QSMat: Query-Based Materialization for Efficient RDF Stream Processing

Christian Mathieu[1], Matthias Klusch[2], and Birte Glimm[3]

[1] Saarland University, Computer Science Department, 66123 Saarbruecken, Germany
[2] German Research Center for Artificial Intelligence, 66123 Saarbruecken, Germany
[3] University of Ulm, Institute of Artificial Intelligence, 89069 Ulm, Germany

Abstract. This paper presents a novel approach, `QSMat`, for efficient RDF data stream querying with flexible query-based materialization. Previous work accelerates either the maintenance of a stream window materialization or the evaluation of a query over the stream. `QSMat` exploits knowledge of a given query and entailment rule-set to accelerate window materialization by avoiding inferences that provably do not affect the evaluation of the query. We prove that stream querying over the resulting *partial* window materializations with `QSMat` is sound and complete with regard to the query. A comparative experimental performance evaluation based on the Berlin SPARQL benchmark and with selected representative systems for stream reasoning shows that `QSMat` can significantly reduce window materialization size, reasoning overhead, and thus stream query evaluation time.

1 Introduction

In many applications, such as machinery maintenance or social media analysis, large volumes of continuously arriving data must be processed in near-realtime. These data streams stem from sources such as thermometers, humidity or flow sensors, social media messages, price updates, news feeds and many others. Often, it is not only necessary to select and filter information from these streams, but also to infer implicit information with the aid of additional domain knowledge. This process of deriving implicit information (*reasoning*) and filtering (*querying*) is called *stream reasoning*. Inferences are drawn using a set of *inference rules*. While there exist increasingly mature solutions for reasoning in static ontologies, stream reasoning poses additional challenges due to the large volume and frequently unreliable and noisy nature of stream data and its transient nature. Stream data is often considered relevant only during a small time interval, called the *stream window*. Data leaving the window and inferences drawn from said data may thus become invalid over time, and newly arriving data may entail new inferences. A standard query language in this context is Continuous SPARQL (C-SPARQL[1]), an extension of SPARQL for continuous queries over streams of RDF data.

Both reasoning and querying of the resulting *materialized stream window* can become significant runtime bottlenecks. Due to the practical relevance of

the problem, there are many existing approaches to speed up either of these two components.

Incremental materialization approaches such as `IMaRS` [2] or `SparkWave` [7] attempt to speed up materialization by eliminating redundant recomputation between subsequent stream windows. As a result, they spend less time materializing each stream window, but they still materialize the window completely. The materialization might depend on the stream window interval defined by the query (or queries) being posed, but usually not on the specific query itself. This materialized window still contains *all* inferences that follow from window content (and static assertions) under the given entailment scheme. Many of these inferences might never be necessary to answer a concrete query, yet they are derived to guarantee completeness with regard to any query.

Query rewriting techniques such as `StreamQR` [4] exploit this, and compile ontological information into one or more rewritten queries. This has the advantage that window materialization is entirely avoided by encoding entailments directly into the query. To allow static rewriting (i.e. the rewriting happens upfront and not while stream data is processed), these approaches usually require that stream data does not contain schema knowledge.

Similar to query rewriting, our approach `QSMat` attempts to speed up reasoning by exploiting knowledge of the specific query, thus bridging querying and reasoning. In contrast to query rewriting, we retain the added flexibility and expressivity of explicit reasoning and we support custom, user-definable inference rules. To reduce window materialization overhead, we analyze background knowledge and the aforementioned provided inference rule-set during a static preprocessing step per query, and discard ontological information and rules that are provably never needed to satisfy a given query. This promises to reduce the problem size for subsequent materialization of each stream window. Query evaluation over each window then only requires the creation of a partial materialization, for which we prove completeness with regard to the query.

The remainder of this paper is structured as follows. Section 2 introduces `QSMat`, a novel approach to query based stream materialization. A comparative experimental evaluation is then presented in Section 3. Section 4 discusses related work, before we conclude in Section 5.

2 Query-Based Stream Materialization

In the following we give a broad overview of the idea, followed by the formal presentation of the algorithm itself in *Section 2.2*. Noteworthy properties of the algorithm are discussed in *Section 2.3*.

