncertainty, RSP-QL

1. Introduction

RDF Stream Processing (RSP) is based on existing Semantic Web standards but extends traditional approaches to support continuous processing of streaming RDF data. While several RSP systems have been inspired by data stream management systems [1,2,3], RSP has also been proposed as a candidate for bringing together the Complex Event Processing (CEP) paradigm and the Semantic Web standards [4,5,6,7] in order to target information integration and stream reasoning. CEP focuses on detecting events from streaming sources, where a high-level event may be viewed as an abstraction of a set of low-level events.

Within the CEP domain, representing and reasoning with uncertainty has been recognized as a critical aspect for dealing with real-world data, which can be imprecise, incomplete, and noisy [8,9,10]. However, within the RSP domain uncertainty has received little attention. In previous work, we evaluated the impact of explicitly managing different uncertainty types in RSP, which showed a need for research on query optimization strategies to improve uncertainty management efficiency [11].

The main contributions of this paper are (1) an extension of the RSP-QL* data model we proposed in [11], to capture the syntax and semantics of uncertainty functions along with filter and bind expressions, (2) two technical optimization strategies for increasing query execution performance, and a heuristic to support reordering of uncertainty filters, and (3) an evaluation of these strategies in a prototype implementation.

The outline of the paper is as follows. Section 2 briefly introduces some basic concepts, and Section 3 extends the syntax and semantics of RSP-QL*. Section 4 provides query re-write rules, describes optimization strategies, and provides a heuristic for reordering uncertainty filters. Section 5 presents an evaluation of the proposed strategies in a prototype implementation. Finally, Section 6 summarizes the findings and outlines future work.

2. Preliminaries

Uncertainty in CEP can broadly be viewed as belonging to three main types: *occurrence uncertainty*, *attribute uncertainty*, and *pattern uncertainty* [8,9,12]. Variations of these uncertainty types have been modeled and implemented in existing CEP systems to deal with data that may be, e.g., incomplete, imprecise, vague, contradictory, or noisy [8,10]. In this paper, we explicitly focus on issues related to attribute uncertainty.

Attribute uncertainty generally refers to uncertainty about the content of event objects [8,12]. For example, when a sensor reports a value, the *true* value is usually assumed to be near the reported value but limited by the precision of the sensor and additional factors of the environment. While such uncertainty is often ignored for simplicity it can have important consequences. For example, consider a case where the task is to generate an alert whenever the oxygen concentration in a room falls below 19.5%. Should an alert be generated if the oxygen concentration is reported to be 19.7%? When also considering that the sensor is only accurate to within 1%, or biased towards higher values?

Attribute uncertainty is often expressed as a distribution around a value, typically described as a probability distribution. While there are no standardized formats for representing probability distributions in RDF, in this paper we will use a literal datatype, defined for this purpose in our previous work [11]. The literal datatype is denoted by the URI *rspu:distribution*, and is of the form $f(p_1, p_2, ..., p_n)$, where f is a string identifier for a probability distribution type, and every p_i is a floating-point number. For example, a normal distribution with a mean μ of 19.7 and variance σ^2 of 1 could be represented using the literal `"N(19.7,1)"`. Similarly, a uniform distribution between 18.7 and 20.7 could be represented as `"U(18.7,20.7)"`.

In order to deal with streaming RDF data, several RSP models and implementations have been proposed over the past decade, and Dell'Aglio et al. defined RSP-QL as a way of unifying the syntax and semantics of these initial proposals [13,14]. The RSP-QL language extends SPARQL 1.1 to enable querying of streaming data by allowing discrete portions of RDF streams to be defined and queried. In earlier work we proposed RSP-QL*, which extends RSP-QL to support statement-level annotations as an alternative to RDF reification [15]. Listing 1 provides an example query demonstrating the main features of the RSP-QL* language, leveraging two uncertainty functions described later in Section 3.1. For an in-depth description of the RSP-QL* data model and syntax, the reader is referred to the original paper [15].

```
REGISTER STREAM <warning/oxygen> COMPUTED EVERY PT10S AS
SELECT ?value
FROM NAMED WINDOW <w> ON <http://stream/oxygen> [RANGE PT10S STEP PT10S]
WHERE {
    WINDOW <w> {
        GRAPH ?g {
            ?o sosa:hasSimpleResult ?value .
            << ?o sosa:hasSimpleResult ?value >> rspu:uncertainty ?unc .
            BIND(rspu:add(?unc, ?value) AS ?d)
            FILTER(rspu:greaterThan(?d, 0.195) > 0.90)
        }
    }
}
```

Listing 1: An RSP-QL* query that passes events to the output stream when the reported oxygen concentration is greater than 19.5% with a probability greater than 0.90. The measurement error is combined with the reported value before being compared with the threshold value.

3. Syntax and Semantics of RSP-QL*

RSP extends traditional RDF/SPARQL by introducing a time dimension to processing [13]. In RSP-QL, the time dimension is managed via windows that define discrete subsets over RDF streams that can then be queried as regular RDF datasets. RSP-QL* [15] extends RSP-QL by extending it along the lines of RDF*/SPARQL* [16, 17]. Essentially, RDF* allows RDF triples to be used as subjects and objects in triples, while SPARQL* supports the querying of such triples. We use RSP-QL* as the starting point for this work. For the SPARQL-specific constructs, we adopt the algebraic SPARQL syntax introduced by Pérez et al. [18]. Due to space constraints, we here limit ourselves to presenting only the core concepts of the language and an extension to a subset of the SPARQL algebra.

The basic building block of SPARQL [19] is a *basic graph pattern* (BGP), that is, a finite set of *triple patterns*. A triple pattern is a tuple $(s,p,o) \in (\mathcal{V} \cup \mathcal{B} \cup \mathcal{I}) \times (\mathcal{V} \cup \mathcal{I}) \times (\mathcal{V} \cup \mathcal{B} \cup \mathcal{I} \cup \mathcal{L})$, where $\mathcal{V}$ is the set of query variables disjoint from $\mathcal{I}$ (all IRIs), $\mathcal{B}$ (all blank nodes) and $\mathcal{L}$ (all literals), respectively. A *solution mapping* is a partial function that maps query variables to blank nodes, IRIs, or literals $\eta : \mathcal{V} \rightarrow (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$.