2.1 Overview

`QSMat`[4] is an approach for continuous query evaluation over RDF streams under reasoning. Its goal is accelerating reasoning for each stream window by suppress-

[4] Github: https://github.com/cmth/qsmat

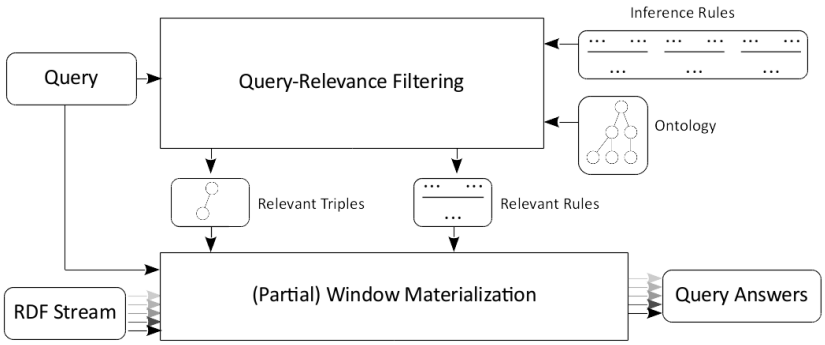


Fig. 1: The `QSMat` architecture

ing inferences during window materialization that are not necessary to answer the query. In other words, it generates incomplete *partial window materializations* that are still sufficient to maintain completeness with respect to the query. Figure 1 illustrates the `QSMat` architecture. `QSMat` is given a query, a static ontology (in the form of triples) expressing background knowledge, and an inference rule-set. The algorithm extracts all triple patterns from the query's where clause, to analyze which triples of a future window materialization are potentially relevant. For each triple pattern, the algorithm guarantees that if a complete window materialization would have generated a triple matching the pattern, then the partial window materialization still generates that triple to maintain completeness. In a one-time per query preprocessing step, `QSMat` performs a backward search over inference rules to find all rules and all triples of the materialized static ontology (also called *static materialization*) that could be needed during window materialization to deduce query-relevant conclusion triples. Since schematic information is usually provided by background knowledge, but not from stream data (e.g. the stream usually does not define new subclasses), the user can tag rule premises that can only match schematic information. Such premises are called *cut-premises*. Using this cut-information, `QSMat` can restrict the relevance search further, since it can exclude inference trees which are not possible with the given background knowledge. This restricted search results in a more aggressive filtering that can exclude more unneeded inferences during subsequent partial window materializations.

Afterwards, the algorithm has a triple and rule subset sufficient to materialize the triples needed for a triple pattern. By merging all of these per-pattern subsets, a set of triples and rules sufficient to answer the entire query is formed. Subsequently, the stream window is materialized partially using this combined triple and rule-set and the query is evaluated over the result.

2.2 The `QSMat` Technique

The main procedure QSMat is shown in Algorithm 1. At the highest level, a query string ($\mathcal{Q}_{str}$), a static ontology (O, interpreted as a set of triples), and a set

Algorithm 1 The main procedure of `QSMat`

```
1: procedure QSMat
2:     Input: Query string Q_str, rule-set R, static ontology O, stream S
3:
4:     Query Q ← parse(Q_str)
5:     (M′, R′) ← filterForQuery(Q, R, O)
6:
7:     for each stream window w ∈ S do
8:         M_w ← materialization of (M′ ∪ w) under R′
9:         Q.eval(M_w)
10:    end for
11: end procedure
```

of inference rules (R) are passed to QSMat. Each rule $r \in R$ may have premise patterns tagged as a *cut-premise*, $r.P_{\text{cut}}$; other patterns are referred to as *live-premises*, denoted $r.P_{\text{live}}$. Both $r.P_{\text{cut}}$ and $r.P_{\text{live}}$ are treated as lists of patterns. The stream S is also treated as a parameter to ease notation. The procedure first parses the query string, yielding $\mathcal{Q}$, a representation of the abstract query tree. It then calls filterForQuery (cf. Algorithm 2). This function handles the relevance filtering and extracts a set M' of relevant triples from the materialization of the static ontology O, and a subset R' of relevant rules from the rule-set R needed for the query Q. The algorithm then listens to the stream and partially materializes each stream window w with M' and R', yielding M_w. Afterwards, M_w contains all triples potentially necessary to answer the query. The procedure then evaluates $\mathcal{Q}$ over M_w, handling the result as appropriate for the given query type.

The function filterForQuery (Algorithm 2) first computes the materialized schema M using the static ontology O and rule-set R. It then traverses the abstract query tree Q to find all triple patterns of the query. For each such triple pattern p, the function filterForPattern (Algorithm 3) then extracts all triples from M and all rules from R that are (transitively) necessary to satisfy p, i.e. to create a partial materialization that still contains all triples of a complete materialization which match p. After filterForPattern returns, filterForQuery aggregates the newly extracted relevant triples (in M') and rules (in R'). After all query triple patterns are processed, filterForQuery returns M' and R', which now contain all triples and rules necessary to satisfy any pattern of the query.