RSP-QL* extends triple patterns to support the concept of *triple* patterns* [17,16], which allow triple patterns to be nested (arbitrarily deep). A triple* pattern is defined recursively as follows: i) any triple pattern is a triple* pattern, and ii) given two triple* patterns tp and tp', and $s \in (\mathcal{I} \cup \mathcal{V})$, $p \in (\mathcal{I} \cup \mathcal{V})$, and $o \in (\mathcal{I} \cup \mathcal{L} \cup \mathcal{V})$, then (tp,p,o), (s,p,tp), and (tp,p,tp') are triple* patterns. A finite set of triple* patterns is referred to as a *BGP**.

Similarly, the notion of solution mappings is extended to *solution* mappings* that allow both RDF terms and RDF* triples to be bound to query variables. A solution* mapping is defined as a partial mapping $\eta : \mathcal{V} \rightarrow (\mathcal{T} \cup \mathcal{I} \cup \mathcal{L})$, where $\mathcal{T}$ is an RDF* triple.

We use the notion of *comparison terms* to denote the terms that can be used in SPARQL *built-in conditions*. For a complete list of the built-in predicates, we refer the reader to the SPARQL specification [19].

Definition 1. Comparison terms are defined recursively as follows:

- a variable is a comparison term,
- a URI is a comparison term,
- a literal is a comparison term,
- if f denotes a SPARQL built-in predicate [19] and $x_1,...,x_n$ are comparison terms, then $f(x_1,...,x_n)$ is a comparison term, and
- if f is a URI that denotes an uncertainty function defined in Section 3.1 and $x_1,...,x_n$ are comparison terms, then $f(x_1,...,x_n)$ is a comparison term.

Definition 2. Built-in conditions are defined recursively as follows:

- if x and y are comparison terms, then $x = true$, $x = y$, $x < y$, $x \leq y$, $x > y$, $x \geq y$, $x \neq y$ are built-in conditions, and
- if c_1 and c_2 are built-in conditions, then $c_1 \wedge c_2$, $c_1 \vee c_2$ and $\neg c_1$ are built-in conditions.

3.1. Syntax of Uncertainty Functions

We here leverage the set of extension functions for managing probability distributions introduced in our previous work [11]. The functions support some useful operations on probability distributions. For these definitions, let $\mathcal{L}_d$ be the set of all *rspu:distribution* literals and $\mathcal{L}_n$ be the set of all literals with numeric data types. For every literal $l \in \mathcal{L}_d$, we write $val(l)$ to denote the probability distribution that l represents, and for every $l \in \mathcal{L}_n$, $val(l)$ denotes the numeric value represented by l.

Definition 3. The URI **rspu:add** denotes a function f_{add} that, for every $c_1 \in \mathcal{L}_d$ and $c_2 \in \mathcal{L}_d \cup \mathcal{L}_n$, returns a literal $c_3 \in \mathcal{L}_d$ such that $val(c_3)$ is the probability distribution obtained by adding $val(c_1)$ to $val(c_2)$.

Definition 4. The URI **rspu:subtract** denotes a function f_{subtract} that, for every $c_1 \in \mathcal{L}_d$ and $c_2 \in \mathcal{L}_d \cup \mathcal{L}_n$, returns a literal $c_3 \in \mathcal{L}_d$ such that $val(c_3)$ is the probability distribution obtained by subtracting $val(c_2)$ from $val(c_1)$.

Definition 5. The URI **rspu:greaterThan** denotes a function $f_{\mathsf{greaterThan}}$ that, for every $c_1 \in \mathcal{L}_d$ and $c_2 \in \mathcal{L}_d \cup \mathcal{L}_n$, returns a literal $c_3 \in \mathcal{L}_n$ such that $val(c_3)$ is the probability that $val(c_1)$ is greater than $val(c_2)$.

Definition 6. The URI **rspu:lessThan** denotes a function f_{lessThan} that, for every $c_1 \in \mathcal{L}_d$ and $c_2 \in \mathcal{L}_d \cup \mathcal{L}_n$, returns a literal $c_3 \in \mathcal{L}_n$ such that $val(c_3)$ is the probability that $val(c_1)$ is less than $val(c_2)$.

Definition 7. The URI **rspu:between** denotes a function f_{between} that, for every $c_1 \in \mathcal{L}_d$ and $c_2, c_3 \in \mathcal{L}_d \cup \mathcal{L}_n$, returns a literal $c_4 \in \mathcal{L}_n$ such that $val(c_4)$ is the probability that $val(c_1)$ is greater than $val(c_2)$ and less than $val(c_3)$.

3.2. Syntax of RSP-QL*

RSP-QL* extends RSP-QL to support all the forms of graph patterns that have been introduced for SPARQL and SPARQL* [15]. For brevity, we here cover only the core constructs of the language. An RSP-QL* query consists of two parts: an RSP-QL* pattern, and a set of *window declarations* associated with IRIs that serve as names for the corresponding windows in the query.

Definition 8. An **RSP-QL* pattern** is defined recursively as follows:

- Any BGP* is an RSP-QL* pattern.
- If $n \in (\mathcal{V} \cup \mathcal{I})$ and P is an RSP-QL* pattern, then (WINDOW n P) and (GRAPH n P) are RSP-QL* patterns.
- If P_1 and P_2 are RSP-QL* patterns, then (P_1 AND P_2), (P_1 OPT P_2), and (P_1 UNION P_2) are RSP-QL* patterns.
- If P is an RSP-QL* pattern and R is a built-in condition, then the expression (P FILTER R) is an RSP-QL* pattern.
- If P is an RSP-QL* pattern, C is a comparison term and $?v$ is a variable that neither occurs in P nor in C, then (P BIND$_{?v}$ C) is an RSP-QL* pattern.

Definition 9. A **window declaration** is a tuple $(u_S, \alpha, \beta, \tau_0)$ where $u_S \in \mathcal{I}$ is an IRI (representing the name of a named RDF* stream), α is a time duration (representing a window width), β is a time duration (representing a slide parameter), and τ_0 is a timestamp (representing a start time).