The function filterForPattern (Algorithm 3) finds all triples in M and all rules in R necessary during window materialization to fulfill a given pattern p_{goal}. It does this by recursively backtracking over rules and finding the sets of triples that could match a rule premise in such a way that the rule can produce relevant output. Note that a pattern can be visualized as the (possibly infinite) set of all triples that match this pattern. By restricting a rule premise with a variable binding consistent with the rule conclusion, this restricted premise then subsumes all triples that potentially fulfill it during a rule application yielding

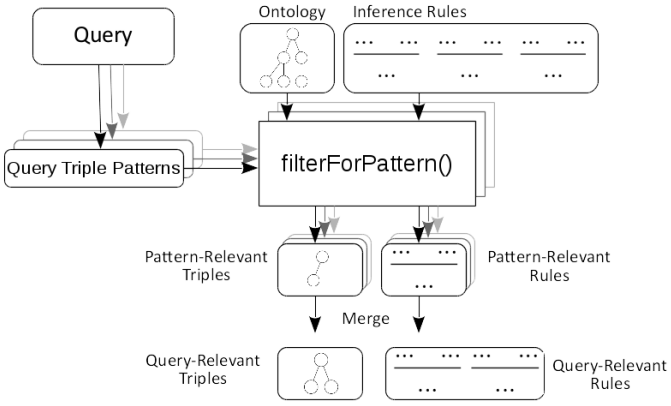


Fig. 2: FILTERFORQUERY concept

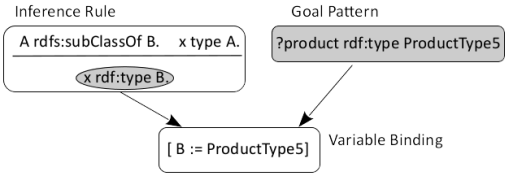
Algorithm 2 Extracts static ontology triples and rules relevant for a query

```
1: function FILTERFORQUERY
2:     Input: Query 𝒬, rule-set R, static ontology O
3:     Output: Filtered static materialization M′, filtered rule-set R′
4:
5:     M′ ← ∅, R′ ← ∅
6:     M ← materialization of O under R
7:     for each triple pattern p ∈ 𝒬 do
8:         (M″, R″) ←FILTERFORPATTERN(R, M, p)
9:         M′ ← M′ ∪ M″
10:        R′ ← R′ ∪ R″
11:    end for
12:    return (M′, R′)
13: end function
```

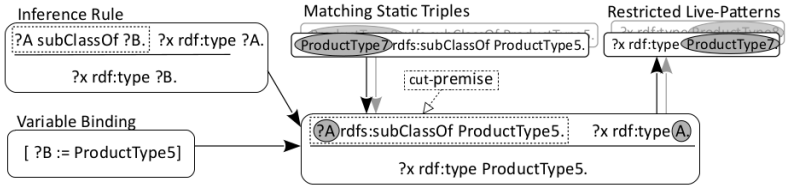
a conclusion that matches the goal pattern (i.e. a potentially relevant pattern). A cut-premise can be thought of as the set of schema triples that may lead to relevant rule firings given the provided ontology, which in turn allows restricting live-premises to match only those triples from a stream that can possibly become relevant given the schema information. Since the latter is statically derivable, it is already explicitly contained in M and can be exhaustively enumerated.

The function maintains a set P of patterns already processed *for the current query triple pattern* to avoid infinite recursion. If the current pattern p_{next} was not already processed before, then the function iterates over all rules that might yield triples matching p_{next}. Rules with no live-premises are skipped since they cannot fire due to triples derived during window materialization by definition. It then binds all variables in cut-premises of the rule r ($r.P_{\text{cut}}$) to matching ground terms in p_{next} and calls the function GROUNDRULE (Algoritm 4), which exhaustively backtracks over all restrictions of this rule given the triples in M

matching the rule's cut-premises. In other words, it finds all ways this rule can be relevant during window materialization to produce a triple matching p_{next}, and thus transitively p_{goal}.



(a) goal pattern variable binding (FILTERFORPATTERN)



(b) rule grounding and premise restriction (GROUNDRULE)

Fig. 3: FILTERFORPATTERN rule predecessor search

The function GROUNDRULE (Algorithm 4) is responsible for finding all ways a rule r can fire yielding a relevant conclusion consistent with a supplied variable binding b. It then returns both the relevant triples from the static materialization and the transitive predecessor patterns needed in the derivation. In other words, it guarantees that all triples matching the goal pattern are still derivable during partial window materialization. It iterates over all combinations of static triples that consistently match all cut-premises, i.e. all ways the rule can fire given the static schema. To do so, it first finds all triples t in M matching the first cut-premise. It then extends the variable binding to be consistent with this triple and then subsequently processes the following cut-patterns in the same manner. Due to the refined variable binding, each following cut-pattern only matches static triples consistent with all previous patterns. For each consistent grounding, i.e. each way to satisfy *all* cut-patterns without disagreeing on variable bindings, a restricted live-pattern p' is generated for each live-premise by applying the variable binding. The pattern p' is then added to Q'. Each t involved in a consistent grounding is added to T'. Afterwards, T' and Q' contain the predecessors needed to guarantee relevant rule firings of r. The set T' contains all static triples needed for cut-premises of r, while Q' contains patterns subsuming all triples that may fulfill a live-premise of r after being newly derived during window materialization. Both T' and Q' are then returned.