Definition 10. An **RSP-QL* query** is a pair (ω, P) where ω is a partial function that maps some IRIs in $\mathcal{I}$ to window declarations, and P is an RSP-QL* pattern such that for every sub-pattern (WINDOW n P') in P it holds that if $n \in \mathcal{I}$, then ω is defined for n, i.e., $n \in \mathrm{dom}(\omega)$.

3.3. Semantics of RSP-QL*

The semantics of RSP-QL* has been described in previous work [15], but is here extended to cover the notions of comparison terms (cf. Definition 1), bind expressions, and filter expressions. The standard notions of *compatibility*, *merging*, and *application* of solution mappings in SPARQL are adapted for solution* mappings as follows.

Definition 11. Two solution* mappings η, η' are **compatible**, denoted $\eta \sim \eta'$, if $\eta(?v) = \eta'(?v)$ for every variable $?v \in dom(\eta) \cap dom(\eta')$.

Definition 12. The **merge** of two compatible solution* mappings η and η', denoted by $\eta \cup \eta'$, is a solution* mapping η'' with the following three properties:

- $dom(\eta'') = dom(\eta) \cup dom(\eta')$,
- $\eta''(?v) = \eta(?v)$ for all $?v \in dom(\eta)$, and
- $\eta''(?v) = \eta'(?v)$ for all $?v \in dom(\eta') \setminus dom(\eta)$.

Definition 13. The **application** of a solution* mapping η to an RSP-QL* pattern P, denoted by $\eta[P]$, is the RSP-QL* pattern obtained by replacing all variables in P according to η.

Next, we define the evaluation function for comparison terms, with respect to a given solution* mapping. This can be regarded as a *lifting* of the definitions of comparison terms and uncertainty functions into the SPARQL context.

Definition 14. Let $eval(c, \eta)$ denote the evaluation function of a comparison term c given a solution* mapping η:

- if c is a variable and $c \in dom(\eta)$, then $eval(c, \eta)$ is $\eta(c)$;
- if c is a URI or a literal, then $eval(c, \eta)$ is c;
- if c is of the form $f(c_1, ..., c_n)$ where f denotes a SPARQL built-in predicate and c_i are comparison terms, then $eval(c, \eta)$ is $f(eval(c_1, \eta), ..., eval(c_n, \eta))$, where f is evaluated according to the SPARQL specification [19];
- if c is of the form $\mathsf{rspu{:}add}(c_1, c_2)$ such that $eval(c_1, \eta) \in \mathcal{L}_d$ and $eval(c_2, \eta) \in \mathcal{L}_d \cup \mathcal{L}_n$ then return $f_{\mathsf{add}}(eval(c_1, \eta), eval(c_2, \eta))$;
- if c is of the form $\mathsf{rspu{:}subtract}(c_1, c_2)$ such that $eval(c_1, \eta) \in \mathcal{L}_d$ and $eval(c_2, \eta) \in \mathcal{L}_d \cup \mathcal{L}_n$ then return $f_{\mathsf{subtract}}(eval(c_1, \eta), eval(c_2, \eta))$;
- if c is of the form $\mathsf{rspu{:}greaterThan}(c_1, c_2)$ such that $eval(c_1, \eta) \in \mathcal{L}_d$ and $eval(c_2, \eta) \in \mathcal{L}_d \cup \mathcal{L}_n$ then return $f_{\mathsf{greaterThan}}(eval(c_1, \eta), eval(c_2, \eta))$;
- if c is of the form $\mathsf{rspu{:}lessThan}(c_1, c_2)$ such that $eval(c_1, \eta) \in \mathcal{L}_d$ and $eval(c_2, \eta) \in \mathcal{L}_d \cup \mathcal{L}_n$ then return $f_{\mathsf{lessThan}}(eval(c_1, \eta), eval(c_2, \eta))$;
- if c is of the form $\mathsf{rspu{:}between}(c_1, c_2, c_3)$ such that $eval(c_1, \eta) \in \mathcal{L}_d$, $eval(c_2, \eta) \in \mathcal{L}_d \cup \mathcal{L}_n$ and $eval(c_3, \eta) \in \mathcal{L}_d \cup \mathcal{L}_n$ then return $f_{\mathsf{between}}(eval(c_1, \eta), eval(c_2, \eta), eval(c_3, \eta))$;
- else return an error.

Finally, let $filter(R, \eta)$ be the evaluation function for the built-in condition R w.r.t. to the solution* mapping η that returns a literal, IRI, or an error. We say that a *filter condition R satisfies* some solution* mapping η if the evaluation of the constraint is true. That is, a filter condition eliminates any solutions that, when substituted into the expression, either result in an effective boolean value of false or produce an error. For a complete definition of the evaluation of SPARQL built-in conditions we refer the reader to the work of Pérez et al. [18] and the SPARQL specification [19]. The corresponding algebraic operations *join* ($\bowtie$), *union* ($\cup$), *difference* ($\setminus$), *left join* ($\mathbin{⟕}$), *projection* (π), *selection* (σ), and *bind* (ρ) are then defined as follows.

Definition 15. Let Ω, Ω_1, and Ω_2 be sets of solution* mappings, $S \subset \mathcal{V}$ be a finite set of variables, R denote a built-in condition, and η_0 be the empty solution* mapping (i.e., $dom(\eta_0) = \emptyset$).

$$\Omega_1 \bowtie \Omega_2 = \{\eta_1 \cup \eta_2 \mid \eta_1 \in \Omega_1, \eta_2 \in \Omega_2,\ \eta_1 \sim \eta_2\}$$
$$\Omega_1 \cup \Omega_2 = \{\eta \mid \eta \in \Omega_1 \text{ or } \eta \in \Omega_2\}$$
$$\Omega_1 \setminus \Omega_2 = \{\eta \in \Omega_1 \mid \text{ for all } \eta' \in \Omega_2,\ \eta \not\sim \eta'\}$$
$$\Omega_1 \mathbin{⟕} \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$$
$$\pi_S(\Omega) = \{\eta \mid \exists \eta' \in \Omega : \eta \text{ is a restriction of } \eta' \text{ to the variables in } S\}$$
$$\sigma_R(\Omega) = \{\eta \in \Omega \mid \eta \text{ satisfies } R\}$$
$$\rho_{?v \leftarrow C}(\Omega) = \{\eta \cup \eta' \mid \eta \in \Omega,\ ?v \notin dom(\eta), \text{ and } \eta' = \eta_0 \text{ if } eval(C, \eta) \text{ returns an error, otherwise } dom(\eta') = \{?v\} \text{ and } \eta'(?v) = eval(C, \eta)\}$$

Based on the definitions of the algebra operators above, RSP-QL* patterns are evaluated over a background dataset and a set of named windows at a given timestamp.