Algorithm 3 Finds triples and rules relevant for a triple pattern

```
1: function filterForPattern
2:     Input: Static materialization M, rule-set R, goal pattern p_goal
3:     Output: Relevant triples M'', relevant rules R''
4:
5:     P ← ∅, Q ← ∅, M'' ← ∅, R'' ← ∅
6:     Q.append(p_goal)
7:     while Q not empty do
8:         p_next ← Q.pop()
9:         if p_next ∉ P then
10:            P ← P ∪ {p_next}
11:            M'' ← M'' ∪ {t ∈ M | t matches p_goal}
12:            for each rule r ∈ R, |r.p_live| > 0 do          ▷ Skips static-only rules
13:                Binding b ← bind(r.p_c, p_next)
14:                if b ≠ ⊥ then   ▷ otherwise r cannot produce triples matching p_next
15:                    (T', Q') ← groundRule(M, r, b, 0)
16:                    if Q' ≠ ∅ then   ▷ Q' empty ⇒ r cannot yield t ∈ p_next given M
17:                        Q ← Q ∪ Q'
18:                        M'' ← M'' ∪ T'
19:                        R'' ← R'' ∪ {r}
20:                    end if
21:                end if
22:            end for
23:        end if
24:    end while
25:    return (M'', R'')
26: end function
```

2.3 Correctness of `QSMat`

We next show that the partial materialization computed by `QSMat` is indeed sufficient to answer the given query. Since DESCRIBE queries are implementation dependent, we do not consider them here.

Theorem 1. *Let $\mathcal{Q}$ be any (non-DESCRIBE) C-SPARQL[1] query, R a monotonic inference rule-set, O a static ontology, and S a stream. Then O and S entail an answer to Q under R iff `QSMat` computes this answer given $\mathcal{Q}$, R, O, and S as input.*

Proof (Sketch). The *if direction* corresponds to the *soundness* of `QSMat`: Note that the complete window materialization is the transitive closure of an inference rule-set on the union of static and window triples. Since the inference rules are monotonic and the filtered ontology and rule-set created by `QSMat` are subsets of O and R, respectively, soundness is trivial.

The *only if direction* corresponds to the *completeness* of `QSMat`: Since all C-SPARQL query types depend solely on triples matching query triple patterns (QTPs), we focus, w.l.o.g., on single QTPs. A QTP can be visualized as the set of all triples that would match this pattern. A triple of the complete materialization

Algorithm 4 Finds predecessors necessary for a rule to fire and create relevant conclusions during partial window materialization

```
1: function groundRule
2:     Input: static materialization M, rule r, variable binding b, premise index i
3:     Output: relevant static triples T', relevant predecessor patterns Q'
4:
5:     if i < |r.P_cut| then
6:         p_cut ← apply b to r.P_cut[i]
7:         T ← {t ∈ M | t matches p_cut}
8:         for t ∈ T do
9:             b' ← b ∪ bind(p_cut, t)
10:            (T'', Q'') ← groundRule(M, r, b', i + 1)
11:            if Q'' ≠ ∅ then                    ▷ Q'' empty ⇒ grounding failed
12:                T' ← T' ∪ T'' ∪ {t}
13:                Q' ← Q' ∪ Q''
14:            end if
15:        end for
16:    else                                       ▷ consistent grounding
17:        for each p_live ∈ r.P_live do
18:            p' ← apply b to p_live
19:            Q' ← Q' ∪ {p'}
20:        end for
21:    end if
22:    return (T', Q')      ▷ Note that Q' is empty if r cannot yield relevant output
23: end function
```

matching a QTP can only come from one or more of the following three sources: (i) It can be contained in the stream window directly, (ii) it can be entailed purely from static ontology triples, or (iii) it can be derived transitively from at least one stream triple and zero or more static triples. For (i), the QTP matches the triple since it is contained in the partial materialization because of Line 8 of `QSMat`. For (ii), the triple is marked by `QSMat` in Line 11 of filterForPattern. Case (iii) covers newly derived query-relevant triples. If one such triple was missing from the partial window materialization, then there must have been at least one rule application that fired during complete materialization, but not during partial materialization. Hence, either (a) a rule was not marked or (b) a triple matching a rule premise was not contained in the partial materialization. For (a), the rule can only have fired during complete window materialization if it had at least one live-premise, by definition of live-premises.[5] Since the rule must have fulfilled all cut-premises to fire during complete window materialization, it has at least one consistent grounding using the complete static materialization, thus it created at least one live-pattern in groundRule[6]. This means filterForPattern

[5] Otherwise the stream would have to contain or imply schematic information that was specified as cut in the rule-set, which is a design error.