Definition 16. Let W be a partial function that maps some IRIs in $\mathcal{I}$ to a window over some RDF* stream, respectively, and P be an RSP-QL* pattern such that for every sub-pattern (WINDOW n P') in P with $n \in \mathcal{I}$, it holds that W is defined for n, i.e., $n \in \text{dom}(W)$. Furthermore, let D be an RDF* dataset, G be an RDF* graph, and τ be a timestamp. Then, the **evaluation of P** over D and W at τ with G, denoted by $[\![P]\!]_G^{D,W,\tau}$, is defined recursively as follows:

1. If P is the empty BGP*, then $[\![P]\!]_G^{D,W,\tau} = \{\eta_0\}$ where η_0 is the empty solution* mapping (i.e., $dom(\eta_0) = \emptyset$).
2. If P is a non-empty BGP*, then $[\![P]\!]_G^{D,W,\tau} = \{\eta \mid dom(\eta) = var(P) \text{ and } \eta[P] \in G\}$ where $var(P)$ denotes the set of variables occurring in P.
3. If P is (GRAPH u P'), then $[\![P]\!]_G^{D,W,\tau} = [\![P']\!]_{G'}^{D,W,\tau}$ where $(u, G') \in D$.
4. If P is (GRAPH $?x$ P'), then $[\![P]\!]_G^{D,W,\tau} = \bigcup_{(u,G')\in D} [\![(\text{GRAPH } u \ P')]\!]_{G'}^{D,W,\tau}$.
5. If P is (WINDOW u P'), then $[\![P]\!]_G^{D,W,\tau} = [\![P']\!]_{G'}^{DS(\mathcal{W}),\emptyset,\tau}$ where $\mathcal{W} = W(u)$ and G' is the default graph of the window dataset denoted by $DS(\mathcal{W})$.
6. If P is (WINDOW $?x$ P'), then $[\![P]\!]_G^{D,W,\tau} = \bigcup_{u\in\text{dom}(W)} [\![(\text{WINDOW } u \ P')]\!]_G^{D,W,\tau}$
7. If P is (P_1 AND P_2), then $[\![P]\!]_G^{D,W,\tau} = [\![P_1]\!]_G^{D,W,\tau} \bowtie [\![P_2]\!]_G^{D,W,\tau}$.
8. If P is (P_1 UNION P_2), then $[\![P]\!]_G^{D,W,\tau} = [\![P_1]\!]_G^{D,W,\tau} \cup [\![P_2]\!]_G^{D,W,\tau}$.
9. If P is (P_1 OPT P_2), then $[\![P]\!]_G^{D,W,\tau} = [\![P_1]\!]_G^{D,W,\tau} \mathbin{⟕} [\![P_2]\!]_G^{D,W,\tau}$.
10. If P is (P' FILTER R), then $[\![P]\!]_G^{D,W,\tau} = \sigma_R([\![P']\!]_G^{D,W,\tau})$.
11. If P is (P' BIND$_{?v}$ C), then $[\![P]\!]_G^{D,W,\tau} = \rho_{?v\leftarrow C}([\![P']\!]_G^{D,W,\tau})$.

It remains to define the semantics of RSP-QL* queries, which contain window declarations in addition to an RSP-QL* pattern (cf. Definition 10).

Definition 17. Let $\mathcal{S}$ be a finite set of named RDF* streams and $q = (\omega, P)$ be an RSP-QL* query such that for every IRI $u_S \in \text{dom}(\omega)$ there exists a named RDF* stream $(u_S, S) \in \mathcal{S}$. Furthermore, let D be an RDF* dataset and τ be a timestamp. The **evaluation of q** over D and $\mathcal{S}$ at τ, denoted by $[\![q]\!]^{D,\mathcal{S},\tau}$, is defined as $[\![q]\!]^{D,\mathcal{S},\tau} = [\![P]\!]_G^{D,W,\tau}$ where G is the default graph of D and W is a partial function such that $\text{dom}(W) = \text{dom}(\omega)$ and for every IRI $u \in \text{dom}(W)$ it holds that $W(u)$ is the time-based window $\mathcal{W}(S, x-\alpha, x)$ with $(u_S, S) \in \mathcal{S}$, $(u_S, \alpha, \beta, \tau_0) = \omega(u)$ and $x = \tau_0 + \alpha + \beta \times i$ for the greatest possible value of $i \in \mathbb{N}$ for which $x < \tau$.

4. RSP-QL* Query Optimization

Following the definitions of the syntax and semantics of RSP-QL*, we now define a set of RSP-QL* algebra equivalences. These equivalences can be applied to arbitrary RSP-QL* patterns to support query rewriting. We say that two patterns P_1 and P_2 are equivalent, denoted by $P_1 \equiv P_2$, if $[\![P_1]\!]_G^{D,W,\tau} = [\![P_2]\!]_G^{D,W,\tau}$ for every RDF* dataset D, RDF* graph G, timestamp τ, and set of named windows W.

Table 1 includes a subset of the equivalence rules that have previously been described for the SPARQL algebra [18,20]. These rewrite rules also apply to RSP-QL* patterns. For proofs of these equivalences, the reader is referred to Peréz et al. [18] and Schmidt et al. [20]. Group I describes the common algebraic laws for query expressions. For example, *JAss* and *JComm* show the commutativity and associativity of join expres-

Table 1. Algebraic equivalence rules or RSP-QL*. P, P_1 and P_2 are RSP-QL* patterns, R, R_1 and R_2 are built-in conditions, n is a URI or variable, and C is a comparison term.