[6] That is why rules that never produced any restricted live-pattern during static search are not needed to satisfy the query, hence are safely removed by QSMat.

added it to the set of relevant rules in Line 19. For (b), the only way to have a query-relevant triple missing from the partial materialization is if a rule entailed it, and was missing a premise triple. This premise triple matched either a cut-premise or a live-premise (or both, which is subsumed by the former two cases). If it was a cut-premise, the triple must have been part of a consistent grounding, and was contained in the complete static materialization. Thus it was added to the set of relevant triples by GROUNDRULE in Line 12. If it was a live-premise, it was subsumed by the live-premise created by GROUNDRULE in Line 18 and added to the set of relevant patterns in Line 19. In this case, the premise triple can either be a static triple, a window triple, or newly derived in the window materialization. The same argument above can be applied to the created live-pattern instead of the query triple pattern, which leads to a proof by structural induction over all possible inferences trees. This proof must terminate, since an inference tree that completed during complete materialization can only have finite depth and width. Either way, we have a contradiction with the assumption that the partial materialization employed by `QSMat` was not complete with regard to the query triple pattern. Since this holds for all query triple patterns, `QSMat` is complete with regard to the query.

3 Comparative Performance Evaluation

In this section we present a comparative experimental evaluation between `QSMat` and the selected state-of-the-art approaches `C-SPARQL` using `Jena` for materialization, `SparkWave` and `StreamQR`.

3.1 Experimental Setting

The core idea behind `QSMat` is avoiding unnecessary inferences by calculating a partial materialization that is sufficient to answer a given query. This can only yield noteworthy performance benefits if the query is independent of many inferences generated by a complete materialization[7] and if this is statically provable by `QSMat`. Conceptually, `QSMat` is similar both to query rewriting approaches, which attempt to speed up evaluation by analysis of the query, and to optimized reasoners, which attempt to reduce runtime overhead of the reasoning step.

Competitive approaches. To evaluate the feasibility of `QSMat`'s query relevance filtering, we compare its performance to several state-of-the-art approaches with `C-SPARQL` [1, 3] serving as a baseline. Since the `C-SPARQL` engine does not offer reasoning support as of the time of this writing, we extended it with a complete materialization step using `Jena` as a back-end. As a competitive reasoning approach, `SparkWave` [7] was chosen, a fast stream reasoning engine using RETE-based inference [5] (the so-called *epsilon network*) and query pattern matching. Query rewriting approaches are represented by `StreamQR` [4], a stream reasoning engine that uses ontological background knowledge to translate the original

[7] As a pathological counterexample take a query with the where clause (?s ?p ?o): since the query matches all derivable triples, none can be excluded.

query into a union of conjunctive queries, which explicitly express all ways query triple patterns can be satisfied under reasoning.[8]

Testing environment. The dataset used for testing is derived from the *Berlin SPARQL Benchmark* (BSBM). While BSBM is a SPARQL benchmark, and thus not stream oriented, it features a flexible data generator that allows for creating variable-size problem instances simulating e-commerce use-cases in the context of vendors offering products. To adapt these datasets to a suitable stream setting, offers and the underlying product data are treated as stream data, while the concept hierarchy of product types and all information not directly related to products and offers are used as background knowledge. To allow for comparable measurements between approaches, a test framework using `Jena` was created, where the triples contained in each window are first aggregated, and the complete window is then passed to a stream-enabled reimplementation of each approach. While this does effectively process data in a one-shot fashion per window, instead of an admittedly more natural streaming scheme, it allows much more transparent performance measurements on equal footing.

The BSBM benchmark defines parametric query templates, which are used to create concrete, randomized benchmark queries by sampling template parameters (e.g. %ProductFeature1%) over the corresponding input data (e.g. product features). For each such query instance, all algorithms are evaluated on identical background knowledge and identical stream window content, and for an equivalent query in the respective query language. This avoids sampling noise due to differing problem sizes or real-time-dependent window semantics. It further allows a direct measurement of the elapsed wall-clock time spent during query evaluation, is independent of triple arrival rate, and also offers scaling information for problem sizes where e.g. a throughput metric might reach the point of saturation first for given window sizes. Before the actual evaluation, a warm-up phase is performed. This gives the system time to stabilize, and reduces sampling noise e.g. due to initialization effects or at-runtime optimizations from the Java Just-In-Time compiler. The result times for individual query instances are averaged per query template. All tests were run on a system using an Intel Xeon W55990 3.33GHz CPU with 32GB of DDR3-1333 RAM.