I. Associativity, Commutativity, Distributivity	
$((P_1 \text{ UNION } P_2) \text{ UNION } P_3) \equiv (P_1 \text{ UNION } (P_2 \text{ UNION } P_3))$	(UAss)
$((P_1 \text{ AND } P_2) \text{ AND } P_3) \equiv (P_1 \text{ AND } (P_2 \text{ AND } P_3))$	(JAss)
$(P_1 \text{ UNION } P_2) \equiv (P_2 \text{ UNION } P_1)$	(UComm)
$(P_1 \text{ AND } P_2) \equiv (P_2 \text{ AND } P_1)$	(JComm)
$((P_1 \text{ UNION } P_2) \text{ AND } P_3) \equiv ((P_1 \text{ AND } P_3) \text{ UNION } (P_2 \text{ AND } P_3))$	(JUDistR)
$((P_1 \text{ UNION } P_2) \text{ OPT } P_3) \equiv ((P_1 \text{ OPT } P_3) \text{ UNION } (P_2 \text{ OPT } P_3))$	(LUDistR)
II. Filter decomposition	
$(P \text{ FILTER } R_1 \wedge R_2) \equiv ((P \text{ FILTER } R_1) \text{ FILTER } R_2)$	(FDecompI)
$(P \text{ FILTER } R_1 \vee R_2) \equiv ((P \text{ FILTER } R_1) \text{ UNION } (P \text{ FILTER } R_2)$	(FDecompII)
$(P \text{ FILTER } R_1 \wedge R_2) \equiv (P \text{ FILTER } R_2 \wedge R_1)$	(FReordI)
$(P \text{ FILTER } R_1 \vee R_2) \equiv (P \text{ FILTER } R_2 \vee R_1)$	(FReordII)
III. Filter pushing	
$((P_1 \text{ UNION } P_2) \text{ FILTER } R) \equiv ((P_1 \text{ FILTER } R) \text{ UNION } (P_2 \text{ FILTER } R))$	(FUPush)
If $\forall\ ?v \in vars(R)\ : ?v \in cVars(P_1) \wedge ?v \notin pVars(P_2)$	
$((P_1 \text{ AND } P_2) \text{ FILTER } R) \equiv ((P_1 \text{ FILTER } R) \text{ AND } P_2)$	(FJPush)
$((P_1 \text{ OPT } P_2) \text{ FILTER } R) \equiv ((P_1 \text{ FILTER } R) \text{ OPT } P_2)$	(FLPush)
If $\forall\ ?v \in vars(R)\ : ?v \in cVars(P)$	
$((\text{WINDOW } n\ P) \text{ FILTER } R) \equiv ((\text{WINDOW } n\ (P \text{ FILTER } R))$	(FWPush)
IV. Bind pushing	
$((\text{GRAPH } n\ (P \text{ BIND}_{?v}\ C)) \equiv ((\text{GRAPH } n\ P) \text{ BIND}_{?v}\ C)$	(BGPush)
$((\text{WINDOW } n\ (P \text{ BIND}_{?v}\ C)) \equiv ((\text{WINDOW } n\ P) \text{ BIND}_{?v}\ C)$	(BWPush)
If $\forall\ ?v \in vars(C)\ : ?v \in cVars(P_1) \wedge ?v \notin pVars(P_2)$	
$((P_1 \text{ AND } P_2) \text{ BIND}_{?v}\ C) \equiv ((P_1 \text{ BIND}_{?v}\ C) \text{ AND } P_2)$	(BJPush)
$((P_1 \text{ OPT } P_2) \text{ BIND}_{?v}\ C) \equiv ((P_1 \text{ BIND}_{?v}\ C) \text{ OPT } P_2)$	(BLPush)

sions. Group II and III contain rules for the manipulation of filters, including filter decomposition, filter reordering, and filter pushing. The final rule in group III has been defined as part of this work and shows how filter pushing may be applied for window patterns. Group IV includes additional equivalence rules for pushing bind expressions, where *BGPush* and *BWPush* have been defined as part of this work.

Following the notation used by Schmidt et al. [20], we write $cVars(P)$ to denote the subset of variables that are *certain* to be bound when evaluating the pattern P, and use $pVars(P)$ to represent the set of variables that are *possibly* bound when evaluating the pattern P. By $vars(x)$ we denote the set of variables occurring in x, where x is either a built-in condition, a comparison term, or an RSP-QL* pattern.

4.1. Query Optimization

SPARQL queries are built up around triple patterns, which result in a large number of join operations. In SPARQL, these joins generally dominate the query execution time, and optimizing queries with respect to these joins has received considerable attention in literature [21,22]. Many of the heuristics that have been proposed for SPARQL can also be applied for RSP-QL* queries, such as triple pattern reordering based on *variable counting*. We use the most common heuristics proposed for SPARQL to provide a baseline in this work, and we assume that data is stored in triple tables (or some similar structure) that provide fast index-based access to stored RDF* data for each possible order of

subject, predicate, and object (i.e., spo, sop, pso, pos, osp, and ops). Below, we describe the heuristics considered in this work.

H1: Triple patterns should be ordered based on their *selectivity*, i.e., based on how likely they are to produce smaller intermediate results [21,22]. Given the position and the number of variables in a triple* pattern, we use the following order starting from the most selective: $(s,p,o) \prec (s,?,o) \prec (?,p,o) \prec (s,p,?) \prec (?,?,o) \prec (s,?,?) \prec (?,p,?) \prec (?,?,?)$. We consider a nested triple* pattern to be a variable if it contains at least one variable.

H2: Graph patterns should be ordered based on their selectivity. A triple pattern evaluated over a specific named graph has higher selectivity than one executed over all graphs: (GRAPH *u P*) $\prec$ (GRAPH ?*x P*). A triple pattern that does not appear in a graph pattern is evaluated over the default graph.

H3: Filters should be broken up into their constituent pieces and applied as *early* as possible. The idea of pushing filters is one of the most commonly applied optimization strategies in database systems since filters reduce size of intermediate results.

H4: Filters containing references to uncertainty functions (cf. Definition 3.1) should be applied as *late* as possible. The intuition is that there exists some trade-off between evaluating a computationally expensive filter condition early and the degree to which the size of the incoming results is reduced. If a filter condition only marginally decreases the size of the intermediate results, it may be more efficient to apply the filter later in the query evaluation when additional query constraints have been applied. Conversely, if the number of join partners increases, the number of times the filter condition needs to be evaluated increases. However, by employing caching, as we shall see later, the cost of these additional filter evaluations is expected to be comparably small. When applied, the heuristic supersedes H3 with respect to filters that reference uncertainty functions.