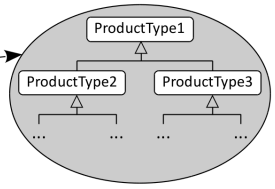
3.2 Evaluation Results

Scalability with regard to ontology size and query. As mentioned above, performance of `QSMat` depends on the reasoning demands of the specific query. Product information generated by BSBM is classified in a product type hierarchy. Leaf products in this hierarchy are classes without subclasses, and class membership of instances cannot be derived in any way except by explicit assertion, while product types further toward the root of the product type hierarchy have increasing numbers of subclasses and larger reasoning demands. As end-points

[8] To give an example: Assume a query matches (?x rdf:type A), and A has subclasses B and C. If no other way to derive membership in A, B or C exists, then it is sufficient to find all (?x rdf:type A), (?x rdf:type B) and (?x rdf:type C).

of this, the root product type offers the least, while leaf product types offer the most opportunity for optimization, both for QSMat and for StreamQR. Deeper product hierarchies further increase the reasoning demands for product types near the root, while the opportunity for optimization in more shallow hierarchies is expected to be lower. A query template chosen to verify this assumption is shown in Listing 3.1.

```
[...]
SELECT DISTINCT ?product ?label
WHERE {
    ?product rdfs:label ?label .
    ?product rdf:type %ProductType% .
    ?product bsbm:productFeature %ProductFeature1% .
    ?product bsbm:productFeature %ProductFeature2% .
    ?product bsbm:productPropertyNumeric1 ?value1 .
        }
ORDER BY ?label
LIMIT 10
```



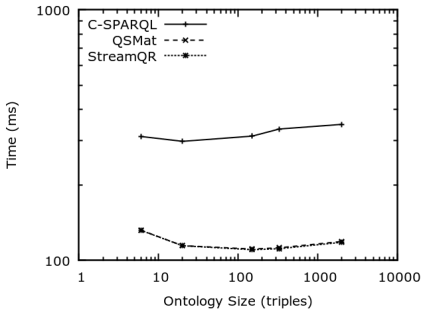
Listing 3.1: Query Template

The results of the static scaling test based on the query template in Listing 3.1 for the Jena-based implementations is shown in Figure 4a and Figure 4b for leaf and for root product types respectively. Both QSMat and StreamQR result in a substantial speedup compared to the completely materializing reference implemenation C-SPARQL, especially for leaf types. This is because for leaves, the complete materialization needs to derive all superclasses of each type in the product type hierarchy, although none of them are ever relevant to the query. Noteworthy is the extremely similar performance of QSMat and StreamQR. This is no coincidence. For this type of query, both QSMat and StreamQR converge to the same behavior: QSMat excludes all inference rules for this type of query, avoids materialization entirely, and then evaluates the query over the raw stream window content. StreamQR creates a trivially rewritten query that is identical to the original query, since there are no relevant schematic dependencies to be compiled into the query. While they arrive at this result differently, both algorithms evaluate the original query over the raw stream window, resulting in almost identical performance.

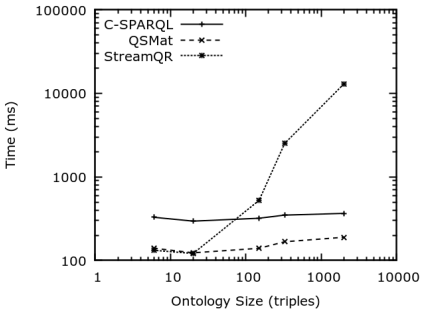
For root types, the performance of C-SPARQL is comparable to the leaf case, which is to be expected, since the cost of the complete materialization does not depend on the specific product type queried. QSMat still outperforms C-SPARQL, but to a lesser degree than for the leaf case, especially for large type hierarchies and thus higher reasoning demand per stream triple. While QSMat can still exclude most rules not needed for the query, all product types are relevant for this type of query. This means that QSMat cannot exclude any part of the product type hierarchy and needs to derive all superclass memberships. As a consequence, a higher fraction of inferences in C-SPARQL's complete materialization is relevant to the query, which is why QSMat's advantage compared to the complete materialization is less dramatic.

It is also apparent that the performance of StreamQR is far more sensitive to ontology size for this type of query instance. Since all product types are

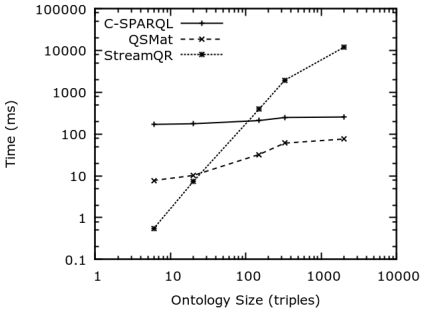
relevant when querying instances of the root type, a larger type hierarchy implies that `StreamQR` must create a larger number of sub-queries to encode subclass relations, which results in increasingly costly query evaluation. These sub-queries need to replicate triple patterns even if they are not related to the rewritten rdf:type pattern, which becomes quite costly for very large and deep hierarchies. As a consequence, its runtime increases with ontology size to a much larger degree than for the leaf case.



(a) Leaf type (low reasoning demand)



(b) Root type (high reasoning demand)

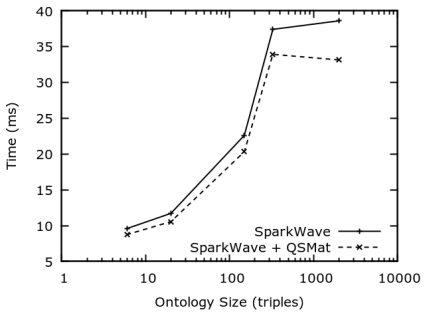


(c) Root type (excl. indexing overhead)

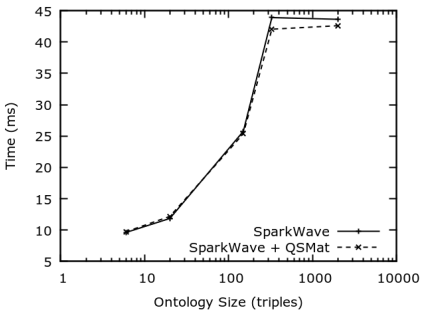
Fig. 4: Query evaluation time with regard to hierarchy size (Jena-based implementations, 100k stream triples / window)

The indexing of stream triples employed by our `Jena`-based query processor dominated query evaluation time for small ontologies, which masked differences between the evaluated algorithms for small ontology sizes and thus low reasoning demands. To emphasize these differences, Figure 4c shows the results for the more interesting root product type while excluding indexing overhead. This reveals that low hierarchy sizes are most beneficial to `StreamQR`, which then needs to create only a small number of sub-queries, leading to a performance advantage over the algorithms employing explicit reasoning. Results of the same static scaling test for a `SparkWave`-based reasoning backend with and without query-relevance filtering of background knowledge with `QSMat` are shown in Figure 5a and Figure 5b for leaf and root type products respectively. Since inference rules are fixed for `SparkWave`, the rule-set was not subjected to relevance-filtering. As above, leaf type queries allow `QSMat` to exclude more of the type hierarchy than

root type queries, which translates into a larger performance benefit compared to regular `SparkWave` for leaf types. The fairly small performance benefit for root types can be explained by `QSMat` discarding background knowledge that is irrelevant to the query even for root types.

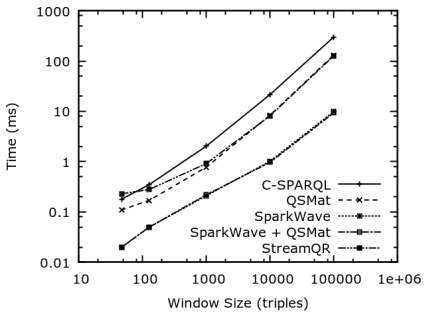


(a) Leaf product type (low reasoning demand)

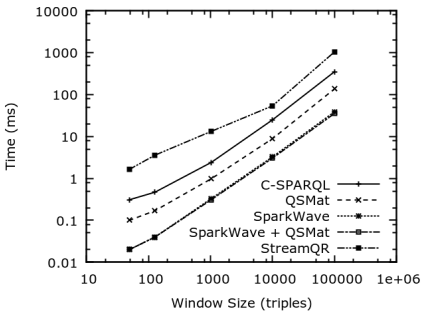


(b) Root product type (high reasoning demand)

Fig. 5: Query evaluation time with regard to hierarchy size (Sparkwave-based Implementations, 100k stream triples / window)



(a) Small type hierarchy (10 products in 7 product types)



(b) Large type hierarchy (10k products in 329 product types)

Fig. 6: Query evaluation time with regard to hierarchy size (Sparkwave-based Implementations, 100k stream triples / window)

Scalability with regard to stream window size. Another important consideration is scaling with regard to window size. While a larger ontology and more complex type hierarchy can require more reasoning overhead per stream triple, a larger window contains more stream triples triggering said reasoning. Since this benchmark solely requires independently classifying streamed instances in a static schema, reasoning demands per window triple do not depend on window size. The *query template* from above was reused for this test, but product types were sampled uniformly both among hierarchy levels and among types in each level, instead of the more pathological root and leaf types above. Figure 6a and Figure 6b show the results for a small and a large static ontology and thus hierarchy size. The cost per stream triple varies between algorithms, and is dependent on the static ontology size, but all evaluated algorithms scale fairly consistently

with growing window size, leading to the conclusion that per-triple reasoning demands are the primary factor regarding scaling.

4 Related Work

There are several existing approaches to accelerate stream reasoning. For our purposes, they can be broadly classified into approaches that focus either on faster query evaluation or reasoning.