Further, we provide two technical optimization techniques related to uncertainty functions. *Lazy variables* provide a mechanism for just-in-time evaluation of values calculated using uncertainty functions. A lazy variable is resolved only when the variable value is requested, and if the value is never used the evaluation can be skipped. The hypothesis is that it will have a positive impact on query execution performance when uncertainty functions are used in bind expressions, since unnecessary evaluations may be avoided.

The second optimization involves the use of a *cache* to store calls to uncertainty functions within a single query execution (i.e., the cache is cleared between query evaluations). Rather than evaluating a given uncertainty function multiple times for the same input, the results can then be looked up in the cache. The cache in the implementation is a basic in-memory hash table, where a serialization of the function calls and referenced nodes are used as keys. If the query plan results in multiple calls to uncertainty functions with the same inputs, e.g., if a filter is preceded by an increase in the number of join partners, the hypothesis is that the cost of these additional filter evaluations will be small.

H1–H3 help minimize the impact of the order of operations in the original queries, and provide the baseline for the evaluation presented in the next section.

5. Evaluation

In this section, we present an evaluation of the proposed heuristic (H4) and the two optimization techniques. The experiments were performed on a MacBook Pro 2015, with

a quad-core 2.8 GHz Intel Core i7 processor, 16 GB of 1600MHz DDR3, and 8 GB of memory allocated to the JVM. The prototype, along with the experiment files and queries, is available under the MIT License[1]. The implementation of the engine has been described in previous work [15,11] but has been extended to support the technical optimizations and heuristics described in the previous section.

5.1. Experiment Setup

While a number of benchmarks have been proposed for evaluating RSP systems, such as LSBench [23], SRBench [24], and CityBench [25], adopting these for the evaluation of performance under uncertainty is out-of-scope for this work. Instead, we provide a concrete scenario for the evaluation based on a variation of the Tunnel Ventilation System (TVS) scenario from Cagula et al. [10]. A TVS uses several types of sensors to detect possible failures in tunnels, such as TVS malfunctioning. For the evaluation, we consider two cases: 1) detect when the oxygen concentration is less than 18% while the temperature is above 30 degrees within the same tunnel sector, and 2) detect when two oxygen sensors in a location report conflicting values, while the temperature is above 30.

We assume that sensors for measuring temperature and oxygen concentration are evenly distributed along the length of a tunnel. A total of 1000 tunnel sectors are equipped with four different sensor types each: two sensors measuring oxygen concentration, and two measuring temperature. Each sensor type generates data into a separate stream at a rate of 1 observation/second. The static dataset, consisting of around 23k triples, provides descriptions of tunnel sectors, observable properties, and sensors. Measurement uncertainty is modeled as part of this static data for two of the sensor types (i.e., similar to how accuracy is often represented in sensor data sheets). For the other two sensor types, uncertainty is instead modeled as annotations on the streamed values. The generated data streams were randomly sampled, such that 95% of the reported values were within the scenario thresholds.

We define a total of 6 queries that are executed under different conditions. Queries 1–3 focus on filters containing calls to uncertainty functions. Query 1 is evaluated over two sensor streams and generates a notification if the oxygen concentration threshold is violated above some probability threshold and the reported temperature value is above 30. Query 2 is evaluated over all four streams and requires all values to simultaneously violate the scenario thresholds with a probability greater than some threshold. Query 3 is evaluated over three of the sensor streams and generates a notification if two reported oxygen concentrations differ with a probability greater than some threshold and the reported temperature is above 30 degrees.

Queries 4–6 are similar to queries 1–3, but rather than filtering on probability thresholds they bind the uncertainty function results to variables and report these as part of the query result.

The query heuristics do not perform any reordering of window patterns, and to reduce ordering bias, we execute two versions of each query (a and b) with reversed window orders. Due to space constraints, we include here only the query shown in Listing 2[2].

[1] `https://github.com/keski/RSPUEngine`

[2] The full list of queries are available at `https://github.com/keski/RSPUEngine`

```
REGISTER STREAM <warning/tvs> COMPUTED EVERY PT4S AS
SELECT ?location ?oxValue ?tempValue
FROM NAMED WINDOW <w1> ON <http://stream/oxygen> [RANGE PT10S STEP PT1S]
FROM NAMED WINDOW <w2> ON <http://stream/temperature> [RANGE PT10S STEP PT1S]
WHERE {
    WINDOW <w1> {
        GRAPH ?g1 {
            ?o1 sosa:hasSimpleResult ?oxValue ;
                sosa:hasFeatureOfInterest ?location .
            << ?o1 sosa:hasSimpleResult ?oxValue >> rspu:uncertainty ?unc .
            # The oxygen is below 0.18 with a probability of at least 80%
            FILTER(rspu:lessThan(rspu:add(?unc, ?oxValue), 0.18) >= 0.80)
        }
    }
    WINDOW <w2> {
        GRAPH ?g2 {
            ?o2 sosa:hasSimpleResult ?tempValue ;
                sosa:hasFeatureOfInterest ?location .
            # The reported temperature is greater than 30
            FILTER(?tempValue > 30)
        }
    }
}
```

Listing 2: The query passes the location, oxygen concentration, and temperature to the output stream when the reported oxygen concentration is less than 18% with a probability of at least 0.80, and the reported temperature at the same location is above 30 degrees. Prefixes left out for brevity.

5.2. Results

Query 1–3 are used to evaluate the impact of the H4 heuristic for different probability thresholds. An increase in probability threshold value corresponds to an increase in *filter selectivity*. The term *selectivity* is used to refer to the degree to which the result size is reduced when applying the filter. A small selectivity value is thus *highly* selective (i.e., it greatly reduces the size of the result). The caching optimization is applied both for the baseline and the H4 heuristic, while the lazy variable mechanism has no impact for these queries.

The results of executing queries 1–3 for varying thresholds are presented in Figure 1. The impact of applying the H4 heuristic differs between queries, and with respect to the selectivity of the uncertainty filters. For example, in query 2b applying the H4 heuristic leads to an increase in query execution times for all probability thresholds. On the other hand, in queries 1a and 3a, the H4 heuristic reduces query execution time by up to an order of magnitude across all probability thresholds.