The `C-SPARQL` [1, 3] engine is a query execution engine for C-SPARQL queries. Its query semantics distinguish between *logical windows*, which are time-based, and *physical windows*, which contain a given number of triples. Logical windows are advanced with a given time-step and can either overlap (*sliding*), or perfectly tile (*tumbling*). Unlike `QSMat`, the currently available implementation does not include materialization.

The `CQELS` [8] engine is a query processor employing window semantics and query operators similar to `C-SPARQL`, including tumbling and sliding windows. Query evaluation is triggered by the arrival of new triples (*streaming*) and not at the end of each window as with `C-SPARQL`. The engine further supports reordering of query operators during query runtime to yield a less costly execution order. However, unlike `QSMat`, it provides no reasoning support.

`SparkWave` [7] uses a query language closely related to `C-SPARQL` and supports time-based windows. In contrast to `C-SPARQL` and `CQELS`, it supports reasoning for a subset of RDFS entailment plus inverse and symmetric properties. The RETE-based algorithm combines a reasoning layer with a pattern matching layer. Both reasoning and query matching is streaming, i.e. triggered by newly arriving triples. As a tradeoff for increased performance, its query format is more restrictive than that of `C-SPARQL` (and thus `QSMat`) and `CQELS`. Unlike `QSMat`, the fixed subset of RDFS entailment rule cannot be customized.

The `Hoeksema S4 Reasoner` [6] is a C-SPARQL engine that supports a fixed subset of the C-SPARQL query language and RDFS entailment. Unlike `QSMat`, it is implemented as a distributed system of processing elements that are realized in S4. The approach only supports time-based windows, since count-based windows would require more synchronization between individual processing elements.

`IMaRS` [2] is a C-SPARQL engine with reasoning support for generic rule systems and incrementally maintains a full materialization. The approach supports only logical windows, which allows it to exploit the specific window semantics of C-SPARQL to precompute expiration times for newly derived triples. However, `QSMat` only needs to perform a partial materialization depending on the needs to satisfy the given stream query.

`StreamQR` [4] is a query rewriting approach that aims to circumvent the need for explicit reasoning by encoding schematic knowledge in a set of conjunctive sub-queries. That is possible, since this conversion is syntactical and independent of the specific stream triples arriving during query runtime. The sub-queries are then evaluated over the raw stream without the need for reasoning. This is both an advantage and a disadvantage compared to materialization like in `QSMat`:

The number of sub-queries generated depends on the complexity of the relevant background knowledge, hence it is most beneficial if this number is low. Both approaches perform analysis on a syntactic level, where `StreamQR` preprocesses the query with the help of relevant static information, while `QSMat` preprocesses static information with the help of the query.

5 Conclusions

We presented a flexible approach, `QSMat`, to accelerate stream reasoning by exploiting knowledge of the query and a configurable inference rule-set. It works backwards from the query, and extracts parts of the static materialization and the rule-set that are not provably irrelevant to the query, which allows it to maintain completeness with regard to the query. The performance evaluation of `QSMat` based on the Berlin SPARQL Benchmark (BSBM) revealed that it can reduce reasoning overhead significantly both on its own or as a preprocessing step for another state-of-the-art reasoner. Future work is concerned with cross-dependencies between multiple query triple patterns, which could yield a more restrictive filtering, thus exclude unneeded inferences that cannot provably be discarded on a per-pattern basis yet.

Acknowledgments: This research was partially supported by the German Federal Ministry for Education and Research (BMB+F) in the project INVERSIV and the European Commission in the project CREMA.

References

1. Barbieri, D. F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M. (2009): C-SPARQL: SPARQL for Continuous Querying. Proc. 18th International Conference on World Wide Web (WWW), ACM.
2. Barbieri, D., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M. (2009): Incremental Reasoning on Streams and Rich Background Knowledge. Proc. International Semantic Web Conference, LNCS 6088, Springer.
3. Barbieri, D. F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M. (2010): C-SPARQL: A Continuous Query Language for RDF Data Streams. *Semantic Computing*, 4(1):3–25.
4. Calbimonte, J.-P., Mora, J., Corcho, O. (2016): Query Rewriting in RDF Stream Processing. Proc. 13th Extended Semantic Web Conference, Springer.
5. Forgy, C.L. (1982): Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37
6. Hoeksema, J., Kotoulas, S. (2011): High-performance distributed stream reasoning using S4. Proc. of Workshop OrdRing at International Semantic Web Conference
7. Komazec, S., Cerri, D., Fensel, D.: (2012) Sparkwave: Continuous Schema-Enhanced Pattern Matching over RDF Data Streams. Proc. 6th ACM International Conference on Distributed Event-Based Systems; ACM.
8. Le-Phuoc, D., Dao-Tran, M., Parreira, J.X., Hauswirth, M. (2011): A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. Proc. 10th International Semantic Web Conference (ISWC); Springer