Queries 4–6 focus instead on the impact of the lazy variable mechanism and caching. These queries include no uncertainty filters, and the application of the H4 heuristic has no impact on these queries. The results of executing the queries are shown in Figure 2. The lazy variable mechanism reduces query execution time across all test queries, from a few percents up to an order of magnitude in query 6a. The lazy variable mechanism improves query execution performance whenever unresolved variable bindings are trimmed from

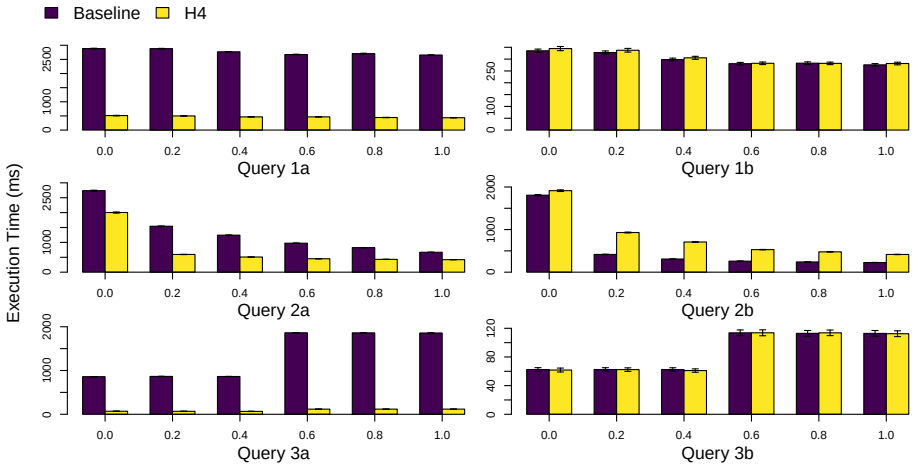

Figure 1. Average query execution times over 20 query executions for queries 1–3. Letters a and b indicate semantically equivalent queries but with reversed window ordering.

the result, and that the overhead of using lazy variables is negligible. The results also show that caching of uncertainty functions generally has a positive impact on query execution performance. However, in query 4a we see that the cache increases overall query execution, showing that the cost of maintaining the cache can add overhead to overall execution if the number of cache hits is low.

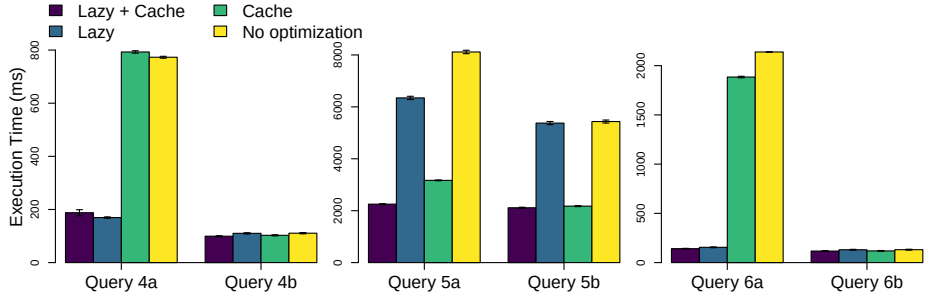

Figure 2. Average query execution times over 20 query executions for queries 4–6, where a and b represent queries with reversed window ordering.

5.3. Follow-up Experiment

To further study how the H4 heuristic impacts performance under different conditions, we generate an independent dataset to support a more precise measure of filter selectivity, and as well as control of the join cardinality between streams (i.e., the factor by which the results will increase when the two windows are joined). We produce two event streams that report randomly sampled values annotated with measurement uncertainty. Each event in the first stream contains four *cardinality properties* that link it to events in

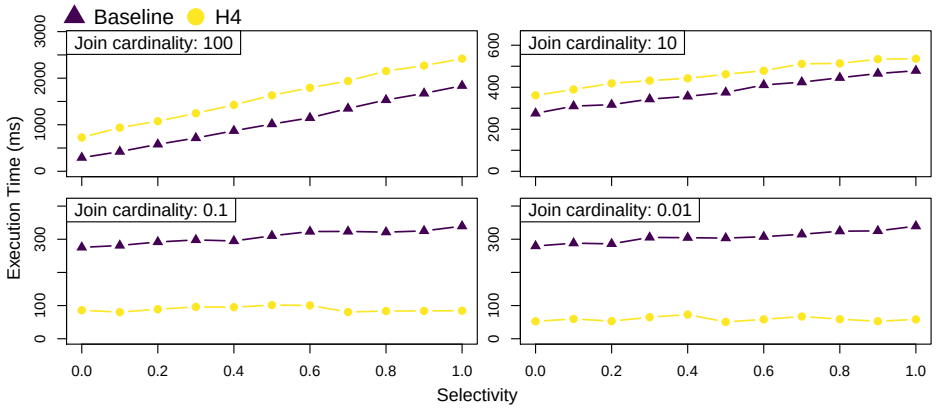

Figure 3. Query execution time for different join cardinalities with varying filter selectivity.

the second stream occurring at the same timestamp. For example, the cardinality property representing a 1-to-100 cardinality maps every event in stream 1 to exactly 100 events in stream 2. An uncertainty filter is applied to the contents of the first stream window. The results of the experiments are presented in Figure 3. The caching optimization is applied both for the baseline and the H4 heuristic.

The results show that when the join cardinality between the windows is high (i.e., the join increases the intermediate result size), the cost of performing the join operation dominates the query execution time, and the baseline outperforms the use of H4, regardless of filter selectivity. However, as the join cardinality decreases, the cost of applying the uncertainty filter becomes the dominating factor and the H4 heuristic then outperforms the baseline across all selectivity thresholds. Generally, applying the H4 heuristic will reduce the number of uncertainty functions that need to be resolved whenever there is a reduction in the number of join partners in the rest of the query pattern. Caching ensures that no uncertainty function will have to be evaluated more than once for the same input, but an increase in the number of join partners can still lead to a high number of cache look-ups that can be detrimental to query execution performance.

6. Conclusions and Future Work

The time required for performing joins between basic graph patterns is generally the dominating factor of execution times in SPARQL processing [21,22]. Pushing filters in order to apply them as early as possible to reduce intermediate results is therefore a common optimization technique. This is true also in the RSP context, but when filters contain calls to uncertainty functions that may be associated with relatively high costs, filter pushing can have the opposite effect. In this paper, we have evaluated a heuristic that instead pulls filters containing references to uncertainty functions, and thereby executes these filters late. The impact of the heuristic depends on both the selectivity of the uncertainty filters, the join cardinalities of subsequent query patterns, and the cost of evaluating the uncertainty filters. Generally, query execution times are reduced when uncertainty filters are pulled if the number of join partners are reduced by subsequent query patterns, since some filter executions can then be avoided.

The two technical optimization techniques proposed to improve query execution performance, have a positive impact on query execution times under most conditions. Caching avoids repeated evaluation of uncertainty functions for the same input, but maintaining the cache and performing cache look-ups also adds to overall execution time. The use of lazy variables reduces the cost of query execution for all affected queries, with performance gains ranging from a few percent to an order of magnitude since no uncertainty function will be executed unless its results are actually used.

In order to effectively combine these findings with other query optimization techniques, such as reordering of operators during query execution, the order of window operations should also be taken into consideration, since it could significantly improve query execution performance. Methods for estimating filter selectivity, improving join cardinality estimation, and switching between query execution strategies to adapt to changing data characteristics also remain important areas for future research.

Acknowledgments

Olaf Hartigs's contributions to this work has been funded in equal parts by the Swedish Research Council (Vetenskapsrådet, project reg. no. 2019-05655) and the CENIIT program at Linköping University (project no. 17.05).

References

[1] 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 *The Semantic Web – ISWC 2011*, pages 370–388, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[2] Calbimonte, Jean-Paul and Corcho, Oscar and Gray, Alasdair J. G. Enabling Ontology-Based Access to Streaming Data Sources. In *The Semantic Web – ISWC 2010*, pages 96–111, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[3] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying RDF Streams with C-SPARQL. *ACM SIGMOD Record*, 39(1):20–26, 9 2010.

[4] Darko Anicic, Sebastian Rudolph, Paul Fodor, and Nenad Stojanovic. Stream Reasoning and Complex Event Processing in ETALIS. *Semant. Web*, 3(4):397–407, 2012.

[5] Minh Dao-Tran and Danh Le-Phuoc. Towards Enriching CQELS with Complex Event Processing and Path Navigation. In *HiDeSt 2015*, pages 2–14, 2015.

[6] Daniele Dell'Aglio, Minh Dao-Tran, Jean-Paul Calbimonte, Danh Le Phuoc, and Emanuele Della Valle. A Query Model to Capture Event Pattern Matching in RDF Stream Processing Query Languages. In *EKAW 2016*, pages 145–162, 2016.

[7] Syed Gillani, Antoine Zimmermann, Gauthier Picard, and Frédérique Laforest. A Query Language for Semantic Complex Event Processing: Syntax, Semantics and Implementation. *Semant. Web*, 10:53–93, 2019.

[8] Elias Alevizos, Anastasios Skarlatidis, Alexander Artikis, and George Paliouras. Probabilistic Complex Event Recognition: A Survey. *ACM Comput. Surv.*, 50(5), 2017.

[9] Alexander Artikis, Opher Etzion, Zohar Feldman, and Fabiana Fournier. Event Processing Under Uncertainty. In *DEBS'12*, pages 32–43, 2012.

[10] Gianpaolo Cugola, Alessandro Margara, Matteo Matteucci, and Giordano Tamburrelli. Introducing Uncertainty in Complex Event Processing: Model, Implementation, and Validation. *Computing*, 97(2):103–144, 2015.

[11] Robin Keskisärkkä, Eva Blomqvist, Leili Lind, and Olaf Hartig. Capturing and Querying Uncertainty in RDF Stream Processing. In *EKAW 2020*, pages 37–53, 2020.

[12] Alessandro Margara, Jacopo Urbani, Frank van Harmelen, and Henri Bal. Streaming the Web: Reasoning over Dynamic Data. *J. Web Semant.*, 25:24–44, 2014.

[13] Daniele Dell'Aglio, Jean-Paul Calbimonte, Emanuele Della Valle, and Oscar Corcho. Towards a Unified Language for RDF Stream Query Processing. In *Revised Selected Papers of the ESWC 2015 Satellite Events on The Semantic Web*, pages 353–363, 2015.

[14] 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. *Int. J. Semant. Web Inf. Syst.*, 10(4):17–44, 2014.

[15] Robin Keskisärkkä, Eva Blomqvist, Leili Lind, and Olaf Hartig. RSP-QL*: Enabling Statement-Level Annotations in RDF Streams. In *SEMANTiCS 2019*, 2019.

[16] Olaf Hartig and Bryan Thompson. Foundations of an Alternative Approach to Reification in RDF. *CoRR*, abs/1406.3399, 2014.

[17] Olaf Hartig. Foundations of RDF* and SPARQL* – An Alternative Approach to Statement-Level Metadata in RDF. In *Proc. of the 11th AMW Workshop*, 2017.

[18] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.

[19] Eric Prud'hommeaux, Steve Harris, and Andy Seaborne. SPARQL 1.1 Query Language. Technical report, W3C, 2013.

[20] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL Query Optimization. In *Proc. of the 13th Int. Conf. on Database Theory*, pages 4–33, 2010.

[21] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In *WWW '08*, 2008.

[22] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter Boncz. Heuristics-Based Query Optimisation for SPARQL. In *Proc. of EDBT*, 2012.

[23] Danh Le-Phuoc, Minh Dao-Tran, Minh-Duc Pham, Peter Boncz, Thomas Eiter, and Michael Fink. Linked Stream Data Processing Engines: Facts and Figures. In *The Semantic Web – ISWC 2012*, volume 7650, pages 300–312. Springer, Berlin, Heidelberg, 2012.

[24] Haopeng Zhang, Yanlei Diao, and Neil Immerman. Recognizing Patterns in Streams with Imprecise Timestamps. *Proc. VLDB Endow.*, 3(1-2):244–255, 2010.

[25] Muhammad Intizar Ali, Feng Gao, and Alessandra Mileo. CityBench: A Configurable Benchmark to Evaluate RSP Engines Using Smart City Datasets. In *The Semantic Web - ISWC 2015*, pages 374–389, Cham, 2015. Springer.